Cooperari: A Tool for Cooperative Testing of Multithreaded Java Programs

Eduardo R. B. Marques   Francisco Martins   Miguel Simões

Large-Scale Informatics Systems Laboratory (LASIGE), Departamento de Informática, Universidade de Lisboa, Portugal

Abstract

Bugs in multithreaded application can be elusive. They are often hard to trace and replicate, given the usual non-determinism and irreproducibility of scheduling decisions at runtime. We present Cooperari, a tool for deterministic testing of multithreaded Java code based on cooperative execution. In a cooperative execution, threads voluntarily suspend (yield) at interference points (e.g., lock acquisition), and code between two consecutive yield points of each thread always executes serially as a transaction. A cooperative scheduler takes over control at yield points and deterministically selects the next thread to run. An application test runs multiple times, until it either fails or the state-space of schedules is deemed as covered by a configurable policy that is responsible for the scheduling decisions. Beyond failed assertions in software tests, deadlocks and races are also detected as soon as they are exposed in the cooperative execution. Cooperari effectively finds, characterizes, and deterministically reproduces bugs that are not detected under unconstrained preemptive semantics, as illustrated by standard benchmark examples.

Categories and Subject Descriptors D2.5 [*Testing and Debugging*]: Testing Tools; D3.3 [*Language Constructs and Features*]: Concurrent programming structures

General Terms Languages, Reliability

Keywords concurrency, threads, software testing, cooperative execution

1. Introduction

Multithreaded programs are hard to test. A multithreaded system's scheduler typically performs context switches in a non-deterministic and irreproducible manner. As a result, it becomes impossible to control and observe thread interleaving appropriately. At the same time, multithreaded bugs are typically manifested only for a subset of all possible schedules, frequently relying on a very particular thread interleaving, that also tends to be hard to trace through debugging ("heisenbugs" are common). Even simple bug patterns [8, 9] may be elusive to detect and replicate precisely.

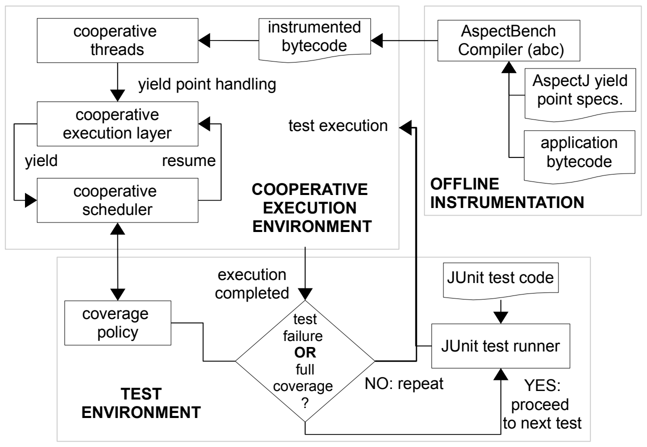


Figure 1: The Cooperari framework

The relevant context switches in a program are those that cause thread interference, e.g., access to shared data or lock operations. Under cooperative semantics [18, 19], context switches occur only at yield points that induce thread interference. A program's execution is cooperative if the code between two yield points of the same thread executes serially as a transaction without any interference from other threads. In the context of software testing, cooperative execution may be exploited to attain a reproducible and systematic coverage of the state-space of thread schedules, e.g., [4, 7, 15]. The idea is to schedule threads cooperatively, such that (deterministic) context switches are only done at yield points. Repeated executions of a program (test) may then potentially explore the state-space of thread schedules in a customized manner. We implemented a tool called Cooperari that enables this type of methodology for Java programs (https://bitbucket.org/edrdo/cooperari). To the best of our knowledge, it is the first of the kind for Java.

Cooperari works on top of an unmodified Java Virtual Machine (JVM) and employs offline instrumentation of JVM bytecode, coupled with cooperative execution and test environments at runtime. Its overall framework is illustrated in Fig. 1. Cooperari defines the interception of yield points using AspectJ (Fig 1, top-right). The yield points comprise Java monitor operations, thread lifecycle methods, and field and array data accesses. The AspectJ code is fed together with application bytecode to the AspectBench compiler (abc) [1] to produce instrumented bytecode that is loaded by a cooperative execution environment (Fig 1, top-left). When a thread reaches a yield point, the instrumented bytecode is executed, delegating control to a cooperative execution layer that ends up causing a voluntary thread yield. A cooperative scheduler becomes aware of this event, and deterministically decides the thread to run in succession. The cooperative scheduler has full control over the resumption

PPPJ'14, September 23 - 26 2014, Cracow, Poland.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2926-2/14/09…$15.00.
http://dx.doi.org/10.1145/2647508.2647523

of threads, ruling out interference from the built-in JVM scheduler. For this scheme to work, we resort to a cooperative implementation of monitor and thread primitives, invoked at yield points in place of the built-in, non-deterministic JVM support, e.g., for lock acquisition or condition-based notifications [14]. The final trait of the cooperative execution layer is the on-the-fly detection of deadlocks, signaled for lock acquisition cycles or blocked program states, and data races, signaled when a write yield point refers to the same data of a simultaneous write or read yield point of another thread.

The framework is completed by a testing environment (Fig. 1, bottom) that integrates with the popular JUnit testing library for Java (http://junit.org). Each test in a given suite may be executed several times, in the manner dictated by a configurable coverage policy. A test executes until it fails, presumably meaning that a bug has been found, or the coverage policy in place deems the state-space exploration of thread schedules as concluded. At this point, we implemented two policies: a pseudo-random (and deterministic) choice of thread to run coupled with a bound on the number of test executions, and a history-dependent policy that tries to avoid repeated scheduling decisions.

Cooperari is able to effectively detect, characterize, and deterministically reproduce bugs that are elusive or difficult to replicate in a preemptive execution. We demonstrate this with an evaluation of standard benchmark examples taken from the ConTest [8] and SIR [16] suites. In the rest of the paper, we begin by surveying related work (Section 2). We then describe Cooperari by example (Section 3), its implementation (Section 4), and benchmark evaluation (Section 5). We conclude with a short discussion of future work (Section 6).

2. Related work

A number of tools employ cooperative execution for deterministic testing of multithreaded applications. CHESS [15] is a tool for multithreaded Windows programs. It operates as a stateless model checker, enumerating all possible thread schedules, while a test repeatedly executes. CHESS relies on thin wrappers for multithreading primitives to identify non-deterministic choices at yield points, and various techniques to curb state-explosion such as iterative preemption bounding. Cloud9 [4] is a parallel symbolic execution engine for multithreaded C POSIX programs. During state-space exploration for symbolic assertion checking, Cloud9 relies on cooperative scheduling and a cooperative/symbolic implementation of a portion of the POSIX system calls, in particular for the pthreads API. CONCURRIT [7] is a tool for C++ multithread programs, employing a DSL for imposing thread schedule constraints. Tests execute repeatedly, with the guidance of a model checker that covers all schedules defined by the DSL constraints. The execution is cooperative, relying on yield calls at execution points in user-level code specified by the DSL (specific details are given in [5]). Compared with these tools, Cooperari works for Java programs, and allows a partial exploration of possible thread schedules through a deterministic pseudo-random search and a structural program coverage criterion. Like CHESS and Cloud9, Cooperari enables cooperative execution of multithreading primitives.

The use of cooperative execution rests on the premise of semantics preservation. The execution of a program under cooperative semantics should be equivalent to that obtained using a traditional preemptive scheduler. This happens if the considered yield points characterize all possible thread interference, a property Yi et al. call cooperability [18, 19]. The authors define a formal framework to reason on cooperative semantics of Java programs, plus tools for inference of yield points and assertion of cooperability. The yield point types we consider are in line with these works, and the discussion of synchronization coverage criteria in [3].

Beyond cooperative execution, a number of approaches can be considered in relation to thread scheduling and bug detection. Basic scheduling constraints for program events may be specified, e.g., as in [12]. Thread interference points may be instrumented with "noise" to induce random context switches [6, 17], and expose bugs effectively [9], though in non-deterministic manner. Active testing techniques, e.g., [13], employ biased random scheduling to cause cause thread yields at pre-determined program locations with high probability.

3. Cooperari by example

We illustrate the main features of Cooperari using two examples in this section: the classic dining philosophers problem, and a semaphore implementation.

Dining philosophers The dining philosophers problem is well-known. A group of N philosophers sits at a round table where N forks are placed in between each plate. To eat, a philosopher first grabs his left fork and then his right one. After eating, he puts both forks down. The philosophers get stuck if they all simultaneously grab the left fork, leaving no right fork available to be picked by anyone. The code in Fig. 2 (a) defines a Philosopher Java class. A Philosopher object is created using supplied left and right fork objects, and the run() method defines the behaviour of the philosopher. During execution, exclusive access to the forks is ensured by acquiring the corresponding locks using Java **synchronized** blocks, lines 12–16 for the left fork and 13–15 for the right fork. If no deadlock occurs in run(), the thoughts and food fields, initially set to **false**, will both be **true** at the end. From a cooperative execu-

(a) Philosopher code

```
1  package philosophers;
2  class Philosopher implements Runnable {
3   private Object left, right; // forks
4   private boolean thoughts = false;
5   private boolean food = false;
6   Philosopher(Object left, Object right) {
7    this.left = left;
8    this.right = right;
9   }
10  public void run() {        // 1. think
11   thoughts = true;
12   synchronized(left) {      // 2. get left fork
13    synchronized(right) {    // 3. get right fork and eat
14     food = true;
15    }                        // 4. release right fork
16   }                         // 5. release left fork
17  }
18  boolean hadThoughts() { return thoughts; }
19  boolean hadFood() { return food; }
20 }
```

(b) Dining philosophers test

```
@RunWith(CTestRunner.class)
@CTestOptions(coverage=RANDOM,
              instrument="philosophers")
public class TestPhilosophers {
 static final int N = ...;
 @Test public void testDinner() {
  Object f[] = new Object[N];
  for (int i = 0; i < N; i++)
   f[i] = new Object();
  Philosopher[] p = new Philosopher[N];
  for (int i=0; i < N; i++)
    p[i] = new Philosopher(f[i], f[(i+1)   runThreads(p);
  for (int i=0; i < N; i++) {
   assertTrue(p[i].hadThoughts());
   assertTrue(p[i].hadFood());
  }
 }
}
```

Figure 2: Dining philosophers example

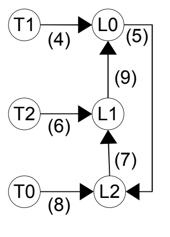
tion perspective, the beginning of each serial code transaction is marked 1 to 5 in the run() method, in line with the thread interference (yield) points defined by fork acquisition and release.

Fig. 2 (b) shows a JUnit test class, TestPhilosophers, with a test method, testDinner. The test defines the usual round table setup comprising N forks and N philosophers (lines 7–12). In the code, runThreads(p) (13) abstracts the launch of N philosopher threads, followed by a wait for the termination of all of them. When the threads complete, a sequence of JUnit-style assertions (14–17) verifies that each philosopher had a round of thought and food. The test almost always passes when executed under normal conditions: on a standard JVM, we observed a deadlock roughly once in 20000 test trials for N=2 philosopher threads, and we failed to observe the bug at all for a higher number of threads. In alternative to a standard test execution, the @RunWith(CTestRunner.**class**) annotation tells JUnit to use Cooperari's custom test runner (CTestRunner) to enable cooperative execution. In complement, the @CTestOptions annotation defines options for cooperative execution: the coverage=RANDOM option specifies that pseudo-random, deterministic scheduling decisions should be made at yield points, and the instrument="philosophers" option indicates the Java package whose classes should be instrumented for cooperative execution. The Cooperari test runner will repeatedly execute each test in the suite, testDinner alone in this case, until it either fails or reaches a maximum number of trials (by default 1000). The output of an execution is as follows, considering N=3 in the test code.

```
Instrumented code must be generated, please wait.
Changes have been detected.
Instrumentation completed in 5486 ms.
testDinner: executed 3 times in 68 ms [failed]
Failure trace for testDinner written to
  'log/TestPhilosophers_testDinner.trace.txt'
There was 1 failure:
1) testDinner(TestPhilosophers)
DeadlockError:  L0/T0/Philosopher.java:13
              > L1/T2/Philosopher.java:12
              > L2/T0/Philosopher.java:12
              > L0/T1/Philosopher.java:12
```

In the execution, bytecode instrumentation takes place (done just once). The actual test execution then proceeds and reports a failure after three executions of method testDinner (taking 68 milliseconds). Information is given regarding the location of a cooperative trace file, shown in Fig 3, where the actual thread schedule can be inspected, but the test output already allows for a partial understanding of the failure. All three threads are stopped with a DeadlockError exception. The error refers to a lock acquisition cycle found at line 13 in Philosopher during the execution of one of the threads (T0), at a time where every thread (T0/1/2) had each previously acquired a lock for the left fork (line 12 in Philosopher); the locks are identified as L0, L1, and L2, each referring to a distinct fork object. The same outcome will be reproduced the next time the test suite is executed, but without instrumenting code again, as long as the Philosopher code is not changed.

```
<step> <thread> <yield point>
1 0 [begin]
2 1 [begin]
3 2 [begin]
4 1 monitorenter(L0) Philosopher.java:12
5 1 monitorenter(L2) Philosopher.java:13
6 2 monitorenter(L1) Philosopher.java:12
7 2 monitorenter(L0) Philosopher.java:13
8 0 monitorenter(L2) Philosopher.java:12
9 0 monitorenter(L1) Philosopher.java:13
```



$$\begin{tikzcd}
T1 \arrow[r, "(4)"'] & L0 \arrow[dd, bend left, "(5)"] \\
T2 \arrow[r, "(6)"'] & L1 \arrow[u, "(9)"'] \\
T0 \arrow[r, "(8)"'] & L2 \arrow[u, "(7)"']
\end{tikzcd}$$

Figure 3: Cooperative execution for the dining philosophers

The cooperative trace of Fig. 3 provides more detail. In the figure, lock acquisition yield points are identified by monitorenter, the name of the corresponding JVM bytecode instruction. All threads are initially suspended at startup (steps 1–3). After startup, thread 1 is allowed to run for two steps (4–5): the first step takes the thread to the point of left fork acquisition, and the second one (with the left fork lock now effectively acquired) to the point of right fork acquisition. Thread 1 then yields, and the same pair of steps is allowed in succession for threads 2 (6–7) and 0 (8–9). A deadlocked state is thus reached, as every philosopher holds the left fork, but neither is able to acquire the right one. For deadlock detection, Cooperari progressively builds a resource graph [11], as lock-related yield points are intercepted. The graph ends up having a cyclic configuration, shown right in the figure, in significance of a deadlock.

Semaphore example Our second example concerns a semaphore implementation and an associated test, shown in Fig. 4. The example illustrates different types of yield points and deadlocks, as well as data race detection by Cooperari. In line with the traditional formulation, a semaphore represents a non-negative integer that can only be atomically incremented or decremented, using up() and down(), respectively. Whenever a thread tries to down() a zero-valued semaphore, it blocks until the semaphore is increased by a call to up() by another thread. The test shown in Fig. 4 (b) creates a semaphore with an initial value of N−1, shared by N Client threads. Each thread proceeds (in the run() method) by decrementing the semaphore, doing some work, and incrementing the semaphore back before terminating. At the end, the test passes if the semaphore's value is equal to the initial one, N−1.

(a) Semaphore code

```
class Semaphore {
 private int value;
 Semaphore(int initial) { value = initial; }
 int getValue() { return value; }
 void down() throws InterruptedException {
  synchronized (this) {
   while (value == 0) { wait(); }
   value--;
  }
 }
 void up() {
  synchronized (this) { value++; }
  if (value == 1) {
   synchronized (this) { notify(); }
  }
 }
}
```

(b) Semaphore test

```
static class Client implements Runnable {
 private Semaphore sem;
 Client(Semaphore sem) { this.sem = sem; }
 public void run() {
  try {
   sem.down();
   // do some work
   sem.up();
  }
  catch(InterruptedException e) { }
 }
}
static final int N = ...;
@Test public final void test() {
 Semaphore sem = new Semaphore(N-1);
 Client[] c = new Client[N];
 for (int i=0; i < N; i++) c[i] = new Client(sem);
 runThreads(c);
 assertEquals(N-1, sem.getValue());
}
```

Figure 4: Semaphore example

The semaphore class employs the condition-based wait() and notify () methods associated to Java monitors, the core synchronization primitives used by Java applications and thread-safe Java API classes [14] (along with notifyAll ()). The wait() call in down() causes the calling thread to block until it receives: (1) a notification from another thread through notify, called within up(); (2) a spurious wakeup, an odd but possible event, and the reason for the standard wait-loop pattern shown [14]; or (3) an interrupt resulting from a call to Thread.interrupt(), a case not exercised by the code at stake. To resume from wait(), the thread then needs to compete with other threads to reacquire the lock.

In the semaphore code of Fig 4 (a), the up() operation only calls notify () when the semaphore increments to 1 (lines 13–14). The code would be correct if a single **synchronized** block covered all instructions in up(). Instead, two blocks of the kind are used, a "two-stage access" pattern [9]. Moreover, the notification event relies on an unsynchronized read access to value (line 13), which may race with simultaneous write accesses. It is then possible that two or more increments in up() take the semaphore value from 0 to a value greater than 1. A required notification, in case some thread is blocked in down(), may be skipped and, as a result, a waiting thread may block forever. Skipped notifications would also be possible if the read access to value was part of the second **synchronized** block, but the purpose of the example is also to illustrate data races.

Cooperari detects the race condition and the wait deadlock, considering the yield points defined by lock acquisition and release, wait(), notify (), and the read/write accesses to the semaphore value. The output of an execution for N=3 is as follows:

```
Race: T0 at Semaphore.java:12 over Semaphore.value
Race: T1 at Semaphore.java:13 over Semaphore.value
Failure trace for test  written to
    'log/examples.semaphore.TestSemaphore_test.trace.txt'
test: executed 36 times in 578 ms [failed]
1) test(examples.semaphore.TestSemaphore)
    WaitDeadlockError: { T2/Semaphore.java:7 }
```

The output reports a race over the semaphore field and a test failure due to a WaitDeadlockError for a thread identified as T2. A fragment of the cooperative trace is shown in Fig. 5, where each step is annotated with pending reads and writes (`r/w`) for the semaphore value. In the execution at stake, thread 0 and 1 (T0, T1) completed down(), thus value will be 0, and began executing up(), whilst thread 2 (T2) is executing down(). At step 23, T0 and T1 are suspended at the first lock acquisition point in up(), and T2 is blocked at the call to wait() in down(). In steps 23–27 and 28–31, T0 and T1 are in succession able to acquire the lock, increment the semaphore value, relinquish the lock, and suspend again before reading value. When they do, just before terminating (steps 32 and 33), value is 2, hence they fail to deliver the notification to T2. Thus, T2 will block forever on wait(). The deadlock is detected at this point. As for the race, it is signaled at step 29, for a pending write by T0 and a pending read by T1.

```
<step> <thread> <yield point>                    # r/w
15 0 monitorenter(L0) Semaphore.java:12          # {}/{}
...
22 1 monitorenter(L0) Semaphore.java:12          # [value=0]
23 2 wait(L0) Semaphore.java:7                   #
24 1 get(Semaphore.value) Semaphore.java:12 # 1/{}
25 1 set(Semaphore.value) Semaphore.java:12 # {}/1
26 1 monitorexit(L0) Semaphore.java:12           # [value=1]
27 1 get(Semaphore.value) Semaphore.java:12 # 1/{}
28 0 get(Semaphore.value) Semaphore.java:12 # 0,1/{}
29 0 set(Semaphore.value) Semaphore.java:12 # 1/0 [race]
30 0 monitorexit(L0) Semaphore.java:12           # [value=2]
31 0 get(Semaphore.value) Semaphore.java:13 # 0,1/{}
32 0 <end> # {}/{};read 2; no call to notify()
33 1 <end> # read 2; no call to nofity()
```

Figure 5: Cooperative execution for the semaphore test

4. Implementation

Yield point instrumentation Cooperari employs AspectJ to instrument yield points. An aspect (on what concerns Cooperari) is a collection of pointcuts and advices. A pointcut is an expression that describes well-defined program execution points (e.g., a method call), called join points. An advice defines code that is executed when the instrumented program reaches a join point that matches the advices' pointcut. We use the AspectJ abc compiler [1] that supports **lock**() and **unlock**() pointcut extensions [2] for intercepting the execution of the JVM lock acquisition (monitorenter) and release (monitorexit) instructions.

Presently, Cooperari defines aspects for specifying yield points related to: (1) lock acquisition, release, and condition-based synchronization through wait(), notify (), and notifyAll ()); (2) thread interruption, sleep, join, yield hints, startup, and termination [14]; (3) object field reads and writes; and, finally, (4) array reads and writes. The time-based primitive Thread.sleep() and the time-based variants of Object.wait() and Thread.join() are executed within the cooperative framework, but not deterministically, since their completion depends on the elapse of time measured by the JVM and, obviously, on all program/JVM actions within that interval. In complement, the tool intercepts thread state methods, like Thread.holdsLock() or Thread.getState(), to attain a coherent cooperative semantics, even if these calls are not yield points.

```
 1  void around(Object o) : lock() && args(o) {
 2    CThread t = getCThread(thisJoinPoint, o);
 3    if (t != null) t.cMonitorEnter(o);
 4    else proceed(o);
 5  }
 6  void around(Object o) : unlock() && args(o) {
 7    CThread t = getCThread(thisJoinPoint, o);
 8    if (t != null) t.cMonitorExit(o);
 9    else proceed(o);
10  }
11  boolean around(Object o) :
12   call(boolean Thread.holdsLock(Object)) && args(o) {
13    CThread t = CThread.intercept(thisJoinPoint, o);
14    if (t != null) return t.cHoldsLock(o);
15    else return proceed(o);
16  }
```

Figure 6: Sample AspectJ instrumentation code

Fig. 6 presents the general pattern of instrumentation we follow. It contains three advices: the first (lines 1–5) associates code that is executed upon lock acquisition, specified by pointcut **lock**(); the second (6–10) intercepts lock release, described by pointcut **unlock**(); finally, the third (11–16) associates with a Thread.holdsLock() method call pointcut. The **around** keyword in the code specifies that these advices run in place of the join point. Looking at the first advice, the code starts by determining if the current thread is subject to cooperative execution semantics, through a call to getCThread() (the **thisJoinPoint** and o arguments are used to initialize profile information for the yield point). If so, the execution is diverted to the cMonitorEnter() method of the cooperative thread to run in place of JVM lock acquisition. Otherwise, the **proceed** AspectJ keyword specifies normal execution of the join point instead, i.e., lock acquisition proceeds normally; note that instrumented code may run in non-cooperative manner in the JUnit runner thread before or after the invocation of method runThreads() that creates the cooperative threads. The other two advices in the figure are defined similarly, and so is the remaining Cooperari instrumentation.

The cooperative scheduler We now turn to the essential traits of the implementation of cooperative scheduling. The scheduler runs in its own thread and only one cooperative thread is active at any time. A cooperative thread voluntarily suspends its execution

(a) Thread yield and resumption

```
private boolean yield;
public void cYield() {
  ... // yielding
  yield = true;
  syncYield();
  while (yield) {
    LockSupport.park();
  }
  syncResume();
  ... // resumed
}
public void cResume() {
  ...
  yield = false;
  LockSupport.unpark(this);
}
```

(b) Scheduler step

```
CoveragePolicy policy;
List<CThread> ready;
...
public void cStep() {
 ...
 syncYield();
 CThread t =
   policy.decision(ready);
 t.cResume();
 syncResume();
 ...
}
```

Figure 7: Implementation of cooperative scheduling

when it reaches a yield point. The cooperative scheduler assumes control at this point, deciding the next thread to run, and resuming the execution of the chosen thread. The built-in JVM scheduler is conditioned in this process by the actions of the cooperative scheduler and cooperative threads. The implementation is outlined in Fig. 7, comprising the definition of the two core primitives for thread yield and resumption, cYield() and cResume() methods in a CThread class (a), and for the scheduler step, method cStep() in a CScheduler class (b).

During the yield procedure (Fig. 7 (a)) a thread starts by enabling the yield condition, and synchronizes with the scheduler with a call to syncYield(), a barrier handshake. The thread then suspends execution using the LockSupport.park() Java API call that disables the thread for execution by the JVM scheduler. In symmetry, during a scheduling step (b), the cooperative scheduler begins by acknowledging a thread yield (syncYield()). It then employs the coverage policy in place (policy) to select the next thread t to run amongst the ready thread set (ready), and executes t.cResume(), disabling the yield condition for t and unblocking it through the unpark() call (a). The JVM scheduler may now consider t for execution. The resumption process ends with a handshake between the cooperative scheduler and t, syncResume() in (a) and (b).

Cooperative implementation of multithreading primitives The implementation of multithreading primitives is defined in class CThread, helped by a CMonitor class for representing monitors, and a COperation base class that implements the behavior of primitives. Each primitive executes an action before thread suspension and another after resumption. For instance, the CThread.cMonitorEnter() method (Fig. 6) for lock acquisition is implemented by: (1) initializing a COperation object for the pending monitor acquisition; (2) yielding (invoking cYield()); and (3) completing the lock acquisition when resumed (through cResume()).

Fig. 8 outlines part of the support for monitor operations. The CMonitor data (a) is manipulated for distinct types of COperation corresponding to monitor acquisition, release, notification, and wait wakeup. The support for lock acquisition is shown as an example in (b). As illustrated, each operation comprises initialization, state report, and completion methods. Only ready-state (CREADY) operations enable resumption for the associated thread. The lock acquisition operation is implemented as follows:
— A monitor for o is created by getMonitor(o) for the first pending lock on o, in the initialization step of lock acquisition, otherwise the reference count for the existing monitor object is incremented. Symmetrically, the reference count will be decremented in the completion of the mirror operation of lock release, and the monitor will be disposed at that time if the reference count reaches 0.
— A CREADY state is signaled when the monitor has no owner (is **null**), letting the thread be considered for execution. The thread

(a) Monitor data

```
class CMonitor {
 // Reference count.
 int refCount;
 // Owner thread
 CThread owner;
 // Wait count
 int waitCount;
 // Notification epoch
 long nEpoch;
 // Notification queue
 Queue<Integer> nQueue;
}
```

(b) Lock acquisition

```
void init(Object o) {
  m = getMonitor(o);
  m.refCount++;
}
CState getState() {
 return
   m.owner == null ?
     CREADY : CBLOCKED;
}
void complete(CThread t) {
  m.owner = t;
}
```

Figure 8: Implementation of monitor operations

may go back to a CBLOCKED state, if a competing thread is chosen instead by the scheduler and acquires the lock.
— If the scheduler lets the thread run, the completion stage of the operation marks the monitor as owned by the thread.

Test execution and state-space exploration The cooperative scheduler uses a coverage policy to determine the next thread to run at each step. The coverage policy is also responsible for tracking state-space exploration and bounding the required number of test trials. After executing each test, the test runner evaluates if the test failed. If so, no more trials are executed. Otherwise, the runner asks the coverage policy if more test trials are required. If so, the test is repeated. At this point we implemented two coverage policies: a memoryless pseudo-random choice of threads, and a history-dependent one that seeks to avoid repeated scheduling decisions for the same program state.

The random policy is implemented using a pseudo-random number generator, always initialized with a fixed seed for reproducible test sessions, and a bound on the number of test trials. The history-dependent policy improves on the latter, by trying to avoid repeated scheduling decisions. To implement this policy, the state of running threads is abstracted as a set of the form $s = \{(n_1, pc_1), \ldots, (n_k, pc_k)\}$. Each $(n, pc) \in s$ defines $n > 0$ threads suspended at location pc, where pc corresponds to the stack trace information obtained via Thread.getStackTrace() augmented with information for the current yield point. History is maintained as a set of pairs (s, pc), where s is a state abstraction and pc is a program location representing a past scheduling decision. At each step, for state s, the policy deterministically tries to find a thread t at location pc such that (s, pc) is not part of the history set. If so, it decides on scheduling t. If no such thread can be found, the selection is random as in the first policy.

The aim of the program state abstraction scheme is to curb state-space explosion by avoiding sequences of scheduling decisions that commute [10]. However, it may rule out relevant schedules as it only accounts for simple structural program information (the stack trace, and yield point information). In any case, the history-dependent policy cannot guarantee an enumeration of all thread schedules, as no backtracking mechanism ensures a visit to states where past unexplored scheduling decisions lie. A model checker can be considered in principle to address this issue, and integrate orthogonally with the remaining infrastructure of Cooperari, e.g., see [7, 15].

Deadlock detection Cooperari employs two mechanisms for deadlock detection. The first monitors if all threads are blocked on a multithreaded synchronization primitive. This means alive threads are unable to progress. The second uses a resource graph [11] to keep track of cyclic dependencies through lock acquisitions. Edges are added to the graph in the initialization stage of lock acquisitions, and removed in the completion stage of lock releases. The graph works in tandem with a chain of lock acquisitions maintained

Example	LOC	Inst. Time	hist. dep. coverage					random coverage					unconstrained execution				
			2	4	8	16	32	2	4	8	16	32	2	4	8	16	32
Alarm Clock [16]	210	18.9	*1000*	7	3	1	2	*1000*	8	13	8	6	*1000*		—— **1000** ——		
			64.9	1.4	1.1	0.7	1.2	62.3	1.4	2.2	1.9	1.8	51.0	51.4	51.6	52.0	52.7
Apache Common Lang [16]	398	26.0	1	1	1	1	1	1	1	1	1	1			—— **1000** ——		
			0.2	0.3	0.4	0.6	0.8	0.2	0.2	0.4	0.5	0.8	0.3	0.6	0.9	1.4	2.2
Bank [8]	77	11.7	1	1	1	3	1	1	1	1	2	1			—— **1000** ——		
			0.1	0.2	0.4	1.0	1.0	0.1	0.2	0.3	1.1	1.0	1.4	1.5	1.9	2.6	3.7
Clean [8, 16]	63	11.4	3	4	2	1	1	21	2	1	1	1	25	3	1	1	1
			0.4	2.0	1.7	1.4	1.6	2.9	0.9	1.1	1.3	1.6	0.1	0.1	0.1	0.1	0.1
Dining Philosophers	29	5.6	2	9	48	581	**1000**	4	13	165	**1000**	**1000**			—— **1000** ——		
			0.1	0.2	1.0	18.8	189.4	0.1	0.2	2.4	23.8	46.2	0.3	0.4	0.8	1.3	2.3
Linked List [8, 16]	150	13.3	2	1	1	1	1	2	4	1	1	1			—— **1000** ——		
			0.1	0.2	0.4	1.0	1.2	0.1	0.2	0.1	0.2	0.4	0.1	0.6	0.9	1.4	1.7
Merge Sort [8]	98	12.1	99	117	95	54	4	99	11	51	14	35			—— **1000** ——		
			4.6	6.8	17.7	26.9	3.2	4.2	2.5	4.6	3.5	9.5	0.3	0.4	0.7	1.3	2.3
Piper [8, 16]	102	14.0	*1000*	2	2	1	1	*1000*	3	1	1	1	*1000*	2	1	1	1
			7.7	0.3	0.3	0.6	0.8	7.2	0.1	0.2	0.4	0.7	0.7	0.1	0.1	0.1	0.1
Reorder [8, 16]	48	11.0	50	13	4	7	20	2	23	26	17	10			—— **1000** ——		
			0.4	0.2	0.2	0.5	1.5	0.1	0.3	0.7	0.8	0.9	0.3	0.4	0.8	1.4	2.3
Semaphore	29	5.6	*1000*	37	137	**1000**	**1000**	*1000*	8	249	**1000**	**1000**	*1000*		—— **1000** ——		
			6.5	0.7	2.7	34.4	73.1	6.0	0.2	2.4	32.1	63.0	0.3	0.5	0.8	1.2	1.9
Two Stage [8, 16]	70	13.3	52	57	11	15	28	324	141	157	92	46			—— **1000** ——		
			1.1	1.9	1.1	0.3	3.2	3.6	2.5	3.6	0.3	4.0	0.3	0.4	0.7	1.3	2.4
Wrong Lock [8, 16]	63	12.5	5	1	2	7	1	5	1	2	3	1			—— **1000** ——		
			0.1	0.1	0.2	0.9	0.5	0.1	0.1	0.2	0.5	0.5	0.3	0.4	0.8	1.3	2.2

Table 1: Benchmark results

for each thread. Edges are added to the graph for a thread t locking monitor m in the following cases: (1) t $\rightarrow$ m for a thread t and monitor m if t owns no locks (the lock chain of t is empty) in the current state; or (2) m' $\rightarrow$ m, if m' is the last lock acquired by t (the lock chain of t ends with m') before trying to acquire m. In both cases, m is appended to the lock chain of t. When t releases m, in reverse manner, the edge and the lock chain's tail are removed. Deadlocks are easily monitored by checking for the existence of cycles in the graph.

Race detection Cooperative scheduling naturally exposes race conditions, as illustrated for the semaphore example in Section 3, enabling a simple detection scheme. Cooperari records information about pending read and write yield points for each data item. Just before a thread yields on a data access, we increment a read or write counter for the data item, and the same counter is decremented when the thread resumes. A race is signaled whenever a completing write detects a pending read or write, or when a completing read detects a pending write. The read-write information is discarded if both read and write counters reach 0, meaning that there are no pending operations for the data item. The race detection scheme is costly in the sense that every field or array access is instrumented and monitored in the current implementation, and only a small portion of them may refer to effectively shared data. To deal with this issue, thread-locality static analysis [2] and yield point inference [19] can potentially be employed. In any case, the shortcomings are partially mitigated by the simplicity and preciseness of race detection, which can otherwise be more complex and imprecise in other approaches, e.g., see [2, 13].

5. Evaluation

We conducted an evaluation over 12 multithreaded program examples, with results given in Table 1. The examples are from the ConTest suite [8] and the SIR repository [16], plus the dining philosophers and semaphore examples of Section 3. Some ConTest examples are also in the SIR repository, and we used the SIR versions in those cases. The SIR/Contest test subjects' original code was used, and associated test programs were converted into JUnit tests. The bugs at stake comprise deadlocks for Bank, Clean, Dining Philosophers, Piper, and Semaphore, and failed test assertions for the remaining examples. Monitor-based synchronization is employed in all cases except for Apache Common Lang, Merge Sort, and Reorder, where plain data races lead to failed test assertions. All examples are parametric in the number of threads.

We ran the tests using a standard Java 7 JVM on a lightly loaded Linux machine with a dual-core 3 GHz CPU and 4 GB of RAM. For each example, we indicate in Table 1 the lines of code (LOC) in the test subject, roughly 30 to 400 LOC, and the execution time for bytecode instrumentation in seconds (Inst. time), less than 30 seconds in all cases. The remaining columns compare cooperative test execution, using the history-dependent or random coverage policies, versus unconstrained execution, with varying number of threads from 2 to 32. For each test setting, the results indicate the number of test trials executed, on top for each entry, and the execution time in seconds, at bottom. The maximum number of trials considered was 1000. The times and test trials in the unconstrained execution are the average of 10 executions for each setting. Entries in italic for some 2-thread settings (Alarm Clock, Piper, and Semaphore) indicate that the bug at stake is guaranteed not to occur, hence 1000 test trials are expected. Bold entries indicate that the bug may occur but is not reproduced after 1000 trials.

In all but two cases, Clean and Piper, cooperative execution exposes bugs that unconstrained execution cannot. Moreover, cooperative executions require a small number of trials in many of the examples. The bugs fail to reproduce after 1000 trials only for the dining philosophers' 32-thread setting and the semaphore's 16/32 thread settings. These two examples require a very precise schedule for deadlock, as discussed in Section 3. We can also observe that the history-dependent policy is more effective in exposing them by avoiding repeated scheduling decisions, but otherwise the random policy has comparable performance. The overhead imposed by cooperative execution is noticeable in the 1000-trial runs, e.g., approximately 20 times slower for the 2-thread setting in the semaphore example, an issue that is mitigated by the execution of a smaller number of trials in all other cases.

6. Conclusion

We presented Cooperari, a tool for reproducible, deterministic testing of multithreaded Java software, with the following core traits:

the instrumentation of thread interference points for cooperative execution; an execution environment established by a cooperative scheduler, a cooperative implementation of multithreading primitives, and runtime detection of deadlocks and races; and an environment for reproducible tests, in association to custom coverage policies for the exploration of the state-space of thread schedules. In the future, we plan on covering a wider set of multithreading Java primitives, e.g., atomic operations or barrier synchronization. For systematic state-space coverage, we wish to embed a model checker in the framework, along with other features for robust testing, such as yield point inference. We wish to drive these efforts with larger, real-world applications in mind.

References

[1] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Abc: An Extensible AspectJ Compiler. volume 3880 of *LNCS*, pages 293–334. Springer-Verlag, 2006.

[2] E. Bodden and K. Havelund. Racer: Effective Race Detection Using AspectJ. In *Proc. 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 155–166. ACM, 2008.

[3] A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur. Applications of synchronization coverage. In *Proc. 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 206–212. ACM, 2005.

[4] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel symbolic execution for automated real-world software testing. In *Proc. 6th European Conference on Computer Systems*, EuroSys '11, pages 183–198. ACM, 2011.

[5] J. Burnim, T. Elmas, G. Necula, and K. Sen. CONCURRIT: Testing concurrent programs with programmable state-space exploration. In *Proc. 4th USENIX Conference on Hot Topics in Parallelism*, HotPar '12. USENIX, 2012.

[6] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for testing multi-threaded Java programs. *Concurrency and Computation: Practice and Experience*, 15(3–5):485–499, 2003.

[7] T. Elmas, J. Burnim, G. Necula, and K. Sen. CONCURRIT: a domain specific language for reproducing concurrency bugs. In *Proc. 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 153–164. ACM, 2013.

[8] Y. Eytani and S. Ur. Compiling a benchmark of documented multi-threaded bugs. In *Proc. 18th International Parallel and Distributed Processing Symposium*, IPDPS '04, pages 266–273. IEEE Computer Society, 2004.

[9] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *Proc. 17th International Symposium on Parallel and Distributed Processing*, IPDPS '03, pages 286.2–. IEEE Computer Society, 2003.

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

[11] R. C. Holt. Some deadlock properties of computer systems. *ACM Computing Surveys*, 4(3):179–196, 1972.

[12] V. Jagannath, M. Gligoric, D. Jin, Q. Luo, G. Rosu, and D. Marinov. Improved multithreaded unit testing. In *Proc. 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 223–233. ACM, 2011.

[13] P. Joshi, M. Naik, C. Park, and K. Sen. CalFuzzer: An extensible active testing framework for concurrent programs. In *Proc. 21st International Conference on Computer Aided Verification*, CAV '09, pages 675–681. Springer-Verlag, 2009.

[14] D. Lea. *Concurrent programming in Java: design principles and patterns, 2nd edition*. Addison-Wesley, 1999.

[15] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *Proc. 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI '08, pages 267–280. USENIX, 2008.

[16] SIR. Software-artifact Infrastructure Repository. http://sir.unl.edu.

[17] D. Stoller. Testing Concurrent Java Programs using Randomized Scheduling. In *Proc. 2nd Workshop on Runtime Verification*, RV '02, pages 142–157. Elsevier, 2002.

[18] J. Yi, C. Sadowski, and C. Flanagan. Cooperative reasoning for preemptive execution. In *Proc. 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 147–156. ACM, 2011.

[19] J. Yi, T. Disney, S. N. Freund, and C. Flanagan. Cooperative types for controlling thread interference in Java. In *Proc. 2012 International Symposium on Software Testing and Analysis*, ISSTA '12, pages 232–242. ACM, 2012.