A Scala Library for Testing Student Assignments on Concurrent Programming

Mikaël Mayer
EPFL, Switzerland
mikael.mayer@epfl.ch

Ravichandhran Madhavan
EPFL, Switzerland
ravi.kandhadai@epfl.ch

Abstract

We present a lightweight library for testing concurrent Scala programs by systematically exploring multiple interleavings between user-specified operations on shared objects. Our library is targeted at beginners of concurrent programming in Scala, runs on a standard JVM, and supports conventional synchronization primitives such as `wait`, `notify`, and `synchronized`. The key component of the library is the trait *SchedulableMonitor* that accepts a thread schedule, and interleaves as per the schedule all user-specified operations invoked through multiple threads on objects implementing the trait. Using our library, we developed a unit test engine that tests concurrent operations on shared objects on thousands of schedules obtained by bounding the number of context-switches. If a unit test fails on a schedule, the test engine offers as feedback the interleaved traces of execution that resulted in the failure. We used our test engine to automatically test and evaluate two assignments: (a) lock-based producer/-consumer problem, and (b) lock-free sorted list implementation, offered to a class of 150 under-graduate students of EPFL. Our evaluations show that the system is effective in detecting bugs in students' solutions.

Categories and Subject Descriptors D.1.3 [*Software*]: Concurrent Programming; D.2.5 [*Software Engineering*]: Testing and Debugging

General Terms Languages, Reliability, Verification

Keywords Testing, Debugging, Concurrency, Synchronization, Scheduling, Deadlocks, Grading

1. Introduction

The increasing adoption of massively parallel and distributed hardware has created a huge demand for softwares that are able to improve their performance by exploiting the available parallel hardware. In response to this demand, languages and runtimes have evolved significantly in providing higher-level language abstractions for creating concurrent computations, such as *C# Tasks* and *Scala Futures*, as well as for coordinating accesses to resources shared across concurrent threads of execution, such as `synchronized`, `wait/notify`, and atomic integers supported by Java virtual machine (JVM). While these features have largely simplified programming concurrent softwares, discovering and fixing bugs in concurrent programs remains a considerable challenge, especially when they result from interleavings of effectful computations executed concurrently.

Automated detection of such *concurrency bugs* has, justifiably, attracted much research, and has witnessed significant advances in the recent years: [6, 16–19, 21] to name a few. These systems have been demonstrated to be effective in detecting complex concurrency bugs on large code bases. However, when seen from a larger perspective of software testing, the existing systems mostly address the problem of bounded model checking of concurrent programs through either symbolic or explicit state exploration. They employ heavy-weight static analyses like symbolic execution, and/or run on instrumented runtime infrastructures with dedicated schedulers for exploring multiple interleaving between instructions of concurrent operations. They typically target fine-grained hazards such as data races, memory safety, or exceptions/assertion-failures in the program. In comparison, less attention has been paid to designing unit testing frameworks for concurrent programs that allow programmers to check high-level properties of code snippets on user-provided inputs, analogous to `JUnit` and `ScalaTest`. Such unit testing libraries, despite being less automated than bounded model checkers, offer more flexibility to users, and do not require additional infrastructure to execute tests. Furthermore, they serve as a way to specify requirements of a software under development, and are integral to test-driven software development methodologies such as extreme programming [3]. In particular, our motivation for exploring such frameworks is to create a grading system where students can automatically test their (partial) solutions of the

SCALA'16, October 30–31, 2016, Amsterdam, Netherlands

http://dx.doi.org/10.1145/2998392.2998395

assignments, and use the failed tests as a guidance to solve the assignments. While there are approaches that try to combine model checking with unit testing frameworks [14, 22], it is typically hard to preserve the benefits of both due to above mentioned incompatibilities in their goals.

However, a factor limiting the usefulness of unit testing frameworks in a concurrent setting is that it is often necessary to test an operation when its instructions interleave with that of other operations that may execute concurrently. Yet, existing unit testing frameworks do not offer any support for scheduling (or ordering) operations executed by multiple threads in a systematic way. In this paper, we propose a library for Scala that provides functionalities for sequentializing operations executed by multiple threads, based on an ordering provided as input. Users of the library can programmatically mark the operations, such as reads/writes of shared variables, whose interleavings need to be tested, and can write unit tests that checks properties that has to hold for all interleavings of the marked operations executed concurrently from user-defined threads. The properties are tested on thousands of different interleavings between the marked operations executed by the threads, referred to as schedules. Moreover, our library is implemented using standard Scala features, compiles with standard `scalac`, and can be imported into any Scala project, much like `JUnit` and `ScalaTest` library. These aspects increase the portability of the test suites written using our library across different versions of Scala compiler, and/or JVMs.

The design of our library is primarily motivated by the need to specify properties about concurrent operations without relying on their concrete implementations. This allows programmers (students in our case) to program against a predefined set of test cases, and hence continuously check the code under development for compliance with the requirements, and identify bugs early. This approach has been particularly helpful for students to learn and solve programming assignments more effectively, and has been a standard practice in Scala programming courses, especially those offered as MOOCs (`https://www.coursera.org/specializations/scala`). However, being completely opaque to the implementation is impossible to achieve in our setting, since the library unlike a runtime cannot keep track of the writes/reads to shared memory locations that an unknown implementation could possibly access. Nonetheless, we circumvent this problem by providing students with a well-marked set of (atomic) operations that update shared memory locations to which they are required to restrict their implementations. To summarize, the following are the main contributions of this paper:

- We present an algorithm that uses standard Scala constructs to enforce an ordering between explicitly marked operations in a program, in the presence of synchronization primitives: `synchronized`, `wait`, `notify` and `notifyall`.
- We present an unit testing library for Scala (`https://github.com/epfl-lara/muscat`) capable of checking properties on multiple interleavings of operations explicitly marked by the user.
- We use the unit testing library to test and evaluate two assignments on concurrent programming offered to a class of 150 under-graduate students of EPFL. Our evaluations show that the unit tests were able to unravel deep concurrency bugs in students' solutions, as well as provide them useful feedback to locate and fix bugs.

To make things concrete, in following section we present an overview of the library using an implementation of a blocking, concurrent, circular queue shown in Fig. 1, and a unit test suite for the queue shown in Fig. 2.

2. Illustration: Blocking, Concurrent Queue

Fig. 1 shows the definition of a class `Queue` that implements a finite queue of capacity `size`. It uses an array `buffer` to store the elements of the queue, and maintains an index `_head` to the head of the queue, and a counter `_count` that tracks the number elements in the queue. These fields constitute the mutable state of the queue. We define a set of accessor and mutator functions to access and update the state as shown in lines 7 to 20. Each accessor function reads or writes exactly one memory location of the mutable state, and hence constitutes a logically atomic operation (even though the operation may be compiled down to several bytecode instructions). The queue supports `enqueue` and `dequeue` operations, which can be invoked concurrently. The operations are guarded by the synchronized construct to prevent race conditions on the mutable state of the queue. However, if an operation cannot be completed because the queue does not have enough elements to dequeue or has no space to enqueue, the thread executing the operation is blocked using `wait` until it is notified by a subsequent change to the state of the queue. Note that the `wait` construct releases the lock acquired by `synchronized`, and on notification would try to re-acquire the lock before resuming the execution. (The Java documentation [1] provides more details.)

This example is based on one of the assignment offered to the students of an under-graduate course on concurrent programming (`http://lara.epfl.ch/w/parcon16:problem_statement_and_handout`). The students were asked to implement the two queue operations `enqueue` and `dequeue`, as well as helper functions such as `isFull`, `isEmpty` or `tail` if needed, using only the accessor and mutator functions. Our real design was slightly more advanced so that students have the impression of working directly with an array of nullable elements.

Observe that the class `Queue` extends a trait `Monitor` which is defined in the library, whose implementation shown in Fig. 5. This binds the `synchronized`, `wait`, and `notify` constructs to the function defined in the trait `Monitor` in-

```
class Queue[T](val size: Int) extends Monitor {
  /** Shared mutable state **/
  private var _buffer = new Array[Option[T]](size)
  private var _head = 0;
  private var _count = 0;
  /** Accessor functions for the shared state **/
  protected def update(index: Int, elem: T) {
      _buffer(index) = Some(elem)
  }
  protected def apply(index: Int): T =
    _buffer(index) match {
      case Some(e) ⇒ e
    }
  protected def delete(index: Int) {
      _buffer(index) = None
  }
  protected def head: Int = _head
  protected def head_=(h: Int): Unit = _head = h
  protected def count: Int = _count
  protected def count_=(c: Int): Unit = _count = c
  /** Externally visible queue operations **/
  def tail: Int = (head + count) % size
  def isFull: Boolean = count == size
  def isEmpty: Boolean = count == 0

  def enqueue(e: T): Unit = synchronized {
    while (isFull()) wait()
    this(tail) = e
    count += 1
    notifyAll()
  }
  def dequeue(): T = synchronized {
    while (isEmpty) wait()
    val elem = this(head)
    this.delete(head)
    head = (head + 1) % size
    count -= 1
    notifyAll()
    elem
  } }
```

Figure 1. Implemenation of a blocking, concurrent queue.

stead of their definitions in the class `Object`, thus making them *overridable*. The trait `Monitor` by default binds the constructs to their standard implementation. However, another trait, namely `SchedulableMonitor` (described shortly), provides a different implementation for the synchronization constructs, and can be mixed in with the class `Queue` to override the standard definition. As described in section 3, we were able to achieve this kind of overridability of synchronization primitives due to the syntactic extensibility offered by Scala through features such as *implicit parameters* and *call-by-name*.

Fig. 2 shows a unit test suite `QueueTestSuite` that tests the implementation of a `Queue`. The test suite uses a sub-class of the `Queue` class, namely `SchedulableQueue`, whose accessor and mutator functions are wrapped inside a

```
class SchedulableQueue[T](val size: Int,schedr: Scheduler)
 extends Queue[T](size) with SchedulableMonitor {
 val scheduler = schedr
 /** Marking interleavable expressions **/
 override def update(index: Int, elem: T): Unit = exec {
     super.update(index, elem)
 }{ /** optional log message **/ }
 override def apply(index: Int): T = exec {super(index)}
 override def delete(index: Int): Unit = exec {
   super.delete(index) }
 override def head = exec{ super.head }
 override def head_=(h: Int) = exec{ super.head_=(h) }
 override def count = exec{ super.count }
 override def count_=(c: Int) = exec{ super.count_=(c)}
 override def tail = exec{ super.tail }
}
class QueueTestSuite extends FunSuite {
  test("Testing FIFO behavior on a queue of size 3") {
    testManyScedules(4, // number of threads
      sched ⇒ {
        val q = new SchedulableQueue[Int](3, sched)
        // operations per thread
        val threadOps = List( // th.1 code
          () ⇒ { q.enqueue(1); q.enqueue(2) }
          () ⇒ q.dequeue(),  // thread 2 code
          () ⇒ q.enqueue(3), // thread 3 code
          () ⇒ q.enqueue(4)) // thread 4 code
        // validate that thread 2 cannot return 2
        val resultValidator =
            results ⇒ (results(1).asInstanceOf[Int] ≠ 2,
                       "Should not obtain 2")
        (threadOps, resultValidator)
    })
  } }
```

Figure 2. Sample units test for the concurrent queue.

```
test("An exhaustive test of FIFO property") {
  testManySchedules(4, sched ⇒ {
   val q = new SchedulableQueue[(Char, Int)](3, sched)
   val (n, chars) = (2, List('a', 'b', 'c'))
   val threadOps = chars.map(c =>
       () ⇒ for (i ← 1 to n) { q.enqueue((c, i)) }
     ) :+ (() ⇒ {
       val counts = HashMap.empty[Char, Int]
       for(c ← chars) counts(c) = 0
       for (i ← 1 to 3 * n)) {
         val (c, n) = q.dequeue()
         counts(c) += 1
         assert(counts(c) == n)
       }
     })
   (threadOps, results ⇒ (true, ""))
  })}
```

Figure 3. An exhaustive unit test for the concurrent queue example shown in Fig. 1

library function `exec`, but otherwise inherits the definitions of the `Queue`. The `exec` construct serves to mark operations that access shared state, and whose interleavings have to be tested. It also accepts an optional message that is appended to a log (specific to a schedule) with the thread id once the operation wrapped inside `exec` completes. We refer to the operations wrapped inside `exec` as *marked operations*. It is desirable, but not binding, that the marked operations are atomic, because our scheduling algorithm allows context switches only at the start of `exec`. (Readers may wonder that it may be simpler to wrap the accessors and mutators of the `Queue` class with `exec`, rather than doing so in a subclass that extends the `Queue`. We resort to this strategy since it restricts the overheads of the scheduling algorithm to the subclass, which is used only in the unit tests, insulating other parts of the applications that may use the `Queue` class.)

Observe that the class `SchedulableQueue` also mixes in the trait `SchedulableMonitor`. This trait defines the `exec` construct, and also overrides the standard implementations of `synchronized`, `wait`, and `notify` constructs. The trait `SchedulableMonitor` implements the necessary coordination for ordering the operations wrapped inside `exec` using the *scheduler* provided as input, which stores a *schedule*. We define a *schedule* as a list of thread ids that specifies the order in which the marked operations of each thread should interleave. For instance, given two threads with ids T_1 and T_2, a schedule: `List(`T_1, T_2, T_1, T_2, $\cdots$`)` indicates that T_1 should execute first until encountering its second marked operation, then should yield control to T_2, which may execute until its second marked operation, and then yields the control back to T_1, and so on. That is, T_1 and T_2 alternatively execute their instructions yielding control to the other thread at the beginning of marked operations.

Similar to marked operations, the entry and exit points of synchronization primitives such as `synchronized`, `wait`, and `notify` are also considered as points of context switch. For instance, if threads T_1 and T_2 race to enter a common `synchronized` block (a critical section) when they begin execution, the thread T_1 should be allowed to enter the block, and T_2 should be blocked as per the above schedule. Incidentally, the presence of synchronization primitives makes certain schedules impossible to follow. For instance, in the current example, if T_1 encounters a second marked operation while being in the synchronized block, it has to yield control to T_2. However, T_2 cannot execute since T_1 hasn't exited the critical section. In our library, we dynamically alter such schedules so that it conforms to the locking protocol, by allowing threads to execute out of turn.

The schedules can be manually constructed by the users of the library, or can be autogenerated using the methods provided in the library that are parametrized by the length of the schedule (as described shortly). Moreover, the thread ids in the schedules need not necessarily match the number of marked operations executed by the threads at run time. (a) If a schedule has more entries for a thread T_1 than its marked operations, then the additional slots of T_1 are ignored by the algorithm. (b) One the other hand, if a schedule has fewer slots for a thread T_1 then T_1 is blocked until the entire schedule has been consumed, and is allowed to complete concurrently with other threads that may also have unscheduled marked operations. In Fig. 6 we describe the implementation of `exec`, and present the algorithm for ordering the marked operations.

Consider now the unit test shown in Fig. 2 at lines 18 to 34. It is meant to check the first-in-first-out (FIFO) behavior of the queue in the presence of concurrent queue operations. The test uses a library function `testManyScedules` which accepts as arguments (a) the number of threads that should be created, (b) a function: `Schedule` $\Rightarrow$ `(List[Unit` $\Rightarrow$`Any], List[Any]`$\Rightarrow$`(Boolean, String))` that takes a schedule and returns a list of operations that should be executed by each thread (`List[Unit`$\Rightarrow$`Any]`) and a validator function of type `List[Any]`$\Rightarrow$`(Boolean, String)`. Given a list of return values of each thread, the validator function checks a property and returns its truth value along with an error message that should be used if the truth value is false. Note that the validator function can also access other variables that are in scope. For instance, it can access the internal state of the queue `q`. In the unit test shown in Fig. 2, we test the queue on four threads that manipulate a single shared queue `q`, as shown by lines 24 to 27. The property that is checked here is that the dequeue operation executed by thread 2 should never return the number two. The function `testManyScedules` randomly samples few thousand schedules from the space of possible schedules with four threads, having a bounded number of context switches and a fixed length. The bound on the number of context switches, and the length are configurable by the user. For instance, for evaluating students' solutions, we fixed the number of schedules to be sampled as 10k, the length of the schedules as 20, and the context switch bound as 10, since the code-snippets that the students had to implement were small. Fig. 3 shows another unit test that tests FIFO propery more exhaustively using assertions and thread local mutable state.

For interested readers, it was shown by J. Hamza and others [4, 9] that to ensure linearizability of a concurrent queue it suffices to verify that certain bad patterns, which are similar to the property tested by the unit tests, do not occur when the operations are executed concurrently. Though such theorems may not exist for other concurrent data structures, it is still possible to check some deep correctness properties using similar unit tests. For example, for a concurrent list, one can check that after executing a sequence of insert operations concurrently with delete operations, the list must contain all elements belonging to inserts but not to deletes.

A schedule is considered to have failed a test iff one of the following scenarios manifests on a schedule: (a) an uncaught exception is thrown by a thread, or (b) the time taken by

the overall execution of the schedule exceeds a user-defined timeout (which is needed to handle non-terminating computations and livelocks), or (c) the threads are caught in a deadlock (which will be detected by our framework), or (d) the validator function returns false. When a schedule fails a test, `testManyScedules` displays the interleaved log of the `exec` and synchronization operations that led to the failure, and flags the unit test as failed.

Detecting and Reporting Bugs. Fig. 4 present a simplified version of a buggy solution provided by a student for the concurrent queue example, during our evaluations. The bug is due to the use of the construct `notify` instead of the construct `notifyAll`, and was detected by one of our unit tests. As per the JVM specification [1], the construct `o.notify` notifies an arbitrarily chosen thread waiting on the receiver object `o`, whereas `notifyAll` notifies every thread waiting on the receiver object `o`. The error trace outputted by the failed test is shown in Fig. 4. The error trace is a log of the events, qualified by the thread ids, generated by the interleaving that led to a failure. The unit test that failed on this buggy solution created five concurrent threads with ids ranging from 1 to 5, each invoking an operation on a shared queue `q` of unit capacity. The threads 1, 2, and 3 concurrently invoked `q.enqueue` on values one, two and three, respectively, and the threads 4 and 5 concurrently invoked `q.dequeue`. The error trace depicts a scenario in which the operations executed by the threads interleave in a way that results in a deadlock, as described below. Initially, all threads try to enter the `synchronized` block (a critical section) as shown by the event logs `i: synchronized → check`. Thread 4 succeeds in entering the `synchronized` block (as shown by `4: synchronized → enter`), and finds that the queue is empty and enters the wait state. In the following steps, thread 1 succeeds in enqueuing an element as shown by the event logs between `1: synchronized → enter`, and `1: synchronized → exit`. Subsequently, threads 2 and 3, which are trying to enqueue an element, enter the critical section and transition to the wait state finding that the queue is full. Later, thread 5 succeeds in dequeuing the element enqueued by thread 1. It also issues a notification before exiting (see event `5:notify`), which is delivered to one thread, namely thread 4. However, thread 4, which is trying execute a dequeue operation, immediately enters the wait state again, since the queue is now empty. Neither of the threads 2, 3, or 4 are able to make progress from this state, resulting in a deadlock. Interestingly, analyzing the history of the source control repository used by the student to submit the solution revealed that the student was able to correctly change `notify` to `notifyAll` in his subsequent attempts. With this overview of the functionality offered by the library, in the subsequent sections, we explain the underlying algorithms and techniques that enable these features.

```
class Queue[T](val size: Int) extends Monitor {
  def enqueue(e: T): Unit = synchronized {
    while (isFull()) wait()
    this(tail) = e
    count += 1
    notify()
  }
  def dequeue(): T = synchronized{
    while (isEmpty()) wait()
    def result = this(head)
    count -= 1
    head = (head + 1) % size
    notify()
    result
  }}
 Error trace reported for a failed test:
Thread 4 crashed on the following schedule:
          5:synchronized check
1:synchronized check
     3:synchronized check
        4:synchronized check
  2:synchronized check
        4:synchronized → enter
        4:Read count → 0
        4:wait
1:synchronized → enter
1:Read count → 0
1:Read head → 0
1:Read count → 0
1:Write buffer(0) = 1
1:Read count → 0
1:Write count = 1
1:notify
1:synchronized → exit
     3:synchronized → enter
     3:Read count → 1
     3:wait
  2:synchronized → enter
  2:Read count → 1
  2:wait
          5:synchronized → enter
          5:Read count → 1
          5:Read count → 1
          5:Write count = 0
          5:Read head → 0
          5:Write head = 0
          5:notify
          5:Read head → 0
          5:Read buffer(0)
          5:synchronized → exit
        4:Read count → 0
        4:wait
        4:throw java.lang.Exception: Deadlock:
        Threads 2, 4, 3 are waiting but all othes
        have ended and cannot notify them.
```

Figure 4. A buggy implementation of the queue, and the error trace reported for a failed test.

```
1  class Dummy
2  trait Monitor {
3    implicit val d: Dummy = new Dummy
4    def wait()(implicit d: Dummy) = waitFun()
5    def synchronized[T](e: ⇒ T) = synchronizedFun(e)
6    def notify()(implicit d: Dummy) = notifyFun()
7    def notifyAll()(implicit d: Dummy) = notifyAllFun()
8    // default implementations
9    def waitFun(): Unit = super.wait()
10   def synchronizedFun[T](toExecute: ⇒T): T =
11     super.synchronized(toExecute)
12   def notifyFun(): Unit = super.notify()
13   def notifyAllFun(): Unit = super.notifyAll()
14 }
```

Figure 5. trait `Monitor` that binds synchronization primitives to methods defined in the trait, which are overridable.

3. Library Implementation

We classify the implementation of the library into parts that deal with (a) enforcing the order specified by a schedule on the marked operations executed by the threads, and (b) generating context-bounded schedules through random sampling. We first focus on part (a), which is the main contribution of this paper, and briefly discuss part (b), which is fairly straightforward. As mentioned in section 2, most of the functionality necessary for ordering the operations of a shared data structure is provided by the trait `Schedulable Monitor`. The trait also overrides the definitions of the synchronization primitives, which is not directly supported by the Scala language, since the synchronization primitives are declared as *final* methods of the class `Object`. Below we explain a way to override their definitions.

Overriding synchronization primitives. We define a trait `Monitor` that has methods named identical to the synchronization primitives, whose definition is shown in Fig. 5. The first four methods of the trait are carefully crafted so that their signatures are different from the signatures of the synchronization primitives defined in class `Object`, but can be used by a client in exactly the same way as the standard synchronization primitives. In particular, the functions `wait` and `notify` uses implicit parameters that are bound to a dummy object, and `synchronized` accepts a call-by-name parameter (the standard definition uses call-by-value).

When these definitions are available in scope, the Scala compiler prefers to bind uses of the primitives to these definitions, rather than the standard ones defined in class `Object`. For example, consider the `Queue` class shown in Fig. 1. Normally, the call to `this.synchronized` (say at line 26) would refer to `Object.synchronized`, but since `Queue` extends `Monitor` the call to `this.synchronized` will now bind to `Monitor.synchronized`. These definitions allows users to the use the standard Scala syntax for writing programs testable using our library, even in the presence of synchronization primitives. Note that, by default, the trait `Monitor` redirects these methods to their standard implementation (see lines 9 - 13). Nonetheless, any subtype of `Monitor` can override the standard definition by overriding the methods defined in lines 9 - 13.

```
1  trait SchedulableMonitor extends Monitor {
2    /** abstract method that returns a scheduler **/
3    def scheduler: Scheduler
4    /** Overriden synchronization primitives **/
5    override def waitFun() = { ... } ...
6    def exec[T](op: => T)(msg: => String): T = { ... }
7  }
8  /** A class used by 'SchedulableMonitor' **/
9  class Scheduler(fullSchedule: List[Int]) {
10   /** Mutable private state **/
11   var remSchedule = fullSchedule // remaining schedule
12   var numThreads = 0 // #threads running concurrently
13   val threadStates = new HashMap[Int, ThreadState]()
14   def waitForTurn = { ... }
15 }
16 /** Classes representing thread states **/
17 abstract class ThreadState // Abbreviated TS below:
18 type L = AnyRef
19 type LS = Seq[AnyRef]
20 case object Starting extends TS
21 case class Running(locks: LS) extends TS
22 case class Sync(lock: L, locks: LS) extends TS
23 case class Waiting(lock: L, locks: LS) extends TS
24 case class SyncUnique(lock: L, locks: LS) extends TS
25 case class VariableRW(locks: LS) extends TS
26 case object Ending extends TS
```

Figure 6. Outline of the trait `SchedulableMonitor`, and the helper classes `Scheduler`, and `ThreadState`.

3.1 Implementation of SchedulableMonitor

Fig. 6 shows the outline of the definition of the trait `SchedulableMonitor`, and a class `Scheduler`. The class `Scheduler` stores the schedule, and implements a few helper functions. For exposition purposes, in figures 7 and 8 we present the definition of the synchronization functions and `waitForTurn` function as high-level pseudocode, abstracting away some of the intricacies and low-level details. Our actual implementation is slightly more sophisticated than the algorithm presented here. In particular, it can dynamically alter a schedule if it is not feasible due to locking dependencies. These features are omitted for clarity. Interested readers are requested to refer to the source repository: `https://github.com/epfl-lara/muscat`. Our scheduling algorithm crucially relies on tracking the state of execution of threads that are created by the unit tests. Below we describe the thread states tracked by our library.

Thread states. The `Scheduler` class maintains an abstract state with each thread created via the `testManySchedules` method of a unit test, as illustrated in Fig. 2. The states are tracked by the variable `threadStates` which is a mutable

```
def exec[T](op: ⇒ T)(msg: ⇒ Option[String]): T = {
 val t = current thread
 change state of t from 'Running' to 'VariableRW'
 scheduler.waitForTurn()
 val res = op
 msg match {
   case Some(m) ⇒ log(m(res))
   case None =>⇒
 }}
def synchronizedFun[T](toExecute: ⇒T): T = {
  val locks = locks held by the current thread
  change current thread state to 'Sync(this, locks)'
  scheduler.waitForTurn()
  val s = toExecute
  change thread state to 'Running(locks)'
  s
}
def waitFun() = {
  val locks = locks held by the current thread
  change current thread state to
    'Waiting(this, locks.filter(_ != this))'
  scheduler.waitForTurn()
}
def notifyFun() = {
  For every thread in state 'Waiting(this, locks)'
    change its state to 'SyncUnique(this, locks)'
}
def notifyAllFun() = {
  For every thread in state 'Waiting(this, locks)' or
    'SyncUnique(this, locks)'
        change its state to 'Sync(this, locks)'
}
```

Figure 7. Pseudocode of the trait `SchedulableMonitor`.

hash map from thread ids to an abstract class `ThreadState`. Each abstract state is represented by a case class that extends `ThreadState` as shown in Fig. 6, and is designed based on the JVM documention [1]. Initially, all threads are initialized to the state `Starting`, and transition to other states as they execute a shared operation or a synchronization primitive. Many states store a *sequence* of locks, which are objects on which the thread would have acquired a lock when executed on the JVM. For instance, when a thread enters a block `o.synchronized`, it takes a lock on the object `o`. Hence, `o` is added to the list of locks stored in the thread's state until the thread exits the synchronized block. The locks are stored as a sequence (instead of a set) to correctly model *re-entrant* calls to `synchronized` i.e, calls of `o.synchronized` within an `o.synchronized` block. Below we describe each state:

- A thread that is started would be in the state `Starting`. It will transition to `Running(∅)` once all threads start.
- A thread that is executing its instructions would be in the state `Running(locks)`, where `locks` is a list of object references on which the thread holds a lock.

```
def waitForTurn() = {
 wait until every thread other than the executing thread
     is blocked on 'waitForTurn'
//last thread calling 'waitForTurn' executes this code
  val remSched = the remaining schedule 'remSchedule'
  val t = remSched.head
  val state = state of the thread t
  state match {
    case Sync(l, P)) ⇒
      if (l ∈ P or there exists no other thread is
        in a state with locks Q such that l ∈ Q) {
          remSched = remSched.tail
          change state of 't' to 'Running(P :+ l)'
          notify thread 't', and block other threads
      }
      else abort the execution
    case VariableRW(locks) ⇒
      remSched = remSched.tail
      change state of 't' to 'Running(locks)'
      notify thread 't', and block other threads
    case SyncUnique(l, P) ⇒
      change the state of every thread other than 't'
        in state 'SyncUnique(l, Q)' to 'Waiting(l, Q)'
      change state of 't' to 'Sync(l, P)'
      waitForTurn()
    case Waiting(_, _) ⇒
      if no other thread can make progress
        report a deadlock
      else
        discard the schedule as it cannot be followed
} }
```

Figure 8. Pseudocode of the function `waitForTurn`.

- The state `VariableRW(lock, locks)` denotes that the thread is about execute a marked operation wrapped inside the `exec` construct.
- The state `Sync(lock, locks)` denotes that the thread is *attempting* to enter a `lock.synchronize` block.
- The state `Waiting(lock, locks)` denotes that the thread is waiting due to an invocation of `lock.wait()`, while holding locks on the list of objects given by `locks`.
- The state `SyncUnique(lock, locks)` is an transition state that results after a call to `lock.notify` by a thread. In this state, only one thread waiting on `lock.wait` will acquire the lock and enter the critical section, and others fall back to the waiting state.
- A thread that is about to terminate will enter the state `Ending`. This state is necessary to initiate certain clean up operations, and invoke the `waitForTurn` function to schedule the next thread to execute.

Implementation of synchronization primitives. *(A) Implementation of Exec.* As shown in Fig. 7, the `exec` function first looks up the id of the executing thread, and changes its state from `Running(locks)` to `VariableRW(locks)`

by updating the mutable hash map `threadStates` of the helper class `Scheduler` shown in Fig. 6. (Since this is an update to a shared state, in the implementation it is actually enclosed within a synchronized block.) The function then invokes `waitForTurn` which blocks the thread until it is the turn of the thread to execute as per the schedule stored in the `scheduler`. Once the thread is unblocked, it executes the marked operation `op`, and records a log message. It is guaranteed by the `waitForTurn` function that `op` is executed in isolation without interference from other threads.

(B) Implementation of synchronizedFun. The function `synchronizedFun` first computes the set of locks that are held by the thread under execution. Recall that this information is available in the state of the thread, which should be `Running(locks)`. The function changes the state of the thread to `Sync(this, locks)` indicating that the thread wishes to take a lock on `this` object, and then invokes `waitForTurn` function. When the thread is granted access, it can acquire the lock and execute the critical section.

(C) Implementation of waitFun. This function changes the state of the executing thread to the `Waiting` state and releases *all* the locks on the `this` object. The thread executing this function will be blocked until it is notified by another thread, at which point it will transition to `SyncUnique` or `Sync` state.

(D) Implementation of notifyFun. The `notifyFun` changes state of every thread in state `Waiting(this,locks)`, where `locks` is unconstrained, to the state `SyncUnique(this, locks)`. Among the threads in state `SyncUnique(this, locks)` exactly one of them would be chosen to execute by `waitForTurn` based on the schedule. The state of the remaining threads will be reset to `Waiting`.

(E) Implementation of notifyAllFun. The `notifyAllFun` function changes the state of every thread in state `Waiting( this, locks)` to `Sync(this,locks)`. Unlike the previous case, all threads in state `Sync(this,locks)` would be ready to execute, and would do so eventually, unless the execution is aborted due to a faulty schedule. However, the threads will be permitted to execute by `waitForTurn` only in the order specified by the schedule.

WaitForTurn Algorithm. Fig. 8 shows the pseudocode of this function. Initially, the function blocks until all threads have invoked the `waitForTurn` function, and are waiting for the function's response. The thread that enters the function last will execute the remaining code of the `waitForTurn` function. This step is much like a *barrier*, and is implemented using the mutable fields of the `Scheduler` class (like `numThreads` shown in Fig. 6) that keeps track of the number of active threads that are executing.

In the following steps, the function retrieves the identifier `t` of the thread that should be allowed to execute next using `remSchedule`. It then chooses an appropriate action to perform based on the state of the thread. For instance, if the thread `t` is in the state `Sync(l, P)`, then it is allowed to take a lock on `l` iff no other thread is holding the lock. Once the next thread is chosen, the function advances the schedule one step (line 12), and changes the state of the chosen thread to `Running`, augmenting the locks field `P` with `l` (line 13). However, to correctly model re-entrant calls to `synchronized` [1], while augmenting `P` with `l` at line 13 we ensure that if `l` was released by the thread due to a call to `l.wait()`, the same number of `l`'s that was held by the thread before calling `l.wait()` is added back to `P`. (For brevity we omit the additional components of the state required to track this information.) Finally, the thread that is chosen is notified so that it can execute, and every other thread remains blocked. However, since a direct notification from one thread to another is not permitted by the JVM, in the actual implementation, the protocol for notifying threads works through mutable fields and objects shared among the threads. (We use the `Scheduler` itself for this purpose, as it is shared among all threads under execution.)

The remaining cases of the `waitForTurn` function are similar. In the case of `SyncUnique`, the thread `t` that is scheduled changes state from `SyncUnique` to `Sync`, while the remaining threads transition to the `Waiting` state. This precisely captures the behavior of `notify` function. In the `Waiting` case, our algorithm detects a deadlock by checking whether there is any thread, other than the one scheduled, that can make progress. This is ascertained by looking for cycles among the locks stored in the threads that are in the `Sync` state, and checking whether all threads are in `Waiting` state. If there is no deadlock, we discard the schedule as it doesn't conform to the locking protocol. However, in our implementation, in this case the schedule is dynamically altered by allowing a thread that can execute (but not at the head of the schedule) to execute out of turn. This helps in utilizing all schedules automatically generated by our tool.

Fig. 9 pictorially depicts the state transitions with respect to a thread, described in this section, in a state transducer like syntax. The arrows represent transitions that happen between the states as the thread executes its instructions. The label A on top of an arrow, if any, implies that the transition is triggered by a call/return event or a condition A, and a label B below the arrow implies that the transition has an effect B on the fields of the target state. For example, the labels on the arrow from `Running` to `Sync` imply that the transition happens on seeing a call `t.synchronized`, and sets the `lock` field of the state `Sync` to `t`. A transition happening on a thread can also trigger transitions in other threads. Such triggers are denoted using $\bullet\!\!-$, and the transitions they induce (in other threads) using dotted arrows . For example, when a thread transitions from `SyncUnique` to `Running`, it triggers a transition in other threads holding the same lock from state `SyncUnique` to `Waiting`. The transition $\diamond\!\!\rightarrow$ depicts the state changes that happen when the scheduling algorithm (`waitForTurn`) signals the thread to execute. We represent changes to `locks` using the following notation.

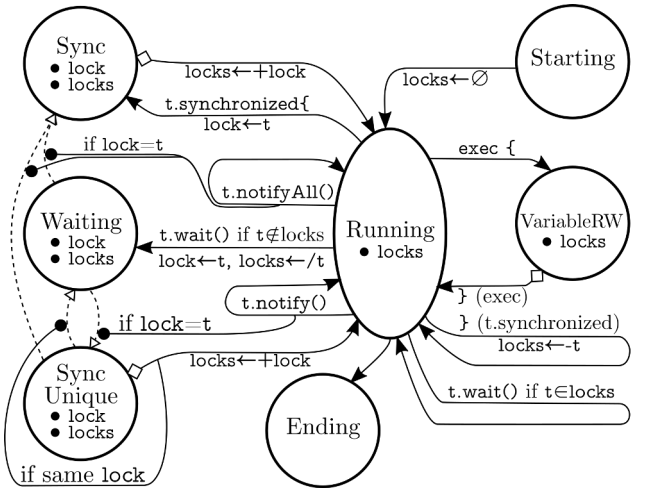

Figure 9. State transitions of a thread.

`locks←/lock` means that all occurrences of `lock` are removed from `locks`. `locks←-lock` means that `lock` was the last element of `locks` and is removed. `locks←+lock` means that `locks` is augmented with `lock`. As mentioned earlier, this operation may add multiple copies of `lock` if there were re-entrant calls to `synchronized`.

3.2 Schedule Exploration

Our library supports deterministic as well as probabilistic exploration of interleavings of marked operations executed concurrently by the threads provided in the unit test. As mentioned in section 2, we provide a function `testManySchedules` that samples context-bounded schedules of a specified length uniformly at random. For instance, given five threads with ids 1 to 5. A schedule with a context bound of 10 has at most 10 points where the adjacent thread ids are not identical. We resort to this strategy as it is widely accepted that most concurrency bugs can be detected with fewer context switches. However, our system can also perform more exhaustive enumeration of schedules.

4. Evaluation

We used our tool to test two assignments offered to students. The first assignment was a producer-consumer problem with a solution such as the one given in Fig. 1. In the second assignment, the students were required to implement a lock-free linked list of sorted numbers that supports an `insert` operation for inserting an element in the list in the sorted order, a `find` operation, and a `delete` operation that deletes one occurrence of a given element. The content of a list node comprises a value, a boolean flag (that marks nodes that are deleted), and a pointer to the next node of the list. Students were tasked with implementing the three list operations using only an atomic `CompareAndSet` and `get` operation, implemented similar to the `AtomicVar` shown below. (Note that `AtomicVar.CompareAndSet` operations executed concurrently will be ordered as per a given schedule, which enables us to test multiple interleavings of the operations executed by the `insert` and `delete` functions.)

```
class AtomicVar(init: Int) {
 private val a = new java.util.concurrent.AtomicInteger(init)
 def get = exec { a.get }(r ⇒ s"Read: $r")
 def compareAndSet(x: Int, y: Int) = exec{
   a.compareAndSet(x, y)
 }(r ⇒ s"CAS $x to $y, success: $r")}
```

The detailed descriptions of the assignments are available as Assignment 5 and 6 in the webpage `lara.epfl.ch/w/parcon16:top`.

Students of our course were required to develop their solutions using the *git* source control system. To evaluate the usefulness of our tool, we analyzed the commits made by the students to determine (a) the errors encountered by them while solving the assignments and (b) weather the feedbacks reported by the tool enabled students to fix the bugs in the subsequent commits. We now discuss the results of our evaluation. The first assignment had a total of 138 submissions, and the second assignment 121 submissions. We tested the first assignment using 12 unit tests, which were similar to the ones shown in Figures 2 and 3. Despite a few minor errors in our initial implementation, which we corrected a posteriori, we found that our tool provided some feedback for 24 students. In total, it outputted 46 traces of failed tests of which 9 were deadlocks and the other 37 were due to wrong outputs generated by the students code. The reasons for the deadlocks were mostly one of following: (a) breaking down the use of `synchronized` blocks into smaller blocks, (b) using `notify` instead of `notifyAll`, and (c) not using a while loop around `wait`.

We evaluated the second assignment on 14 unit tests. Our tool detected an error trace at least once in solutions submitted by 76 students. In total, it outputted 663 error traces. Although hard to read at first sight, we used such traces to walk through their lock-free code during lab sessions, and in a few minutes help them find where the bug was. For this assignment, it appears that most of the bugs came from the expectation that two immediate reads of a shared field of a list node would result in the same value. In both assignments, we found that students also added their own tests using our library by modifying the tests given to them.

5. Related Work

Model checkers for concurrent programs. Much of the research in testing concurrent programs has been directed towards bounded and symbolic model checking approaches. Due to the plethora of tools available in this space it is difficult to exhaustively mention them. Therefore, we restrict the discussion to a few interesting tools in this area. CHESS [18] is a model checker for .NET programs that allows a systematic exploration of concurrent executions. COLT [23] is a tool for compositionally testing linearizability of Java programs by exploring multiple schedules. Both these ap-

proaches do not support user-defined unit tests, and rely on assertions in the program to unravel bugs. COLT may also raise false alarms. Java PathFinder [24] presents an approach for model checking concurrent Java code using a specialized JVM. Goldilocks [6] is a tool for run-time data race detection in Java. In comparison to this work, our library can be used to detect any kind of synchronization problems, and is applicable even to lock-free algorithms where data-races can be tolerated. CSEQ [8] is a model checker for concurrent C programs that reduces a concurrent program to a sequential one and applies model checkers for sequential programs such as CBMC [15, 20]. Zing [2] is a bounded model checker for concurrent programs that supports inferring bounds on the state-space exploration. Emmi et al. [7] propose an extended depth-first search strategy for exploring interleavings of asynchronous programs in the presence of a blocking `await` operation, especially for programs that create a tree of asynchronous tasks. In comparison, our tool targets concurrent data structures accessed by multiple threads. Moreover, our scheduling algorithm handles all the synchronization primitives offered by the JVM. Spin [10, 11] is a model checker that verifies programs written in Promela against linear temporal logic specifications. Verifast [12] uses specifications to prove properties of concurrent C and Java programs.

Concurrent unit testing frameworks. Although scarce, unit testing frameworks for concurrent programs have been explored by prior works. Parallel-junit allows running unit tests in parallel, which may expose certain concurrency bugs. CONCURRIT [5] and [13] offer support for expressing concurrent schedules, but requires the programmer to insert appropriate synchronization constructs in body of the threads. In contrast, we explore thousands of schedules automatically, even without the knowledge of the code executed by the threads.

6. Conclusion and Future Work

We presented a lightweight unit testing library that is suitable for testing Scala programs written by students learning concurrent programming. Our tool allows testing multiple thread interleavings by generating and enforcing schedules, and produces an interleaved error trace when a test fails. Nonetheless, there are certain limitations to our tool which we would like to address in the future. The tool does not explore reorderings possible due to relaxed (or weak) memory models. We believe our tool can be extended to handle this by modeling the weak writes to memory as asynchronous writes to a shared, concurrent buffer. We would also like to the add the ability to replay the error traces.

Acknowledgments

We thank Jad Hamza for providing valuable suggestions and ideas for this work. This work is partly supported by the Swiss National Science Foundation grant: *Constraint Solving Infrastructure for Program Analysis*.

References

[1] Java documentation. URL `docs.oracle.com/javase/8`.

[2] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: Exploiting program structure for model checking concurrent software. In *CONCUR*, 2004.

[3] K. Beck. Embracing change with extreme programming. '99.

[4] A. Bouajjani, M. Emmi, C. Enea, and J. Hamza. On reducing linearizability to state reachability. In *ICALP*, 2015.

[5] J. Burnim, T. Elmas, G. Necula, and K. Sen. Concurrit: testing concurrent programs with programmable state-space exploration. In *USENIX Workshop*, 2012.

[6] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A race-aware java runtime. *Commun. ACM*, Nov. 2010.

[7] M. Emmi, B. K. Ozkan, and S. Tasiran. Exploiting synchronization in the analysis of shared-memory asynchronous programs. In *SPIN*, 2014.

[8] B. Fischer, O. Inverso, and G. Parlato. Cseq: a sequentialization tool for C. In *TACAS*, 2013.

[9] J. Hamza. *Algorithmic Verification of Concurrent and Distributed Data Structures*. PhD thesis, 2015.

[10] G. J. Holzmann. The model checker spin. *TSE*, 1997.

[11] G. J. Holzmann. *The SPIN model checker: Primer and reference manual*. Addison-Wesley Reading, 2004.

[12] B. Jacobs, J. Smans, and F. Piessens. A quick tour of the verifast program verifier. In *APLAS*, 2008.

[13] V. Jagannath, M. Gligoric, D. Jin, Q. Luo, G. Rosu, and D. Marinov. Improved multithreaded unit testing. In *FSE'11*.

[14] M. Kebrt and O. Šerỳ. Unitcheck: Unit testing and model checking combined. In *ATVA*, 2009.

[15] D. Kroening and M. Tautschnig. CBMC – C bounded model checker. In *TACAS*, 2014.

[16] A. Lal, T. Touili, N. Kidd, and T. Reps. Interprocedural analysis of concurrent programs under a context bound. In *TACAS*, 2008.

[17] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI '07*.

[18] M. Musuvathi and S. Qadeer. CHESS: systematic stress testing of concurrent software. In *LOPSTR*, 2006.

[19] S. Qadeer. Poirot – a concurrency sleuth. In *ICFEM*, 2011.

[20] I. Rabinovitz and O. Grumberg. Bounded model checking of concurrent programs. In *CAV*, 2005.

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

[22] K. Sen and G. Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *CAV*, 2006.

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

[24] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. In *ASE*, 2003.