Patterns in Testing Concurrent Programs with Non-deterministic Behaviors*

Samira Tasharofi
University of Illinois at Urbana-Champaign
Urbana, IL, 61801, USA
tasharo1@illinois.edu

Ralph Johnson
University of Illinois at Urbana-Champaign
Urbana, IL, 61801, USA
rjohnson@illinois.edu

ABSTRACT
Concurrent programs are hard to test because of the non-determinism inherit in them. Since non-determinism is one of the major sources of bugs, it is important to to be tested. We studied the test suites of four open source concurrent libraries and discovered that the tests fall into three different patterns. One of the patterns avoids testing non-determinism while the other two control the schedule of the program execution to manage that.

Keywords
concurrent programs, testing patterns, testing non-determinism

1. INTRODUCTION
Testing is important for concurrent programs and also a hard problem due to non-determinism, race conditions, deadlock, and live-lock. Non-determinism in these programs means that it is possible with the same input, they can lead to different outputs. This happens because the schedule of the events during the execution might be different. In message-passing style, this schedule is the order of the messages sent to each component. However, in shared-memory style, this schedule is the order of the read/write accesses on the shared variables. The test cases written for these programs should not only cover different inputs but also cover different schedule of the program for a given input.

In order to understand how the programmers write tests for concurrent programs, we inspected the test suite repositories of four open source libraries including:

- Java concurrent package library: this package includes features/contrsucts for concurrent programming in Java. The test repository [6] contains tests for testing different features in this package.
- Groovy actor library: Groovy [4] is a Java-based language that has a library for message-passing programming in the style of actors [13]. The code repository [5] includes a great amount of tests for testing actors.
- Akka actor library: Akka [2] is a framework written in Scala [9] and gives developers a simpler programming model to develop highly reliable applications for concurrent or parallel operation. Using the popular Actor model as a basis, Akka extends the concept to provide high availability and fault tolerance. The test suite we inspected is written for actors and available in [3].
- Smile: Smile [12] is a memcache library for scala which is intended to be fast and highly concurrent.

The tests are written using the most popular testing frameworks in those languages, e.g. JUnit [7] and ScalaTest [10]. We also contacted some developers (Daniel Spiewak) of concurrent closed source applications such as Novell Vibe [8] to know about the way that they test their programs. Novell Vibe is an actor-based social collaboration platform for the enterprise that's available both in the cloud and on premise.

The tests that we studied can be categorized into three patterns: *Deterministic black-box*, *Delayed Execution*, and *Wait-notify Coordination*. In the first pattern, programmers try to get rid of non-determinism by writing tests in which the results are the same regardless of the schedule. In the second pattern, testers control the schedule of the tests by inserting delays in the test. In the third pattern, testers control the schedule by having parts of the program generate events and other parts wait for them.

What can be concluded from these solutions is that none of them is suitable for testing concurrent programs. The first pattern avoids testing non-determinism. The second one makes the tests unreliable or very slow; and the third one does not handle all the situations that must be tested and can make the tests complex. This shows the deficiencies of

*Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission. A preliminary version of this paper was presented in a writersŠ workshop at the 3rd Annual Conference on Parallel Programming Patterns (ParaPLoP). ParaPLo Copyright 2011 is held by the author(s). ACM 978-1-4503-0127-5.

current testing frameworks which are mostly developed for testing sequential programs. It turns out that we need a framework for testing concurrent programs that can provide better ways of testing non-determinisim.

2. PATTERNS

2.1 Deterministic Black-box

2.1.1 *Problem*

How to test a concurrent program while keeping the tests simple and using the same approach as testing sequential programs.

2.1.2 *Context*

In concurrent programs it is possible that with some given input, the results are different in different runs. Therefore, the tests might pass in one run and fail in the other run. This non-determinism is due to the order of the events in the execution (schedule of the execution). However, since it is difficult to control the schedule, some programmers want to write tests without dealing with different possible order of events in the test execution. They want to keep the tests simple and similar to the tests written for sequential programs.

2.1.3 *Solution*

The solution consists of two parts: (1) choose inputs for the program under test whose final result does not depend on the schedule; (2) run the program, wait for the results and check if they are correct.

In shared-memory multi-threaded programs, the requests (and input) to the program under test are usually given by method calls. Therefore, the returned results can be checked at the same place as the methods are called. However, in message passing programs, after sending a request (message) to the program under test, the test should wait to receive the result (which is in the form of message) from the program and then check the result. This solution is based on black-box testing. The reason is that white-box testing needs to deal with the order of the events. In fact, if the programmers want to check some internal properties of the components in the middle of the test execution, they need to be aware of the events happened by that time.

Example 1.1. As an example, consider the test in Listing 1 written in JUnit [7]. It is written for `PriorityQueue` class in Java which is an unbounded priority queue based on a priority heap. The goal of this test is to check the correctness of the `isEmpty` function by putting and removing elements from the queue. It sequentially adds and removes elements and calls `isEmpty` function without considering any concurrency.

Example 1.2. As the second example, consider the test in Listing 2 which is written for testing `FSM` actor in Akka using ScalaTest [10]. This actor simulates a finite state machine by defining its states and the transitions among the states. It also accepts a set of subscribers actors and informs them about the current state and the transitions that happened by sending `CurrentState` and `Transition` messages respectively.

Listing 1 Test for PriorityQueue class in Java using Deterministic Black-box pattern

```
1  /**
2   * isEmpty is true before add, false after
3   */
4  public void testEmpty() {
5    PriorityQueue q = new PriorityQueue(2);
6    assertTrue(q.isEmpty());
7    q.add(new Integer(1));
8    assertFalse(q.isEmpty());
9    q.add(new Integer(2));
10   q.remove();
11   q.remove();
12   assertTrue(q.isEmpty());
13 }
```

The aim of the test in Listing 2 is to test a simple basic functionality of a `FSM` actor. It creates a `FSM` actor that is initially in state 0 and by receiving a `"tick"` message it goes to state 1 and by receiving another `"tick"` message it comes back to state 0 again. The test first subscribes itself in `fsm` and waits to receive a message from that actor about its current state by calling `expectMsg`. Subsequently, it sends two `"tick"` messages and waits to receive appropriate messages regarding the transitions happened in `fsm`.

2.1.4 *Forces*

The advantage of this solution is that it keeps test cases very simple and understandable. However, it is a very weak way of testing concurrent programs that cannot cover many aspects of the program behavior. In fact, by this approach, the programmers just test the deterministic parts of the program behavior and leave it to the users to detect the bugs related to non-deterministic parts of the behavior.

The programmers decide about the inputs (whose final results do not depend on the schedule of the test execution) by using the program specification; but it might not be true if there is any bug in the implementation. Therefore, they might need to run each test multiple times with the hope that if there is any bug, it will be revealed.

2.1.5 *Known Uses*

According to the discussions that we had with some companies that use actor-based languages for their products, e.g. Novell Vibe, this approach is the only solution that they use for testing their products. They also mentioned that they just test one component (actor) at a time to keep the tests simple and manageable. More than 80% of the test cases written for Smile and more than 50% of the tests written for Java concurrent package library are based on this solution. However, we found a few of them in Akka and Groovy repositories.

2.2 Delayed Execution

2.2.1 *Problem*

How to write a test for concurrent programs in which an event e should happen after some component is blocked or terminated.

2.2.2 *Context*

Listing 2 Test for FSM actor in Akka using Deterministic Black-box pattern

```
1  object FSMTransitionSpec {
2  //...
3    class MyFSM(target : ActorRef) extends Actor
         with FSM[Int, Unit] {
4      startWith(0, Unit)
5      when(0) {
6        case Ev("tick") => goto(1)
7      }
8      when(1) {
9        case Ev("tick") => goto(0)
10     }
11     // ....
12   }
13 //...
14 }
15
16 class FSMTransitionSpec extends WordSpec with
       MustMatchers with TestKit {
17   import FSMTransitionSpec._
18
19   "A FSM transition notifier" must {
20     "notify listeners" in {
21       val fsm = Actor.actorOf(new MyFSM(testActor)
           ).start()
22       within(1 second) {
23         fsm ! SubscribeTransitionCallBack(
             testActor)
24         expectMsg(CurrentState(fsm, 0))
25         fsm ! "tick"
26         expectMsg(Transition(fsm, 0, 1))
27         fsm ! "tick"
28         expectMsg(Transition(fsm, 1, 0))
29       }
30     }
31 }
```

Concurrent components that are based on synchronous communications are more likely to get blocked during the execution. For example, in shared-memory style of programming most of the communications are based on synchronous method calls and the threads are usually blocked on their requests until they receive their responses. The same issue occurs in message-passing programs with synchronous send/receive of the messages. In general, it is hard to enforce a schedule in which some events (such as checking assertions in the test) should happen after some component is blocked or terminated. The reason is that the blocked or terminated components are not able to communicate with other components to comply with some specific schedule.

2.2.3 *Solution*

Ensure that event e happens after the desired component is blocked or terminated by delaying e. Therefore, the concurrent components in the program under test will not communicate with each other to preserve the order; but the desired order is enforced by implicit coordination. To implement this pattern, put delay commands in the test and/or program under test at appropriate places such that the desired schedule is met. Depending on the language different constructs might be provided for this purpose. For example, in Java (and other Java-based languages such as Scala [9], ActorFoundry [1], Groovy [4], etc.) `Thread.sleep` can be used which accepts a time interval and delays the execution by that time interval.

Listing 3 Test for ArrayBlockingQueue class in Java using Delayed Execution pattern

```
1  public void testBlockingTake() throws
       InterruptedException {
2      final ArrayBlockingQueue q = populatedQueue(
         SIZE);
3      Thread t = new Thread(new CheckedRunnable() {
4          public void realRun() throws
               InterruptedException {
5              for (int i = 0; i < SIZE; ++i) {
6                  assertEquals(i, q.take());
7              }
8              try {
9                  q.take();
10                 shouldThrow();
11             } catch (InterruptedException success)
                   {}
12         }});
13
14     t.start();
15     Thread.sleep(SHORT_DELAY_MS);
16     t.interrupt();
17     t.join();
18 }
```

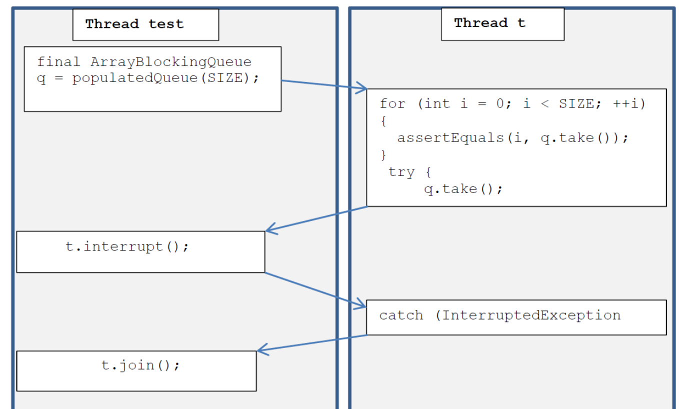

Figure 1: The desired schedule of the test in Listing 3

Example 2.1. As an example consider the test in Listing 3 written in JUnit for testing `ArrayBlockingQueue` class in Java concurrent package. `ArrayBlockingQueue` is a bounded blocking queue backed by an array. Attempts to put an element to a full queue will result in the `put` operation blocking; attempts to retrieve an element (by `take` operation) from an empty queue will similarly block. In this test, the goal is to check when the queue gets empty, the `take` operation would block the calling thread. For that, the test creates a blocking queue and populates the queue with some default values. Then it creates a thread `t` that takes all the elements out of the queue and then calls one additional `take`. The test starts the thread, waits for the thread to reach to line 9 (using `Thread.sleep`) and then interrupts the thread. In Java, if you interrupt a thread which is blocked it will throw `InterruptedException`. In this code this exception in catched successfully. If the thread is not blocked, it will reach to line 10 which throws an exception that is not catched. The schedule that should be satisfied in this test is graphically represented in Figure 1.

Listing 4 Test for Actor class in AKKA using Delayed Execution pattern

```
1  // ...
2  class CrashingTemporaryActor extends Actor {
3    self.lifeCycle = Temporary
4     def receive = {
5       case "Die" =>
6         throw new Exception("Expected exception")
7    }
8
9    @Test
10   def shouldShutdownCrashedTemporaryActor = {
11     val actor = actorOf[CrashingTemporaryActor].
          start
12     assert(actor.isRunning)
13     actor ! "Die"
14     Thread.sleep(100)
15     assert(actor.isShutdown)
16   }
17   //...
```

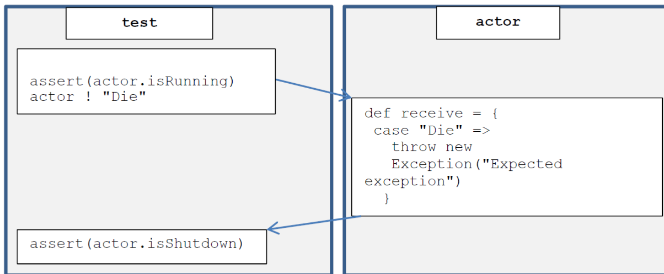

Figure 2: The desired schedule of the test in Listing 4

Example 2.2. As another example, consider the test case in Listing 4 written in JUnit for testing Akka actors . According to the specification of Akka `Actor` class, if their life cycle is defined as *Temporary*, they should terminate if some exception happens in their execution. The goal of the in Listing 4 is to check this behavior. In the test class, a class of `CrashingTemporaryActor` that extends `Actor` is defined and its life cycle property is set to `Temporary`. This actor can receive a `"Die"` message and upon receiving that, it throws an exception. The test method creates an actor from the type `CrashingTemporaryActor`, sends a `"Die"` message to that actor and waits (by `Thread.sleep`) for the actor to receive and process the `"Die"` message. Then, it checks if the state of the actor is `isShutdown` (the actor is terminated). The desired schedule of this test execution in shown in Figure 2.

As can be seen, in both examples 2.1 and 2.2, the time interval should be selected cautiously; if the time interval for delay is not large enough the assertions might be checked while the thread/actor has not executed the desired operations and therefore the test will fail which is a false negative.

2.2.4 Forces

The advantage of this solution is its simplicity; it does need special features supported by the language or special technology (delays can be simulated by loops in any language). However, since it depends on real time intervals it has its own drawbacks; specifically speaking, two extreme situations may happen:

- **Small delay interval**: in this case, running test cases in different situations and different platforms may not satisfy the the desired schedule. It can lead to both false positives (the test cases can pass while they won't pass if they run with the desired schedule) and false negatives (test cases may fail because the test did not execute with the supposed schedule).
- **Large delay interval**: using large time intervals can alleviate the problem of false positive and false negative results of running tests; but at the cost of slowing down the test executions.

In the next section we describe about the other pattern that can alleviate these problems.

2.2.5 Known Uses

This solution is widely used for testing Java concurrent library. We found many of them based on this solution. However, among the test cases written for actor libraries such as Akka and Groovy, we found a few of them based on this solution. The reason might be related to the nature of the message-passing programming which is based on asynchrony and non-blocking. The Smile repository also contains a few tests based on this pattern. But this few amount is due to the fact that most of the test cases written for Smile does not test the concurrency behavior of the program.

2.3 Wait-notify Coordination

2.3.1 Problem

How to write a test for concurrent program in which one event, $e2$ in component $c2$ should happen after another event $e1$ in component $c1$ and event $e1$ does not include blocking or termination of component $c1$.

2.3.2 Context

As mentioned before, due to non-determinism in concurrent programs, with a given input the result might be different depending on the schedule of the test execution. If the events that must be ordered do not include blocking or termination of the concurrent components, the components can communicate with each other to preserve that order.

2.3.3 Solution

Ensure that event $e2$ happens after another event $e1$ by having event $c1$ sent a notification after $e1$ happens and having $c2$ wait for that notification to trigger $e2$. This idea inherits from observer pattern [15] in which some entities can wait (register) for some event and whenever the event happens they will be notified. The advantage of this approach is that it does not depend on real time values that makes the tests more reliable. Depending on the language, this approach can be implemented in different ways. In Java (and some other Java-based languages) there are synchronization constructs that are used for this purpose including `CyclicBarrier` and `CountDownLatch`. `CyclicBarrier` is a synchronization construct that allows a set of threads to all wait for each other to reach a common barrier point. `CyclicBarriers` are useful in programs involving a fixed sized party of

Listing 5 Test for Actor class in Akka using Wait-notify Coordination pattern

```
1  "An Actor" should {
2  "be able to hotswap its behavior with become(..)"
      in {
3     val barrier = new CyclicBarrier(2)
4     @volatile var _log = ""
5     val actor = actorOf(new Actor {
6       def receive = {
7         case "init" =>
8           _log += "init"
9           barrier.await
10        case "swap" => become({
11          case "swapped" =>
12            _log += "swapped"
13            barrier.await
14        })
15      }
16    }).start
17
18    actor ! "init"
19    barrier.await
20    _log must be ("init")
21
22    barrier.reset
23    _log = ""
24    actor ! "swap"
25    actor ! "swapped"
26    barrier.await
27    _log must be ("swapped")
28  }
29 }
```

threads that must occasionally wait for each other. The barrier is called cyclic because it can be re-used after the waiting threads are released. `CountDownLatch` is a synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes. A `CountDownLatch` is initialized with a given count. The `await` methods block until the current count reaches zero due to invocations of the `countDown` method, after which all waiting threads are released and any subsequent invocations of await return immediately. The count in the latch cannot be reset.

Example 3.1. As an example consider the test in Listing 5 which is written for testing Akka actor library using ScalaTest as the testing framework. In this library, each actor should be able to change its behavior by calling `become` method. The test creates a barrier with size two and an actor that can receive two messages namely, `"init"` and `"swap"`. The actor after receiving `"swap"` message, it calls `"become"` and then becomes an actor that can only receive `"swapped"` message. The test sends a message `"init"` to `actor` and waits on the barrier which means that it waits for `actor` to receive `"init"` message (Line 9. Then it checks the `_log` value to see if it is set to `"init"` (which means that the actor is able to receive the proper messages from the test). Then, the test rests the barrier and `_log`, sends `"swap"` and `"swapped"` messages and waits for the actor to process `"swap"` message, calls `become` and becomes an actor that can accept `"swapped"` message and finally process `"swapped"` message. At this point the test will exit from waiting on the barrier and checks the `_log` value. The schedule of this test is shown in Figure 3.

Note that in this test, if there is any bug in `become` method,

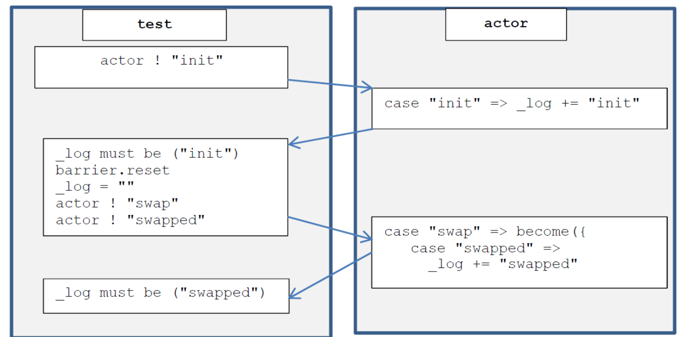

Figure 3: The desired schedule of the test in Listing 5

the test might wait for the barrier forever (deadlock). To solve this problems, latches and barriers can accept a timeout value for `await` method. In that case, if the thread is not released from waiting after the timeout, it will return "false" that can throw an exception by using `assertTrue` on `await` method.

Example 3.2. The test in Listing 6 shows another example. This test is written for testing `ScheduledThreadPoolExecutor` class in Java concurrent package. `ScheduledThreadPoolExecutor` is a `ThreadPoolExecutor` that can additionally schedule commands to run after a given delay, or to execute periodically. The Delayed tasks execute no sooner than they are enabled, but without any real-time guarantees about when, after they are enabled, they will commence. Tasks scheduled for exactly the same execution time are enabled in first-in-first-out (FIFO) order of submission.

In This test the goal is to check if a `ScheduledThreadPoolExecutor` successfully executes a `runnable`. For that, it creates a executor will a pool size of one, a latch with count one, and a `Runnable` task. Then it asks executor to execute the task. The latch will be count down when the task is executed (method `run` is called). The test will wait for the latch to reach zero. In this test, `await` is used with a timeout value which causes `assertTrue` throws assertion error if the latch does not reach zero after the time out.

2.3.4 Forces

Wait-notify Coordination is a more reliable alternative to delay style. Although it can alleviate the problems associated with *Delayed Execution*, it has its own limitations as follows:

- In order to be able to use that, the language should support the required synchronization constructs. Therefore, it is not such simple that can be used in every language.item If it is not used with cautions, deadlock can occur in test execution, e.g. some threads might wait for a barrier or latch and they won't be notified.
- It is not straight forward to be used in the cases that the events subjected to be in a specific order include blocking/termination of components. The reason is

Listing 6 Test for ScheduledThreadPoolExecutor class in Java using Wait-notify Coordination pattern

```
1  public void testExecute() throws
       InterruptedException {
2    ScheduledThreadPoolExecutor executor = new
         cheduledThreadPoolExecutor(1);
3    final CountDownLatch done = new CountDownLatch
         (1);
4    final Runnable task = new CheckedRunnable() {
5      public void realRun() {
6        done.countDown();
7      }};
8    try {
9      executor .execute(task);
10     assertTrue(done.await(SMALL_DELAY_MS,
           MILLISECONDS));
11   } finally {
12     joinPool(p);
13   }
14 }
```

that blocked or terminated component cannot notify other components. In these situations, *Delayed Execution* is a better solution.

2.3.5 Known Uses

This pattern is widely used for testing message-passing (actor-based) libraries. More than 95% of the tests written for Groovy actors use this solution. Additionally, more than 60% of the tests written for Akka actors are based on this solution. The reason is that in message-passing environment most of the communications are asynchronous and hence the events do not cause the components to be blocked. However, a few of test cases written for Java concurrent package library and Smile test cases are also based on this approach.

3. MIXING PATTERNS

As can seen, each one of the patterns described in this paper have their own advantages and disadvantages. A good practical approach would be using a combination of them in a test to alleviate their problems and exploit their advantages. This approach can also be seen in some of the test cases written for testing concurrent programs. An example is shown in Listing 7.

This test is written for testing `BlockingQueue` class in Java concurrent package. Similar to `ArrayBlockingQueue`, `BlockingQueue` is a thread-safe queue with bounded capacity and calling `take` method on an empty `BlockingQueue` would block the calling thread until the queue becomes non-empty (some element is put in the queue).

The goal of this test is to check the blocking of `take` operation on an empty queue. For that, it uses a mixture of latches and `Thread.sleep` to enforce the desired order. The test creates an empty queue and then starts a new thread `t` which tries to take some elements out of the empty queue. This test is very similar to the test in in Listing 3 except that initially the queue is not populated. When the thread `t` starts its execution, it counts down the `threadStarted` latch such that the test thread would be informed that the thread `t` has started its execution. Then, the test calls `Thread.sleep` for some time intervals to let thread `t` calls `q.take`. The latch in this test is used to make the test more reliable: the test

Listing 7 Test for Blocking class in Java using a mix of both Delayed Execution and Wait-notify Coordination patterns

```
1  /**
2   * take() blocks interruptibly when empty
3   */
4  public void testTakeFromEmptyBlocksInterruptibly()
5      throws InterruptedException {
6    final BlockingQueue q = emptyCollection();
7    final CountDownLatch threadStarted = new
         CountDownLatch(1);
8    Thread t = newStartedThread(new CheckedRunnable
         () {
9      public void realRun() {
10       long t0 = System.nanoTime();
11       threadStarted.countDown();
12       try {
13         q.take();
14         shouldThrow();
15       } catch (InterruptedException success) {}
16       assertTrue(millisElapsedSince(t0) >=
             SHORT_DELAY_MS);
17     }});
18
19   threadStarted.await();
20   Thread.sleep(SHORT_DELAY_MS);
21   assertTrue(t.isAlive());
22   t.interrupt();
23   awaitTermination(t, MEDIUM_DELAY_MS);
24 }
```

won't call `Thread.sleep` until thread `t` starts its execution to increase the chance of delay interval to be sufficient.

The mix of the patterns can also be seen in a few of the tests written for Akka and Groovy actor libraries.

4. CONCLUSION AND RELATED PATTERNS

In this paper we discussed about three patterns used for writing tests for concurrent programs. These patterns are extracted by inspecting the test suite of four open source libraries and the discussion that we had with the developers of some closed-source concurrent applications.

As noted before, each solution has its own advantages and limitations. But in general, none of them is well-suited for testing concurrent programs even if they are mixed in a test. The tests written based on these solutions can get very complex and unreliable. In other words, these patterns cry out for a better solution. It turns out the frameworks which are developed for testing sequential programs are not suitable to be reused for testing concurrent programs. Some prior works tried to alleviate these problems by creating a framework for testing multi-threaded programs [18, 17, 14]. But still in these frameworks it is not easy to specify the order of the events in the test and hence they have not got popular enough to be used by the programmers in practice. To the best of our knowledge, even there is no such priori work in the area of message-passing actor programs.

Fortunately, people are working on better solutions, and with any luck, in a few years these patterns will be superceded by better ones. A group of people at UIUC are working on creating a framework for testing multi-threaded programs [16]. We also have a work in progress [19, 11] to develop a framework for testing Scala actors. Although it is specific for Scala programs, but we believe that the idea is general enough to be applied to other object-oriented ac-

tor languages. By this framework, the user can specify the order of the events in the test in an elegant and clean way. Instead of using latches, barriers, and/or Thread.sleep, what the user needs to do is to 1) define the events (messages) that should be ordered (not necessarily all events); and 2) specify the order of the defined events that must be satisfied in the test execution. The framework would execute the test according to the desired order. This will make the test cases more reliable and on the other hand, there is no need to put delay/latch commands which can make the tests very complex. The framework can also handle the blocking or termination of actors.

5. REFERENCES

[1] ActorFoundry, http://osl.cs.uiuc.edu/af/

[2] Akka, http://akka.io/

[3] Akka Project, https://github.com/jboner/akka/tree/master/akka-actor-tests

[4] GPars, http://gpars.codehaus.org/

[5] Groovy DSLs for concurrent processing, http://code.google.com/p/gparallelizer/source/browse/trunk/src/test/groovy/groovyx/gpars/actor/

[6] Java Community Process. JSR 166: Concurrency utilities., http://g.oswego.edu/dl/concurrency-interest/

[7] JUnit, http://www.junit.org/

[8] Novell Vibe, http://www.novell.com/products/vibe-cloud/

[9] Scala, http://www.scala-lang.org/

[10] ScalaTest, http://www.scalatest.org/

[11] Setac, http://mir.cs.illinois.edu/setac/

[12] Smile, https://github.com/robey/smile

[13] Agha, G.: Actors: a model of concurrent computation in distributed systems. MIT Press (1986)

[14] Dantas, A., Brasileiro, F., Cirne, W.: Improving automated testing of multi-threaded software. In: ICST. pp. 521–524 (2008)

[15] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)

[16] Jagannath, V., Gligoric, M., Jin, D., Rosu, G., Marinov, D.: IMUnit: improved multithreaded unit testing. In: Proceedings of the 3rd International Workshop on Multicore Software Engineering. pp. 48–49. IWMSE '10, ACM (2010)

[17] Long, B., Hoffman, D., Strooper, P.: Tool support for testing concurrent Java components. IEEE Trans. Softw. Eng. 29, 555–566 (2003)

[18] Pugh, W., Ayewah, N.: Unit testing concurrent software. In: ASE. pp. 513–516 (2007)

[19] Tasharofi, S., Gligoric, M., Marinov, D., Johnson, R.: Setac: A Framework for Phased Deterministic Testing of Scala Actor Programs. In: 2nd Scala Workshop (2011)