Setac: A Framework for Phased Deterministic Testing of Scala Actor Programs

Samira Tasharofi, Milos Gligoric, Darko Marinov, and Ralph Johnson
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
{tasharo1, gliga, marinov, rjohnson}@illinois.edu

Abstract

Scala provides an actor library where computation entities, called actors, communicate by exchanging messages. The schedule of message exchanges is in general non-deterministic. Testing non-deterministic programs is hard, because it is necessary to ensure that the system under test has executed all important schedules. *Setac* is our proposed framework for testing Scala actors that (1) allows programmers to specify constraints on schedules and (2) makes it easy to check test assertions that require actors to be in a stable state. Setac requires little change to the program under test and requires no change to the actor run-time system. In sum, Setac aims to make it much simpler to test non-deterministic actor programs in Scala.

Keywords actor programs, testing framework, message-passing programs, concurrency bugs

1. Introduction

With the advent of multicore processors and networked computing, parallel and distributed programming is becoming an increasingly important topic. While such programming is often done with the shared-memory paradigm, it leads to problems such as low-level data races. The message-passing paradigm is an alternative that reduces some problems associated with parallel and distributed programming.

The actor-based programming model [2] is one way of programming in the message-passing paradigm. Each *actor* is a concurrent computation entity that communicates with other actors by exchanging messages. Each actor has a *mailbox* which stores the messages delivered to the actor but not yet processed. The *message handlers* in the actors determine which messages should be picked from the mailbox for processing. A message handler removes from the mailbox a message that can be processed and then processes the message. We distinguish between *message delivery*, which refers to adding the message to the mailbox, and *message processing*, which refers to removing the message from the mailbox and executing a handler on it.

Scala provides an actor library in the `scala.actors` package in the standard Scala distribution [5]. (There are also other actor libraries for Scala, such as Akka [3] and Lift [9], but our focus in this paper is on the standard library.) The library provides a number of features, including dynamically creating and destroying actors, sending asynchronous and synchronous messages, handling exceptions, garbage collecting actors, remote actors, and some customization of the thread pool executing actors. However, the Scala library does not provide any special support for testing actors. The common approach is to use single-threaded execution of actors, which does not allow the programmer to examine *different schedules* but does allow the test to be executed deterministically for *one schedule*.

Testing actor programs is challenging precisely due to non-determinism associated with message schedules. Namely, an actor system can have different behavior based on the order in which the messages are processed by each actor. (If the mailbox is a queue and processes the first available message, then the order of message deliveries matters.) It is, therefore, important to explore multiple schedules/orders of messages. Using model checking and verification approaches that try to explore (almost) all possible schedules [6, 7] leads to state space explosion that makes it impossible to check larger programs, even when advanced pruning techniques are used [8]. As a result, our analysis of publicly available, manually written actor tests shows that they focus on very simple cases (test one actor for one message at a time) and encounter problematic issues in *controlling/constraining schedules* and *checking assertions*.

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 and/or a fee.
The Scala Workshop'11 June 2, Stanford, CA, USA.
Copyright © 2011 ACM [to be supplied]. . . $10.00

One issue in writing test cases for actors, using just the standard library, is that constraining the schedule is difficult and sometimes impossible. The difficulty stems from the fact that enforcing specific schedules requires: (1) changing the application (e.g., using synchronous communication instead of asynchronous communication, thread sleeps, latches, barriers, etc. to enforce the order of messages delivered to the actors), (2) changing the environment that runs the application (e.g., changing the system scheduler), and/or (3) creating mock actors to control the order of messages sent to the actor such that one actor is tested at a time. Changing the application or the environment requires great effort on the programmer's side, while creating mock actors can make the test cases very complex and cannot be applied for programs with multiple communicating actors.

Another issue in writing test cases for actors, using just the standard library, is checking assertions, namely checking that the program state satisfies certain conditions at appropriate points in execution. Assertions that check the effect of sending some messages should not be checked until after those messages have been processed by the actors. However, the standard library does not provide a functionality to check if messages have been processed. The solutions that are currently used include: (1) using explicit delays with the hope that the desired messages are processed by the actors, (2) using barriers in the message handlers of the actors after some messages are processed, and/or (3) checking just the final results returned from the actors in the form of messages and not any internal states of the actors.

The issues of controlling schedules and checking assertions during tests were previously addressed for shared-memory programming, in particular for Java. For example, ConAn [10] and MultithreadTC [12] are two frameworks that allow controlling schedules, i.e., specifying the order of accesses to shared data and synchronization variables, and ThreadControl [4] provides a framework for checking assertions at appropriate points in execution. However, to the best of our knowledge, there was no such framework for the actor systems.

We have developed a framework, called Setac, for phased deterministic testing of Scala actor programs. In our framework, the programmer can define certain messages in the program under test as relevant for controlling schedules and checking assertions. Specifically, these are messages whose delivery/processing status is important for the purpose of testing. We refer to these messages as *test messages*. Not all messages in a test execution need to be test messages. The programmer can use Setac to enforce a delivery order between test messages and to obtain some information about their delivery and processing status. Controlling the schedule (to enforce a particular order between some messages) brings an element of *determinism* to test execution.

Assertions should only be checked when the results are ready. In actor programs this usually happens when the individual actors process all the messages they can, i.e., the system reaches a *stable state* where no actor is processing any message and no message can be processed (until new messages are delivered). This point of stability can be viewed as the end of a *phase* in the test execution. Indeed, Setac allows the programmer to break the entire execution of the test into multiple phases. Each phase starts by delivering some messages, then lets the program run, and finally checks assertions. This feature creates a *phased* execution.

Compared to the current approaches for writing actor tests, Setac provides several advantages:

- Setac allows the programmer to easier write tests with some explicit constraints on the message schedule. The schedule is the order of messages delivered to the actors. The constraints do not need to enforce the order of all messages from a test execution but can enforce ordering constraints only on a selected set of *test messages*. Allowing the programmer to enforce some partial constraints avoids two extreme situations: (1) the programmer has no control over the schedule, or (2) the programmer should specify all the details of the schedule although the order of some messages does not matter.
- Setac makes it easy to check assertions in the stable state, which is usually the appropriate time for checking assertions. Without Setac, checking that the system is in a stable state is not straightforward in most cases.
- Setac allows accessing the delivery/processing status of messages defined as test messages, which enables new types of assertions to be succinctly encoded. Additionally, providing a way to define test messages eliminates the need to change the program under test to keep track of the messages.
- Setac reduces the cost of changing the program for the purpose of testing: the only change needed to use Setac is to make each actor class from the code under test a subclass of a Setac's class rather than the standard `Actor`. Note that the programmer need not change literally every class in the program, but only the superclasses of those actors that are direct subclasses of `Actor`. Also note that this change can be automated easily.
- Setac requires no change in the environment that runs the actor program to preserve the portability: since the entire Setac framework is implemented on top of the standard Scala environment that runs the Scala actor programs, Setac makes the testing framework portable.

While this paper presents a high-level overview of our Setac framework, the exact design and implementation of Setac are still evolving. The latest code for Setac and several examples is available from:

`http://mir.cs.illinois.edu/setac`

2. Example

To better describe the problems in testing actor systems, we start with an example. Consider a server that processes clients' requests using a divide-and-conquer recursive algorithm: when the server receives a request that needs a large amount of work, it distributes the work among two children, waits for them to finish their work, merges the results, and returns it as the final result of the request. To make the example more concrete, we assume a simple QuickSort server as shown in Listing 1. It accepts an array of integers as input (through a `Sort` message) and returns an array that is sorted input (through a `Result` message).

When the `QuickSort` actor receives a `Sort` message (line 13), it sets the `requester` variable to the sender of the message and calls the `split` method. This method chooses a pivot, divides the array into two subarrays whose elements are smaller/greater than the pivot, and creates two (children) quick sort actors to sort each subarray. Note that before dispatching the array among the two children, `split` sets a boolean variable `enableProcessingMessage` to false, which disables processing any `Sort` message. It also sets the counter for partial results, `resultCount`, to zero.

After receiving a partial result (through `Result`, line 17), the actor increments `resultCount`, and when the counter reaches two, the actor calls the `mergeResults` method to merge the partial results and returns the final result. After that, it sends the final result to the requester, and sets `enableProcessingMessage` to true which enables the actor to process another sort request. For the sake of simplicity, we omitted the code for merging the partial results.

We want to test the following two properties for the QuickSort example. First, the basic functional correctness: if the actor receives one `Sort` message with some input array, the result should be the sorted array of the input. Second, if it receives multiple `Sort` messages, it should process them one by one and return correct result for each message. Specifically, if it receives a `Sort` message while it is waiting for the partial results of the previous `Sort` message, it should *not* process the second sort message until it finishes its work with the previous `Sort` message.

Writing test code for the first property is quite straightforward; however, testing the second property might not be easy. We need to write a test in which two `Sort` messages are sent to the actor and enforce that the second `Sort` message be delivered before both partial results of the first sort message.

A message sequence diagram that corresponds to such a test is shown in Figure 1; `Sort(input1)` and `Sort(input2)` show the two messages sent from the test to the `qsort` actor, while `Result(part1)` and `Result(part2)` show the two messages sent from the children that are created by the `qsort` actor. As mentioned before, assertions break execution into *phases* in the test. The diagram in Figure 1 consists of two phases: in the first phase, a schedule is enforced in which

Listing 1 Simple QuickSort actor

```
1  class QuickSort extends Actor {
2    var part1 = Array[Int]()
3    var part2 = Array[Int]()
4    var middle = Array[Int]()
5
6    var requester: OutputChannel[Any] = null
7    var resultCount = 0
8    var enableProcessingMessage = true
9
10   start
11   def act() = loop {
12     react {
13       case Sort(input) if (enableProcessingMessage) => {
14         requester = sender
15         split(input)
16       }
17       case Result(res) => {
18         resultCount += 1
19         if (resultCount == 1) {
20           part1 = res
21         } else if (resultCount == 2) {
22           part2 = res
23           val finalResult = mergeResults
24           requester ! Result(finalResult)
25           enableProcessingMessage = true
26         }
27       }
28     }
29   }
30
31   private def split(xs:Array[Int]) {
32     if (xs.length <= 1) {
33       requester ! Result(xs)
34     } else {
35       enableProcessingMessage = false
36       resultCount = 0
37
38       val pivot = xs(0)
39       val left = (xs filter (pivot >))
40       val right = (xs filter (pivot <))
41       middle = (xs filter (pivot ==))
42       new QuickSort ! Sort(left)
43       new QuickSort ! Sort(right)
44     }
45   }
46
47   private def mergeResults:Array[Int] = {
48     // merge part1, part2, and middle
49   }
50 }
```

the `qsort` actor receives the first sort message, then a partial result from one of its children, and finally the second sort message. At the end of the phase, it is checked that the actor has not processed the second `Sort` message. In the second phase, the second partial result is delivered, and it is checked that the resulting arrays are sorted.

2.1 Problems with Writing Test for QuickSort

Listing 2 shows a possible implementation of the test depicted in Figure 1 *without* using Setac but using only the current testing frameworks, e.g., JUnit. To impose the desired order of message delivery, we had to change the code under test as shown in Listing 3.

In Listing 3, we added some arguments to the constructor: `childId` to distinguish the first and second child from the parent, `res1Latch` to be informed about sending `part1` (by one of the children) and then send `sort2` (because we want

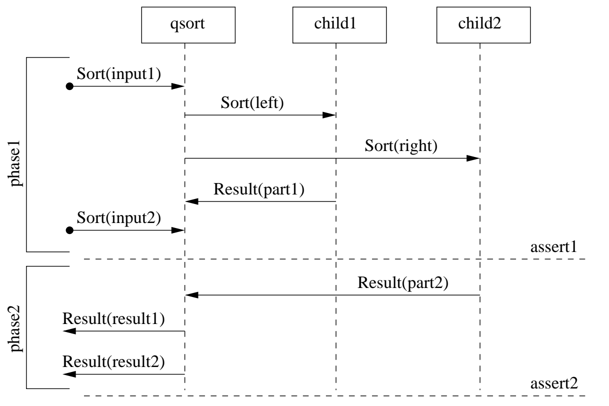

Figure 1. Sequence diagram describes a test which checks that processing multiple Sort messages do not overlap

Listing 2 Test written in the current testing frameworks

```
class QuickSortTest extends TestCase {
  var qsort: QuickSort = null
  var input1 = Array[Int](2,3,1)
  var input2 = Array[Int](4)

  var res1Latch: CountDownLatch = null
  var assert1Latch: CountDownLatch = null
  var sortLatch: CountDownLatch = null
  var finishLatch: CountDownLatch = null

  override def setUp {
    res1Latch = new CountDownLatch(1)
    assert1Latch = new CountDownLatch(1)
    sortLatch = new CountDownLatch(2)
    finishLatch = new CountDownLatch(1)
    qsort = new QuickSort(0,res1Latch,assert1Latch,
        sortLatch)
  }

  def testTwoSortMessages {
    actor {
      // phase1
      qsort ! Sort(input1)
      // wait for the first partial result
      res1Latch.await()
      qsort ! Sort(input2)
      Thread.sleep(1000)
      // assertion1: actor does not process sort2
      assert(sortLatch.getCount == 1)

      // phase2
      assert1Latch.countDown
      // assertion2: resulting arrays are sorted
      receive { case Result(result) =>
          assert(isSorted(result,input1))
      }
      receive { case Result(result) =>
          assert(isSorted(result,input2))
      }
      finishLatch.countDown
    }
    finishLatch.await
  }

  def isSorted(result:Array[Int],input:Array[Int]):
      Boolean = {
    // check if the result is sorted array of input
  }
}
```

Listing 3 QuickSort actor changed for testing

```
class QuickSort(childId:Int,res1Latch:CountDownLatch,
    assert1Latch:CountDownLatch,sortLatch:CountDownLatch
    ) extends Actor {
  var part1 = Array[Int]()
  var part2 = Array[Int]()
  var middle = Array[Int]()

  var requester: OutputChannel[Any] = null
  var resultCount = 0
  var enableProcessingMessage = true

  start
  def act() = loop {
    react {
      case Sort(input) if (enableProcessingMessage) => {
        // if parent, track processed messages
        if (childId == 0)
          sortLatch.countDown
        requester = sender
        split(input)
      }
      case Result(res) => {
        resultCount += 1
        if (resultCount == 1) {
          part1 = res
        } else if (resultCount == 2) {
          part2 = res
          val finalResult = mergeResults()
          requester ! Result(finalResult)
          enableProcessingMessage = true
        }
      }
    }
  }

  private def split(xs:Array[Int]) {
    if (xs.length <= 1) {
      // wait for phase1 and assertion1 to be finished
      if (childId == 2)
        assert1Latch.await()
      requester ! Result(xs)
      // signal on sending the first partial result
      if (childId == 1)
        res1Latch.countDown
    } else {
      enableProcessingMessage = false
      resultCount = 0

      val pivot = xs(0)
      val left = (xs filter (pivot >))
      val right = (xs filter (pivot <))
      middle = (xs filter (pivot ==))
      new QuickSort(1,res1Latch,assert1Latch,sortLatch) !
          Sort(left)
      new QuickSort(2,res1Latch,assert1Latch,sortLatch) !
          Sort(right)
    }
  }

  private def mergeResults():Array[Int] = {
    // merge part1, part2, and middle
  }
}
```

an order in which the second `Sort` message is delivered after the first partial result), `assert1Latch` such that the other child can finish the first phase (checking assertion1) and send `part2` for starting the second phase, and `sortLatch` to keep the number of `Sort` messages processed by the `qsort` actor and to infer the number of `Sort` messages remaining in its mailbox. The body of QuickSort actor is also changed to `countDown` or `wait` on the latches at appropriate times. Note

that these changes might be different for different test cases. For example, if `input2` has more than one element or the order of elements in `input1` is changed, then we will have more children that send the partial results, and so we need more latches.

The test case from Listing 2 creates the latches and passes them to the constructor of the `qsort` actor. We set the variable `childId` to zero to mark it as the parent. In `testTwoSortMessages` method, line 20, a mock actor is created that sends (receives) messages to (from) `qsort` actor. For checking assertion1, we use thread sleep for a time interval to make sure that the `qsort` actor cannot process any more messages and then check the value of `sortLatch`. Because we know that we sent two `Sort` messages, if the value of `sortLatch` equals to one, it means that one of the `Sort` messages still remains in the mailbox and has not been processed by the actor. The `finishLatch` is used to wait for the mock actor to finish its work before the test terminates.

In summary, writing the test described in Figure 1 using current testing frameworks has the following challenges:

- Reasoning about the delivery of important messages (test messages): how can we define messages that are important in the test and be informed about their delivery? As shown, using latches requires changes in the program under test, which makes the test very complex.
- Enforcing the desired order between the messages: if all of the actors in the system were created by the user, sometimes it is possible to impose some orders between the messages, but it is not possible in all situations. In the QuickSort example, the issue is that children, created for providing partial results, are not under the control of the user. So, the user cannot easily impose the schedule in which the second sort message is delivered before the second partial result of the first sort message. As shown, changing the program under test is not an easy and practical solution.
- Checking the assertions: how do we know when to check `assertion1`? How long do we have to wait to make sure that the `qsort` actor has processed as many messages as it can and thus cannot process more messages?
- Accessing the contents of the mailbox: in `assertion1`, how can we check that the second sort message remains in the mailbox and will not be processed until `phase2` starts (the actor will not start processing the second sort message until it finishes its work with the first sort message)? We added `sortLatch` to the program under test to keep track of the `Sort` messages processed by the actor and then reason about the messages remaining in the mailbox of the actor. So, in order to check the contents of the mailbox, we need to keep track of the messages sent to the actor and processed by the actor via adding some extra variables and changing the application under test.

3. Example in Setac

Listing 4 shows the test from Figure 1 written in Setac. Each test class is a test suite that consists of three parts: `setUp`, test cases, and `tearDown`. Before and after running each test case, the `setUp` and `tearDown` methods are executed, respectively In our QuickSort example, `tearDown` is not needed, therefore it is not shown in Listing 4.

To allow Setac to control the execution of the program under test, each class that extends `Actor` should replace it with `TestSubject`. Specifically, in Listing 1 we will have `class QuickSort extends TestSubject`, but note that this is literally the *only* change to the original code under test.

3.1 Defining Test Messages

The first step in writing a test with Setac is to define the *test messages*, which can be of two kinds: *schedule messages* are used in constraining the schedule, and *watch messages* are only checked for their delivery or processing status. In our QuickSort example, we need four schedule messages—`sort1`, `sort2`, `part1`, and `part2`—that are all sent to the `qsort` actor. There are other, non-test messages in the system, e.g., the `Sort` messages sent from `qsort` to the children, but we do not care about the order of their delivery.

Each test message is identified by three parameters: *sender*, *receiver*, and *content*. The sender and the receiver are the IDs of the respective actors. The value `ANY_ACTOR` stands for the wildcard that can be matched with any actor. The content can be an object or a partial function that should be matched with the messages in the system. The power of partial functions for pattern matching brings a lot of flexibility in defining test messages, especially when some values are not known statically but determined at run time.

In Setac, test messages need to be defined before the main test execution. Therefore, the `setUp` method is a good place to define them, although they can be defined at the beginning of the test method itself.

In Setac, schedule messages are defined, and the references to them are created, with the `createScheduleMessage` method that takes the sender, receiver, and content. This method returns a reference to a schedule message that can be used in the test to refer to that message. Note that *the messages that are not matched with any of the schedule messages will be delivered without any constraints*. (Watch messages are explained in more detail in Section 5.)

For our QuickSort example, we define four schedule messages in the `setUp` method in Listing 4 using the method `createScheduleMessage`. For two of these message, the content parameter is known statically (the arguments of two `Sort` messages are `input1` and `input2`), so they are defined as objects. For the other two messages, the content is not known statically (the arguments of two `Result` messages), so they are defined as partial functions that can match any `Result` message sent to the `qsort` actor. The senders of these messages are not important and are thus set to `ANY_ACTOR`.

Listing 4 Example test written in Setac

```
class QuickSortTest extends Test {
  var qsort: QuickSort = null
  var input1 = Array[Int](2,3,1)
  var input2 = Array[Int](4)

  // test messages
  var sort1: TestMessage = null
  var sort2: TestMessage = null
  var part1: TestMessage = null
  var part2: TestMessage = null

  override def setUp() {
    qsort = new QuickSort
    sort1 = createScheduleMessage(ANY_ACTOR, qsort,
                                  Sort(input1))
    sort2 = createScheduleMessage(ANY_ACTOR, qsort,
                                  Sort(input2))
    part1 = createScheduleMessage(ANY_ACTOR, qsort)
                               ({case Result(_) => ()})
    part2 = createScheduleMessage(ANY_ACTOR, qsort)
                               ({case Result(_) => ()})
  }

  def testTwoSortMessages() {
    // phase1
    setSchedule(sort1 -> part1 -> sort2)
    qsort ! Sort(input1)
    qsort ! Sort(input2)
    // assertion1: actor does not process sort2
    assertWhenStable("mailbox has the second sort message
         ",getMsgCount(sort2, qsort) == 1)

    // phase2
    setSchedule(part2)
    // assertion2: resulting arrays are sorted
    receive { case Result(result) =>
        assertTrue("result1 is not sorted",isSorted(
             result,input1))
    }
    receive { case Result(result) =>
        assertTrue("result2 is not sorted",isSorted(
             result,input2))
    }
  }

  def isSorted(result:Array[Int],input:Array[Int]):
       Boolean = {
    // check if the result is sorted input array
  }
}
```

3.2 Controlling Schedule and Checking Assertions

A test will execute the code under test with some inputs. The advantage of Setac is that the programmer can (1) control the test execution by enforcing some specific order of (schedule) messages and (2) check assertions at some points in the middle of test execution (not just at the end). These features allow overcoming some non-determinism in actor programs and having a more fine grain control of the code under test.

Controlling Schedule After defining the schedule messages, the order of their delivery can be enforced by the `setSchedule` method. Listing 4 uses this method, lines 26 and 33. In general, this method accepts a set of constraints among schedule messages, where each constraint is specified with a precedence operator, '->', between schedule messages. Note that if a reference to a schedule message is created but not used in any `setSchedule` method, this message will not be delivered by Setac. Instead, it will give a warning indicating that there are some messages that are not delivered. In other words, creating a reference to a schedule message means that the programmer wants to determine when the message should be delivered.

Checking Assertions Setac provides two methods for checking assertions: `assertWhenStable` and `assertAfter`. Both methods take a name and a boolean expression, and `assertAfter` additionally takes a *time out* value. The difference between these methods is in the execution point at which they evaluate the boolean expression: `assertWhenStable` waits for the system to get stable (no actor is processing any message and all the actors are suspended, blocked, or terminated or have not been started yet) and then evaluates the expression; and `assertAfter` evaluates the expression after a specified amount of time. In our QuickSort example, for checking `assertion1` the system should be stable (to ensure that the `qsort` actor cannot process any more messages), while `assertion2` can be evaluated immediately (because the results are ready). So, as shown at the end of `phase1` and `phase2` in Listing 4, `assertWhenStable` is used for `assertion1`, and `assertTrue` is used for `assertion2`. (Section 5 explains `assertAfter` in more detail.)

Accessing the contents of mailbox Setac also provides methods for inspecting the content of mailboxes. One of the methods is `getMsgCount`, used in `assertion1`. This method takes a test message and an actor as inputs, and returns the number of the messages in the mailbox of the actor that match the test message. In our example, at the end of `phase1`, we need to check that the second sort message is still in the mailbox and has not been processed by the actor.

4. Architecture of Setac

We next briefly describe the architecture of our framework. Setac comes in a package named `edu.illinois.cs.setac`, shown in Figure 2. Note that `Actor` is the actor class from the standard Scala library. Setac consists of several components.

`TestScheduler` is a singleton actor that coordinates the communication of `TestSubject` actors. More specifically, `TestScheduler` is responsible for enforcing the schedule provided by the programmer. `TestSubject` overrides the `send` method, which is used in the standard `Actor` to deliver a message, in order to ask `TestScheduler` for permission to deliver a message. If the message should not be delivered (based on the constraints specified in the test), then `TestScheduler` keeps that message in its cloud (a container for the messages that should be delivered later) and delivers it at the appropriate later time.

`TestSubject` is a trait that extends `Actor`. `TestSubject` communicates with `TestScheduler` upon sending and receiving messages. `TestSubject` also communicates with `TestSubjectRegistry` to register itself in the registry list. To allow Setac to manage messages, the actors in the program under test should extend `TestSubject`.

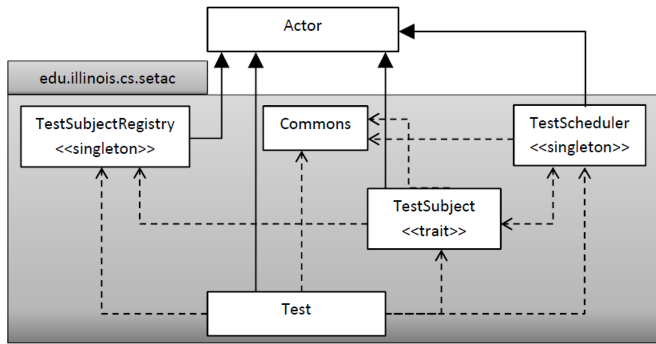

Figure 2. Architecture of the Setac framework

`TestSubjectRegistry` is a singleton actor which is responsible for registering each `TestSubject` in its registry list. This registry list is used to keep track of the actors in the test and also to kill them when the test finishes.

`Test` is a class that extends `Actor` and provides the functionality for writing test cases. It uses `TestScheduler`, `TestSubjectRegistry`, and `TestSubject` to provide the necessary functionality.

`Commons` contains all the common types used in our `edu.illinois.cs.setac` package, e.g., `TestMessage`.

5. Other Features

The key Setac features are explained and used in our QuickSort example. We explain here the additional Setac features. Note that Setac is still under development, so the list of features that are described may not be complete. The latest version of the framework is available online [16].

5.1 Watch Messages

Some messages do not require a specific schedule, but it is important to check the delivery or processing status of these messages. In Setac, we call these test messages *watch messages*. They are defined via the `createWatchMessage` method that takes the same parameters as `createScheduleMessage`. Watch messages are lighter than schedule messages in that the framework need not interfere with the delivery of watch messages but only observes them to collect the information of their delivery/processing. Because schedule messages are also observed as watch messages, the programmer can obtain the delivery/processing status of schedule messages without defining them as watch messages.

5.2 Status of Test Messages

In Setac, there are different methods for obtaining the delivery/processing information of test messages, including whether a test message is processed/delivered or not, and the number of test messages processed/delivered.

5.3 Status of Actors

During test execution, the actors can have different execution status. There are five methods—`isBlocked`, `isSuspended`, `isRunning`, `isTerminated`, and `isNotStarted` (the actor is created but not started yet)—that can be used to access the execution state of the actors. These methods are particularly useful for checking deadlock in the system, i.e., when all or some actors in the system are blocked.

Considering that the mailbox of an actor is a part of the actor's state, `getMailboxSize` and `getMsgCount` can be used to obtain the total number of messages in an actor's mailbox and the number of a particular test message in the actor mailbox, respectively. Our QuickSort example shows how to use `getMsgCount` to access the number of `sort2` test messages in the mailbox of the `qsort` actor.

5.4 Assertions for Non-Stable Systems

The method `assertWhenStable` waits for the system to get stable and then evaluates the expression. However, sometimes the system may not get stable. An example is an actor that has a loop in receiving `TIMEOUT` message, e.g., using `receiveWithin` or `reactWithin` in a `loop`. In these cases, `assertAfter` should be used, which takes a timeout interval as the argument and checks the assertion after that timeout.

6. Limitations

Although some of the problems with testing Scala actor programs are addressed in Setac, our framework has its own limitations. As mentioned before, only the actors that extend `TestSubject` can be controlled by Setac. This means that anonymous actors which do not extend `TestSubject` are out of control of Setac. An example can be `Futures` in Scala. This limitation also carries over to the cases where the source code of the program is not available. One solution for this limitation would be to provide some of the Setac functionality directly into the standard `Actor` class. We believe that Setac is useful enough to consider this approach.

Another issue is with the systems that do not get stable. In that case, the programmer should either use some black-box testing that does not depend on the schedule or provide a time out for checking assertions. This will reduce the robustness of the test cases because they will become dependent on some real time values.

7. Related Work

Our design and implementation of Setac are inspired by testing frameworks for shared-memory, multithreaded Java code such as ConAn [10] and MultithreadedTC [12]. In ConAn, the programmer can specify the order of the events in the test by using a global clock value. A difficulty of writing test cases in ConAn is that the user needs to provide the order of *all* events in the test, but the order of some events might not be important (e.g., messages sent from `qsort` to the children in our running example). Another issue with ConAn is that tests are written as scripts in a language different from the language of the program under test. In MultithreadedTC, tests are written in Java, which is the same

as the language of the program under test. The programmer can specify the order of some (not necessarily all) events with respect to a global clock. However, tracking the exact value of the global clock can make it complex and difficult to specify the order of events. We are not aware of any similar framework for testing actor programs.

There are frameworks for model checking and automated test generation for actor programs. For example, McErlang [11] and QuviQ [13], which are used for Erlang programs, mostly deal with the abstract model of system in the form of an FSM. Writing test cases in these frameworks is not easy and requires creating a correct model of the system. Also, the programmer needs to annotate the program to trigger the appropriate events to map the current state of the program to a state in the FSM model. The programmer can provide a random seed for the scheduler; obviously specifying a random seed will not give any intuition to the programmer about the order of the events, but it is useful for trying different random schedules and establishing deterministic replay with respect to a random seed. Finally, these frameworks are targeting Erlang programs and not Scala (actors). Basset [6, 7] is a framework for state-space exploration of actor programs written in Scala and ActorFoundry [1]. Basset tries to explore (almost) all possible schedules, which can lead to state-space explosion and make it impossible to check larger programs.

There are some tools for testing Scala programs, including Java tools, such as JUnit, and the tools written in Scala such as ScalaTest [15], specs [17], and ScalaCheck [14]. ScalaTest is an alternative to JUnit for writing test cases with fewer lines and more clarity. Specs is a framework that facilitates writing test cases in the behavior-driven development testing style that extends the test-driven development by using natural language for writing test cases. ScalaCheck is a tool for automatic unit testing of Scala and Java programs which is based on property specifications and automatic test data generation. In this tool, the programmers specify the behavior of the code in terms of some properties, and the tool generates random test data and checks that the properties hold. Although these tools facilitate testing Scala programs, none of them is designed and specialized for dealing with non-determinism in the actor programs in terms of controlling schedules and checking assertions like Setac.

8. Conclusions and Future Work

We presented our initial work on creating a testing framework, called Setac, for testing Scala actor programs. The main goal of Setac is to alleviate the problems with non-determinism in testing actor programs. The features provided by Setac make it possible for the programmer to write test cases with some constraints on the schedule, and to check the assertions when the system reaches a stable state.

In the future, we plan to integrate Setac with ScalaTest [15] such that the programmers can benefit from the advantages of both frameworks for testing Scala actors. We hope that this integration will provide us with a user base and result in more tests written using Setac. The tests will be used to evaluate the expressiveness of Setac and likely to revise its design. We will also rewrite more of the existing test cases for Scala actors that we have collected or will collect using Setac. Finally, we plan to compare the test cases written with and without Setac in terms of various metrics.

Acknowledgments

We thank Philipp Haller and Frank Sommers for providing a draft of their book on Scala actors, Daniel Spiewak for sharing information about testing Novell Vibe actors, and Nicholas Chen and Steven Lauterburg for providing feedback on an earlier draft of this paper. This material is based upon work partially supported by the Department of Energy under Grant No. DE-FC02-06ER25752 and the National Science Foundation under Grant Nos. CCF-1012759, CNS-0958199, and CCF-0916893.

References

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

[2] G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, 1986.

[3] Akka. `http://akka.io/`.

[4] A. Dantas, F. Brasileiro, and W. Cirne. Improving automated testing of multi-threaded software. In *ICST*, pages 521–524, 2008.

[5] P. Haller and F. Sommers. *Actors in Scala - Concurrent programming for the multi-core era*. Artima, 2011.

[6] S. Lauterburg, M. Dotta, D. Marinov, and G. Agha. A framework for state-space exploration of Java-based actor programs. In *ASE*, pages 468–479, 2009.

[7] S. Lauterburg, R. K. Karmani, D. Marinov, and G. Agha. Basset: a tool for systematic testing of actor programs. In *FSE*, pages 363–364, 2010.

[8] S. Lauterburg, R. K. Karmani, D. Marinov, and G. Agha. Evaluating ordering heuristics for dynamic partial-order reduction techniques. In *FASE*, pages 308–322, 2010.

[9] Lift. `http://liftweb.net/`.

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

[11] McErlang Project. `https://babel.ls.fi.upm.es/trac/McErlang/`.

[12] W. Pugh and N. Ayewah. Unit testing concurrent software. In *ASE*, pages 513–516, 2007.

[13] QuviQ QuickCheck. `http://www.quviq.com/`.

[14] ScalaCheck. `http://code.google.com/p/scalacheck/`.

[15] ScalaTest. `http://www.scalatest.org/`.

[16] Setac. `http://mir.cs.illinois.edu/setac/`.

[17] specs. `http://code.google.com/p/specs/`.