

Bitá: Coverage-guided, Automatic Testing of Actor Programs

Samira Tasharofi ^{*}, Michael Pradel [†], Yu Lin ^{*}, and Ralph Johnson ^{*}

^{*}Department of Computer Science

University of Illinois, Urbana, IL 61801, USA

Email: {tasharo1,yulin2,rjohnson}@illinois.edu

[†]Department of Computer Science, ETH Zurich, Switzerland

Email:michael@binaervarianz.de

Abstract—Actor programs are concurrent programs where concurrent entities communicate asynchronously by exchanging messages. Testing actor programs is challenging because the order of message receives depends on the non-deterministic scheduler and because exploring all schedules does not scale to large programs. This paper presents Bitá, a scalable, automatic approach for testing non-deterministic behavior of actor programs. The key idea is to generate and explore schedules that are likely to reveal concurrency bugs because these schedules increase the schedule coverage. We present three schedule coverage criteria for actor programs, an algorithm to generate feasible schedules that increase coverage, and a technique to force a program to comply with a schedule. Applying Bitá to real-world actor programs implemented in Scala reveals eight previously unknown concurrency bugs, of which six have already been fixed by the developers. Furthermore, we show our approach to find bugs 122x faster than random scheduling, on average.

I. INTRODUCTION

Concurrent programs are becoming increasingly important as multi-core and networked computing systems become the norm. Testing concurrent programs is challenging because a single input may exhibit different behavior due to non-deterministic scheduling. A model of concurrent programming that has been gaining popularity is the *actor model* [6], [23]. Actor programs consist of computing entities called actors—each with its own local state and thread of control—that communicate exclusively by exchanging messages. Since actors do not share state, the actor model reduces the potential for *data races*, a common bug in the shared-memory model. The widespread use of message-passing concurrency in industrial software development [21] and the growing number of libraries and languages that support actor-based programming [7], [8], [22], [37], [11], [28] evidence the popularity of the actor model.

Despite the lack of shared state, testing actor systems is difficult because the order in which actors receive messages—the *schedule* of the execution—is non-deterministic. This non-determinism leads to race conditions at the level of messages. For example, consider the Scala code in Listing 1, which is a simplified version of a bug we found in the real-world actor program Gatling [3]. There are three actor classes: *Writer*, which stores information to external storage, *Action*, which sends its results to the writer, and *Terminator*, which is responsible for proper termination of the program. A program

has *actionNum* instances of *Action* and exactly one instance of each *Writer* and *Terminator*.

Figure 1 shows the message sequence diagram of an execution of a program with one action. When the action receives an *Execute* message, it sends *Write* to the writer and *ActionDone* to the terminator. When the terminator receives the *ActionDone* message, it decreases the number of current actions. If this number reaches zero, the terminator sends *Flush* to the writer, which causes the writer to write all results into external storage, to assign *null* to the *results* variable, and to send a *Flushed* message to the terminator. This execution is successful, because the writer receives *Write* before *Flush*. However, an execution with a different schedule may reorder *Write* and *Flush*. In this case, not only the flushed records is incorrect but the *results* variable is *null* and the program throws an exception.

This paper presents *Bitá*, a scalable, automatic approach to test different schedules of an actor program. The key idea is to leverage *schedule coverage* to focus the exploration of possible schedules on those schedules that are likely to be of interest for exposing bugs. Schedule coverage describes the extent to which a set of possible schedules has already been explored. We present three schedule coverage criteria for actor programs. Bitá exploits these criteria to tests a program in three steps. First, it runs the program to obtain an arbitrary initial schedule. Second, it uses the initial schedule to generate schedules that increase the coverage. Finally, it runs the program with each generated schedule.

Existing approaches to test actor programs by exhaustively exploring all possible schedules [31] do not scale to real-world programs, even with advanced partial order reduction techniques [41], [45]. Another approach is to let developers explicitly specify which schedules to explore during test execution [38], [26], [44]. In contrast, Bitá explores interesting schedules automatically. For testing shared-memory programs (and not actor programs), several schedule coverage criteria have been proposed [46], [48], [9], [33] and leveraged for exploring schedules [24], [49]. To the best of our knowledge, no existing work offers a scalable, automatic technique for testing actor programs based on schedule coverage criteria.

In this paper, we take advantage of schedule coverage criteria to propose a practical technique for automated test-

```

1 class Writer extends Actor {
2   var results = ArrayBuffer[String]()
3   def receive() = { // called when a message is removed from the mail box
4     case Write(result:String) => // if the message is Write(result)
5       results.append(result)
6     case Flush => { // if the message is Flush
7       writeToExternal(results) // write the results into the external storage
8       results = null
9       sender ! Flushed // send message Flushed to the sender
10    }
11  }
12 }
13 class Action(name:String, terminator:Terminator, writer:Writer) extends Actor {
14   def receive() = { // called when a message is removed from the mail box
15     case Execute => { // if the message is Execute
16       writer ! Write(name) // send message Write to the writer
17       terminator ! ActionDone // send message ActionDone to the terminator
18     }
19   }
20 }
21 class Terminator(actionNum:Int, writer:Writer) extends Actor {
22   var curActions = actionNum
23   def receive() = { // called when a message is removed from the mail box
24     case ActionDone => { // if the message is ActionDone
25       curActions -= 1
26       if (curActions == 0) writer ! Flush // send message Flush to the writer
27     }
28   }
29 }

```

Listing 1. Real-world example of a message ordering bug.

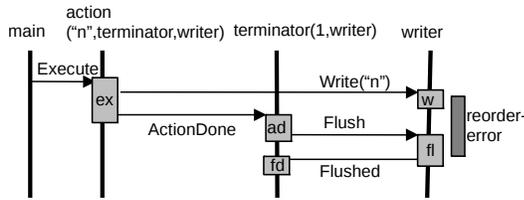


Fig. 1. Sequence diagram of an executions of the code in Listing 1 with one action. Rectangles represent receive events.

ing of actor programs. Specifically, we make the following contributions:

- (1) **Schedule coverage criteria for actor systems.** We present three schedule coverage criteria for actor programs that address common bugs patterns and a technique for measuring the coverage achieved by a set of schedules (Section III).
- (2) **Coverage-guided schedule generation.** We present a coverage-guided approach for automatically generating schedules based on an initial schedule of a program (Section IV). The schedule generator guarantees that each generated schedule is feasible and that it contributes to higher coverage. Previous work for shared-memory programs [49], [24] generates infeasible schedules, which reduces the efficiency of the testing process. Instead, our approach creates feasible schedules by considering must-happen-before relations between messages. Generated schedules can be stored and served as a part of test cases for reproducing bugs or for validating the absence of a particular bug.
- (3) **A technique for forcing a schedule at run-time.** We

present a runtime scheduler that intercepts messages to force a specified schedule while preserving the semantics of the actor model (Section V).

- (4) **Implementation and evaluation with real-world programs.** We implement the Bita approach for Akka [8], a popular actor library for Scala, and apply it to five real-world programs and three smaller benchmarks (Section VI). Bita detects eight previously unknown bugs of which six have already been fixed by the developers. Compared to a scheduler that perturbs the execution by introducing random delays [17], Bita finds bugs substantially faster (122x on average).

II. BACKGROUND

Actors [23], [6] are concurrently executing entities that do not share state but communicate through exchanging messages. Each actor has a mail box for incoming messages and a *message handler* that determines which actions to perform for processing messages. For example, in Akka as shown in Listing 1, the behavior of each actor is specified by a *receive* method and each *case* in the *receive* method is a message handler.

Each step of computation is a *receive* event in which the actor removes a message from its mail box and processes that message. Upon processing a message, the actor may update its local state, change its behavior, send more messages, or create more actors. The message processing is performed in an *atomic step* and without any interruption. This property removes the fine-grained non-determinism and leaves the concurrency non-determinism in the order of messages received by the actors. In Figure 1, each rectangle indicates a receive event.

Although the actor model is built upon asynchronous (non-blocking) communications, it is possible to implement synchronous communications by composing multiple steps of asynchronous communications [6]. In synchronous communication, the sender actor blocks until it receives the reply from the receiver. If a receive event is the reply of a synchronous message sending, then we call it *synchronous receive*; otherwise, we refer to it as *asynchronous receive*.

III. SCHEDULE COVERAGE CRITERIA FOR ACTOR SYSTEMS

This section presents three schedule coverage criteria for actor programs (Section III-A) and how to measure the coverage achieved by a set of executions of an actor program (Section III-B). Sections IV and V use the coverage criteria to automatically generate and execute schedules that explore a subset of all possible schedules that is likely to trigger bugs.

Our approach builds upon the notion of a *schedule*. The schedule of a concurrent program is the order of all concurrency-related events in the program execution. For actor programs, the schedule is given by the sequence of receive events. Formally, a schedule s is a finite sequence of receive events $s = \langle r_1, r_2, \dots, r_n \rangle$ where each receive event r is identified by the sender actor, $sender(r)$, the receiver actor,

$rec(r)$, and the message, $msg(r)$. We ignore send events and actor creation events in the schedule because these events are caused by receive events. That is, scheduling receive events indirectly also schedules send and actor creation events.

There is a trade-off between the bug detection capability of a criterion and the cost of fulfilling the criterion [33]. That is, while satisfying a criterion that requires exploring a larger number of schedules increases the probability to detect bugs, it also increases the cost of the testing process and limits scalability. To balance this trade-off, we focus on criteria that consider pairs of asynchronous receive events of a single actor. The rationale for this decision is threefold. First, considering pairs of receive events is beneficial because the cost of fulfilling these criteria is at most quadratic in the number of concurrent events. Second, considering asynchronous receive events is sufficient because every synchronous receive in an actor is caused by an asynchronous receive and it happens just after that asynchronous receive and before any other asynchronous receive in that actor. Therefore, each ordering of synchronous receive events can be achieved by at least one ordering of asynchronous receive events. Third, considering events of a single actor is beneficial because actors do not share state, that is, only the receive events of a particular actor change the state of the actor.

A. Coverage Requirements

The coverage requirements presented in the following are inspired by common bug patterns of actor programs and shared-memory programs. Each criterion defines *ordering goals* to be achieved by schedules.

1) *Pair of Consecutive Receives*: Many concurrency bugs are triggered when two accesses to a shared resource occur in a particular order [10], [34]. Inspired by this observation, we define the following coverage criterion:

Definition 1 (Pair of Consecutive Receives (PCR)): A schedule s that contains two asynchronous receive events r_i and r_j achieves the ordering goal $r_i \rightarrow_{PCR} r_j$ if and only if

- $rec(r_i) = rec(r_j)$ and
- r_i appears before r_j in s and
- there exists no asynchronous receive event r_k in s so that $rec(r_k) = rec(r_i)$, and that r_k appears between r_i and r_j in s .

As an example, consider a program based on the code in Listing 1 with *two* actions. The writer receives two *Write* messages, with receive events w_1 and w_2 , and one *Flush* message, with receive event fl . The number of *PCR* ordering goals for the writer actor is 6 and one of the possible sets of schedules that covers them is: $S = \{\langle w_1, w_2, fl \rangle, \langle fl, w_2, w_1 \rangle, \langle w_1, fl, w_2 \rangle, \langle fl, w_1, w_2 \rangle\}$.

The *PCR* criterion relates to coverage criteria for shared memory programs that consider pairs of accesses to a shared object [49], [25], [33] and adapts the idea to actor programs.

2) *Pair of Receives*: This criterion is a less restrictive version of *PCR*, in which the two receives for an actor do not need to be consecutive. The variant of *PCR* is useful to detect concurrency bugs that manifest when changing the order of

two receives in a single actor, even if the actor receives other messages between the two receives. For example, consider an initialization bug where a receive r initializes a field in an actor. If a receive r' that dereferences that field appears before r , the invalid *null* value is read and leads to an exception, even if other receive events happen between r' and r .

Definition 2 (Pair of Receives (PR)): A schedule s that contains two asynchronous receive events r_i and r_j achieves the ordering goal $r_i \rightarrow_{PR} r_j$ if and only if

- $rec(r_i) = rec(r_j)$; and
- r_i appears before r_j in s .

While the number of ordering goals in the domain of *PR* and *PCR* for a given program are the same, *PCR* may be satisfied by fewer schedules which brings a merit for *PR* over *PCR*. For the example in subsection III-A1, the number of ordering goals for the writer actor is 6 for both *PR* and *PCR*. However, the ordering goals of *PR* can be covered by the first two schedules, $\langle w_1, w_2, fl \rangle$ and $\langle fl, w_2, w_1 \rangle$, of the four schedules required for *PCR*.

3) *Pair of Behavior Change and Receive*: The behavior of an actor may be changed during its lifetime by changing the actor's message handler. Sending a message to an actor that does not have a compatible handler for the message is a common bug pattern in actor programs [13]. Depending on the actor system, such unsuccessful receives may lead to different kinds of unexpected program behavior. For example, in Erlang [7] and Scala Actors [22], the message will stay in the mailbox, which may lead to mailbox overflow; in Akka 1.3, an exception is thrown; in Akka 2.x, the message will be discarded, which may confuse the sender because it assumes that the receiver has received the message [8]. We define the following criterion aimed at detecting this kind of potential error:

Definition 3 (Pair of Behavior Change and Receive (PBR)): For each receive event r , let $cb(r)$ denotes whether r changes the actor behavior or not. A schedule s with two asynchronous receive events r_i and r_j achieves the ordering goal $r_i \rightarrow_{PBR} r_j$ if and only if

- $rec(r_i) = rec(r_j)$; and
- $cb(r_j) = true$ or $cb(r_i) = true$; and
- there exists no asynchronous receive event r_k in s such that r_k appears between r_i and r_j in s , $rec(r_k) = rec(i)$, and $cb(r_k) = true$.

The domain of *PBR* is a subset of the domain of *PCR* and therefore, the cost of satisfying *PBR* is smaller than *PCR*. For the example in subsection III-A1, suppose the writer changes its behavior when it receives the *Flush* message. The set of *PBR* ordering goals for the writer actor would be achieved by two schedules of $\langle w_1, w_2, fl \rangle$ and $\langle fl, w_1, w_2 \rangle$.

B. Measuring Coverage

The following describes how to quantify the coverage achieved by a set of schedules for the criteria explained in Section III-A. In general, it is impractical to compute the coverage domain of a criterion—all possible ordering goals for

a given program and input—because it requires exploring all possible schedules. Instead, we compute the coverage achieved by different sets of schedules and compare them to each other.

Definition 4 (Coverage of a Set of Schedules): For a coverage criterion cr , a set S of schedules covers a pair $(r_i, r_j)_{cr}$ of receive events if and only if there exist schedules $s_1, s_2 \in S$ such that s_1 covers $r_i \rightarrow_{cr} r_j$ and s_2 covers $r_j \rightarrow_{cr} r_i$.

That is, to increase coverage, one must cover both possible orders of a pair: $r_i \rightarrow_{cr} r_j$ and $r_j \rightarrow_{cr} r_i$. The rationale for this definition is that we are interested in detecting non-deterministic bugs that may manifest only with one of the two orders.

To measure the coverage achieved by a set of schedules, we must match equivalent receive events across multiple executions. A simple approach would be to match equivalent receive events based on the source code location of message handlers. Unfortunately, this approach is very imprecise because a single message handler may execute many times. For example, in a program in which the actor receives thousands of messages of type M , all of the receive events would be identical and hence the coverage for this actor would be equal to another program in which the actor receives one message of type M .

To address the problem of precisely identifying receive events across executions, we compute a hash value for each actor and each message, and identify a receive event via its sender hash value, receiver hash value, and message hash value. By assuming that the application entry point is a receive event with the hash value of zero, we can compute the hash value of all receive events in an execution. The hash value of an actor A is computed based on A 's dynamic type, the hash value of the receive event r that creates A , and the number of actors that r has created before A . Similar, the hash value of a message M is computed based on M 's dynamic type, the hash value of the receive event r that sends M , and the number of messages that r has sent before M .

IV. COVERAGE-GUIDED SCHEDULE GENERATION

Bitá leverages the coverage criteria from Section III to automatically explore potentially bug-revealing schedules. Therefore, it combines a technique for automatically generating schedules, explained in this section, with a run time scheduler that forces a test execution into a specified schedule, explained in Section V. The basic idea of the schedule generator is to capture the schedule of an arbitrary execution of the program, called the *initial schedule*, and to create new schedules by modifying the initial schedule.

The schedule generation approach provides two guarantees. First, each generated schedule increases the coverage compared to the already generated schedules. Second, each generated schedule is *feasible*, that is, there exists at least one execution of the program that satisfies all the ordering constraints of the schedule. These guarantees are based on the assumption that the program does not have any other sources of non-determinism except for message ordering. More formally, let $s = \langle r_1, r_2, \dots, r_n \rangle$ be a schedule and

$s' = \langle r'_1, r'_2, \dots, r'_n \rangle$ be a schedule captured from an execution of the program. We say s' satisfies s iff

- for all $r_i \in s$, $\exists r'_j \in s'$ such that $r_i = r'_j$; and
- let s'_s be the sequence obtained from s' by retaining only the events in s , then $s'_s = s$.

A schedule does not need to contain all events of an execution to be feasible. For example, in Figure 1, the schedule $\langle w, fl \rangle$ is a feasible schedule although it contains only two events. However, the schedule $\langle fd, ad \rangle$ which requires the receive of message *Flushed* to happen before the receive of message *ActionDone* in the terminator is infeasible. The reason is that the receive of *ActionDone* triggers the *Flush* message which itself triggers the *Flushed* message. The schedule generator in Bitá avoids producing such infeasible schedules by analyzing the ordering constraints of all events.

A. Overview of Schedule Generation

The schedule generation algorithm is shown in Algorithm 1. An initial schedule s and the coverage criterion cr are the inputs of the algorithm. The algorithm uses s as the foundation to construct other schedules. The list of generated schedules are kept in S and the set of ordering goals that have been achieved by s and S are held in a global set O . The algorithm updates O whenever it adds a schedule to S .

For each pair of receives r_i and r_j in s , the algorithm checks (line 5) if the events are related to the coverage metric cr and whether swapping them yields a feasible schedule, that is, r_i is not required to happen before r_j . These two conditions are explained in detail in Section IV-B and Section IV-C.

If these conditions hold, the algorithm tries to achieve both ordering goals related to r_i and r_j . It checks whether any of the goals $r_i \rightarrow_{cr} r_j$ (line 6) and $r_j \rightarrow_{cr} r_i$ (line 10) has not yet been achieved by the schedules in S and the initial schedule s , that is, whether the goal is not yet in O . Although r_i appears before r_j in s , the ordering goal $r_i \rightarrow_{cr} r_j$ may not be covered by s if the criterion is *PCR* or *PBR* and if the events are not consecutive. For an ordering goal that has not yet been achieved, the algorithm calls *schedule* to generate a new schedule that achieves the goal (details in Section IV-D). The fourth argument of *schedule* indicates whether the events should be swapped. After generating a new schedule, the algorithm updates the list of generated schedules S and the covered ordering goals O .

B. Identifying Coverage-related Events

To ensure that each generated schedule increases coverage compared to the already generated schedules, Algorithm 1 checks whether two events r_i and r_j are related to the coverage criterion cr . To contribute to the *PR* and *PCR* criteria, the events must have the same receiver; to contribute to the *PBR* criterion, the events must have the same receiver and at least one of them must change the receiver's behavior. Function *isCrRelated*(r_i, r_j, cr) implements this check as follows:

- If $cr = PR$ or $cr = PCR$, it returns *true* if and only if $rec(r_i) = rec(r_j)$.

Algorithm 1 *generateSchedules*(s, cr)

Input: Initial schedule s , coverage criterion cr **Output:** List S of generated schedules

```
1:  $S \leftarrow \emptyset$ 
2:  $O \leftarrow cr$ -ordering goals achieved by  $s$ 
3: for all  $r_i$  in  $s$  so that  $0 < i < |s|$  do
4:   for all  $r_j$  in  $s$  so that  $i < j \leq |s|$  do
5:     if  $isCrRelated(r_i, r_j, cr)$  and  $(r_i, r_j) \notin mustHB$ 
       then
6:       if  $r_i \rightarrow_{cr} r_j \notin O$  then
7:          $s' \leftarrow_{cr} schedule(s, i, j, false)$ 
8:          $S \leftarrow S \cup \{s'\}$ 
9:          $O \leftarrow O \cup cr$ -ordering goals achieved by  $s'$ 
10:      if  $r_j \rightarrow_{cr} r_i \notin O$  then
11:         $s' \leftarrow_{cr} schedule(s, i, j, true)$ 
12:         $S \leftarrow S \cup \{s'\}$ 
13:         $O \leftarrow O \cup cr$ -ordering goals achieved by  $s'$ 
14: return  $S$ 
```

- If $cr = PBR$, it returns *true* if and only if $rec(r_i) = rec(r_j)$ and if either $cb(r_i)$ or $cb(r_j)$.

C. Must-Happen-Before Constraints

To avoid creating infeasible schedules, Algorithm 1 checks whether two events can be reordered or whether the first must happen before the second. For this purpose, we compute the *mustHB* relation by considering all ordering constraints that exist in actor programs. A pair of events (r_i, r_j) is in *mustHB*(s) if r_i must happen before r_j based on the ordering constraints inferred from s . The following explains two kinds of ordering constraints in actor programs and how we compute must happen before relations from them.

1) *Causality Constraints*: If one receive event causes another, then these two events cannot be reordered. In actor programs, such *causality constraints* occur in two cases. First, a message receive r_i directly causes another message receive r_j if executing r_i sends the second message or creates the receiver of the second message. Second, a message receive r_i may indirectly cause another message receive r_j if there is a third message receive r_k between r_i and r_j that has the same receiver as r_i and if r_k directly causes r_j . The second case implies causality because executing r_i may change the receiver's state in a way that causes r_k to send the message of r_j .

Definition 5 (Causality Constraints): The causality constraints $mustHB_{causality}(s)$ of a feasible schedule s contain all pairs (r_i, r_j) with $0 < i < j \leq |s|$ for which one of the following conditions holds:

- $msg(r_j) \in sent(r_i)$
- $rec(r_j) \in created(r_i)$
- $\exists k : i < k < j$ so that $rec(r_i) = rec(r_k)$ and $msg(r_j) \in sent(r_k)$
- $\exists k : i < k < j$ so that $rec(r_i) = rec(r_k)$ and $rec(r_j) \in created(r_k)$

For the example in Figure 1, $mustHB_{causality} = \{(ex, w), (ex, ad), (ad, fl), (fl, fd), (w, fd)\}$.

2) *Sender-Receiver Constraints*: Some actor systems, including the system we use for the evaluation [8], guarantee that for a given pair of actors, messages sent from the first to the second will not be received out-of-order.

Definition 6 (Sender-Receiver Constraints): The sender-receiver constraints $mustHB_{sendRec}(s)$ for a feasible schedule s contain all pairs (r_i, r_j) with $sender(r_i) = sender(r_j)$ and $rec(r_i) = rec(r_j)$.

These constraints only concern the order of messages between a pair of actors. Messages received from different actors can be reordered unless this violates a causality constraint.

For the example in Figure 1, $mustHB_{sendRec}$ is empty. However, suppose an extension of the example where handling the *Execute* message triggers two *Write* messages w_1 and w_2 . In this case, $mustHB_{sendRec} = \{(w_1, w_2)\}$.

3) *Computing Must-Happen-Before Constraints*: Based on the constraints imposed by causality and by sender-receiver message ordering, we compute the set of all must-happen-before constraints as the transitive closure of the union of these constraints.

Definition 7 (Must-Happen-Before Constraints): For a feasible schedule s , the must-happen-before constraints are $mustHB(s) = (mustHB_{causality}(s) \cup mustHB_{sendRec}(s))^+$

Our approach extracts these ordering constraints from the initial schedule and when writing *mustHB* we mean $mustHB(s)$, where s is the initial schedule. In addition to causality constraints and sender-receiver constraints, our implementation also considers ordering constraints imposed by synchronous communication between actors. Details are omitted for lack of space.

D. Generating a Feasible Schedule that Increases Coverage

Once the schedule generator has determined that bringing two events r_i and r_j in a particular order is feasible and that doing so achieves a not yet achieved ordering goal, the schedule generator creates a new schedule to achieve this goal (lines 7 and 11 in Algorithm 1). Our approach to schedule generation addresses two important challenges. First, to create a schedule with enough information for the runtime scheduler to guarantee that it will succeed in forcing the schedule. Second, to create a schedule that achieves not only a single new ordering goal but multiple new ordering goals.

To illustrate these challenges, consider the sequence diagram in Figure 2, which is an extended version of Figure 1. For the extended example, suppose that if the writer receives a *Write* message and the *results* variable is *null*, it sends an *Error* message to the terminator. Moreover, the terminator accepts an additional message *CheckForError* from the application entry point, which checks whether the terminator knows about an error. Figure 2 shows an execution where the writer sends an *Error* message and where the check for errors occurs before the terminator receives this *Error* message.

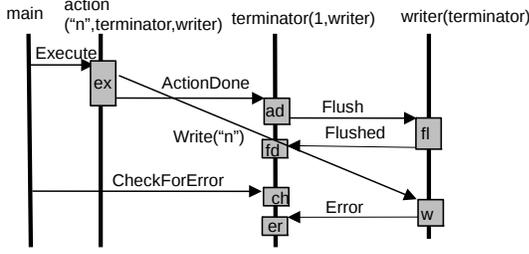


Fig. 2. Sequence diagram that shows an execution of the code in Listing 1 with some extensions in which the terminator accepts a *CheckForError* message and the writer sends an *Error* message to the terminator if *results* is *null*.

Suppose that, based on the execution in Figure 2, the schedule generator tries to reorder *ch* and *er*, so that the *CheckForError* occurs after the terminator has received the *Error* message. A naïve approach would be to create a schedule that specifies only the two events $\langle er, ch \rangle$. Unfortunately, this schedule does not address the two challenges. First, the schedule does not contain enough information for the runtime scheduler to force the schedule. Since there is no constraint on the order of *fl* and *w*, they may happen in the order of $\langle w, fl \rangle$, which results in no *Error* message and hence no *er* event. Previous work shows that this problem can significantly reduce the efficiency of the testing process [49], [24]. Second, the schedule achieves only a single additional ordering goal. Instead, a longer schedule can achieve multiple additional goals at once. For example, a schedule $\langle w, fl, ch, fd \rangle$ for *PR* covers the two ordering goals $w \rightarrow_{PR} fl$ and $ch \rightarrow_{PR} fd$.

Our schedule generation approach addresses both challenges. First, to create a schedule that contains enough information for the runtime scheduler to enforce it, the approach uses the must-happen-before constraints to include all events necessary to make a pair of events happen. Second, to achieve multiple not yet achieved goals in a single schedule, the scheduler does not focus on only two events, but it searches through all remaining events and reorders them to increase coverage.

Algorithms 2 and 3 summarize our approach for generating a schedule. The approach consists of two parts, the first of which is described in Algorithm 2. As its input, the algorithm takes a schedule *s*, two indices *i* and *j*, and a flag *swap* that indicates whether to swap the events at *i* and *j*. The algorithm computes a schedule that brings r_i and r_j in the desired order by scheduling all events that must happen before these two events and by appending r_i and r_j in the desired order.

For example, consider a program based on Listing 1 with two actions and a schedule $s = \langle ex_1, w_1, ex_2, w_2, ad_1, ad_2, fl, fd \rangle$. The other inputs are $i = 1, j = 3, cr = PR$, and $swap = true$, that is, the goal is to swap w_1 and w_2 to achieve an additional *PR* ordering goal. At first, the algorithm copies all the events before r_i to the generated schedule s' (line 2), giving $s' = ex_1$. Next, it searches through all events between *i* and *j* in *s* and copies

Algorithm 2 $schedule(s, i, j, swap, cr)$

Input: feasible schedule *s*, indices *i, j* of events to schedule, flag *swap* that indicates whether to swap these events, coverage criterion *cr*

Output: feasible schedule s' in which the event at *j* comes before the event at *i* if and only if *swap* is true

- 1: $s' \leftarrow$ empty list
 - 2: append all r_k with $0 < k < i$ to s'
 - 3: **for** $k = i + 1$ **to** $j - 1$ **do**
 - 4: **if** $(r_k, r_j) \in mustHB$ **then**
 - 5: append r_k to s'
 - 6: **if** *swap* **then**
 - 7: append r_j, r_i to s'
 - 8: **else**
 - 9: append r_i, r_j to s'
 - 10: *tail* \leftarrow compute tail of s'
 - 11: **return** $scheduleTail(s', tail, cr)$
-

those events that must happen before r_j to s' (lines 3 to 5), which gives $s' = \langle ex_1, ex_2 \rangle$. At this point, all events required for r_i and r_j to happen have been added to s' . Now, the algorithm copies either r_i, r_j (line 7) or r_j, r_i (line 9) to s' , resulting in $s' = \langle ex_1, ex_2, w_2, w_1 \rangle$.

After creating a scheduling that brings two events in a particular order, Algorithm 2 invokes Algorithm 3 (line 11), which considers the remaining events and tries to order them in a way that achieves additional ordering goals. The *tail* of events (line 10) contains all events that are not yet scheduled, excluding events r_k with $(r_i, r_k) \in mustHB$ or $(r_j, r_k) \in mustHB$. It is crucial to exclude such events from the tail because these events may not be valid anymore after reordering r_i and r_j . Depending on the coverage criterion, the tail also contains the last event of the so far generated schedule s' . For *PCR* and *PBR*, r_i and r_j must be consecutive, that is, they should not be reordered as part of the tail. For *PR*, r_i and r_j need not be consecutive, that is, the second of the two events can be reordered is part of the tail.

For the example, the tail of events to reorder is $\langle w_1, ad_1, ad_2, fl \rangle$. Event w_1 is part of the tail because we consider the *PR* criterion, which does not require w_2 and w_1 to be consecutive. Event *fd* is excluded from the tail because (w_1, fd) and (w_2, fd) are in *mustHB*.

Algorithm 3 takes a prefix *p* of scheduled events, the tail *t*, and the coverage criterion *cr* as its input. The basic idea is to append the tail to the generated schedule s' and to try to reorder the events in the tail to increase coverage as much as possible. The algorithm uses an approach similar to Algorithm 1 for finding pairs of events to reorder. Once such a pair of events is found, the algorithm reorders them by passing the concatenation $p + t$ of the prefix and the tail to Algorithm 2. Since Algorithm 2 generates a schedule in which all events before the first event to schedule have the same order as the given schedule, calling Algorithm 2 for the tail events guarantees that the events in the prefix remain in

Algorithm 3 *scheduleTail*(p, t, cr)

Input: schedule prefix p , tail t , coverage criterion cr **Output:** feasible schedule that appends the tail events to p

```
1: for  $r_i$  in  $t$  so that  $0 < i < |t|$  do
2:   for  $r_j$  in  $t$  so that  $i < j \leq |t|$  do
3:     if  $isCrRelated(r_i, r_j, cr)$  and  $(r_i, r_j) \notin mustHB$ 
       then
4:       if  $r_i \rightarrow_{cr} r_j \notin O$  then
5:          $s' \leftarrow schedule(p + t, i + |p|, j + |p|, false)$ 
6:         return  $s'$ 
7:       else if  $r_j \rightarrow_{cr} r_i \notin O$  then
8:          $s' \leftarrow schedule(p + t, i + |p|, j + |p|, true)$ 
9:         return  $s'$ 
10: return  $p$ 
```

the given order. If the algorithm does not find any tail events to reorder, then it omits the tail from the schedule, that is, the schedule does not specify the order of events in the tail.

Algorithms 2 and 3 recursively call each other until the tail does not contain any events to reorder. Since the tail is becoming shorter for each recursive invocation of Algorithm 3 this recursion is guaranteed to terminate for every finite initial schedule.

For our example, the first call to Algorithm 3 selects w_1 and fl for reordering and calls Algorithm 2 to create a schedule where fl precedes w_1 . This call results in $\langle ex_1, ex_2, w_2, ad_1, ad_2, fl, w_1 \rangle$. After this step, the tail is empty and Algorithm 2 returns the generated schedule. This schedule achieves two new ordering goals $w_2 \rightarrow_{PR} w_1$ and $fl \rightarrow_{PR} w_1$.

The schedule generation (Algorithm 1) guarantees that each generated schedule increases the coverage compared to the already covered schedules and that each generated schedule is feasible. The first property holds because the algorithm only generates a new schedule for an ordering goal if the goal has not been covered yet. After generating each new schedule, it adds all the orderings covered in the schedule to the current covered orderings. Therefore, it will never generate a schedule for an ordering that has been already covered. The second property holds because the schedule generator respects all must-happen-before relations. For each event r_i that it adds to the schedule, it also adds all the events that must happen before r_i . Moreover, when it reschedules an event r_i in the schedule, it eliminates all the events r_j that require r_i to happen before them in specific order, i.e., $(r_i, r_j) \in mustHB$.

V. RUN-TIME SCHEDULER

This section describes the third part of Bita which is the run time scheduler that runs the program with a given schedule. The inputs of the scheduler are the program, the test input, and a feasible schedule. We instrument the actor system to intercept all calls to the actor system for sending, receiving, actor creation, and behavior change. For each such call, the scheduler updates its information about the program execution and forces the execution to follow the schedule. To force a

specific schedule, the scheduler interferes with the send events and delivers messages one by one according to the schedule. For each receiver actor, the scheduler holds the next message until the last sent message—whose receive event is the head of the schedule—is processed by the actor.

For sending messages, the scheduler is called when the message is going to be placed in the mail box. When the schedule is not empty, it compares the corresponding receive of the message with the receive event at the head of the schedule. If it matches, the scheduler allows the message to be placed in the receiver mail box; otherwise it keeps the message in a pool of held messages to be delivered later. Note that even if the message is not placed in the mail box, the sender actor does not block for sending messages.

Upon receiving messages, the scheduler compares the receive event with the head of the schedule. If it matches, the scheduler updates the current schedule by removing the head of the schedule. After each receive event, the scheduler compares the current head of the schedule with the corresponding receive events of the messages held in the pool. If any held message is eligible to be sent, it delivers them to the receiver.

In addition to forcing schedules, we leverage the infrastructure of the scheduler to measure coverage and to gather information for computing ordering constraints.

VI. EVALUATION

To evaluate the effectiveness of our approach, we have implemented it for Akka [8], a popular, commercially supported actor library for Scala, and apply it to five real-world actor programs and three smaller actor programs. In summary, we have the following results:

- Bita detects twelve bugs, including eight previously unknown bugs. Six of seven bugs that we reported to the developers have already been fixed.
- Bita is more effective in finding bugs than existing approaches: it finds bugs 122x faster than a random scheduler and 656x faster than the default scheduler.
- Within a given time, Bita gives higher coverage than existing approaches, for example, 3x higher *PR* coverage.

A. Experimental Setup

1) *Programs*: Table I lists the programs used in the experiments. The first five programs are open-source, real-world programs. For Fyrie Redis, we use two independent branches of the program. The other three programs are implementations of classical actor problems and the translation of a program used in earlier work [15]. For two of the real-world programs we use the same inputs provided in the test suites of the source repository and for the others we use variants of the inputs in the test suites. The test oracle checks for crashes in the program with a valid input.

Bita relies on the assumption that the tested programs' only source of non-determinism is message ordering. To match this assumption, we must deal with programs that interact with external entities, such as an HTTP server or the actor system scheduler, or that have time-dependent behavior. For example,

Program	LOC	Description
Gatling-v1.4 (Ga)	11,902	Stress testing tool for HTTP servers [3]
GeoTrellis-v0.9 (Geo)	20,706	Geographic data processing engine [4]
Fyrie Redis-v1.2 (FR1)	3,517	Redis client written in Scala [2]
Fyrie Redis-v2.0(FR2)	2,788	Redis client written in Scala [2]
SignalCollect-v2.1(SC)	6,315	Framework for scalable graph computing [5]
Barber (Ba)	329	Sleeping barber problem
Messenger (Ms)	197	Instant messaging application
ProcReg (PR)	322	Translation of process registry in Erlang [15]

TABLE I
PROGRAMS USED IN THE EXPERIMENTS.

some actors in SignalCollect send messages depending on the time passed between the last receive and the current receive. To deal with programs that interact with external entities we extend Bita so that for each external entity, we can define artificial actors as the senders of the external messages. As a result, Bita can treat the external messages as regular messages. To deal with time-dependent behavior, we introduce a logical time and replace checks for the system time by checks for the logical time. The logical time is a counter that is increased when a message is received.

2) *Baselines*: We compare our approach with two other ways to explore the schedules of an actor program: (i) repeated execution with a scheduler that adds random delays before delivering a message, similar to what [17] describes for shared-memory programs, and (ii) repeated executions with Akka’s default scheduler. The random scheduler respects message ordering constraints when it chooses the delay value. For example, to respect sender-receiver constraints for a particular pair of sender and receiver the scheduler always delays a later messages long enough to arrive after an earlier message. The effectiveness of the random scheduler depends on the range from which delays are taken. We experiment with delays in the range $[0, d_{max}]$ for three values of d_{max} : $100ms$, $200ms$, and $300ms$. Larger delays are impractical because of the timeout of synchronous communications.

B. Bug Detection

1) *Real-World Bugs*: Applying Bita to the programs in Table I reveals twelve bugs, as shown in the first column of Table II. We experiment with four known bugs and Bita finds all of them. For example, the developers of the known bug SC3 mention that “In rare cases the test fails in the following way [...]” [1], which means they cannot reproduce the bug easily but they occasionally observed the bug. Bita finds this bug in every experiment and takes 176 seconds, on average. Since Bita stores the schedules, bugs can be easily reproduced and their absence can be verified after fixing them.

In addition to previously known bugs, Bita detect eight previously unknown bugs: four in Gatling, two in SignalCollect, one in Fyrie Redis, and one in Barber. Except for the bug that we found in Barber, which is implemented by the authors, we reported these bugs to the respective developers in the form of six issues. All but one bug has already been confirmed and fixed by the developers.

2) *Comparison with Baselines*: To compare our approach with random scheduling and Akka’s default scheduler, we measure for each bug how long each approach takes to find it. For each approach, we stop testing if the bug is found or after a timeout of one hour. For programs that contain more than one bug, such as Gatling, we fix all but one bug at a time.

The schedules generated by Bita depend on the schedule from the initial execution. To address this source of non-determinism, we run Bita ten times for each bug. Similar, the random scheduler depends on a random seed and the default scheduler may be influenced by various system effects. We repeat each experiment ten times, giving different random seeds to the random scheduler.

Given the three coverage criteria, which criterion should developers use when testing with Bita? We prioritize criteria based on their cost, which is the number of generated schedules. Based on the discussion in Section III and initial experiments, the number of generated schedules for *PR* and *PBR* are usually smaller than for *PCR*. The number of generated schedules for *PR* and *PBR* may not be comparable. We configure Bita to obtain an initial schedule and to use at first *PR*, then *PBR*, and finally *PCR* until a bug is found. The bug detection time is the sum of the time for obtaining the initial schedule, the time for generating schedules, and the time for executing the program with the generated schedules until the bug is found or timeout is reached.

Table II summarizes how long each approach requires to find each bug. For all measured values, we give the arithmetic mean and confidence intervals (95% confidence level). All times are in seconds. “TO” means the approach does not find the bug before timeout in any of our experiments. If an approach finds a bug in some but not all runs, we compute the average time by optimistically using the timeout value for the runs that do not detect the bug. This situation happened only for *PR* and the default scheduler. The “Tried Criteria” column shows the set of criteria tried by Bita until it finds the bug. For programs where *PBR* is not applicable because these programs do never change the behavior of an actor, Bita skips *PBR* and uses *PCR* after *PR*. The “Schedule” column indicates the number of schedules tested by Bita. The “Exec” column for the baselines shows the number of executions until the bug is detected or until the timeout is reached.

The results show that Bita finds all bugs within a time that is reasonable for an automatic testing tool, whereas the other approaches miss most bugs within the one hour timeout. The best configurations of the random scheduler, $d_{max} = 200ms$ and $d_{max} = 300ms$, detect only three bugs. The default scheduler finds only one out of twelve bugs. Bita finds ten of the twelve bugs with the first criterion, *PR*, and by running the program with at most three schedules. The *PCR* and *PBR* criteria each detect one bug missed by *PR*. The small confidence intervals for Bita and the large confidence intervals for the baselines show the stability of Bita in detecting bugs.

The bottom of Table II summarizes the results for all twelve bugs and for all ten repetitions per bug. For each approach, we give four values: (i) the total time that the approach spends

Bug	Issue	Bita			Random Scheduler						Default Scheduler	
		Tried Criteria	Time	Schedule	$d_{max}=100ms$		$d_{max}=200ms$		$d_{max}=300ms$		Time	Execs
Ga1(U)	1019	<i>PR</i>	36±1	1	TO	219	TO	204	TO	191	TO	265
Ga2(U)	1018	<i>PR</i>	37±1	1	1,448±672	90±48	137±56	8±3	163±75	8±4	TO	269
Ga3(U)	1018	<i>PR</i>	26±1	1	860±671	53±41	104±64	6±4	100±40	6±2	TO	270
Ga4(U)	1116	<i>PR</i>	25±1	1	TO	216	1,321±515	75±29	326±98	18±5	TO	270
SC1(U)	80	<i>PR</i>	102±15	2±1	TO	181	TO	177	TO	158	TO	182
SC2(U)	81	<i>PR</i>	86±32	2±1	TO	120	TO	111	TO	104	TO	219
SC3(K)	58	<i>PR</i>	176±29	3±1	TO	99	TO	91	TO	90	TO	257
FR11(U)	13	<i>PR</i>	43±6	1	TO	192	TO	181	TO	206	TO	225
FR12(K)	12	<i>PR</i>	36±1	1	TO	495	TO	476	TO	471	TO	594
Ba(U)		<i>PR,PBR</i>	250±43	28±5	TO	334	TO	295	TO	263	TO	532
Ms(K)		<i>PR</i>	14	1	TO	832	TO	878	TO	703	TO	1788
PR(K)		<i>PR,PCR</i>	263±151	32±21	TO	282	TO	256	TO	235	2,268±782	557±180

Summary of all bugs with ten repetitions per bug: Total time – Total bugs – Avg. time to detect a bug – Slowdown														
10,939 – 120 – 91 – 1x			371,543 – 20 – 18,577 – 203x			339,622 – 30 – 11,320 – 124x			335,903 – 30 – 11,196 – 122x			419,020 – 7 – 59,860 – 656x		

TABLE II

BUGS DETECTED AND COMPARISON OF OUR APPROACH TO OTHER APPROACHES. TIMES ARE IN SECONDS. ABBREVIATIONS: “U” MEANS UNKNOWN AND “K” MEANS KNOWN BUG; “TO” MEANS TIMEOUT.

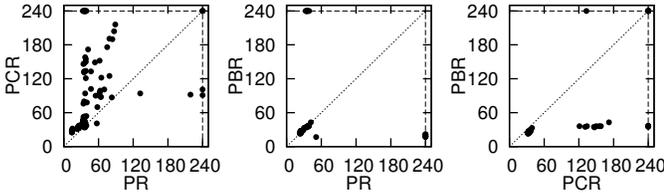


Fig. 3. Pairwise comparison of the four configurations.

when trying to find each of the twelve bugs ten times; (ii) the number of times the approach finds a bug; (iii) the average time to find a bug; (iv) the slowdown of the approach relative to Bita. The results show that Bita clearly outperforms the other approaches. Compared to the best configuration of the random scheduler, Bita finds bugs 122x faster. Compared to the default scheduler, Bita is even 656x faster.

3) *Comparison of Coverage Criteria*: To compare the three coverage criteria to each other, we measure how long Bita takes to find the bugs in Table II if it generates schedules for only one criterion. For each bug, Bita obtains an initial schedule, analyzes the program with each of the three criteria, and measures the time to detect the bug, using a timeout of one hour. We run this experiment for all twelve bugs and repeat it ten times. Figure 3 compares pairs of criteria to each other. Each point (x, y) corresponds to one experiment. The x and y values respectively show the time in which the bug is detected by the criterion at the X axis and the criterion at the Y axis. Points on the dashed line are runs where a criterion does not detect the bug due to timeout. That is, if most of the dots are in the upper-left part of the graph, the criterion at the X axis is better, and if most of the dots are in the lower-right part of the graph, the criterion at the Y axis is better.

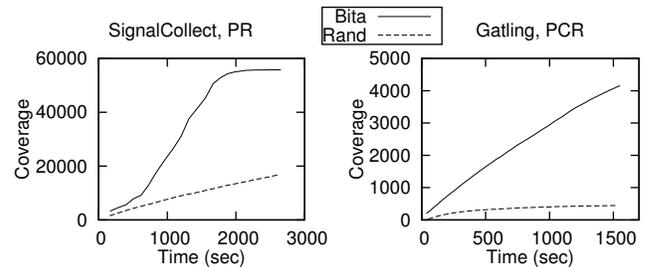
Figure 3 shows that both *PR* and *PBR* perform much better than *PCR*. Because most programs do not change actor behavior at runtime, *PBR* is not applicable for them, and there are fewer points in the plots for *PBR*. For the programs

Progr.	<i>PR</i>				<i>PCR</i>				<i>PBR</i>			
	Time	Bita	Rand Impr.	Impr.	Time	Bita	Rand Impr.	Impr.	Time	Bita	Rand Impr.	Impr.
Ga	732	5,106	525	9.7	1,552	4,158	450	9.2	201	27	1	27
Geo	2,851	2,740	2,039	1.3	3,483	2,557	2,097	1.2	N/A	N/A	N/A	N/A
SC	2,654	55,759	17,422	3.2	3,525	4,814	2,974	1.6	N/A	N/A	N/A	N/A
FR2	674	9,288	4,324	2.1	1,134	10,885	6,992	1.6	N/A	N/A	N/A	N/A

GeoM	3.0	2.3	27
------	-----	-----	----

TABLE III

COMPARISON OF THE COVERAGE ACHIEVED BY BITA AND RANDOM SCHEDULING WITH $d_{max} = 300ms$. TIMES ARE IN SECONDS. THE LAST COLUMN OF EACH CRITERION SHOWS THE IMPROVEMENT OF BITA OVER RANDOM SCHEDULING. THE LAST ROW IS THE GEOMETRIC MEAN.

Fig. 4. Comparison of coverage achieved by Bita and random scheduling with $d_{max} = 300ms$.

that change actor behavior at runtime, *PR* and *PBR* perform similarly. In summary, the results suggest that among all three criteria, *PR* is the most effective criterion for detecting bugs.

C. Coverage

To evaluate Bita’s effectiveness in increasing schedule coverage, we experiment with non-buggy versions of the real-world programs, excluding FR1 because the developers have not yet provided a fixed version. For each criterion, we run a program with Bita and measure the coverage achieved by all schedules generated for the criterion, and the time Bita needs for testing all of them. Then, we repeatedly run the test with random scheduling and with the default scheduler for the same amount of time and measure the achieved coverage.

Table III compares the average over five runs of coverage achieved by Bita and random scheduling with $d_{max} = 300ms$. We only report the best configuration for random scheduling and omit the default scheduler, which performs worse. On average, Bita achieves at least twice the coverage of random scheduling for all three criteria. For *PR*, which is the most effective criterion in detecting bugs, Bita gains coverage three times faster than random scheduling. Figure 4 illustrates the coverage achieved by Bita and random scheduling for two programs and two criteria. The figure illustrates that Bita achieves coverage much faster than random scheduling.

The coverage improvement of Bita over random scheduling is smaller than the improvement in bug finding ability (Table II) because Bita’s coverage domain is smaller. There are two reasons. First, our approach is based on a single initial schedule, whereas random scheduling can discover additional ordering goals in later executions. Second, our approach conservatively considers must-happen-before relations and therefore may miss feasible ordering goals. Despite the smaller coverage domain, Bita clearly outperforms random scheduling in both coverage and bug finding ability.

VII. RELATED WORK

A. Exploring Schedules of Concurrent Programs

One baseline for our evaluation are random delays of message delivery, which is similar to existing work that introduces random delays before thread synchronization points [17]. Other random-based approaches schedule threads based on the partial order of events [39] or based on heuristics [10].

Software model checkers, for shared memory programs [47], [16] or for actor programs [41], [19], [31], [45] explore all possible schedules exhaustively, possibly optimized through partial order reduction to avoid redundant schedules. In contrast to exhaustive exploration, our scheduling approach scales to large programs. To reduce the complexity of exhaustive exploration, one can bound the search space, for example, by bounding the number of preemptions [34]. We are not aware of a bounded model checker for actor programs.

Active testing combines an analysis that finds potential concurrency bugs with schedule generation aimed at exposing these potential bugs [27]. The idea has been applied to data races [40], [12], deadlocks [27], atomicity violations [12], [35], [42], [30], memory-related errors [51], concurrent access anomalies [25], and other errors [50], [20]. In contrast, our approach is not focused on a particular kind of error, but it generates schedules to increase coverage. Another stream of work forces shared memory programs into unusual schedules under the assumption that these schedules are not tested sufficiently [36], [18]. Adapting the ideas of these approaches to actors is subject to future work. Several testing frameworks have been developed for both shared-memory programs [32], [38], [26] and actor programs [44] that run the tests with a schedule given by the programmers. In contrast, our approach automatically generate the schedules of interest without requiring the programmers to write the schedules.

Finally, the idea to use coverage to guide schedule generation has recently been proposed for shared memory programs [49], [24]. Hong et al. generate schedules based on synchronization-pair coverage [24]. Yu et al. consider multiple interleaving idioms to construct a coverage domain [49]. Similar to our work, both approaches define ordering goals based on one or more initial runs. Our work differs by considering actor programs instead of shared memory programs, and by generating only feasible schedules.

B. Coverage of Concurrent Programs

Measuring coverage is widely accepted to assess the effectiveness of tests and to guide the creation of a test suite. Taylor et al. are the first to propose to apply coverage criteria to concurrent programs [46]. Yang et al. adapt all-definition-use pair coverage to concurrent programs and show how to measure it [48]. Synchronization coverage is a set of coverage criteria focused on synchronization primitives of shared memory programs [9]. Lu et al. propose a hierarchy of seven schedule coverage criteria and theoretically analyze the cost of each criterion [33]. Krena et al. propose a set of saturation-based coverage metrics that are derived from dynamic analyses to find concurrency errors [29]. In contrast to all the above approaches, this paper addresses the problem of coverage for actor programs.

Souza et al. propose structural coverage criteria for MPI (Message Passing Interface) programs [43]. Their coverage criteria are based on MPI synchronization primitives and cannot be directly applied to actor programs. In contrast to their work, we leverage coverage criteria for schedule generation.

C. Debugging Actor Programs

Claessen et al. propose to detect bugs in actor programs by automatically generating tests and by using the linearizations of a concurrent execution as an oracle for the concurrent execution [15]. In contrast to our approach, their technique relies in finite state models that formally specify how the program under test should behave. Christakis and Sagonas describe a static analysis of Erlang programs to find message passing errors based on four common bug patterns [14]. In contrast to their analysis, which suffers from false positives, our approach guarantees that each detected bug is feasible and our approach is not limited to particular bug patterns.

VIII. CONCLUSION

This paper presents Bita, an automatic testing approach to efficiently explore the non-deterministic behavior of actor programs. Guided by three novel coverage criteria, the approach automatically generates schedules and forces the program execution to follow these schedules. Each generated schedule is feasible and achieves coverage goals not achieved by previous schedules. Applying the approach to real-world actor programs shows that it finds bugs substantially faster than random scheduling and than repeated execution with the default scheduler. Bita reveals eight previously unknown concurrency bugs, six of which have already been fixed by the developers in reaction to our bug reports.

REFERENCES

- [1] <https://github.com/uzh/signal-collect/issues/58>.
- [2] Fyrie redis. <https://github.com/derekjw/fyrie-redis/tree/akka-1.2>.
- [3] Gatling stress tool. <http://gatling-tool.org/>.
- [4] Geotrellis. <http://www.azavea.com/products/geotrellis/>.
- [5] Signal/collect. <http://uzh.github.io/signal-collect/>.
- [6] G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.
- [7] J. Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, Kungl Tekniska Högskolan, 2003. <http://www.erlang.org>.
- [8] J. Bonér, V. Klang, R. Kuhn, et al. Akka library. <http://akka.io>.
- [9] A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur. Applications of synchronization coverage. In *PPOPP*, pages 206–212, 2005.
- [10] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *ASPLOS*, pages 167–178, 2010.
- [11] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin. Orleans: cloud computing for everyone. In *SOCC*, pages 16:1–16:14, 2011.
- [12] F. Chen, T. F. Serbanuta, and G. Rosu. jPredictor: a predictive runtime analysis tool for java. In *ICSE*, pages 221–230, 2008.
- [13] M. Christakis and K. Sagonas. Detection of asynchronous message passing errors using static analysis. In *PADL*, pages 5–18, 2011.
- [14] M. Christakis and K. F. Sagonas. Detection of asynchronous message passing errors using static analysis. In *PADL*, pages 5–18, 2011.
- [15] K. Claessen, M. Palka, N. Smallbone, J. Hughes, H. Svensson, T. Arts, and U. T. Wiger. Finding race conditions in Erlang with QuickCheck and PULSE. In *ICFP*, pages 149–160, 2009.
- [16] K. E. Coons, S. Burckhardt, and M. Musuvathi. GAMBIT: effective unit testing for concurrency libraries. In *PPOPP*, pages 15–24, 2010.
- [17] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded Java program test generation. *IBM Syst J*, 41(1):111–125, 2002.
- [18] C. Flanagan and S. N. Freund. Adversarial memory for detecting destructive races. In *PLDI*, pages 244–254, 2010.
- [19] L.-Å. Fredlund and H. Svensson. Mcerlang: a model checker for a distributed functional programming language. In *ICFP*, pages 125–136, 2007.
- [20] Q. Gao, W. Zhang, Z. Chen, M. Zheng, and F. Qin. 2ndstrike: toward manifesting hidden concurrency typestate bugs. In *ASPLOS*, pages 239–250, 2011.
- [21] P. Godefroid and N. Nagappan. Concurrency at Microsoft - an exploratory survey. In *EC2*, 2008.
- [22] P. Haller and F. Sommers. *Actors in Scala*. Artima, 2012.
- [23] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI*, pages 235–245, 1973.
- [24] S. Hong, J. Ahn, S. Park, M. Kim, and M. J. Harrold. Testing concurrent programs to achieve high synchronization coverage. In *ISSTA*, pages 210–220, 2012.
- [25] J. Huang and C. Zhang. Persuasive prediction of concurrency access anomalies. In *ISSTA*, pages 144–154, 2011.
- [26] V. Jagannath, M. Gligoric, D. Jin, Q. Luo, G. Rosu, and D. Marinov. Improved multithreaded unit testing. In *ESEC/FSE*, pages 223–233, 2011.
- [27] P. Joshi, M. Naik, C.-S. Park, and K. Sen. CalFuzzer: An extensible active testing framework for concurrent programs. In *CAV*, pages 675–681, 2009.
- [28] R. K. Karmani, A. Shali, and G. Agha. Actor frameworks for the JVM platform: a comparative analysis. In *PPPJ*, pages 11–20, 2009.
- [29] B. Krena, Z. Letko, and T. Vojnar. Coverage metrics for saturation-based and search-based testing of concurrent software. In *RV*, pages 177–192, 2011.
- [30] Z. Lai, S.-C. Cheung, and W. K. Chan. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In *ICSE*, pages 235–244, 2010.
- [31] S. Lauterburg, M. Dotta, D. Marinov, and G. A. Agha. A framework for state-space exploration of Java-based actor programs. In *ASE*, pages 468–479, 2009.
- [32] B. Long, D. Hoffman, and P. Strooper. Tool support for testing concurrent Java components. *IEEE Tr Softw Eng*, 29(6):555–566, June 2003.
- [33] S. Lu, W. Jiang, and Y. Zhou. A study of interleaving coverage criteria. In *ESEC/FSE*, pages 533–536, 2007.
- [34] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *OSDI*, pages 267–280, 2008.
- [35] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *FSE*, pages 135–145, 2008.
- [36] S. Park, S. Lu, and Y. Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *ASPLOS*, pages 25–36, 2009.
- [37] V. Pech, D. König, R. Winder, et al. GPar. <http://gpar.codehaus.org/>.
- [38] W. Pugh and N. Ayewah. Unit testing concurrent software. In *ASE*, pages 513–516, 2007.
- [39] K. Sen. Effective random testing of concurrent programs. In *ASE*, pages 323–332, 2007.
- [40] K. Sen. Race directed random testing of concurrent programs. In *PLDI*, pages 11–21, 2008.
- [41] K. Sen and G. Agha. Automated systematic testing of open distributed programs. In *FASE*, pages 339–356, 2006.
- [42] F. Sorrentino, A. Farzan, and P. Madhusudan. PENELOPE: weaving threads to expose atomicity violations. In *FSE*, pages 37–46, 2010.
- [43] S. R. S. Souza, S. R. Vergilio, P. S. L. Souza, A. S. Simão, and A. C. Hausen. Structural testing criteria for message-passing parallel programs. *Concurr Comput: Pract Exper*, 20(16):1893–1916, Nov. 2008.
- [44] S. Tasharofi, M. Gligoric, D. Marinov, and R. Johnson. Setac: A framework for phased deterministic testing of scala actor programs. In *Scala Days*, 2011.
- [45] S. Tasharofi, R. K. Karmani, S. Lauterburg, A. Legay, D. Marinov, and G. Agha. Transpor: A novel dynamic partial-order reduction technique for testing actor programs. In *FMOODS/FORTE*.
- [46] R. N. Taylor, D. L. Levine, and C. D. Kelly. Structural testing of concurrent programs. *IEEE Tr Softw Eng*, 18(3):206–215, 1992.
- [47] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Autom Software Eng*, 10(2):203–232, 2003.
- [48] C.-S. D. Yang, A. L. Souter, and L. L. Pollock. All-du-path coverage for parallel programs. In *ISSTA*, pages 153–162, 1998.
- [49] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam. Maple: a coverage-driven testing tool for multithreaded programs. In *OOPSLA*, pages 485–502, 2012.
- [50] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. W. Reps. ConSeq: detecting concurrency bugs through sequential errors. In *ASPLOS*, pages 251–264, 2011.
- [51] W. Zhang, C. Sun, and S. Lu. ConMem: detecting severe concurrency bugs through an effect-oriented approach. In *ASPLOS*, pages 179–192, 2010.