Bita: 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
{tasharo1,yulin2,rjohnson}@illinois.edu
†Department of Computer Science, ETH Zurich, Switzerland
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 Bita, 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 Bita 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], [24]. 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 [22] and the growing number of libraries and languages that support actor-based programming [7], [8], [11], [23], [29], [39] 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 logs information to external storage, $Action$, which notifies the writer about its execution, and $Terminator$, which is responsible for flushing the writer when the program terminates. 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. The execution in Figure 1 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 record is incorrect but the $results$ variable is $null$ and the program throws an exception.

This paper presents *Bita*[1], 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 expose 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. Bita exploits these criteria to test 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 [32] do not scale to real-world programs, even with advanced partial order reduction techniques [43], [48]. Another approach is to let developers explicitly specify which schedules to explore during test execution [27], [40], [47]. In contrast, Bita explores interesting schedules automatically. For testing shared-memory programs

The work presented in this paper was partially supported by the Swiss National Science Foundation under grant number 200021-134453.

[1]Bita in Persian means "unique".

```
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       // ... perform the action
17       writer ! Write(name) // send message Write to the writer
18       terminator ! ActionDone // send message ActionDone to the terminator
19     }
20   }
21 }
22 class Terminator(actionNum:Int, writer:Writer) extends Actor {
23   var curActions = actionNum
24   def receive() = { // called when a message is removed from the mail box
25     case ActionDone => { // if the message is ActionDone
26       curActions −= 1
27       if (curActions == 0) writer ! Flush // send message Flush to the writer
28     }
29   }
30 }
```

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

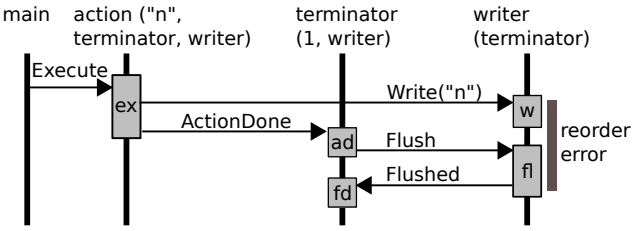

Fig. 1. Sequence diagram of an execution of the code in Listing 1 with one action. Rectangles represent receive events.

(and not actor programs), several schedule coverage criteria have been proposed [9], [35], [49], [51] and leveraged for exploring schedules [25], [52]. 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 automatic testing 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 bug 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 [25], [52] may generate infeasible schedules, which reduces the efficiency of the testing process. Instead, our approach creates feasible schedules by considering must-happen-before relations between message receives. Generated schedules can be stored and provided along with a test case for reproducing a bug or for validating the absence of a particular bug. We also present a run-time scheduler that forces a specified schedule while preserving the semantics of the actor model (Section V).

(3) **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, Bita finds bugs substantially faster (122x on average).

II. Background

Actors [6], [24] 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 set of *message handlers* that determine which actions to perform for processing messages. For example, in Listing 1, the set of message handlers of each actor is specified in 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 the message. Upon processing a message, the actor may update its local state, change its message handlers, send more messages, or create more actors. The message processing is performed as an atomic step. This property removes the fine-grained non-determinism and leaves the concurrency non-determinism in the order of messages received by the actors.

Although the actor model is built upon asynchronous (non-blocking) communication, it is possible to implement synchronous communication by composing multiple steps of asynchronous communication [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 $s = \langle r_1, r_2, ..., r_n \rangle$ 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 schedule coverage criterion and the cost of fulfilling the criterion [35]. 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 that occur in the same 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 happens as part of an asynchronous receive execution. Therefore, each ordering of synchronous receive events can be achieved by at least one ordering of asynchronous receive events. Third, considering events with the same receiver 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 in 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], [36]. 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 six 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 [26], [35], [52] 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 of a single actor, even if the actor receives other messages between the two receives. For example, consider an initialization bug where a receive r_{init} initializes a field. If a receive r_{deref} that dereferences that field appears before r_{init}, the invalid $null$ value is read and leads to an exception, even if other receive events happen between r_{deref} and r_{init}.

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, PR may be satisfied by fewer schedules which brings a merit for PR over PCR. For the example in subsubsection III-A1, the number of ordering goals for the writer actor is six 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 Message Handler Change and Receive : The message handlers of an actor may be changed during its lifetime. 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 [23], the message will stay in the mailbox, which may lead to mailbox overflow; in Akka 1.x, an exception is thrown; in Akka 2.x, the message will be discarded, which may confuse a sender that 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 Message Handler Change and Receive (PMR)): For each receive event r, let $cmh(r)$ denote whether r changes the actor message handlers. A schedule s with two asynchronous receive events r_i and r_j achieves the ordering goal $r_i \rightarrow_{PMR} r_j$ if and only if

- $rec(r_i) = rec(r_j)$; and
- $cmh(r_j) = true$ or $cmh(r_i) = true$; 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 $cmh(r_k) = true$.

The domain of PMR is a subset of the domain of PCR and therefore, the cost of satisfying PMR is smaller than PCR. For the example in subsubsection III-A1, suppose the writer changes its message handler when it receives the $Flush$ message. The set of PMR ordering goals for the writer actor would be achieved by two schedules $\langle w_1, w_2, fl\rangle$ 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. That is, it is generally impossible to compute coverage as a percentage because it is impractical to determine what 100% means. 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 receive events across executions. A simple approach would be to match 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 uniquely 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

Bita leverages the coverage criteria from Section III to automatically explore potentially bug-revealing schedules. Therefore, it combines a technique for automatically generating schedules, presented in this section, with a run time scheduler that forces a test execution into a specified schedule, presented in Section V. The basic idea of the schedule generator is to capture the schedule of an arbitrary execution of a 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 $s = \langle r_1, r_2, ..., r_n \rangle$ be a schedule and $s' = \langle r'_1, r'_2, ..., r'_m \rangle$ be a schedule captured from an execution of the program. We say s' satisfies s if and only if

- 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 Bita avoids producing such infeasible schedules by analyzing the ordering constrains 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 PMR 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 PMR criterion, the events must have the same receiver and at least one of them must change the receiver's message handlers.

Algorithm 1 $generateSchedules(s, cr)$

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

```
1:  S ← ∅
2:  O ← 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 ≤ |s| do
5:      if isCrRelated(r_i, r_j, cr) and (r_i, r_j) ∉ mustHB
        then
6:        if r_i →_cr r_j ∉ O then
7:          s' ←_cr schedule(s, i, j, false, cr)
8:          S ← S ∪ {s'}
9:          O ← O ∪ cr-ordering goals achieved by s'
10:       if r_j →_cr r_i ∉ O then
11:         s' ←_cr schedule(s, i, j, true, cr)
12:         S ← S ∪ {s'}
13:         O ← O ∪ cr-ordering goals achieved by s'
14: return S
```

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

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. 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 but provided in [46]. Note that among these three constraints, only causality constraints are imposed by the basic actor model. In contrast, sender-receiver constraints and synchronous constraints are imposed by the implementation of the actor model in Akka. Other actor systems may impose other constraints that go beyond the basic actor model. Bita could be extended to handle those systems by considering additional constraints.

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.

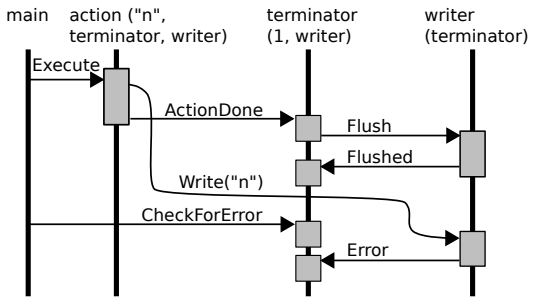

Fig. 2. Sequence diagram that shows an execution of the code in Listing 1 with some extensions in which the terminator accepts a $CheckForError$ message and the writer sends an $Error$ message to the terminator if $results$ is $null$.

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.

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 [25], [52]. 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 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 schedule generation approach. The approach consists of two parts, the first of which is described in Algorithm 2. The algorithm takes a schedule s, two indices i and j, a flag $swap$ that indicates whether to swap the events at i and j, and a criterion cr. 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

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' ← 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) ∈ 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 ←compute tail of s'
11: return scheduleTail(s', tail, cr)
```

$s = \langle ex_1, w_1, ex_2, w_2, ad_1, ad_2, fl, fd\rangle$. The other inputs are $i = 2$, $j = 4$, $cr = PR$, 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 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 PMR, 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

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 ≤ |t| do
3:     if isCrRelated(r_i, r_j, cr) and (r_i, r_j) ∉ mustHB
       then
4:       if r_i →_cr r_j ∉ O then
5:         s' ← schedule(p + t, i + |p|, j + |p|, false, cr)
6:         return s'
7:       else if r_j →_cr r_i ∉ O then
8:         s' ← schedule(p + t, i + |p|, j + |p|, true, cr)
9:         return s'
10: return p
```

to Algorithm 2 (if t contains the last event of p, it will be removed from p). 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 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 shorter for each recursive invocation of Algorithm 3, this recursion terminates 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 $\langle ex_1, ex_2, w_2, ad_1, ad_2, fl, w_1\rangle$. Next, the tail is empty and Algorithm 2 returns the generated schedule, which 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, that is, $(r_i, r_j) \in mustHB$ [46].

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 message handler 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 event 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 messages to be delivered later. Note that even if the is held in the pool, 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 select from the program's test suite a test case that triggers non-deterministic behavior. For the other three programs, we modify an existing test to trigger some non-deterministic features of the program or make the test execution shorter. The test oracle checks for program crashes.

Bita relies on the assumption that the tested programs' only source of non-determinism is message ordering. To match this

TABLE I
PROGRAMS USED IN THE EXPERIMENTS.

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]

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, 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 to 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 [18] describes for shared-memory programs, and (ii) repeated execution 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 observe 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 detects 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 (including the initial execution) 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 PMR are usually smaller than for PCR. The number of generated schedules for PR and PMR may not be comparable. We configure Bita to obtain an initial schedule and to use at first PR, then PMR, 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 PMR is not applicable because these programs do never change message handlers, Bita skips PMR 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 PMR 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

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.

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

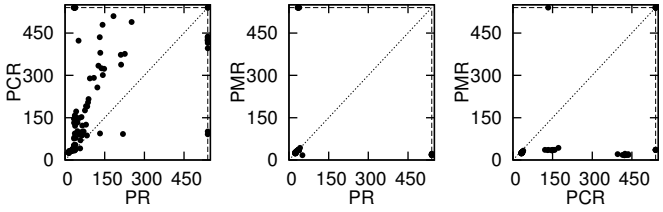

Fig. 3. Pairwise comparison of the three configurations.

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 PMR perform much better than PCR. Because most programs do not change message handlers, PMR is not applicable for them, and there are fewer points in the plots for PMR. For the programs that change message handlers, PR and PMR perform similarly. In summary, the results suggest that among all three criteria, PR is the most effective criterion for detecting bugs.

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.

Progr.	PR				PCR				PMR			
	Time	Bita	Rand	Impr.	Time	Bita	Rand	Impr.	Time	Bita	Rand	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**

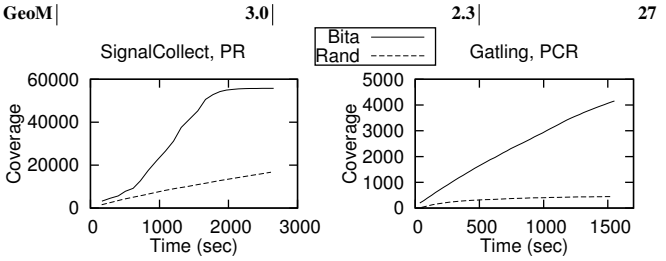

Fig. 4. Comparison of coverage achieved by Bita and random scheduling with $d_{max} = 300ms$.

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, similar to existing work that introduces random delays before thread synchronization points [18]. Other random-based approaches schedule threads based on the partial order of events [41] or based on heuristics [10]. Lei et al. introduce random delays into actor programs to increase the coverage of send-receive pairs [33]. Their coverage criterion considers all permutations of receive events and therefore may not scale to large programs. In contrast to [33], our approach guarantees that each generated schedule increases coverage.

Software model checkers, for shared memory programs [16], [50] or for actor programs [20], [32], [43], [48] explore all possible schedules exhaustively, possibly optimized through partial order reduction. In contrast to exhaustive exploration, our 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 [36]. We are not aware of a bounded model checker for actor programs.

Active testing combines an analysis to find potential concurrency bugs with schedule generation to expose them [28]. The idea has been applied to data races [12], [42], deadlocks [28], atomicity violations [12], [31], [37], [44], memory-related errors [54], concurrent access anomalies [26], and other errors [21], [53]. In contrast, our approach does not focus on a particular kind of error but generates schedules to increase coverage. Other work forces shared memory programs into unusual schedules, under the assumption that these schedules are not tested sufficiently [19], [38]. Testing frameworks, for shared-memory programs [27], [34], [40] and actor programs [47], run tests with a programmer-specified schedule. In contrast, our approach automatically generates schedules.

The idea to use coverage to guide schedule generation has recently been proposed for shared memory programs [25], [52]. Hong et al. generate schedules based on synchronization-pair coverage [25]. Yu et al. consider multiple interleaving idioms to construct a coverage domain [52]. Similar to our work, both approaches define ordering goals based on one or more initial runs. Our work differs by generating only feasible schedules and by considering actor programs.

B. Coverage of Concurrent Programs

Measuring coverage is widely accepted to assess the effectiveness of tests. Taylor et al. are the first to apply coverage criteria to concurrent programs [49]. Yang et al. adapt all-definition-use pair coverage to concurrent programs and show how to measure it [51]. Synchronization coverage [9] is a set of coverage criteria focused on shared memory synchronization primitives. For example, one metric requires that each lock acquisition must be observed as blocked and blocking at least once. The main focus of [9] are metrics that are usable for humans, whereas our approach automates the use of coverage metrics. Lu et al. propose a hierarchy of seven schedule coverage criteria and theoretically analyze their cost [35]. Krena et al. propose saturation-based coverage metrics that are derived from dynamic analyses to find concurrency errors [30]. To identify threads across multiple executions, they compute a hash value from the type of a thread and from the identifiers of the first n methods executed in the thread. We address the problem of identifying actors with a hash value computed based on an actor's dynamic type and on the receive events that create the actor. All the above approaches address shared memory programs and cannot be easily mapped to actor programs.

Souza et al. propose structural coverage criteria for MPI (Message Passing Interface) programs [45]. Their coverage criteria are based on MPI synchronization primitives and cannot be directly applied to actor programs. Deniz et al. describe mutation operators for MCAPI (Multicore Communications API) programs and measure coverage in terms of how many mutants a test kills [17]. In contrast to both approaches, we leverage coverage criteria for schedule generation.

C. Debugging Actor Programs

Claessen et al. propose to detect bugs in actor programs by generating tests and by using linearizations as an oracle for the concurrent execution [15]. Their technique relies on finite state specifications of the program. 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*. Cambridge, MA, USA: MIT Press, 1986.
[7] J. Armstrong, "Making reliable distributed systems in the presence of software errors," Ph.D. dissertation, 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*, 2005, pp. 206–212.
[10] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte, "A randomized scheduler with probabilistic guarantees of finding bugs," in *ASPLOS*, 2010, pp. 167–178.
[11] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin, "Orleans: cloud computing for everyone," in *SOCC*, 2011, pp. 16:1–16:14.
[12] F. Chen, T. F. Serbanuta, and G. Rosu, "jPredictor: a predictive runtime analysis tool for java," in *ICSE*, 2008, pp. 221–230.
[13] M. Christakis and K. Sagonas, "Detection of asynchronous message passing errors using static analysis," in *PADL*, 2011, pp. 5–18.
[14] M. Christakis and K. F. Sagonas, "Detection of asynchronous message passing errors using static analysis," in *PADL*, 2011, pp. 5–18.
[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*, 2009, pp. 149–160.
[16] K. E. Coons, S. Burckhardt, and M. Musuvathi, "GAMBIT: effective unit testing for concurrency libraries," in *PPOPP*, 2010, pp. 15–24.
[17] E. Deniz, A. Sen, and J. Holt, "Verification and coverage of message passing multicore applications." *ACM T Des Automat El*, p. 23, 2012.
[18] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur, "Multithreaded Java program test generation," *IBM Syst J*, vol. 41, no. 1, pp. 111–125, 2002.
[19] C. Flanagan and S. N. Freund, "Adversarial memory for detecting destructive races," in *PLDI*, 2010, pp. 244–254.
[20] L.-Å. Fredlund and H. Svensson, "Mcerlang: a model checker for a distributed functional programming language," in *ICFP*, 2007, pp. 125–136.
[21] Q. Gao, W. Zhang, Z. Chen, M. Zheng, and F. Qin, "2ndstrike: toward manifesting hidden concurrency typestate bugs," in *ASPLOS*, 2011, pp. 239–250.
[22] P. Godefroid and N. Nagappan, "Concurrency at Microsoft - an exploratory survey," in *EC2*, 2008.
[23] P. Haller and F. Sommers, *Actors in Scala*. Artima, 2012.
[24] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in *IJCAI*, 1973, pp. 235–245.
[25] S. Hong, J. Ahn, S. Park, M. Kim, and M. J. Harrold, "Testing concurrent programs to achieve high synchronization coverage," in *ISSTA*, 2012, pp. 210–220.
[26] J. Huang and C. Zhang, "Persuasive prediction of concurrency access anomalies," in *ISSTA*, 2011, pp. 144–154.
[27] V. Jagannath, M. Gligoric, D. Jin, Q. Luo, G. Rosu, and D. Marinov, "Improved multithreaded unit testing," in *ESEC/FSE*, 2011, pp. 223–233.
[28] P. Joshi, M. Naik, C.-S. Park, and K. Sen, "CalFuzzer: An extensible active testing framework for concurrent programs," in *CAV*, 2009, pp. 675–681.
[29] R. K. Karmani, A. Shali, and G. Agha, "Actor frameworks for the JVM platform: a comparative analysis," in *PPPJ*, 2009, pp. 11–20.
[30] B. Krena, Z. Letko, and T. Vojnar, "Coverage metrics for saturation-based and search-based testing of concurrent software," in *RV*, 2011, pp. 177–192.
[31] Z. Lai, S.-C. Cheung, and W. K. Chan, "Detecting atomic-set serializability violations in multithreaded programs through active randomized testing," in *ICSE*, 2010, pp. 235–244.
[32] S. Lauterburg, M. Dotta, D. Marinov, and G. A. Agha, "A framework for state-space exploration of Java-based actor programs," in *ASE*, 2009, pp. 468–479.
[33] Y. Lei and W. E. Wong, "A novel framework for non-deterministic testing of message-passing programs." in *HASE*, 2005, pp. 66–75.
[34] B. Long, D. Hoffman, and P. Strooper, "Tool support for testing concurrent Java components," *IEEE Tr Softw Eng*, vol. 29, no. 6, pp. 555–566, Jun. 2003.
[35] S. Lu, W. Jiang, and Y. Zhou, "A study of interleaving coverage criteria," in *ESEC/FSE*, 2007, pp. 533–536.
[36] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing Heisenbugs in concurrent programs," in *OSDI*, 2008, pp. 267–280.
[37] C.-S. Park and K. Sen, "Randomized active atomicity violation detection in concurrent programs," in *FSE*, 2008, pp. 135–145.
[38] S. Park, S. Lu, and Y. Zhou, "CTrigger: exposing atomicity violation bugs from their hiding places," in *ASPLOS*, 2009, pp. 25–36.
[39] V. Pech, D. König, R. Winder *et al.*, "GPars," http://gpars.codehaus.org/.
[40] W. Pugh and N. Ayewah, "Unit testing concurrent software," in *ASE*, 2007, pp. 513–516.
[41] K. Sen, "Effective random testing of concurrent programs," in *ASE*, 2007, pp. 323–332.
[42] ——, "Race directed random testing of concurrent programs," in *PLDI*, 2008, pp. 11–21.
[43] K. Sen and G. Agha, "Automated systematic testing of open distributed programs," in *FASE*, 2006, pp. 339–356.
[44] F. Sorrentino, A. Farzan, and P. Madhusudan, "PENELOPE: weaving threads to expose atomicity violations," in *FSE*, 2010, pp. 37–46.
[45] 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*, vol. 20, no. 16, pp. 1893–1916, Nov. 2008.
[46] S. Tasharofi, "Efficient testing of actor programs with non-deterministic behavior," University of Illinois at Urbana–Champaign, Tech. Rep., 2013, http://hdl.handle.net/2142/45680.
[47] S. Tasharofi, M. Gligoric, D. Marinov, and R. Johnson, "Setac: A framework for phased deterministic testing of scala actor programs," in *Scala Days*, 2011.
[48] S. Tasharofi, R. K. Karmani, S. Lauterburg, A. Legay, D. Marinov, and G. Agha, "TransDPOR: A novel dynamic partial-order reduction technique for testing actor programs," in *FMOODS/FORTE*.
[49] R. N. Taylor, D. L. Levine, and C. D. Kelly, "Structural testing of concurrent programs," *IEEE Tr Softw Eng*, vol. 18, no. 3, pp. 206–215, 1992.
[50] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda, "Model checking programs," *Autom Software Eng*, vol. 10, no. 2, pp. 203–232, 2003.
[51] C.-S. D. Yang, A. L. Souter, and L. L. Pollock, "All-du-path coverage for parallel programs," in *ISSTA*, 1998, pp. 153–162.
[52] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam, "Maple: a coverage-driven testing tool for multithreaded programs," in *OOPSLA*, 2012, pp. 485–502.
[53] 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*, 2011, pp. 251–264.
[54] W. Zhang, C. Sun, and S. Lu, "ConMem: detecting severe concurrency bugs through an effect-oriented approach," in *ASPLOS*, 2010, pp. 179–192.