

© 2013 Samira Tasharofi

EFFICIENT TESTING OF ACTOR PROGRAMS WITH NON-DETERMINISTIC
BEHAVIORS

BY

SAMIRA TASHAROFI

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Doctoral Committee:

Research Associate Professor Ralph Johnson, Chair
Professor Gul Agha
Associate Professor Darko Marinov
Dr. Madan Musuvathi, Senior Researcher at Microsoft Research

Abstract

The actor model is a model of concurrent programming that consists of concurrent entities called actors. Actors communicate using asynchronous messages, and depending on the order in which they receive messages, they may exhibit different behaviors. This non-determinism brings significant challenges in testing techniques; to make sure that an actor program is correct, conceptually all possible interleavings of message receives should be explored. However, exploring all possible interleavings is not practical because the number of interleavings grows exponentially in the number of messages.

This dissertation targets the problem of efficiently testing actor programs. Specifically, it considers three solutions for reducing the number of explored interleavings.

Partial-order reduction (POR) techniques can substantially prune the number of explored interleavings. These techniques require defining a *dependency relation* on transitions in the program, and exploit independency among certain transitions to prune the state-space. We observe that actor systems exhibit a dependency relation among co-enabled transitions with an interesting property: *transitivity*. We propose a novel dynamic POR technique, TransDPOR, that exploits the transitivity of the dependency relation in actor systems. Empirical results show that leveraging transitivity speeds up exploration by up to two orders of magnitude compared to existing POR techniques.

While TransDPOR makes exploration more efficient, experiments with TransDPOR on large programs reveal that the improvement may not overcome the exponential growth of the state-space. An observation is that a small subset of all possible interleavings might be sufficient to expose potential concurrency bugs. Having the fact that programmers—who have the knowledge of program specification—would be able to identify those interleavings, we developed two testing frameworks, Setac and Setak. These frameworks are developed for testing actor programs implemented in two popular actor libraries in the Scala language, namely Scala actor and Akka respectively. These

frameworks allow programmers to specify constraints on order of messages—called *schedule*—and during execution, they enforce the specified schedule. In addition, Setac/Setak make it easy to check test assertions that require actors to be in a stable state.

From our experience with the users of Setac/Setak, it turns out that in some cases specifying schedules is not straightforward for programmers. To address this problem, we propose Bitac, a scalable and coverage-guided technique which automatically generates schedules. The key idea is to generate schedules that are likely to reveal concurrency bugs because these schedules increase the interleaving coverage. We present three interleaving coverage criteria for actor programs, an algorithm to generate feasible schedules whose executions increase coverage, and a technique to force a program execution to comply with a schedule. Applying Bitac to real-world actor programs implemented in Akka reveals eight previously unknown concurrency bugs, of which six have already been fixed by the developers. Furthermore, we show Bitac finds bugs 122x faster than random scheduling, on average.

To my parents, my husband, and my sister for their love and support.

Acknowledgments

Foremost, I would like to thank my adviser Ralph Johnson for his support and guidance during my Ph.D study. He gave me confidence and taught me how to be patient and independent in research. Without his continuous support, I would not have been able to accomplish my thesis.

Besides my adviser, my sincere gratitude goes to the rest of my thesis committee members, Gul Agha, Darko Marinov, and Madan Musuvathi. They gave me insightful advice and helped me to stay in a right direction and have a right plan for my research. I also got advice from other professors at the Computer Science Department. Madhusudan Parthasarathy, Grigore Rosu, and William D. Gropp kindly scheduled several meetings with me and gave me valuable comments.

I would like to thank my friends in the school including Nicholas Chen, Yun Young Lee, Milos Gligoric, Stas Negara, Mohsen Vakilian, and Maurice Rabb who have been supportive in both emotional and technical aspects of my Ph.D. life. Specially, Nicholas Chen and Yun Young Lee have been my companions in the office, brought me joys, and kept me motivated in my frustrations. My thanks also go to my other colleagues and collaborators, Michael Pradel, Peter Dinges, Rajesh Karmani, Yu Lin, Steven Lauterburg, Axel Legay, Ali Vakilian and my undergrad students—Yanyi Mei, Ehsan Razfar, and Martin Wozniewicz—for their technical help in various projects.

I should also thank the academic staff at the Computer Science Department. Mary Beth Kelly, Rhonda McElroy, Kathy Runck, and Indria F. Clay have been so helpful by answering my questions and resolving the issues that I faced during my time in the school.

I am grateful for the funding sources that allowed me to pursue my graduate school study. I was a recipient of Sohaib Abbasi fellowship in 2008–2009 academic year and Yahoo! Grace Hopper Celebration Scholarship in 2010. I was also funded by Department of Energy grant (DOE DE-FG02-06ER25752) as a research assistant. Department of Computer Science at University of Illinois generously gave me teaching assistantship.

Last but certainly not least, my thanks go to my family, specially my mother, my sister, my sister's husband, and my young brother for their endless love and support. Their calls along with their encouraging words have always kept me motivated. Most of all, I deeply thank my husband who has been my spectacular companion and friend in my life. He drove a long distance every weekend to come and visit me. His tremendous help and support have always made my life blissful.

Table of Contents

List of Tables	ix
List of Figures	x
Chapter 1 Introduction	1
1.1 Overview	1
1.2 Thesis Statement	2
1.2.1 TransDPOR	2
1.2.2 Setac/Setak	4
1.2.3 Bita	5
1.3 Dissertation Organization	7
Chapter 2 Background	8
2.1 The Actor Model and Non-determinism	8
2.1.1 Non-determinism by Example	9
2.1.2 Constraints in Actor Systems	10
2.2 Actor Libraries	10
2.2.1 Scala Actor	11
2.2.2 Akka	12
Chapter 3 TransDPOR: A Dynamic Partial Order Reduction Technique	15
3.1 Motivating Example	15
3.2 The Actor Model Definitions	17
3.3 Partial-Order Reduction (POR)	19
3.4 DPOR for Actor Systems	20
3.4.1 Dependency Relation	20
3.4.2 Equivalent Receive Sequences	21
3.4.3 DPOR Algorithm	22
3.5 TransDPOR	26
3.5.1 Completeness Proof	29
3.6 Implementation and Evaluation	32
3.7 Discussions	35
Chapter 4 Setac/Setak: Frameworks for Exploring User-specified Schedules . .	37
4.1 Setac by Example	40
4.1.1 Problems with Writing Test for QuickSort	42
4.2 Writing Tests with Setac	47

4.2.1	Defining Test Messages	47
4.2.2	Controlling the Order of Message Delivery and Checking Assertions	49
4.2.3	Other Features	50
4.3	Setak	51
4.4	Discussions	52
Chapter 5	Bitac: A Coverage-guided Testing Technique	53
5.1	Bitac by Example	53
5.2	Interleaving Coverage Criteria for Actor Programs	55
5.2.1	Coverage Requirements	56
5.2.2	Measuring Coverage	59
5.3	Coverage-guided Schedule Generation	60
5.3.1	Schedule	60
5.3.2	Overview of Schedule Generation	61
5.3.3	Identifying Coverage-related Events	64
5.3.4	Must-Happen-Before Relation	64
5.3.5	Generating a Feasible Schedule that Increases Coverage	68
5.4	Properties of Generated Schedules	72
5.4.1	Feasible	73
5.4.2	Increase Coverage	75
5.4.3	Sufficiently Detailed	75
5.5	Run-Time Profiler/Scheduler	75
5.6	Implementation and Evaluation	76
5.6.1	Experimental Setup	77
5.6.2	Bug Detection	78
5.6.3	Coverage	83
Chapter 6	Related Work	85
6.1	Exhaustive Exploration	85
6.2	Bounded Explorations and Random Testing	87
6.3	Testing Programmer-specified Interleavings	88
6.4	Coverage Criteria for Concurrent Programs	89
6.5	Coverage-guided and Active Testing	90
6.6	Debugging Actor Programs	91
Chapter 7	Conclusions and Future Work	92
7.1	TransDPOR	92
7.2	Setac/Setak	93
7.3	Bitac	93
References		95

List of Tables

3.1	Comparison of TransDPOR and DPOR	33
3.2	Comparison of TransDPOR+Sleep sets and DPOR+Sleep sets	36
5.1	Programs used in the experiments.	77
5.2	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.	79
5.3	Comparison of three criteria when Bitra is configured to use one criterion. “Prob.“ is the probability of bug detection; “Total Sched.“ is the total number of schedules generated; “Tested Sched.“ is the number of schedules tested until the bug is found; “Time“ is the time taken (in seconds) until the bug is found; “NA“ means not applicable.	83
5.4	Comparison of the coverage achieved by Bitra and random scheduling with $d_{max} = 300ms$. Times are in seconds. The last column of each criterion shows the improvement of Bitra over random scheduling. The last row is the geometric mean.	84

List of Figures

2.1	Communication of actors in bounded buffer program that contains three actors: buffer, producer, and consumer.	9
3.1	The registry example and the messages exchanged.	16
3.2	State-space explored by DPOR and TransDPOR for the registry example in Figure 3.1.	25
4.1	Sequence diagram describes a test which checks that processing multiple Sort messages do not overlap	43
5.1	Sequence diagram of an execution of the code in Listing 5.1 with one action. Vertical rectangles represent receive events and the labels besides rectangles shows the receive events IDs.	55
5.2	The architecture of Bita.	60
5.3	Sequence diagram of an an actor program with three actors. Because of sender-receiver constraint, M_2 is always delivered before M_4	65
5.4	Sequence diagram of an an actor program with synchronous communication.	66
5.5	Sequence diagram that shows an execution of the code in Listing 5.1 with some extensions in which the terminator accepts a <i>CheckForError</i> message and the writer sends an <i>Error</i> message to the terminator if <i>results</i> is <i>null</i>	68
5.6	Pairwise comparison of the three criteria. The X and Y axes show the time (in seconds) that each criterion detects the bug.	82
5.7	Pairwise comparison of the three criteria. The X and Y axes show the number of schedules generated for each criterion.	82
5.8	Comparison of coverage achieved by Bita and random scheduling with $d_{max} = 300ms$	84

Chapter 1

Introduction

1.1 Overview

Concurrent programs are becoming increasingly important as multi-core and networked computing systems become the norm. A model of concurrent programming that has been gaining popularity is the *actor model* [1, 2]. The actor model is used in many systems such as Actor Foundry [3], Akka [4], AmbientTalk [5], Charm++ [6], Erlang [7], GPars [8], Orleans [9], Rebeca [10], and Scala actor [11].

Actor systems consist of computing entities called actors (each with its own local state and thread of control) that communicate by exchanging messages asynchronously. Each actor has a *mailbox* which stores the messages delivered to the actor but not yet processed. A *configuration* of an actor system consists of the local state of the actors and a set of *messages* in the actors mailboxes.

At each step of computation, called *receive event*, an actor removes a message from its mailbox and processes that message. Upon processing a message, an actor may update its local state, send messages, or create new actors. Therefore, actor systems are state-transition systems in which states are configurations and transitions are receive events.

Actor semantics mandate that each actor is perfectly encapsulated, that is, there is no shared state between the actors. While this semantic greatly reduces the potential for *data races*, it does not eliminate *race condition* among receive events. In other words, in an actor program and a given input, the actors may receive messages with different orders and hence exhibit different behaviors.

This non-determinism in the actor model brings significant challenges to the testing community, that is, to make sure that an actor program with a given input is correct, various interleavings of receive events must be explored. Unfortunately, the number of all possible interleavings in an actor program is exponential in the number of exchanged messages. Exploring all possible interleavings

does not scale and in most cases leads to state-space explosion.

Indeed, it is crucial to apply efficient testing techniques that carefully select and explore a subset of all possible interleavings. This dissertation targets the problem of efficient testing of actor systems with non-deterministic behaviors.

1.2 Thesis Statement

This dissertation claims that:

It is possible to apply testing techniques to actor systems that explore a subset of all possible receive interleavings while being effective in revealing concurrency bugs.

To support this statement, this dissertation provides three techniques to reduce the number of explored interleavings:

- TransDPOR: a novel partial-order reduction (POR) technique that exploits properties of the actor model to improve over existing POR techniques;
- Setac/Setak: testing frameworks that provide a language and proper APIs for the programmers to specify the order of message delivery and control the schedule of test execution; and
- Bitac: a practical coverage-guided testing technique that based on some interleaving coverage criteria selects and explores interleavings that increase the coverage.

The rest of this section presents each solution in more details.

1.2.1 TransDPOR

Partial-order reduction (POR) techniques reduce the number of explored interleavings by eliminating equivalent interleavings, i.e., interleavings that lead to the same state. To identify the equivalent interleavings, they require defining a dependency relation among transitions of a concurrent system. The idea is that in a given interleaving, permuting adjacent transitions which are

not dependent yields an equivalent interleaving. A valid dependency relation is a reflexive and symmetric (but *not necessarily transitive*) binary relation on the transitions.

Dependency relation can be computed by static or dynamic analysis. Flanagan and Godefroid introduced a dynamic POR technique, called DPOR [12], that computes the dependency relation by dynamic analysis. The authors show that DPOR improves over POR techniques that compute dependency relation by static analysis. DPOR has been shown to be effective in pruning the state-space exploration when applied to actor systems [13].

In TransDPOR, we leverage the fact that actors do not share their states, and we define a dependency relation between the receive events that is *transitive* on the receives enabled in the same configuration (called co-enabled receives). We exploit transitivity of dependency relation to propose a novel dynamic POR technique that improves over DPOR. Specifically, we make the following contributions:

- (1) **A novel dynamic POR technique.** We present a new stateless dynamic POR algorithm, called *TransDPOR* which extends DPOR to take the advantage of transitive dependency relations in actor systems. We show that TransDPOR in some cases explores fewer configurations/receives than DPOR, but it never explores more.
- (2) **Proof of soundness and completeness.** We prove that TransDPOR is sound and complete like DPOR, that is, every reported bug is a real bug and when the state-space is acyclic—which is the case for actor systems—it can reach every deadlock or local safety violation in the system.
- (3) **Implementation and evaluation.** We implement TransDPOR in Basset [14], a tool for systematic testing of actor programs written in Scala actor [11] or the ActorFoundry [3]. TransDPOR code is publicly available with Basset at <http://mir.cs.illinois.edu/basset>. We compare TransDPOR and DPOR on eight programs without bugs and three programs with bugs. The experimental results show that TransDPOR reduces the number of receive events executed during state-space exploration by $2.39x$ on *average* and *up to* $163.80x$ over DPOR. When we combine TransDPOR and DPOR with sleep sets (a traditional POR technique) [15], we find that TransDPOR can find bugs *up to* $2.56x$ faster than DPOR.

TransDPOR has been published in the form of a conference paper at the 14th joint IFIP WG

6.1 international conference and 32nd IFIP WG 6.1 international conference on Formal Techniques for Distributed Systems (FMOODS’12/FORTE’12) [16].

After experimenting with TransDPOR on larger programs we realized that while TransDPOR is more efficient than DPOR, it still suffers from scalability problems. This motivates us to seek for solutions that explore a small subset of all possible interleavings. One of the solutions is developing two testing frameworks which are explained in the next section.

1.2.2 Setac/Setak

One of the solutions to overcome the scalability problem of POR techniques—and specifically TransDPOR—is to explore a small subset of interleavings which are more likely to reveal potential concurrency bugs.

One way to determine such interleavings is to exploit the knowledge of programmers about program specification and let them specify those interleavings. Unfortunately, current testing frameworks do not provide proper features for actor programmers to specify schedule of a test execution. Our study of publicly available manually written tests shows that actor programmers have difficulties in testing non-deterministic behavior of actor systems using standard libraries [17]. Specifically, to enforce a specific interleaving or check assertions at different points of execution, they use latches and other synchronization constructs from shared-memory world.

For example, usually, the test execution needs to wait for all the messages to be processed by actors—the system reaches a *stable state*—and then checks the assertions. Since standard testing libraries do not provide any feature to wait for a stable state, programmers use `thread.sleep` or latches to make sure that actors have finished their execution and check the assertions. Similar issue exists when programmers want to control the schedule of a test execution and run the test with a specific interleaving.

The issues of controlling schedules of execution and checking assertions in testing concurrent programs were previously addressed for shared-memory programming [18–21]. However, to the best of our knowledge, there was no such framework for the actor systems.

We have developed two testing frameworks, called Setac and Setak¹, for testing actor programs

¹Setac/Setak abbreviated from “Stepwise deterministic testing of Scala/Akka actors”.

written in two popular actor libraries in Scala [22], namely, Scala actor and Akka respectively. The main features in these frameworks which make them appropriate for testing actor programs consist of the following:

- (1) **Schedule specification.** These frameworks provide a language and APIs for programmers to specify the schedule of execution via specifying the delivery order of (some) messages. The run-time schedulers in Setac/Setak enforce the ordering constraints specified in the schedule during test execution. This feature is useful in reproducing a bug or verifying the absence of a fixed bug.
- (2) **Checking actor-related assertions.** These frameworks provide various APIs for checking status of actors and messages at different execution points. This enables new types of assertions to be succinctly encoded.
- (3) **Checking assertions in a stable state.** These frameworks make it easy to check assertions in the stable state, which is usually the appropriate time for checking assertions. Without these frameworks, checking that the system is in a stable state is rather challenging.

Setac was accepted at Second Scala Workshop-Scala Days 2011 [23]. The author of this dissertation presented the work at the workshop. Both frameworks are publicly available online at: <http://web.engr.illinois.edu/~tasharo1/setac/> and <http://web.engr.illinois.edu/~tasharo1/setak/>.

Since Setac/Setak limit the state-space exploration to the schedules specified by the programmers, they solve the scalability problems of POR techniques. Nevertheless, our experiences with the users of these frameworks reveal that specifying bug exposing schedules might not be an easy task for the programmers. To address this deficiency, next section explains another technique that automatically generates schedules and runs the program with those schedules.

1.2.3 Bitā

Bitā² is a coverage-guided technique that tries to address major problems with Setac/Setak and TransDPOR. On one hand, it takes the responsibility of specifying schedules away from program-

²Bitā in Persian means “unique”.

mers by automatically generating schedules; and on the other hand, it keeps the number of generated schedules (and explored interleavings) at most quadratic in the number of exchanged messages.

To identify and generate bug exposing schedules, Bitá exploits three *interleaving coverage criteria*. These criteria are inspired from common concurrency bug patterns, and they are defined in a way that the cost of satisfying them is at most quadratic in the number of messages. Interleaving coverage describes the extent to which a set of possible interleavings has already been explored. We present three interleaving coverage criteria for actor programs. Bitá exploits these criteria to test a program in three main steps. First, it runs the program to obtain an arbitrary execution path. Second, it uses the execution path to generate schedules that increase the coverage. Finally, it runs the program with each generated schedule.

For testing shared-memory programs (and not actor programs), several interleaving coverage criteria have been proposed [24–27] and leveraged for exploring interleavings [28, 29]. To the best of our knowledge, no prior work offered a scalable, automatic technique for testing actor programs based on interleaving coverage criteria.

In Bitá, we take advantage of interleaving coverage criteria to propose a practical technique for automatic testing of actor programs. Specifically, we make the following contributions:

- (1) **Interleaving coverage criteria for actor programs.** We present three interleaving coverage criteria for actor programs that address common bug patterns and a technique for measuring the coverage achieved by a set of execution paths. As mentioned before, the cost of satisfying these criteria is at most quadratic in the number of exchanged messages.
- (2) **Coverage-guided schedule generation.** We present a coverage-guided approach for automatically generating schedules based on an initial execution of a program. Generated schedules can be stored and served as a part of test cases 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.
- (3) **Proof of schedules feasibility.** We prove that the schedule generator guarantees that each generated schedule is feasible. Previous work for shared-memory programs [28, 29] may generate infeasible schedules, which reduces the efficiency of testing process. Instead, our approach

creates feasible schedules by considering must-happen-before relation between receive events. Moreover, we show that each schedule contributes to higher coverage and contains enough details so that the run-time scheduler succeeds in realizing the schedule during execution.

- (4) **Implementation and evaluation with real-world programs.** We implement Bitra for Akka and apply it to five real-world programs and three smaller benchmarks. Bitra 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, Bitra finds bugs substantially faster (122x on average). A prototype of Bitra is publicly available at:

<http://bitra.cs.illinois.edu/>

Bitra has been accepted and will be published in the form of a conference paper at the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13) [30].

1.3 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 describes more details of the actor model and the actor libraries we used in our studies. Chapter 3 presents TransDPOR algorithm along with its implementation and evaluation. The two testing frameworks, Setac and Setak, are presented in Chapter 4. Chapter 5 describes the Bitra technique, its implementation, and its evaluation. Previous related studies and their comparison with the solutions proposed in this dissertation are explained in Chapter 6. Chapter 7 contains concluding remarks and future work.

Chapter 2

Background

This chapter presents an overview of the actor model and how non-determinism can happen in the actor systems. It also gives the description of the actor libraries in the Scala language [22].

2.1 The Actor Model and Non-determinism

The actor model [1, 2] is a model of concurrency in which concurrent entities are actors. Actors communicate by exchanging messages asynchronously. Each actor has a mailbox, which is a place holder for *pending*—delivered but not yet processed—messages, a local state, and a thread of control.

We differentiate between message delivery and processing in that message delivery means placing the message in the mailbox and message processing means removing a message from the mailbox and acting in response to that. Note that in the actor semantics, there is no guarantee that the messages are processed in the order that they are delivered. Depending on the implementation of the mailboxes in the actor systems, the order of messages in the actor mailbox might be the same as or different from the order of their delivery.

In systems with strict actor semantics, each actor is perfectly encapsulated, that is, there is no state shared between actors. The strict semantics furthermore mandate (1) *location transparency*: that actors know each other by opaque addresses that uniquely identify an actor in the system; (2) *address safety*: actors only know the address of other actors they either created, or have been introduced to; and (3) *fairness*: that every message sent is eventually received.

The strict actor semantics form a powerful basis for the theoretical treatment of programs. Fairness can be used, for example, for proving deadlock-freedom; address safety facilitates reasoning about security properties.

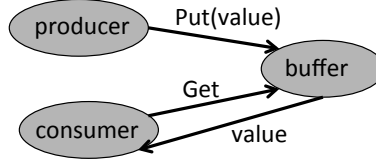


Figure 2.1: Communication of actors in bounded buffer program that contains three actors: buffer, producer, and consumer.

Each step of computation is a *receive* event in which an actor removes a message from its mailbox and processes that message. The message processing is performed in an atomic step [31] and without interruption. Each actor has a set of message handlers that determine what kind of messages can be processed by the actor and which actions should be performed when processing a message. In response to processing a message, an actor may send more messages, create more actors, or change its local state including its message handlers.

2.1.1 Non-determinism by Example

Despite the lack of shared state, testing actor systems is difficult because even for a given input, the order in which actors receive messages is non-deterministic. Hence, the system may exhibit different behaviors depending on the order of message receives.

For example, consider a program which contains three actors: bounded buffer, producer, and consumer. As shown in Figure 2.1, the producer adds a value into the buffer content via a `Put` message and the consumer reads a value from the buffer via a `Get` message. When the buffer receives a `Get` message, it removes a value from its content and sends it in a message to the consumer. Initially, the buffer is empty. If the buffer receives the `Put` message before or after the `Get` message, the consumer may receive different values. For example, there might be a bug in the program that causes invalid value is sent to the consumer when an empty buffer receives a `Get` message.

While the non-determinism in the actor systems helps in modeling real-world concurrency, it makes testing them challenging. The testing techniques not only require to test an actor program with various inputs but also with different interleavings of receive events for each given input.

2.1.2 Constraints in Actor Systems

Although the actor model is established upon asynchronous messaging, it is possible to build other synchronization patterns by composing multiple steps of asynchronous communications [2]. Synchronization patterns impose some constraints on the order of receive events and hence, they may limit the non-determinism in the actor systems. The two common synchronization patterns used in the actor systems are *synchronous communications (remote procedure call-like messaging)* and *local synchronization constraints* [32]. Some actor systems provide primitive constructs for these patterns to facilitate actor programming.

In synchronous communication, the sender actor is blocked (does not process any other messages) 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*. Local synchronization constraints enable or disable a receive event in an actor depending on the actor's local state. Specifically, these constraints depending on the local state of an actor, determine whether a message is allowed to be removed from the mailbox and processed. As a result, local synchronization constraints can be used for changing actor message handlers.

Besides synchronization patterns, the infrastructure of some actor systems may also put some constraints on the order of receive events. One of the popular examples for such constraints is *sender-receiver* constraint. The execution infrastructure of an actor system with sender-receiver constraint guarantees that for each pair of sender and receiver actors, the messages are never delivered out-of-order. For example, if actor A sends messages M_1 and M_2 to actor B , message M_2 is never delivered before message M_1 to actor B .

2.2 Actor Libraries

The growing demand for the actor model has led to the emergent of many actor libraries and languages such as Actor Foundry [3], Akka [4], AmbientTalk [5], Charm++ [6], Erlang [7], GParS [8], Orleans [9], Rebeca [10], and Scala actor [11]. This section describes the two popular actor libraries in the Scala language which we use in our experiments.

Scala [22] is an object-functional language that runs on top of JVM. The two well-known actor

```

1 class BoundedBuffer(size: Int) extends Actor {
2
3   var content = new Array[Int](size)
4   var head, tail, curSize = 0
5   start
6
7   override def act() {
8     loop {
9       react { // called when there is a message in the mail box
10        case Put(x) if (curSize < size) => { // if the message is Put(x) and curSize < size
11          content(tail) = x
12          tail = (tail + 1) % size
13          curSize += 1
14        }
15        case Get => { // if the message is Get
16          if (curSize > 0) {
17            val value = content(head)
18            head = (head + 1) % size
19            curSize -= 1
20            reply(value)
21          } else reply(-1)
22        }
23      }
24    }
25  }
26 }

```

Listing 2.1: Implementation of bounded buffer in Scala actor.

libraries implemented for Scala include: (1) the *Scala actor* package which is included in the official release of Scala; and (2) *Akka*. In the rest of this section we explain each library in more details.

2.2.1 Scala Actor

Scala actors provides constructs for various features of the actor model, including dynamically creating and destroying actors, sending asynchronous and synchronous messages, local synchronization constraints, and remote actors. In addition, it provides features for improving the efficiency and quality of program execution such as handling exceptions, garbage collecting actors, and some customization of the thread pool executing actors. The mailboxes in Scala actor sorts messages in FIFO order.

The code in Listing 2.1 shows an implementation of bounded buffer actor explained in Section 2.1.1 in Scala actor v2.9 . Each actor extends the `Actor` trait from the library. When the actor is created, the `start` method should be called on the actor so that it can start its execution. This method can either be called in the constructor (as shown in Line 5) or from an object that creates the actor.

The message handlers of an actor—which can be changed dynamically—are specified by overriding the `act` method and they are implemented in a `react` or `receive` method. In this example, `react` is used¹. These methods are basically partial functions specified by a sequence of `case` statements. Each `case` statement acts as a message handler. Scala actor implements the local synchronization constraints via the patterns of `case` statements. That is, the patterns of `case` statements determine which receive event is enabled. At each step of computation, a message which is matched with the pattern of at least one `case` statement is removed from the mailbox and the body of the matched `case` statement will be executed. If a message does not match with any of the `case` patterns, it will stay in the mailbox, i.e., its receive event remains disabled.

In Listing 2.1, the buffer message handlers state that the buffer accepts two kinds of messages: `Put(x)` and `Get`. The receive of a `Put` message is enabled (it can be removed from the mailbox) if the size of the buffer is less than the maximum size. The maximum size is determined via variable `size` in the constructor. Therefore, condition in Line 10 acts as a local synchronization constraint. In response to processing a `Put` message (Line 10), it adds `x` into its content. In response to processing a `Get` message (Line 15), if the buffer is not empty, then it removes a value from the head of its content and sends the value to the sender of `Get` message (Line 20); otherwise, it sends `-1` to the sender (Line 21).

2.2.2 Akka

Building upon experience from Scala actors, the Akka library [4] supplies another implementation of actors for Scala. Besides offering better performance, it adds automatic load-balancing, improves the Erlang-style [7] resilience and fault-tolerance, and introduces opaque actor references for better encapsulation.

Akka does not guarantee message delivery and indeed, does not provide fairness of the actor semantics. Unlike Scala actor, Akka infrastructure guarantees sender-receiver constraint would be preserved in the execution. The mailboxes by default sort delivered messages in FIFO order but they can be customized to sort messages with different ordering policies. Another major difference is that Akka does not implement local synchronization constraints. At each step of computation,

¹Refer to [11] for more details about `react` and `receive`

```

1 class BoundedBuffer(size: Int) extends Actor {
2
3   var content = new Array[Int](size)
4   var head, tail, curSize = 0
5
6   def receive = { // called when a message is removed from the mail box
7     case Put(x) if (curSize < size) => { // if the message is Put(x) and curSize < size
8       content(tail) = x
9       tail = (tail + 1) % size
10      curSize += 1
11    }
12    case Get => { // if the message is Get
13      if (curSize > 0) {
14        val value = content(head)
15        head = (head + 1) % size
16        curSize -= 1
17        sender ! value
18      } else sender ! -1
19    }
20  }
21 }

```

Listing 2.2: Implementation of bounded buffer in Akka.

a message is removed from the mailbox, that is, its receive event is always enabled. If the message does not match with any of the `case` statements patterns, it will be discarded either silently (in Akka 2.x) or with exception (in Akka 1.x). Akka provides APIs for changing the actor message handlers at runtime.

The code in Listing 2.2 shows the bounded buffer actor implemented in Akka 2.0. The code is very similar to Listing 2.1. However, the actor message handlers can be specified in a simpler way and via a `receive` method which is a partial function. Moreover, there is no explicit `start` because the actor will be started automatically after creation. In this code, if there is a `Put` message in the mailbox and the buffer is full, then the message will be removed from the mailbox but the body of none of the case statements will be executed.

Note that exploiting the benefits of the actor model for programming does not require a language that enforces *strict* actor semantics; it is sufficient to have a library providing asynchronous messaging between concurrent objects, and to adhere to coding conventions for avoiding shared state. Besides all the features that Scala actor and Akka provide for actor programming, both Scala actor and Akka libraries do not enforce the programmers to adhere to the basics of the actor model.

As our study in [33] shows, many of Scala programmers mix the actor model with other concurrency models such as thread-based, shared-memory model. This problem increases the chance of

introducing data races, and more importantly, causes most of the testing tools specialized for actor systems lose their efficiency and their bug detection capability for these programs.

However, as the study suggests, the problem is mostly due to deficiencies of the actor libraries and improving those libraries may reduce the possibility of mixing models. The trend of improvement in Akka empowers the hope for that.

Chapter 3

TransDPOR: A Dynamic Partial Order Reduction Technique

One of the solutions for testing non-deterministic behavior of an actor system would be exploring all interleavings. A naïve exploration that explores all the interleavings to reach all possible states does not scale. Partial-order reduction (POR) techniques [12,34–44] can be applied to help mitigate the resulting state-space explosion by exploring a representative subset of all possible interleavings. POR techniques have been widely used for testing and verification of concurrent protocols and software, including in tools such as SPIN [45], VeriSoft [35], and Java PathFinder [46].

One of the popular POR techniques, called dynamic POR (DPOR) [12], has shown to be effective in reducing state-space exploration when applying to the actor systems [13]. We observe that DPOR can be improved if some properties of actor systems are taken into account. This chapter presents our new dynamic POR technique, called TransDPOR, that leverages some important properties of the actor systems and improves over DPOR.

This chapter starts by an example to illustrate the motivation of partial order reduction in actor systems (Section 3.1). A brief overview of POR techniques is provided in Section 3.3. Section 3.4 presents a well-known dynamic POR technique—called DPOR—which was originally proposed for shared-memory programs but we adapted for actor systems. Section 3.5 presents a new dynamic POR technique—called TransDPOR—which improves over DPOR. The evaluation of TransDPOR and its comparison with DPOR are presented in Section 3.6. Finally, Section 3.7 discusses about advantages and limitations of TransDPOR.

3.1 Motivating Example

Figure 3.1 shows code snippets of registry program written in Scala actor library. The program contains four actors: a master, a registry, and two workers. The master creates a registry and

```

1 master:
2   ...
3   var registry = ... // create an instance of Registry
4   registry ! Register("master") /*r0*/
5   var worker1 = ... // create an instance of Worker
6   worker1 ! RegisterSelf(registry) /*w1*/
7   var worker2 = ... // create an instance of Worker
8   worker2 ! RegisterSelf(registry) /*w2*/

1 registry:
2   ...
3   def act() = loop {
4     react {
5       case Register(id: String) => {
6         // add id to the registered list
7       }
8     }
9   }

1 worker1:
2   ...
3   var id = "worker1"
4   def act() = loop {
5     react {
6       case RegistrSelf(registry: Registry) => {
7         registry ! Register(id) /*r1*/
8       }
9     }
10  }

1 worker2:
2   ...
3   var id = "worker2"
4   def act() = loop {
5     react {
6       case RegistrSelf(registry: Registry) => {
7         registry ! Register(id) /*r2*/
8       }
9     }
10  }

```

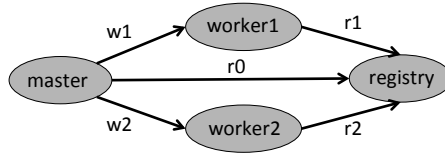


Figure 3.1: The registry example and the messages exchanged.

sends its ID to the registry via a `Register` message. Then, it creates two workers and sends them the reference of the registry actor as the argument of `RegisterSelf` message. When the workers receive the registry reference, they send their IDs to the registry via `Register` messages to register themselves.

In the comments for send statements (the statements that contain `!`), each of the five messages are labeled: `worker1` and `worker2` receive messages `w1` and `w2` respectively, and the `registry` eventually receives three messages—`r0`, `r1`, and `r2`. These three messages can be received in any order and the program could have a bug if it assumes that `r0` is received before `r1` and `r2`.

In this program, as shown in the bottom of Figure 3.1, five messages are exchanged and without any assumption it would have up to $5!$ interleavings of receive events.

However, many of these interleavings are equivalent, i.e., lead to the same configuration. For example, since actors do not share states and only communicate by exchanging messages, processing a message in one actor does not affect the state of other actors, i.e., only the order of receives in each

actor matters. This property implies that for example the two execution paths r_0, w_1, w_2, r_1, r_2 and r_0, w_2, w_1, r_1, r_2 lead to the same configuration and it is unnecessary for the state-space exploration technique to explore both of them.

The goal of partial order reduction techniques is to make the exploration more efficient through removing such redundant interleavings.

3.2 The Actor Model Definitions

This section presents some definitions of the actor model which will be used in the POR algorithms explained in the next sections.

Actor systems can be modeled as state-transition systems. The (global) state of an actor system is called *configuration* which consists of the local states of the actors and a set of pending message:

Definition 1 (Configuration). *A configuration of an actor system, $\kappa = \langle \alpha, \mu \rangle$, consists of a map $\alpha : \mathbb{A} \rightarrow \mathbb{L}$, where $\mathbb{A}$ are actor identifiers and $\mathbb{L}$ are possible local states, and a set of pending messages $\mu \subseteq \mathbb{M}$, where $\mathbb{M}$ is the set of all possible messages in the system.*

We use $\mathbb{K}$ to denote the set of all configurations in a system and $pending(\kappa)$ to denote the set of pending messages for $\kappa \in \mathbb{K}$. Each message is a tuple of *receiver actor*, *content*, and *unique message identifier*. Conceptually, the messages in μ can be partitioned according to their receiver actor, i.e., μ is a union of disjoint message sets, one for every actor in the system.

The transitions of an actor system are receive events. At each step in the computation, an actor from the system is scheduled to process one of its pending messages. Assuming that this processing terminates, the actor system transitions to a new configuration. The actor model allows *constraints* that enable or disable processing of some message by an actor. Formally, for an actor a , its constraint $c_a \subseteq \mathbb{L} \times \mathbb{M}$ is a predicate on the local state of the actor and the set of messages. Note that this constraint can be used in modeling various constraints such as synchronous communication constraints and local synchronization constraints.

Definition 2 (Receive Event). *The receive event r_m for a message m is a partial function $r_m : \mathbb{K} \rightarrow \mathbb{K}$. For a given $\langle \alpha, \mu \rangle \in \mathbb{K}$, let the receiver of m be actor a with the local state s , $\alpha(a) = s$, and constraint c_a ; the receive event r_m is enabled if $m \in \mu$, and $\langle s, m \rangle \in c_a$. If r_m is enabled, it*

can be executed and produces a new configuration, updating the local state of the actor from s to s' , sending messages $\text{sent}_s(r_m)$, and creating new actors with their initial local state $\text{create}_s(r_m)$:

$$\langle \alpha, \mu \rangle \xrightarrow{r_m} \langle \alpha[a \mapsto s'] \cup \text{create}_s(r_m), \mu \setminus \{m\} \cup \text{sent}_s(r_m) \rangle$$

We identify each receive event r_m by the message m it processes. Thus, due to uniqueness of messages, each receive event has also a unique identifier. We denote $\text{msg}(r_m) = m$ and $\text{rec}(r_m) = a$. Furthermore, $\text{sent}(r_m)$ and $\text{create}(r_m)$ stand for the sets of all new messages and actors, respectively, that the receive event r_m can send and create for *any* local state s .

We assume that the execution of all receives terminate—this is a standard assumption in testing programs. Thus, the execution of a receive r in a configuration κ leads to a unique successor κ' (up to the choice of fresh identifiers for new actors and messages). Moreover, as is usual in actor semantics, we assume that the behavior of an actor in response to a message is *deterministic*.

Definition 2 also shows the inherit non-determinism in the actor systems. In each configuration, more than one receive event might be enabled and if that happens, one of them will be executed non-deterministically.

Let τ be the set of all receives in an actor system. By using the above definitions, the state-space of the systems A_G can be defined as a transition system:

$A_G = \langle \mathbb{K}, \Delta, \kappa_0 \rangle$, where $\Delta = \{ \langle \kappa, \kappa' \rangle \mid \exists r \in \tau : \kappa \xrightarrow{r} \kappa' \}$ and κ_0 is the initial configuration. We use A_R to denote the reduced state-space explored by POR techniques.

Proposition 1. *Since each message and each receive event in $\mathbb{K}$ has a unique identifier, there is no cycle in A_G .*

The execution of an actor system can be presented as a sequence of receive events:

Definition 3 (Receive Sequence). *A receive sequence R of an actor system is a (finite) sequence of receives $r_1, r_2, \dots, r_n$ where there exist configurations $\kappa_0, \dots, \kappa_n$ such that κ_0 is the initial configuration and $\kappa_0 \xrightarrow{r_1} \kappa_1 \xrightarrow{r_2} \dots \xrightarrow{r_n} \kappa_n$.*

Let τ^* be the set of all receive sequences. We write $\kappa \xRightarrow{\omega} \kappa'$ to denote that the execution of finite sequence $\omega \in \tau^*$ leads from κ to κ' .

An execution path of an actor system is a receive sequence which ends in a configuration with no enabled receive event.

Definition 4 (Execution Path). *A configuration in which no receive is enabled is called a deadlock or terminating configuration and a receive sequence that ends in a deadlock or terminating configuration is called an execution path of the system.*

3.3 Partial-Order Reduction (POR)

Partial-Order reduction (POR) techniques envision concurrent programs as state-transition systems. They remove redundant interleavings from exploration by identifying equivalent interleavings. Equivalent interleavings are interleavings that lead to the same state with respect to some properties. To identify equivalent interleavings, POR techniques exploit the dependency relation among concurrent transitions. The basic idea is that in a given receive sequence, swapping two adjacent receive events which are not in dependency relation results in an equivalent receive sequence. For example, in one of the well-known POR techniques, called persistent sets [34], in each state, the algorithm selects a subset of enabled transitions which are in dependency relation and explores those transitions.

The following two definitions recap the general definition of a valid dependency relation [12]:

Definition 5. *Let r_1 and r_2 be two receives of an actor system. We say that r_1 and r_2 are independent if for all configurations κ in the state space A_G of the system:*

- *if r_1 is enabled in κ and $\kappa \xrightarrow{r_1} \kappa'$, then r_2 is enabled in κ iff r_2 is enabled in κ' (i.e., independent receives cannot disable or enable each other); and*
- *if r_1 and r_2 are enabled in κ , there is a unique configuration κ' such that $\kappa \xrightarrow{r_1, r_2} \kappa'$ and $\kappa \xrightarrow{r_2, r_1} \kappa'$ (i.e., enabled independent transitions must commute).*

The independence relation is used to define the dependency relation:

Definition 6. *The reflexive and symmetric binary relation D is a valid dependency relation on τ iff $D = \{(r_1, r_2) | r_1, r_2 \text{ are not independent receive events}\}$. The pair of receive events (r_1, r_2) are said to be dependent iff they belong to a valid dependency relation.*

Note that a valid dependency relation is a *reflexive* and *symmetric* but *not necessarily transitive* binary relation on the receive events.

The dependency relation can be computed using static or dynamic analysis. Traditionally, dependencies among transitions, such as in persistent sets [34] proposed by Godefroid, were computed via static analysis. More recently, Flanagan and Godefroid introduced a POR algorithm, called dynamic POR (DPOR), that relies on dynamic analysis for computing dependencies [12]. The persistent set in this algorithm—which is called the *backtrack set*—is computed at run time. Flanagan and Godefroid show that DPOR can significantly improve on POR techniques that compute the persistent set based on static analysis. DPOR was originally proposed for multi-threaded shared-memory programs. Next section presents the adaptation of DPOR for actor systems.

3.4 DPOR for Actor Systems

Applying DPOR for actor systems requires defining a valid dependency relation between the receive events. This section starts with defining a valid dependency relation for actor systems and then, it presents the adaptation of DPOR for actor systems.

3.4.1 Dependency Relation

We observe that for actor systems, a receive r_m cannot be enabled until the receiver actor for m is created, and message m is sent. Moreover, once a message m is sent to an actor a , the only way r_m (which processes m) gets enabled or disabled is when actor a receives another message. In other words, the constraint c_a does not depend on the global state but only on the local state of a and the message m . Therefore, we can easily show that two receives r_1 and r_2 are independent iff (1) $rec(r_1) \notin create(r_2)$; (2) $msg(r_1) \notin sent(r_2)$; and (3) $rec(r_1) \neq rec(r_2)$.

Based on these observations, we can cast Definition 6 in the actor systems setting to obtain the following proposition.

Proposition 2 (Dependent Receives). *Two receives $r_1, r_2 \in \tau$ are dependent iff one of the following conditions holds:*

- $rec(r_1) \in create(r_2)$ or $rec(r_2) \in create(r_1)$; or

- $msg(r_1) \in sent(r_2)$ or $msg(r_2) \in sent(r_1)$; or
- $rec(r_1) = rec(r_2)$.

Based on Proposition 2, one can extract an important property of dependency relation among certain receive events which will be used in TransDPOR to improve over DPOR. Before giving the details, we define the race relation among receive events.

Definition 7 (May Be Co-enabled Receives). *Two receives $r_1, r_2 \in \tau$ may be co-enabled if there may exist some configuration κ in which both r_1 and r_2 are enabled.*

Definition 8 (Race Relation). *Two receives $r_1, r_2 \in \tau$ are in the race relation iff they are dependent and may be co-enabled.*

A key observation is that if (r_1, r_2) are in a race, then $rec(r_1) = rec(r_2)$. Indeed, while our definition of dependency allows two other cases— $msg(r_1) \in sent(r_2)$ or $rec(r_1) \in create(r_2)$ —the receive events that satisfy those two other cases can never be co-enabled (because those cases require that the message or actor for r_1 be created after the execution of r_2). As a result, the following proposition holds.

Proposition 3. *The race relation is reflexive, symmetric, and transitive.*

3.4.2 Equivalent Receive Sequences

Defining the equivalent receive sequences helps POR techniques to remove redundant receive sequences from exploration. To formalize the set of equivalent receive sequences, we need to define the happens-before-relation among receives. The following definition recaps the happens-before relation presented by Flanagan and Godefroid [12]:

Definition 9 (Happens-before Relation). *The happens-before relation $\xrightarrow{hb}_R$ for a receive sequence $R = r_1, \dots, r_n$ is the smallest relation on $\{1, \dots, n\}$ such that (1) if $i \leq j$ and r_i is dependent with r_j , then $i \xrightarrow{hb}_R j$; and (2) $\xrightarrow{hb}_R$ is transitively closed.*

The happens-before relation—which is a partial order—can be used for defining equivalent receive sequences [12]:

Definition 10 (Equivalent Receive Sequences). *Two receive sequences R_1 and R_2 are equivalent iff they have the same set of receives, and they are linearizations of the same happens-before relation.*

We use $[R]$ to denote the set of receive sequences that are equivalent to R .

3.4.3 DPOR Algorithm

Algorithm 1 shows the DPOR algorithm adapted for the actor systems. The input to the algorithm is a receive sequence R which is initially an empty sequence—the initial call is *Explore*($\emptyset$). DPOR maintains a backtrack set in each configuration κ —*backtrack*(κ)—which contains the set of messages to be explored from that configuration in R .

For a receive event sequence $R = r_1 \dots r_n$ the following notations are used:

- $dom(R)$ is the set $\{1, \dots, n\}$;
- R_i for $i \in dom(R)$ is the receive r_i ;
- $pre(R, i)$ for $i \in dom(R)$ is the configuration in which r_i is executed; and
- $last(R)$ is the configuration reached after executing R .

Moreover, $next(\kappa, m)$ denotes the receive event that processes message m in configuration κ .

In this algorithm, it is required to determine if some messages are sent as the result of executing other receive events. Indeed, we define the *message casual relation* for this purpose:

Definition 11 (Message Casual Relation). *In a receive sequence R , the message casual relation $i \xrightarrow{mca}_R m$ holds for $i \in dom(R)$ and message m iff:*

- $m \in sent(R_i)$; or
- $\exists j \in dom(R)$ such that $i \xrightarrow{hb}_R j$ and $m \in sent(R_j)$.

DPOR starts by finding the current configuration κ for the input sequence R (Line 1). Therefore, in the first call in which $R = \emptyset$, κ is the initial configuration.

Lines 2–11 update the backtrack sets of the configurations seen before in R . For every message m in $pending(\kappa)$ (Line 2), it considers the receive event $next(\kappa, m)$ that processes message m . It

Algorithm 1 *Explore(R)*: The DPOR algorithm adapted for actor systems.

```

1:  $\kappa \leftarrow \text{last}(R)$ 
2: for all messages  $m \in \text{pending}(\kappa)$  do
3:   if  $\exists i = \max(\{i \in \text{dom}(R) \mid R_i \text{ is dependent and may be co-enabled with } \text{next}(\kappa, m)\})$  then
4:      $E \leftarrow \{m' \in \text{enabled}(\text{pre}(R, i)) \mid m' = m \text{ or } \exists j \in \text{dom}(R) \mid j > i \text{ and } m' = \text{msg}(R_j) \text{ and } j \xrightarrow{\text{mca}}_R m\}$ 
5:   end if
6:   if  $E \neq \emptyset$  then
7:     add any  $m' \in E$  to  $\text{backtrack}(\text{pre}(R, i))$ 
8:   else
9:     add all  $m$  in  $\text{enabled}(\text{pre}(R, i))$  to  $\text{backtrack}(\text{pre}(R, i))$ 
10:  end if
11: end for
12: if  $\exists m \in \text{enabled}(\kappa)$  then
13:    $\text{backtrack}(\kappa) \leftarrow \{m\}$ 
14:    $\text{done} \leftarrow \emptyset$ 
15:   while  $\exists m \in (\text{backtrack}(\kappa) \setminus \text{done})$  do
16:     add  $m$  to  $\text{done}$ 
17:      $\text{Explore}(R.\text{next}(\kappa, m))$ 
18:   end while
19: end if

```

finds the *last* receive event i in the sequence R which is in the race relation with $\text{next}(\kappa, m)$, i.e., $\text{rec}(R_i) = \text{rec}(\text{next}(\kappa, m))$ and R_i may be co-enabled with $\text{next}(\kappa, m)$ (Line 3).

Next, it finds the messages that should be added to $\text{backtrack}(\text{pre}(R, i))$ by computing the set E (Line 4). If m is enabled in $\text{pre}(R, i)$ it is added to E ($m' = m$); otherwise any message m' is added to E if its transition is the transition after R_i that causes m . If E is not empty, it adds any message in E to $\text{backtrack}(\text{pre}(R, i))$; otherwise, it adds *all* messages from E to $\text{backtrack}(\text{pre}(R, i))$.

Lines 12–19 process the messages from the current configuration κ . The algorithm non-deterministically chooses an enabled message from κ (Line 12) to initialize $\text{backtrack}(\kappa)$ (Line 13). It then processes all messages from the backtrack set that have not been explored before (Line 15).

Every time the algorithm backtracks to κ and explores a new message, it adds that message to the *done* set (Line 16). The algorithm finally recursively calls itself with the transition sequence extended with the $\text{next}(\kappa, m)$ (Line 17). When the **while** loop terminates, the exploration from κ is finished, and the algorithm backtracks to the previous configuration.

We illustrate this algorithm with the example in Figure 3.1. The state-space exploration is shown in Figure 3.2. Each node in this figure represents a configuration, and each edge shows a receive

event labeled with the message being processed. For the sake of simplicity, we use the ID of each message for the receive event that processes that message.

Initially, R is empty and the initial configuration κ_0 would be the configuration reached after executing the entry point (master). Hence, the set of pending messages would be $\{r_0, w_1, w_2\}$.

Since in this call R is empty, Lines 2–11 do not update the backtrack set of any configuration.

In Lines 12–19 the algorithm processes r_0 , adds it to the *done* set, and calls $Explore(r_0)$. In this call, R is not empty and the algorithm checks if any of the pending messages— w_1 and w_2 —is in the race with r_0 (Line 3). Since none is in the race with r_0 , it does not update the backtrack set of any configuration.

Next, w_1 is processed (in Line 12) which causes message r_1 to be sent by *worker1*. Indeed, in the call for $Explore(r_0, w_1)$ the set of pending messages in $\kappa_2 = last(r_0, w_1)$ would be $\{w_2, r_1\}$. In this call, there is a message— r_1 —in the set of pending messages which is in the race with r_0 . Since it r_1 is not enabled in κ_0 and $2 \xrightarrow{mca}_R r_1$, it adds w_1 to E and consequently to $backtrack(\kappa_0)$.

After updating $backtrack(\kappa_0)$, it selects and executes one of the enabled messages, w_2 , and calls $Explore(r_0, w_1, w_2)$. In this call, the set of pending messages in $\kappa_3 = last(r_0, w_1, w_2)$ is $\{r_1, r_2\}$. The algorithm checks if any of them is in the race with w_2 , w_1 , or r_0 . Since r_2 is in the race with r_0 , it is not enabled in κ_0 , and $3 \xrightarrow{mca}_R r_2$, it adds w_2 to the $backtrack(\kappa_0)$.

Next, the algorithm executes r_1 and calls $Explore(r_0, w_1, w_2, r_1)$. In this call, r_2 is the only pending and enabled message. Since r_2 is in the race relation with r_1 and it is enabled in κ_3 , the algorithm adds r_2 to $backtrack(\kappa_3)$. Then, it executes r_2 .

After executing r_2 , one execution path of the program is completed. The algorithm backtracks to the previous configurations and executes the unexplored messages in the backtrack sets of those configurations. In other words, it backtracks to κ_4 which does not have any unexplored message in its backtrack set. Then, it backtracks to κ_3 , executes r_2 , and continues until another execution path is completed. Similarly, it backtracks to κ_0 and executes w_1 and w_2 from that configuration.

The algorithm finishes when there is no message in the backtrack sets of the configurations which is not explored from those configurations. In the end, as shown in Figure 3.2, DPOR explores 24 paths of the state-space.

Note that the algorithm is stateless, i.e., it does not store states/configurations across different

<pre> master: var registry =... //create registry registry ! Register("master")/*r0*/ worker1 =...//create worker1 worker1 ! RegisterSelf(registry)/*w1*/ worker2 =...//create worker2 worker2 ! RegisterSelf(registry)/*w2*/ </pre>	<pre> registry: ... react { case Register(id)⇒ //register id ... } </pre>	<pre> worker1: ... react { case RegisterSelf(registry)⇒ registry ! Register(id)/*r1*/ } </pre>	<pre> worker2: ... react { case RegisterSelf(registry)⇒ registry ! Register(id)/*r2*/ } </pre>
--	---	--	--

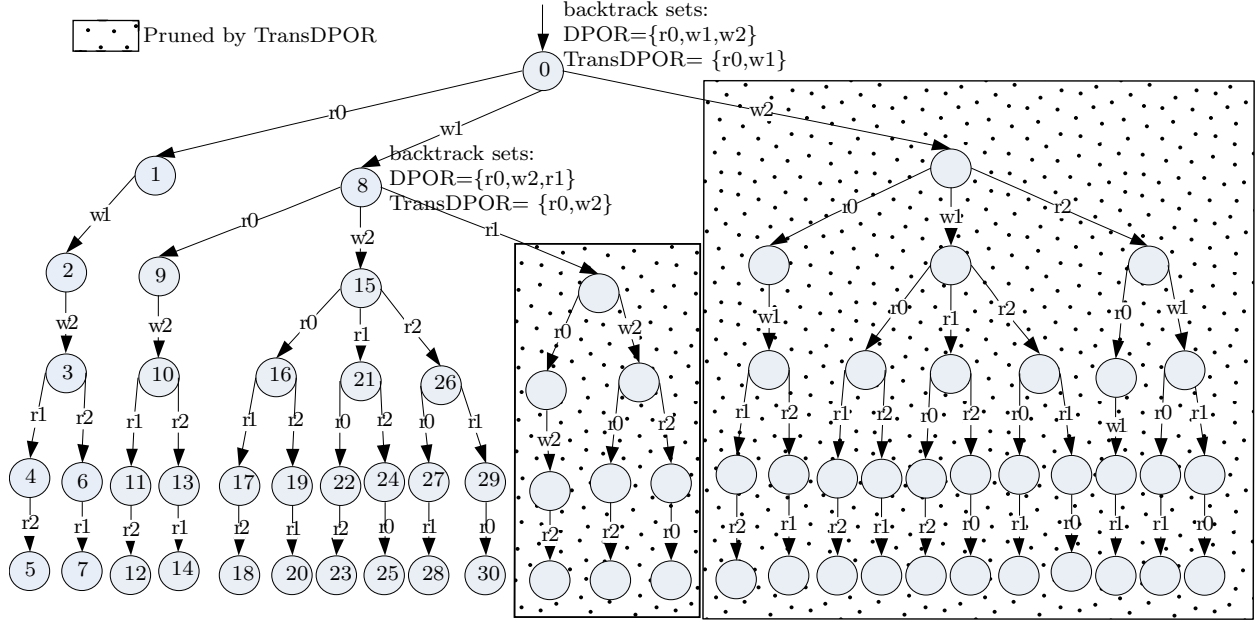


Figure 3.2: State-space explored by DPOR and TransDPOR for the registry example in Figure 3.1.

execution paths. However, it may store configurations for the current path on a stack, depending on the implementation strategy.

While the above pruning improves the full exploration of state-space, it still explores some redundant paths. The reason is that only registry receives more than one message and an efficient exploration should explore permutation of those messages which can be performed in six paths.

Most of these redundant paths can be removed in Figure 3.2 if DPOR does not explore the subtree that starts with w_2 from κ_0 and the subtree that starts with r_1 from κ_8 . When DPOR starts exploring those paths, all the permutations of the three messages in the registry have already been explored in the previous explored paths.

As we explain in Section 3.5, TransDPOR removes these redundant paths by preventing w_2 and r_1 to be added to backtracks of κ_0 and κ_8 respectively. As a result, it only explores 10 paths (those *not* included in dotted boxes in Figure 3.2).

3.5 TransDPOR

This section presents a new algorithm—called TransDPOR—for actor systems exploration that improves on DPOR by creating smaller backtrack set.

Algorithm 2 *Explore(R)*: The TransDPOR algorithm. The underlined parts show the differences between DPOR and TransDPOR.

```

1:  $\kappa \leftarrow last(R)$ 
2: for all messages  $m \in pending(\kappa)$  do
3:   if  $\exists i = max(\{i \in dom(R) \mid R_i \text{ is dependent and may be co-enabled with } next(\kappa, m)\})$  then
4:     if  $\neg freeze(pre(R, i))$  then
5:        $E \leftarrow \{m' \in enabled(pre(R, i)) \mid m' = m \text{ or } \exists j \in dom(R) \mid j > i \text{ and } m' = msg(R_j) \text{ and } j \xrightarrow{mca}_R m \text{ and } j = min(\{j \in dom(R) \mid j > i \text{ and } j \xrightarrow{mca}_R m\})\}$ 
6:     end if
7:   end if
8:   if  $E \setminus backtrack(pre(R, i)) \neq \emptyset$  then
9:     add any  $m' \in E$  to  $backtrack(pre(R, i))$ 
10:     $freeze(pre(R, i)) \leftarrow true$ 
11:   else
12:      $/*add all m in enabled(pre(R, i)) to backtrack(pre(R, i))*/$ 
13:   end if
14: end for
15: if  $\exists m \in enabled(\kappa)$  then
16:    $backtrack(\kappa) \leftarrow \{m\}$ 
17:    $done \leftarrow \emptyset$ 
18:   while  $\exists m \in (backtrack(\kappa) \setminus done)$  do
19:     add  $m$  to  $done$ 
20:      $freeze(\kappa) \leftarrow false$ 
21:      $Explore(R.next(\kappa, m))$ 
22:   end while
23: end if

```

The TransDPOR algorithm is shown in Algorithm 2. Like DPOR, TransDPOR is a dynamic POR technique. The parts which are different from DPOR in Algorithm 1 are underlined.

The key idea in TransDPOR is to add (at most) one new message to the backtrack set of each configuration and add more messages only when the current added message is explored from that configuration. This strategy is implemented via associating a boolean flag, called *freeze* flag, to each configuration which is initially unset. As shown in Line 4, it only updates the backtrack set of a configuration if the *freeze* flag in that configuration is *not* set. This flag is set in Line 10 when a message is added to the backtrack set of a configuration. Whenever TransDPOR backtracks to a configuration and explores a new message from that configuration, it resets the *freeze* flag in

Line 20.

Due to the transitivity of race relation, we prove that the messages not added right away to the backtrack set of a configuration are added later if necessary to explore them from that configuration. However, as TransDPOR does not add them right away, it may terminate faster than DPOR.

Another difference is in computing E (Lines 5 and 12). In Line 5, TransDPOR finds the *minimum* index $j > i$ such that j happens before m , while DPOR finds *all* $j > i$ such that j happens before m . As a result of this change, E in TransDPOR has at most one element. After computing E , if it contains a message that is not already in $\text{backtrack}(\text{pre}(R, i))$, the message is added to $\text{backtrack}(\text{pre}(R, i))$. In contrast, in DPOR, if E is not empty, it can have more than one message, and the algorithm non-deterministically chooses one message to add to $\text{backtrack}(\text{pre}(R, i))$ (hence “add any”). Note that when E is not empty, this change in TransDPOR only gives priority to the element of E which is added to the backtrack set. Hence, it won’t affect the correctness of the algorithm but in some cases it leads to the fewer explored paths.

If E is empty, TransDPOR does not add anything to $\text{backtrack}(\text{pre}(R, i))$ at this point (Line 12 is effectively deleted). In contrast, in DPOR, if E is empty, the algorithm adds *all* messages from E to $\text{backtrack}(\text{pre}(R, i))$. This case happens if m is in $\text{pending}(\text{pre}(R, i))$ but r_m is not enabled in $\text{pre}(R, i)$. Intuitively, because of the transitivity of race relation, every enabled message in $\text{pre}(R, i)$ that can enable r_m would be in the race with R_i (all of them belong to the same actor) and would be added to $\text{backtrack}(\text{pre}(R, i))$ either in the next iteration of the *for* loop or in the recursive calls to *Explore*. Therefore, as we prove in Section 3.5.1 this change in TransDPOR does not affect the correctness of the algorithm.

To give an intuition of these changes, consider the registry example again from the point that the algorithm calls $\text{Explore}(r_0, w_1)$. In this call, the set of pending messages are $\{r_1, w_2\}$ and TransDPOR adds w_1 to $\text{backtrack}(\kappa_0)$ because of the race relation between r_0 and r_1 . Then, it sets $\text{freeze}(\kappa_0)$ which prevents from adding w_2 to $\text{backtrack}(\kappa_0)$ later on upon calling $\text{Explore}(r_0, w_1, w_2)$. After the algorithm explores w_1 from κ_0 , it resets $\text{freeze}(\kappa_0)$, but because none of the messages in paths that start with w_1 from κ_0 is in the race with w_1 , no message is added to the $\text{backtrack}(\kappa_0)$ (w_2 is not added). Similar situation happens in κ_8 , which causes r_1 not to be added to $\text{backtrack}(\kappa_8)$.

As a result, the backtrack sets of κ_0 and κ_8 in TransDPOR are smaller than in DPOR. This

reduction is allowed due to the *transitivity* of the race relation, and we show that this reduction does not miss any bug that DPOR can find.

On the other hand, consider κ_{15} in which all of the three messages r_0 , r_1 , and r_2 are added to the backtrack set of the configuration. In this configuration, after exploring r_0 , both r_1 and r_2 are in the race with r_0 , but TransDPOR only adds r_1 to the backtrack set of κ_{15} . The *freeze* flag prevents the addition of r_2 to the backtrack set of κ_{15} at the same time. Due to the transitivity of the race relation, r_1 and r_2 are also in the race. Thus, after exploring r_1 from κ_{15} , the *freeze* flag is reset and r_2 is added to the backtrack set of κ_{15} . The algorithm will end up with adding all three messages r_0 , r_1 , and r_2 to the backtrack set of κ_{15} and will not miss any permutation of these three messages.

Proving the soundness of TransDPOR is straight forward. Since TransDPOR is a dynamic approach like DPOR, every reported bug is a real bug. In addition, it is trivial to show that TransDPOR never explores more execution paths than DPOR. As a result of the changes in Lines 4 and 12 of TransDPOR, in each call to $Explore(R)$, the algorithm adds either fewer or the same number of messages to the backtrack set of each configuration κ .

We also claim that TransDPOR is complete. Theorem 1 states that by starting from an initial configuration, since A_G is acyclic, TransDPOR will explore at least one execution path from each set of equivalent execution paths in A_G , i.e., it can detect any *deadlock* and *local safety violation* in the program as it is proved by Godefroid [35].

Theorem 1. *In a program $\mathcal{P}$, by starting from an initial configuration, let A_G be the acyclic state-space graph and A_R be the reduced state space explored by Algorithm 2. If Ω_G and Ω_R denote the set of execution paths of $\mathcal{P}$ in A_G and A_R respectively, then $\forall \omega \in \Omega_G, \exists \omega' \in \Omega_R$ such that $\omega' \in [\omega]$.*

An Optimization

We can introduce an optimization that can further reduce the number of explored paths. The idea behind the optimization is that while adding messages to the backtrack set of each configuration, the algorithm prioritizes the messages that belong to a different actor from the actor of the last message executed from that configuration. In this case, a *done* variable is associated to each configuration which contains a list of explored messages from that configuration. This list save the

order of messages executed in the `while` loop of Line 18 from that configuration.

To prioritize messages, if the $freeze(pre(R, i))$ in Line 4 is *true*, then the algorithm continues computing E . If the condition in Line 8 holds— E is not empty and the message in E has not been explored—and the actor of the message in E is different from the actor of the last message in $done(pre(R, i))$, then it removes the unexplored message from the backtrack set and adds the message in E to the backtrack set. Specifically, it removes every message in $backtrack(pre(R, i)) \setminus done$ from $backtrack(pre(R, i))$ and adds E to the backtrack set (Line 9).

3.5.1 Completeness Proof

This section gives the proof of Theorem 1. Before proving Theorem 1, we define some terms and a Lemma. This Lemma relies on *transitivity* of the race relation in actor systems and will be used in the proof of Theorem 1.

Let $Explore(R)$ denote the set of execution paths of the program with prefix R which are explored by TransDPOR. We denote the configuration reached after R as κ_R and the message explored in the last iteration of the `while` loop in Algorithm 2, Line 18 from κ_R as $lmsg(\kappa_R)$. Note that although after executing $lmsg(\kappa_R)$ the *freeze* flag in κ_R is not set, no message is added to $backtrack(\kappa_R)$.

Lemma 1. *For a transition sequence R , let $l = lmsg(\kappa_R)$. If there exists a message m in $pending(\kappa_R)$ such that r_m is enabled in κ_R and $rec(r_m) = rec(r_l)$ then $m \in backtrack(\kappa_R)$.*

Intuitively, if the receiver of the last message which is explored by $Explore(R)$ from κ_R is an actor a , then all enabled messages in $pending(\kappa_R)$ whose receiver is actor a have been explored by the algorithm from κ_R .

Proof of Lemma 1. Since both r_m and r_l are enabled in κ_R , they are in the race relation, i.e., because of the *transitivity of the race relation*, all enabled receive events of an actor in a configuration are in the race relation. Having $m \in pending(\kappa_R)$ implies that $m \in pending(\kappa_{R.r_l})$. Upon calling $Explore(R.r_l)$, $freeze(\kappa_R)$ is not set and r_m is in the race with r_l . Therefore, if m is not in $backtrack(\kappa_R)$ then the algorithm adds m to $backtrack(\kappa_R)$. On the other hand, we know that l is the last message explored from κ_R and no message is added to $backtrack(\kappa_R)$. That means m

already exists in $backtrack(\kappa_R)$ and has been explored before l .

□

Now we exploit Lemma 1 to prove Theorem 1.

Proof of Theorem 1. For each transition sequence R of length k , if the maximum length of the execution paths of the program with prefix R is n , we prove Theorem 1, by induction on the value of $n - k$ (the maximum length of the postfix of the execution paths of the program with prefix R). We refer to this value as $max(R)$.

The base case would be where $max(R) = 0$ which means that R is a an execution path of the program. In the call for $Explore(R)$ the algorithm first updates the backtrack set of the previous configurations in R (Lines 2–14) and then because $enabled(\kappa_R)$ contains no message (Line 15), it returns from the current call and backtracks to the previous configuration.

Assume for each receive sequence R such that $max(R) = n$, $Explore(R)$ has explored at least one path from each equivalent classes of the execution paths with prefix R .

Now, we show that for every receive sequence R such that $max(R) = n + 1$, $Explore(R)$ also explores at least one path from each equivalent classes of execution paths with prefix R .

Suppose there is an execution path, ω , with prefix R that has not been explored by the algorithm: $\omega = R.r.R'$, $max(R) = n + 1$, and $msg(r) \notin backtrack(\kappa_R)$.

We show that either some equivalent execution path from $[\omega]$ is explored by TransDPOR or we have a contradiction. Let $rec(r) = a$ and $lmsg(\kappa_R)$ be l . We consider the following cases for r_l :

- $rec(r_l) = rec(r) = a$: Since r is enabled in κ_R , by Lemma 1 $msg(r)$ should be in $backtrack(\kappa_R)$ and has been explored by TransDPOR. Having $max(R.r) \leq n$ implies that $Explore(R.r)$ has explored at least one execution path from $[\omega]$ or we have a contradiction.
- $rec(r_l) = b, b \neq a$: In this case, since r_l is enabled in κ_R , l is in $pending(\kappa_{R.r})$. We claim that at least one receive event in R' is executed by actor b .

The proof is trivial: since r_l is enabled in κ_R , if no transition of actor b is executed in R' , it will be enabled after R' and can be executed (Note that once a message m is sent to an actor, only the receive events of that actor can enable or disable r_m). This implies that ω

is not an execution path (contradiction). Therefore, R' contains either r_l or other transition from actor b .

Let r_{b_1} be the first transition after r such that $rec(r_{b_1}) = b$ and $\omega = R.r.R_1.r_{b_1}.R_2$. We consider the following cases for r_{b_1} :

- $r_{b_1} = r_l$: Then, ω is equivalent to $\omega' = R.r_l.r.R_1.R_2$ (Definition 10). Moreover, $max(R.r_l) \leq n$ which implies that $Explore(R.r_l)$ has explored at least one execution path from $[\omega']$ (and hence from $[\omega]$) or we have a contradiction.
- $r_{b_1} \neq r_l$ and r_{b_1} is enabled in κ_R : Again, according to Definition 10, ω is equivalent to $\omega' = R.r_{b_1}.r.R_1.R_2$. Since $rec(r_l) = rec(r_{b_1})$ and both r_l and r_{b_1} are enabled in κ_R , according to Lemma 1, b_1 is in $backtrack(\kappa_R)$ and has been explored from κ_R . Having $max(R.r_{b_1}) \leq n$ implies that $Explore(R.r_{b_1})$ has explored at least one path from $[\omega']$ or we have a contradiction.

- $r_{b_1} \neq r_l$ and r_{b_1} is not enabled in κ_R : In this case, since r_{b_1} is enabled in $\kappa_{R.r.R_1}$ and no receive from actor b is executed in $r.R_1$, b_1 is not in pending messages of κ_R . Therefore, there should be some receive r' in $r.R_1$ such that $actor(r') = c$, $c \neq b$ and $r' \xrightarrow{mca}_{\omega} b_1$.

Let the first such receive be r_{c_1} and $r.R_1 = R_3.r_{c_1}.R_4$. Since no receive in R_3 happens before r_{c_1} , ω is equivalent to $\omega' = R.r_{c_1}.R_3.R_4.r_{b_1}.R_2$ (Definition 10).

In this path, no receive in $r_{c_1}.R_3.R_4$ is dependent to r_l (since no receive belongs to actor b) and we know that r_l is enabled after R . This implies that there is an execution path α with prefix $R.r_l$ that contains $r_{c_1}.R_3.R_4$ or its equivalent, e.g., $\alpha = R.r_l.r_{c_1}.R_3.R_4.R_5$.

Having $max(R.r_l) \leq n$ implies α or its equivalent path is explored by $Explore(R.r_l)$.

In this path, upon calling $Explore(R.r_l.r_{c_1}.R_3.R_4)$, b_1 is in the pending messages, $actor(b_1) = actor(r_l)$, and $r_l \xrightarrow{mca}_{R.r_l.r_{c_1}.R_3.R_4} b_1$.

Because r_{b_1} is not enabled in κ_R , the algorithm looks for the *minimum* index j in the sequence $r_{c_1}.R_3.R_4$ such that j have message causal relation with b_1 (Line 5), and adds c_1 to $backtrack(\kappa_R)$. Since nothing is added to $backtrack(\kappa_R)$, c_1 already exists in $backtrack(\kappa_R)$ and has been explored from κ_R . Having $max(R.r_{c_1}) \leq n$ implies that at least one path from $[\omega']$ has been explored by $Explore(R.r_{c_1})$ or we have a contradiction.

Note that if $r_{c_1} = r$ then R_3 is empty and the proof still holds.

□

3.6 Implementation and Evaluation

To compare TransDPOR and DPOR, we implemented TransDPOR in the Basset tool [14]. Basset provides an extensible environment for testing Java-based actor programs written in Scala actor [11] or ActorFoundry [3]. We use vector clocks [47] to track the happens-before relation at runtime as shown in dCute study [42]. TransDPOR code is available with Basset at <http://mir.cs.illinois.edu/basset>

Eight different subject programs are used in our evaluation. Each actor program was either originally implemented in ActorFoundry or ported to it for this evaluation. `fibonacci` computes the n^{th} element in the Fibonacci sequence. In this case, we show the result for $n = 5$. `quicksort` is a distributed sorting implementation using a standard divide-and-conquer strategy to carry out the computation. `pi` is a porting of a publicly available [48] MPI example, which computes an approximation of π by distributing the task among a set of worker actors. The results shown here are for a configuration with five worker actors. `pipesort` is a modified version of the sorting algorithm used in the dCUTE study [42]. `chameneos` is an implementation of the `chameneos-redux` benchmark from the Great Language Shootout (<http://shootout.alioth.debian.org>). `leader` is an implementation of a leader election algorithm previously used in the dCUTE study [42]. `shortpath` is an implementation of the Chandy–Misra shortest path algorithm [49]. This subject appears twice in the results: once for a graph with 4 nodes (`shortpath4`) and once for a graph with 5 nodes (`shortpath5`), where the two graphs are dissimilar. `regsim` is a server registration simulation. The numbers with the name of subjects, if available, represent the values of program parameters. We performed all experiments using Sun’s JVM 1.6.0_20-b02 on a 2.93GHz Intel Core(TM)i7 running Ubuntu release 10.04.

The work in [13] shows that the effectiveness of DPOR techniques is highly sensitive to the *order* in which messages are explored. However, one cannot easily determine before the exploration which order will work the best. For that reason, we present results for three ordering heuristics,

Table 3.1: Comparison of TransDPOR and DPOR

		DPOR				TransDPOR				Speedup				
Heur.	Subject	# of Paths	# of Rec.s	time [sec]	mem [MB]	# of Paths	# of Rec.s	time [sec]	mem [MB]	# of Paths	# of Rec.s	time [sec]	mem [MB]	
FIFO	fib5	40	203	5	176	40	203	4	176	1.00x	1.00x	1.25x	1.00x	
ECA		327	1650	28	455	203	1051	18	377	1.61x	1.57x	1.56x	1.21x	
LCA		16	91	3	173	16	91	3	159	1.00x	1.00x	1.00x	1.09x	
FIFO	quicksort6	368	1586	26	343	368	1586	26	463	1.00x	1.00x	1.00x	0.74x	
ECA		3822	16766	264	381	1519	6992	115	751	2.52x	2.40x	2.30x	0.51x	
LCA		32	156	4	197	32	156	4	179	1.00x	1.00x	1.00x	1.10x	
FIFO	pi5	120	931	16	265	120	931	16	374	1.00x	1.00x	1.00x	0.71x	
ECA		120	931	16	263	120	931	16	374	1.00x	1.00x	1.00x	0.70x	
LCA		19845	156070	2509	451	312	2452	40	376	63.61x	63.65x	62.73x	1.20x	
FIFO	pipesort4	1791	8562	101	375	755	3541	45	375	2.37x	2.42x	2.24x	1.00x	
ECA		288	1293	17	374	288	1293	18	450	1.00x	1.00x	0.94x	0.83x	
LCA		5970	32385	361	375	2221	11999	136	451	2.69x	2.70x	2.65x	0.83x	
FIFO	chameneos2	3240	19459	233	376	600	3673	44	376	5.40x	5.30x	5.30x	1.00x	
ECA		19683	118197	1360	550	1728	10554	123	374	11.39x	11.20x	11.06x	1.47x	
LCA		216	1231	16	375	216	1231	16	453	1.00x	1.00x	1.00x	0.83x	
FIFO	leader4	18098	107780	26872	336	14984	86889	37516	341	1.21x	1.24x	0.72x	0.99x	
ECA		11957	68373	1207	240	11909	68125	1312	266	1.00x	1.00x	0.92x	0.90x	
LCA		39238	236330	4301	634	27287	163030	3120	525	1.44x	1.45x	1.38x	1.21x	
FIFO	shortpath4	238	910	12	261	238	910	12	261	1.00x	1.00x	1.00x	1.00x	
ECA		392	1464	20	262	392	1464	19	260x	1.00x	1.00x	1.05x	1.01x	
LCA		640	2158	27	260	370	1337	17	264	1.73x	1.61x	1.59x	0.98x	
FIFO	shortpath5	528	2443	32	454	528	2443	33	372	1.00x	1.00x	0.97x	1.22x	
ECA		2658	8476	104	368	1170	3737	49	261	2.27x	2.27x	2.12x	1.41x	
LCA		1865	7076	93	375	1272	4704	61	375	1.47x	1.50x	1.52x	1.00x	
FIFO	regsim	211750	590835	14440	989	1320	3607	64	76	160.42x	163.80x	225.63x	13.01x	
ECA		208034	591454	14782	989	1950	5434	93	79	106.68x	108.84x	158.95x	12.52x	
LCA		720	1962	34	64	720	1962	36	66	1.00x	1.00x	0.94x	0.97x	
									Max	160.42x	163.80x	225.63x	13.01x	
									Average	2.39x	2.39x	2.38x	1.18x	

ECA, *LCA*, and *FIFO*. *ECA* sorts messages according to the creation time of the receiving actor in ascending order; messages for the *earliest created actor* are considered first. *LCA* is similar to *ECA* but sorts the actors in descending order of their creation time. *FIFO* sorts the messages based on the time they are sent in the ascending order.

To illustrate the speedup that can be achieved using TransDPOR, we performed a set of nine experiments which compare explorations performed using DPOR with explorations performed using TransDPOR. Table 3.1 shows the results for these experiments. For each subject and DPOR technique, we show the number of paths executed in their entirety while exploring the specified subjects, the total number of receive events executed (across all execution paths), the total exploration time in seconds, and memory usage in MB. Since the length of paths might be different in

a program, and the time is dependent on the platform and noise in the system, we focus on the number of receive events as the primary metric for comparison.

The experiments suggest that TransDPOR can explore up to *over two orders of magnitude* fewer receives than DPOR. In all the experiments TransDPOR has a speedup for at least one heuristic, and it is *never* the case that the use of TransDPOR results in more executed receives than DPOR. Although the speedup in receive events executed can at times be small (e.g., only 1.24x or less for `leader`), it can be also quite significant. For `chameneos`, the speedup is over 11x, and for `regsim`, it is over 163x. The `regsim` experiment using DPOR did not complete in 4 hours for either FIFO or ECA.

Combining with Sleep Sets Sleep sets is a POR technique based on the history of exploration [15]. Specifically, sleep sets record the receive events that have already been explored from a particular configuration, and avoid exploring them in successor configurations until some condition is met. Sleep sets can further prune the number of receive events and configurations that are explored [35]. In the case where the state-space is acyclic (which is the assumption in the model of actor systems presented in this work), sleep sets can be combined with dynamic POR in exactly the same way as static POR [12]. We implemented a variant of TransDPOR that is combined with sleep sets and compared it with the combination of DPOR with sleep sets.

In addition to the eight programs used in our initial experiment, we added three more programs. These programs have such a large state space that the exploration times out without sleep sets. `diningphil` is an implementation of the dining philosopher protocol in ActorFoundry. `minesweeper` is a simulation of the minesweeper game written using the Scala Actors library. `le-erlang` is an implementation of a fault-tolerant leader election algorithm for Erlang that had been running on Ericsson switches. Some bugs were found in the program by Arts et al. [50] in the presence of node failures. We re-implemented the buggy program in ActorFoundry in order to test it using our tool.

The results are presented in Table 3.2. For `le-erlang`, our tool was able to find all the previously known bugs in the algorithm (in the presence of node failures). We also tested the algorithm for four processes and *without a failure-recovery scenario*. To our surprise, our tool detected a *new bug*, which allows the program to reach a state in which no leader is elected. We contacted the

developers and they confirmed the new bug.

The combination with sleep sets reduces the improvement as sleep sets already prune many redundant receives; however TransDPOR is always equal to or better than DPOR in terms of explored paths and receives. For seven experiments, TransDPOR provides the speedup of over $1.20x$ for at least one heuristic. Note that it is not obvious from the program what may be a good heuristic, and the results table suggests the same. For example, ECA is the best heuristic for `1e-erlang`, and FIFO is the best for `minesweeper`. Moreover, different configurations of a single application, such as the `regsim-2-level`, may have different good heuristics for exploration.

Overall, the results suggest that our algorithm combined with sleep sets outperforms the combination of DPOR and sleep sets. We achieved speedup as high as $2x$ for the `regsim` benchmark as shown in Table 3.2. TransDPOR is also very efficient when exploring programs with bugs. We consistently find the bug faster than DPOR. Even in the presence of sleep sets, we were able to find the bugs up to $2.56x$ faster than DPOR.

3.7 Discussions

The TransDPOR technique explained in this chapter has several advantages. It is complete in that it can detect every deadlock and local safety violation in the program. Moreover, as the results suggest, it can significantly improve state-space exploration over DPOR. However, using TransDPOR for programs with large state-space may not be a practical solution since it may not be able to overcome the exponential growth of the state-space. For example, in the experiments with buggy programs such as `1e-erlang4`, although it is combined with sleep sets, it takes around an hour to find the bug for LCA heuristic. Next chapter presents another solution which is not complete like TransDPOR but it is scalable and requires the intervention of programmers.

Table 3.2: Comparison of TransDPOR+Sleep sets and DPOR+Sleep sets

		DPOR+ S				TransDPOR+ S				Speedup			
Heur.	Subject	# of Paths	# of Rec.s	time [sec]	mem [MB]	# of Paths	# of Rec.s	time [sec]	mem [MB]	# of Paths	# of Rec.s	time [sec]	mem [MB]
FIFO	fib5	16	101	3	173	16	101	3	173	1.00x	1.00x	1.00x	1.00x
ECA		16	139	4	159	16	139	4	173	1.00x	1.00x	1.00x	0.92x
LCA		16	91	3	174	16	91	3	174	1.00x	1.00x	1.00x	1.00x
FIFO	quicksort6	32	179	5	181	32	179	5	181	1.00x	1.00x	1.00x	1.00x
ECA		32	272	7	269	32	272	7	270.00x	1.00x	1.00x	1.00x	1.00x
LCA		32	156	5	194	32	156	5	193	1.00x	1.00x	1.00x	1.01x
FIFO	pi5	120	931	17	264	120	931	17	376	1.00x	1.00x	1.00x	0.70x
ECA		120	931	17	266	120	931	17.00	263.00	1.00x	1.00x	1.00x	1.01x
LCA		120	1236	22	377	120	990	18	456	1.00x	1.25x	1.22x	0.83x
FIFO	pipesort4	288	1448	20	378	288	1422	20	377	1.00x	1.02x	1.00x	1.00x
ECA		288	1293	19	376	288	1293	18	376	1.00x	1.00x	1.06x	1.00x
LCA		288	1944	27	376	288	1935	27	376	1.00x	1.00x	1.00x	1.00x
FIFO	chameneos2	216	1681	23	378	216	1453	20	377	1.00x	1.16x	1.15x	1.00x
ECA		216	1826	24	374	216	1530	21	376	1.00x	1.19x	1.14x	0.99x
LCA		216	1231	17	376	216	1231	17	375	1.00x	1.00x	1.00x	1.00x
FIFO	leader4	492	3125	43	454	492	3097	43	372	1.00x	1.01x	1.00x	1.22x
ECA		492	3267	45	376	492	3218	42	377	1.00x	1.02x	1.07x	1.00x
LCA		492	3311	46	680	492	3311	46	377	1.00x	1.00x	1.00x	1.80x
FIFO	shortpath4	126	473	8	261	126	473	8	262	1.00x	1.00x	1.00x	1.00x
ECA		126	489	8	260	126	489	8	260	1.00x	1.00x	1.00x	1.00x
LCA		126	522	9	262	126	502	8	262	1.00x	1.04x	1.13x	1.00x
FIFO	shortpath5	296	1408	22	375	296	1408	22	375	1.00x	1.00x	1.00x	1.00x
ECA		296	1031	16	264	296	997	17	265	1.00x	1.03x	0.94x	1.00x
LCA		296	1228	20	376	296	1218	19	451	1.00x	1.01x	1.05x	0.83x
FIFO	regsim	720	3453	37	376	720	2019	22	377	1.00x	1.71x	1.68x	1.00x
ECA		720	4054	47	375	720	2152	26	452	1.00x	1.88x	1.81x	0.83x
LCA		720	1962	22	264	720	1962	22	375	1.00x	1.00x	1.00x	0.70x
FIFO	regsim-2-level	1296	8636	141	427	1296	5537	92	417	1.00x	1.56x	1.53x	1.02x
ECA		1296	14990	267	558	1296	7486	129	560	1.00x	2.00x	2.07x	1.00x
LCA		1296	6481	115	381	1296	6295	111	381	1.00x	1.03x	1.04x	1.00x
FIFO	diningphil	31	1375	38	524	31	1375	37	524	1.00x	1.00x	1.03x	1.00x
ECA		31	2082	55	348	31	1662	44	366	1.00x	1.25x	1.25x	0.95x
LCA		31	1333	38	374	29	1147	33	407	1.07x	1.16x	1.15x	0.92x
									Max	1.07x	2.00x	2.07x	1.80x
									Average	1.00x	1.13x	1.13x	0.98x
Buggy programs (Exploration stops at first bug instance.)													
FIFO	diningphil (deadlock)	1	15	2	112	1	15	2	112	1.00x	1.00x	1.00x	1.00x
ECA		16	915	29	340	16	767	22	342	1.00x	1.19x	1.32x	0.99x
LCA		1	15	2	112	1	15	2	112	1.00x	1.00x	1.00x	1.00x
FIFO	minesweeper (deadlock)	1	29	3	163	1	29	2	163	1.00x	1.00x	1.50x	1.00x
ECA		2710	15577	484	717	2710	15381	499	744	1.00x	1.01x	0.97x	0.96x
LCA		6993	69350	1950	1041	6993	55430	1651	893	1.00x	1.25x	1.18x	1.17x
FIFO	le-erlang3-failure (safety)	457	1976	41	462	452	1944	27	427	1.01x	1.02x	1.52x	1.08x
ECA		30	174	4	241	30	174	4	236	1.00x	1.00x	1.00x	1.02x
LCA		233738	759109	14401	2020	93640	296229	3557	1557	2.50x	2.56x	4.05x	1.30x
FIFO	le-erlang4 (no leader, new)	2209	11006	169	605	2191	10146	155	586	1.01x	1.08x	1.09x	1.03x
ECA		1	27	2	112	1	27	2	112	1.00x	1.00x	1.00x	1.00x
LCA		198713	698759	14405	2011	88505	277440	3924	1383	2.25x	2.52x	3.67x	1.45x
									Max	2.50x	2.56x	4.05x	1.45x
									Average	1.16x	1.22x	1.40x	1.08x

Chapter 4

Setac/Setak: Frameworks for Exploring User-specified Schedules

Our study of TransDPOR in Chapter 3 shows that exploring conceptually all possible interleavings even when advanced pruning techniques are used, suffers from scalability problems.

An observation is that exploring a small subset of all possible interleavings might be sufficient to expose potential concurrency bugs. Therefore, an alternate solution would be carefully selecting interleavings that may expose bugs and controlling the schedule of execution to run the program with those interleavings. Identifying bug exposing interleavings requires the knowledge of the program specification and the programmers have that knowledge.

Unfortunately, current testing frameworks do not provide proper features for programmers to write such tests for actor programs. Our analysis of publicly available, manually written actor tests shows that they focus on very simple cases (test one actor for one message at a time) and have problems in controlling schedules and checking assertions [17].

One issue in writing test cases for actors, using just the standard library, is that controlling the schedule is difficult and sometimes impossible. The reason is that enforcing a specific schedule requires: (1) changing the application (e.g., using synchronous communication instead of asynchronous communication, thread sleeps, latches, barriers, etc.) to enforce the order of messages delivered to the actors; (2) changing the environment that runs the application (e.g., changing the system scheduler); and/or (3) creating mock actors to control the order of messages sent to the actor such that one actor is tested at a time.

Changing the application or the environment requires great effort on the programmer's side, while creating mock actors can make the test cases very complex and cannot be applied for programs with multiple communicating actors.

Another issue in writing test cases for actors, using just the standard library, is checking assertions, namely checking that the program state satisfies certain conditions at appropriate points in

execution. Assertions that check the effect of sending some messages should not be checked until after those messages have been processed by the actors. However, the standard library does not provide a functionality to check if messages have been processed. The solutions that are currently used include: (1) using explicit delays with the hope that the desired messages are processed by the actors; (2) using barriers/latches in the message handlers of the actors after some messages are processed; and/or (3) checking just the final results returned from the actors in the form of messages and not any internal states of the actors.

The issues of controlling schedules and checking assertions during tests were previously addressed for shared-memory programming, in particular for Java. For example, ConAn [18] and MultithreadTC [19] are two frameworks that allow controlling schedules, i.e., specifying the order of accesses to shared data and synchronization variables, and ThreadControl [21] provides a framework for checking assertions at appropriate points in execution. However, to the best of our knowledge, there was no such framework for any actor system.

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

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

Compared to the current approaches for writing actor tests, Setac/Setak provide several advantages:

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

This chapter presents an overview of Setac/Setak. More details of these frameworks are available online at:

<http://web.engr.illinois.edu/~tasharo1/setac/> and
<http://web.engr.illinois.edu/~tasharo1/setak/>.

This chapter starts by an example to show the difficulty of writing tests for actor programs using standard testing frameworks (Section 4.1). Section 4.2 explains Setac features and how Setac makes it easy to write tests for actor programs. Since Setac and Setak have similar features (with small syntax differences), we present Setac in more details to give the main idea of these testing frameworks. A brief overview of Setak and its differences with Setac are explained in Section 4.3. Finally, Section 4.4 discusses some of the limitations of Setac/Setak.

4.1 Setac by Example

To better describe the problems in testing actor systems, we start with an example. Consider a server that processes clients' requests using a divide-and-conquer recursive algorithm: when the server receives a request that needs a large amount of work, it distributes the work among two children, waits for them to finish their work, merges the results, and returns it as the final result of the request. To make the example more concrete, we assume a simple QuickSort server which is written in Scala actor library and shown in Listing 4.1. It accepts an array of integers as input (through a `Sort` message) and returns an array which is sorted input (through a `Result` message).

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

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

```

1 class QuickSort extends Actor {
2   var part1 = Array[Int]()
3   var part2 = Array[Int]()
4   var middle = Array[Int]()
5
6   var client: OutputChannel[Any] = null
7   var resultCount = 0
8   var enableProcessingMessage = true
9
10  start
11  def act() = loop {
12    react {
13      case Sort(input) if (enableProcessingMessage) => {
14        client = sender
15        split(input)
16      }
17      case Result(res) => {
18        resultCount += 1
19        if (resultCount == 1) {
20          part1 = res
21        } else if (resultCount == 2) {
22          part2 = res
23          val finalResult = mergeResults
24          client ! Result(finalResult)
25          enableProcessingMessage = true
26        }
27      }
28    }
29  }
30
31  private def split(xs:Array[Int]) {
32    if (xs.length <= 1) {
33      client ! Result(xs)
34    }
35    else {
36      enableProcessingMessage = false
37      resultCount = 0
38
39      val pivot = xs(0)
40      val left = (xs filter (pivot >))
41      val right = (xs filter (pivot <))
42      middle = (xs filter (pivot ==))
43      new QuickSort ! Sort(left)
44      new QuickSort ! Sort(right)
45    }
46  }
47
48  private def mergeResults:Array[Int] = {
49    // merge part1, part2, and middle
50  }
51 }

```

Listing 4.1: Simple QuickSort actor written in Scala actor library.

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

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

A message sequence diagram that corresponds to such a test is shown in Figure 4.1; `Sort(input1)` and `Sort(input2)` show the two messages sent from the test to `qsort` actor, while `Result(part1)` and `Result(part2)` show the two messages sent from the children which are created by `qsort` actor. As mentioned before, assertions break execution into *phases* in the test. The diagram in Figure 4.1 consists of two phases: in the first phase, a schedule is enforced in which `qsort` actor receives the first sort message, then a partial result from one of its children, and finally the second sort message. At the end of the phase, it is checked that the actor has not processed the second `Sort` message. In the second phase, the second partial result is delivered, and it is checked that the resulting arrays are sorted.

4.1.1 Problems with Writing Test for QuickSort

Listing 4.2 shows a possible implementation of the test depicted in Figure 4.1 *without* using `Setac` but using one of the standard libraries, e.g., `JUnit`. To enforce the desired order of message delivery, we had to change the code under test as shown in Listing 4.3.

In Listing 4.3, we added some arguments to the constructor: `childId` to distinguish the first and second child from the parent, `res1Latch` to be informed about sending `part1` (by one of the children) and then send `sort2` (because we want an order in which the second `Sort` message is delivered after the first partial result), `assert1Latch` such that the other child can finish the first phase (checking `assertion1`) and send `part2` for starting the second phase, and `sortLatch` to keep

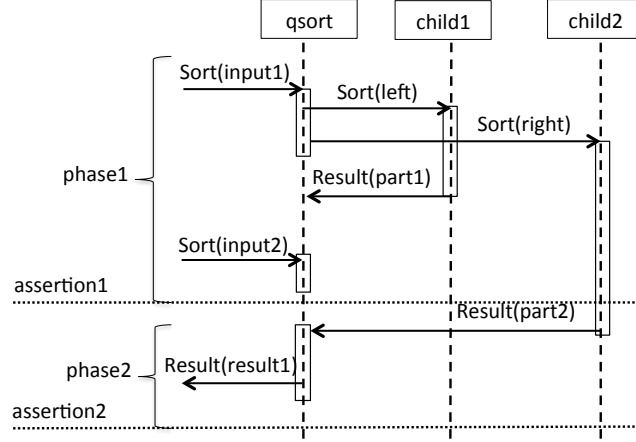


Figure 4.1: Sequence diagram describes a test which checks that processing multiple `Sort` messages do not overlap

the number of `Sort` messages processed by `qsort` actor and to infer the number of `Sort` messages remaining in its mailbox. The body of `QuickSort` actor is also changed to `countDown` or `wait` on the latches at appropriate times. Note that these changes might be different for different test cases. For example, if `input2` has more than one element or the order of elements in `input1` is changed, then we will have more children that send the partial results, and so we need more latches.

The test case from Listing 4.2 creates the latches and passes them to the constructor of `qsort` actor. We set the variable `childId` to zero to mark it as the parent. In `testTwoSortMessages` method, Line 20, a mock actor is created that sends (receives) messages to (from) `qsort` actor. For checking `assertion1`, we use thread sleep for a time interval to make sure that `qsort` actor cannot process any more messages and then check the value of `sortLatch`. Because we know that we sent two `Sort` messages, if the value of `sortLatch` equals to one, it means that one of the `Sort` messages still remains in the mailbox and has not been processed by the actor. The `finishLatch` is used to wait for the mock actor to finish its work before the test terminates.

In summary, writing the test described in Figure 4.1 using current testing frameworks has the following challenges:

- Reasoning about the delivery of important messages: how can we define messages that are important in the test and be informed about their delivery? As shown, using latches requires changes in the program under test, which makes the test very complex.

```

1 class QuickSortTest extends TestCase {
2   var qsort: QuickSort = null
3   var input1 = Array[Int](2, 3, 1)
4   var input2 = Array[Int](4)
5
6   var res1Latch: CountDownLatch = null
7   var assert1Latch: CountDownLatch = null
8   var sortLatch: CountDownLatch = null
9   var finishLatch: CountDownLatch = null
10
11  override def setUp {
12    res1Latch = new CountDownLatch(1)
13    assert1Latch = new CountDownLatch(1)
14    sortLatch = new CountDownLatch(2)
15    finishLatch = new CountDownLatch(1)
16    qsort = new QuickSort(0, res1Latch, assert1Latch, sortLatch)
17  }
18
19  def testTwoSortMessages {
20    actor {
21      // phase1
22      qsort ! Sort(input1)
23      // wait for the first partial result
24      res1Latch.await()
25      qsort ! Sort(input2)
26      Thread.sleep(1000)
27      // assertion1: actor does not process sort2
28      assert(sortLatch.getCount == 1)
29
30      // phase2
31      assert1Latch.countDown
32      // assertion2: resulting arrays are sorted
33      receive {
34        case Result(result) =>
35          assert(isSorted(result, input1))
36      }
37      receive {
38        case Result(result) =>
39          assert(isSorted(result, input2))
40      }
41      finishLatch.countDown
42    }
43    finishLatch.await
44  }
45
46  def isSorted(result: Array[Int], input: Array[Int]): Boolean = {
47    // check if the result is sorted array of input
48  }
49 }

```

Listing 4.2: Quick sort test written in JUnit.

```

1 class QuickSort(childId:Int,res1Latch:CountDownLatch,assert1Latch:CountDownLatch,sortLatch:CountDownLatch) extends Actor {
2   var part1 = Array[Int]()
3   var part2 = Array[Int]()
4   var middle = Array[Int]()
5
6   var requester: OutputChannel[Any] = null
7   var resultCount = 0
8   var enableProcessingMessage = true
9
10  start
11  def act() = loop {
12    react {
13      case Sort(input) if (enableProcessingMessage) => {
14        // if parent, track processed messages
15        if (childId == 0)
16          sortLatch.countDown
17        requester = sender
18        split(input)
19      }
20      case Result(res) => {
21        resultCount += 1
22        if (resultCount == 1) {
23          part1 = res
24        } else if (resultCount == 2) {
25          part2 = res
26          val finalResult = mergeResults()
27          requester ! Result(finalResult)
28          enableProcessingMessage = true
29        }
30      }
31    }
32  }
33
34  private def split(xs:Array[Int]) {
35    if (xs.length <= 1) {
36      // wait for phase1 and assertion1 to be finished
37      if (childId == 2)
38        assert1Latch.await()
39      requester ! Result(xs)
40      // signal on sending the first partial result
41      if (childId == 1)
42        res1Latch.countDown
43    } else {
44      enableProcessingMessage = false
45      resultCount = 0
46
47      val pivot = xs(0)
48      val left = (xs filter (pivot >))
49      val right = (xs filter (pivot <))
50      middle = (xs filter (pivot ==))
51      new QuickSort(1,res1Latch,assert1Latch,sortLatch) ! Sort(left)
52      new QuickSort(2,res1Latch,assert1Latch,sortLatch) ! Sort(right)
53    }
54  }
55
56  private def mergeResults():Array[Int] = {
57    // merge part1, part2, and middle
58  }
59 }

```

Listing 4.3: Quick sort modified for testing with the standard testing frameworks.

- Enforcing the desired order between the messages: if all of the actors in the system were created by the user, sometimes it is possible to impose some orders between the messages, but it is not possible in all situations. In the QuickSort example, the issue is that the children created for providing partial results are not under the control of the user. So, the user cannot easily impose the order in which the second sort message is delivered before the second partial result of the first sort message. As shown, changing the program under test is not an easy and practical solution.
- Checking the assertions: how do we know when to check `assertion1`? How long do we have to wait to make sure that `qsort` actor has processed as many messages as it can and thus cannot process more messages?

Usually, the test execution needs to wait for all actors to become idle before it checks the assertions. We refer to this state of an actor program as a *stable state*. An actor program is in a stable state if there is no actor which is processing a message and all the actors are suspended, blocked, or terminated or have not been started yet.

Common solutions to check assertions in stable states are using latches or thread sleep. As mentioned before, none is a reliable and appropriate solution.

- Accessing the contents of the mailbox: in `assertion1`, how can we check that the second sort message remains in the mailbox and will not be processed until `phase2` starts (the actor will not start processing the second sort message until it finishes its work with the first sort message)? We added `sortLatch` to the program under test to keep track of the `Sort` messages processed by the actor and then reason about the messages remaining in the mailbox of the actor. So, in order to check the contents of the mailbox, we need to keep track of the messages sent to the actor and processed by the actor via adding some extra variables and changing the application under test.

Note that these problems still exist even if other standard testing libraries such as `ScalaTest` [51] are used.

4.2 Writing Tests with Setac

Listing 4.4 shows the test from 4.1 written in Setac integrated with JUnit framework. Each `SetacTest` class is a test case that consists of three parts: `setUp`, tests, and `tearDown`. Before and after running each test, the `setUp` and `tearDown` methods are executed, respectively. In the QuickSort example, `tearDown` is not needed, therefore it is not shown in Listing 4.4.

To allow Setac to control the execution of the program under test, each class that extends `Actor` should replace it with `TestSubject`. Specifically, in Listing 4.1 we will have `class QuickSort extends TestSubject`, but note that this is literally the *only* change to the original code under test.

4.2.1 Defining Test Messages

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

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

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

Schedule messages are defined and the references to them are created via `createScheduleMessage` method. This method takes the sender, receiver, and content of the message as its parameters and returns a reference to a schedule message that can be used in the test to refer to that message. Note that *the messages that are not matched with any of the schedule messages will be delivered*

```

1 class QuickSortTest extends SetacTest {
2   var qsort: QuickSort = null
3   var input1 = Array[Int](2, 3, 1)
4   var input2 = Array[Int](4)
5
6   // test messages
7   var sort1: TestMessage = null
8   var sort2: TestMessage = null
9   var part1: TestMessage = null
10  var part2: TestMessage = null
11
12  override def setUp() {
13    qsort = new QuickSort
14    sort1 = createScheduleMessage(ANY_ACTOR, qsort, Sort(input1))
15    sort2 = createScheduleMessage(ANY_ACTOR, qsort, Sort(input2))
16    part1 = createScheduleMessage(ANY_ACTOR, qsort)({case Result(_) => ()})
17    part2 = createScheduleMessage(ANY_ACTOR, qsort)({case Result(_) => ()})
18  }
19
20  def testTwoSortMessages() {
21    // phase1
22    setSchedule(sort1 -> part1 -> sort2)
23    qsort ! Sort(input1)
24    qsort ! Sort(input2)
25    // assertion1: actor does not process sort2
26    assertWhenStable("mailbox has the second sort message", qsort.msgCountInMailbox(sort2) == 1)
27
28    // phase2
29    setSchedule(part2)
30    // assertion2: resulting arrays are sorted
31    receive {
32      case Result(result) =>
33        assertTrue("result1 is not sorted", isSorted(result, input1))
34    }
35    receive {
36      case Result(result) =>
37        assertTrue("result2 is not sorted", isSorted(result, input2))
38    }
39  }
40
41  def isSorted(result: Array[Int], input: Array[Int]): Boolean = {
42    // check if the result is sorted input array
43  }
44 }

```

Listing 4.4: Quick sort test written in Setac.

without any constraints. Watch messages are explained in details in Section 4.2.3.

For QuickSort example, we define the four schedule messages—`sort1`, `sort2`, `part1`, and `part2`—in the `setUp` method in Listing 4.4. For two of these message, the content parameter is known statically (the arguments of two `Sort` messages are `input1` and `input2`), so they are defined as objects. For the other two messages, the content is not known statically (the arguments of two `Result` messages), so they are defined as partial functions that can match any `Result` message sent to `qsort` actor. The senders of these messages are not important and are thus set to `ANY_ACTOR`.

4.2.2 Controlling the Order of Message Delivery and Checking Assertions

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

Controlling the Order of Message Delivery After defining the schedule messages, the order of their delivery can be enforced by the `setSchedule` method. Listing 4.4 uses this method, lines 22 and 29. In general, this method accepts a set of (comma-separated) constraints among schedule messages, where each constraint is specified with a precedence operator, ‘`->`’, between schedule messages. Note that if a reference to a schedule message is created but not used in any `setSchedule` method, this message will not be delivered by Setac. Instead, it will give a warning indicating that there are some messages that are not delivered. The reason is that creating a reference to a schedule message indicates the programmer wants to determine when the message should be delivered.

Checking Assertions Setac provides two methods for checking assertions: `assertWhenStable` and `assertAfter`. Both methods take a name and a boolean expression, and `assertAfter` additionally takes a *timeout* value. The difference between these methods is in the execution point at which they evaluate the boolean expression: `assertWhenStable` waits for the system to become stable and then evaluates the expression.

`assertAfter` evaluates the expression after a specified amount of time. In QuickSort example, for checking `assertion1` the system should be stable (to ensure that `qsort` actor cannot process any more messages), while `assertion2` can be evaluated immediately (because the results are ready). So, as shown at the end of `phase1` and `phase2` in Listing 4.4, `assertWhenStable` is used for `assertion1`, and `assertTrue` is used for `assertion2`. `assertAfter` is explained in Section 4.2.3 in more details.

Accessing the contents of the mailbox Setac also provides methods for inspecting the content of actor mailboxes. One of the methods is `msgCountInMailbox` which is used in `assertion1` (Line 26). This method is called on a test actor and takes a test message as its input. The returned value is the number of the messages in the mailbox of the actor that match the test message. In the example, at the end of `phase1`, we need to check that the second sort message is still in the mailbox and has not been processed by the actor.

4.2.3 Other Features

QuickSort example illustrates the key features of Setac. This section presents additional features of Setac.

Watch Messages

Sometimes the order of message delivery does not matter in the test but it is important to check the delivery or processing status of these messages. In Setac, the users can define such test messages as *watch messages*. They are defined via the `createWatchMessage` method that takes the same parameters as `createScheduleMessage`. Watch messages are lighter than schedule messages in that the framework does not interfere with the delivery of watch messages but only monitors them to collect the information of their delivery/processing. Because schedule messages are also monitored as watch messages, the programmer can obtain the delivery/processing status of schedule messages without defining them as watch messages.

Status of Test Messages

In Setac, it is possible to check whether a test message is delivered or processed by calling `isProcessed` or `isDelivered` respectively on the test message reference.

Additionally, the number of processing/delivery of a test message can be obtained via calling `numOfDelivery/numOfProcessing` methods on the test message.

Status of Actors

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

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

Assertions for Non-Stable Systems

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

4.3 Setak

Setak is implemented for actor programs written in Akka v1.2. It provides similar features as Setac. The only additional feature is that it allows some actions—such as checking assertions—to be performed after a set of (not all) messages are processed. Two methods `afterMessages(<a sequence of test messages>){body}` and `afterAllMessages{body}` are provided for this purpose.

The body of `afterMessages` is executed after all the messages specified in its argument (a set of comma separated test messages) are processed and the body of `afterAllMessages` is executed

when all of the test messages specified in the test are processed. This feature is a more reliable solution than `assertAfter` for checking assertions in non-stable systems.

In addition, Setak is integrated with ScalaTest [51], a common testing library for Scala, as well as JUnit [52].

4.4 Discussions

Setak and Setac are scalable testing solutions. They help the actor programmers to reproduce a known concurrency bug and facilitate testing actor programs. However, they have their own limitations. As mentioned before, only actors that extend `TestSubject` and messages whose receivers extend `TestSubject` can be monitored and controlled by these frameworks. This means that messages sent to anonymous actors which do not extend `TestSubject` are out of control of these frameworks. This limitation also carries over to the cases where the source code of the actor is not available to the testing frameworks.

Moreover, these testing frameworks require the programmers to specify the order of message delivery. Specifying bug exposing schedules for some programs might be very complicated and require lots of effort. Next chapter presents another technique that automatically generates schedules and executes the program with those schedules.

Chapter 5

Bitá: A Coverage-guided Testing Technique

Our study in Chapter 3 shows that exploring the entire state-space even if partial order reduction techniques are applied, does not scale. A scalable alternative solution is having testing frameworks like Setac/Setak (Chapter 4) that runs the program under test with schedules specified by programmers. However, such testing frameworks put the burden of specifying schedules—which might be very difficult for some programs—on the programmer.

This chapter presents another technique, called *Bitá*¹, that automatically generates schedules and runs the program with those schedules. These schedules are schedules that increase the coverage of the state-space with respect to some interleaving coverage criteria. Since interleaving coverage criteria are established based on common concurrency bugs, executing a program with those schedules guides the execution to interleavings which are more likely to expose bugs. Bitá is implemented for Akka actor programs and publicly available at: <http://bita.cs.illinois.edu/>.

This chapter starts by an example in Section 5.1 to show how Bitá works. Section 5.2 presents three interleaving criteria for actor programs. These criteria are used by schedule generator explained in Section 5.3 to generate schedules. Section 5.4 describes some important properties of the schedule generator which are crucial for the efficiency of testing process. Section 5.5 explains the run-time profiler/scheduler which is responsible for producing an execution path and realizing a schedule in the execution. The implementation and evaluation of Bitá are presented in Section 5.6.

5.1 Bitá by Example

Bitá generates schedules for an actor program by obtaining an execution path of the program and then modifying that execution path to increase coverage.

¹Bitá in Persian means “unique”.

```

1 class Writer extends Actor {
2   var results = ArrayBuffer[String]()
3   def receive() = {
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() = {
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() = {
25     case ActionDone => { // if the message is ActionDone
26       curActions -= 1
27       if (curActions == 0)
28         writer ! Flush // send message Flush to the writer
29     }
30   }
31 }

```

Listing 5.1: Real-world example of a receive ordering bug.

Consider the program shown in Listing 5.1. This code is a simplified version of a bug we found in a real-world actor program, Gatling [53], using Bitacore. There are three actor classes: *Writer*, which logs information to an external storage, *Action*, which performs some actions and notifies the writer about its execution, and *Terminator*, which is responsible for flushing the information stored in the writer when the program terminates. A program has *actionNum*—a parameter of *Terminator*—instances of *Action* and exactly one instance of each *Writer* and *Terminator*. We first give a possible execution of this system and then show how to get other schedules from it.

Figure 5.1 shows the message sequence diagram of an execution of a program with one action. When the action receives an *Execute* message, it performs the action, sends its name in the parameter of a *Write* message to the writer to record its execution, and sends an *ActionDone* message to the terminator to announce its termination. When the terminator receives the *ActionDone* message, it decreases the number of current actions. If this number reaches zero, the terminator sends a *Flush* message to the writer, which causes the writer to write all results into the external

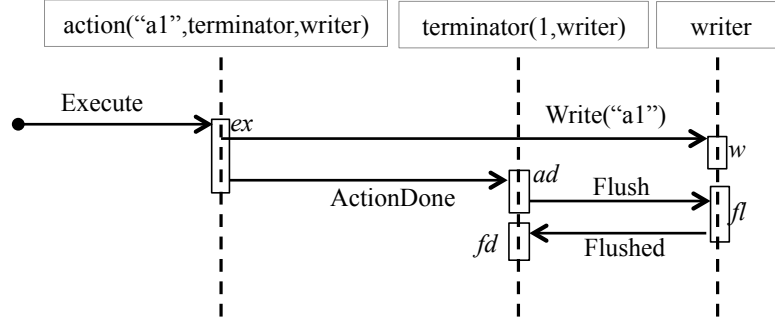


Figure 5.1: Sequence diagram of an execution of the code in Listing 5.1 with one action. Vertical rectangles represent receive events and the labels besides rectangles shows the receive events IDs.

storage, to assign *null* to the *results* variable, and to send a *Flushed* message to the terminator.

The execution in Figure 5.1 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 action will be missing from the written results but the *results* variable will be *null* and the program will throw an exception.

Suppose Bitá is given the execution path in Figure 5.1, $\omega = (ex, w, ad, fl, fd)$. Because Bitá realizes that *fl* may happen before *w*, it generates a schedule in which the receive of message *Flush* happens before the receive of message *Write*. Then, it executes the program with that schedule which reveals the bug.

One of the challenges in generating schedules is avoiding infeasible schedules. For example, in Figure 5.1, any schedule 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 receive events.

5.2 Interleaving Coverage Criteria for Actor Programs

Testing based on coverage criteria is a widely accepted method for both sequential and parallel programs [26,54,55]. The original and most common use of coverage criteria is as a way of evaluating the quality of tests or explored interleavings. But coverage criteria can also be used to generate

tests or schedules, as in Bita. Interleaving coverage criteria [24–27] for concurrent programs have been proposed for measuring the quality of a set of execution paths and also guiding the testing tools for exploring interleavings in the state-space that are more likely to expose bugs.

This section presents three interleaving coverage criteria for actor programs (Section 5.2.1) and how to measure the coverage achieved by a set of execution paths (Section 5.2.2). Section 5.3 uses the coverage criteria to automatically generate schedules.

There is a trade-off between the bug detection capability of an interleaving coverage criterion and the cost of fulfilling the criterion [27]. That is, while satisfying a criterion that requires exploring a larger number of paths 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 receives that occur in the same actor.

The rationale for this decision is threefold. First, considering pairs of receive events makes it scalable 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 happens as a part of an 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, since actors do not share state, considering different interleavings of receives with the same receiver actor is effective for detecting deadlock and local safety violations (see Chapter 3).

5.2.1 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 *interleaving goals* to be achieved by execution paths.

Pair of Consecutive Receives

Many concurrency bugs are triggered when two accesses to a shared resource occur in a particular order [56, 57]. Inspired by this observation, we define the following coverage criterion:

Definition 12 (Pair of Consecutive Receives (*PCR*)). *An execution path ω that contains two*

asynchronous receive events r and r' achieves the interleaving goal $r \rightarrow_{PCR} r'$ if and only if

- $rec(r) = rec(r')$; and
- r appears before r' in ω ; and
- there exists no asynchronous receive event r'' in ω such that $rec(r'') = rec(r)$, and that r'' appears between r and r' in ω .

As an example, consider a program based on the code in Listing 5.1 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* interleaving goals for the writer actor is six and one of the possible sets of execution paths that covers them is: $\Omega = \{(ex_1, ex_2, ad_1, ad_2, w_1, w_2, fl, fd), (ex_1, ex_2, ad_1, ad_2, fl, w_2, w_1, fd), (ex_1, ex_2, ad_1, ad_2, w_1, fl, w_2, fd), (ex_1, ex_2, ad_1, ad_2, fl, w_1, w_2, fd)\}$.

The *PCR* criterion relates to coverage criteria for shared-memory programs that consider pairs of accesses to a shared object [27, 29, 58] and adapts the idea to actor programs.

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_{init} initializes a field in an actor. 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} in the actor.

Definition 13 (Pair of Receives (*PR*)). *An execution path ω that contains two asynchronous receive events r and r' achieves the interleaving goal $r \rightarrow_{PR} r'$ if and only if*

- $rec(r) = rec(r')$; and
- r appears before r' in ω .

While the number of interleaving goals in the domain of PR and PCR for a given program are the same, it may take fewer paths to satisfy PR which brings a merit for PR over PCR . For the example in Section 5.2.1, the number of interleaving goals for the writer actor is six for both PR and PCR . However, the interleaving goals of PR can be covered by the first two paths, $(ex_1, ex_2, ad_1, ad_2, w_1, w_2, fl, fd)$ and $(ex_1, ex_2, ad_1, ad_2, fl, w_2, w_1, fd)$ of the four paths required for PCR .

Pair of Message Handler Change and Receive

An actor's message handlers may be changed during its lifetime. Sending a message to an actor that does not have a compatible handler for that message is a common bug pattern in actor programs [59]. 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 [11], 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 the sender that assumes the receiver has received the message [4]. We define the following criterion aimed at detecting this kind of potential error:

Definition 14 (Pair of Message Handler Change and Receive ($PMHR$)). *For each receive event r , let $cmh(r)$ denote whether r changes the actor message handlers. An execution path ω with two asynchronous receive events r and r' achieves the interleaving goal $r \rightarrow_{PMHR} r'$ if and only if*

- $rec(r) = rec(r')$; and
- $cmh(r) = true$ or $cmh(r') = true$; and
- *there exists no asynchronous receive event r'' in ω such that $rec(r'') = rec(r)$, $cmh(r'') = true$, and r'' appears between r and r' in ω .*

The domain of $PMHR$ is a subset of the domain of PCR and therefore, the cost of satisfying $PMHR$ is smaller than PCR . For the example in Section 5.2.1, suppose the writer changes its message handlers when it receives the *Flush* message. The set of $PMHR$ interleaving goals for the writer actor would be achieved by two execution paths $(ex_1, ex_2, ad_1, ad_2, w_1, w_2, fl, fd)$ and $(ex_1, ex_2, ad_1, ad_2, fl, w_1, w_2, fd)$.

5.2.2 Measuring Coverage

The following describes how to quantify the coverage achieved by a set of execution paths for the criteria explained in Section 5.2.1. In general, it is impractical to compute the coverage domain of a criterion—all possible interleaving goals for a given program and input—because it requires exploring all possible paths in the state-space. Instead, we compute the coverage achieved by different sets of execution paths and compare them to each other.

Definition 15 (Coverage of a Set of Execution Paths). *For a coverage criterion cr , a set Ω of execution paths covers a pair $(r, r')_{cr}$ of receive events if and only if there exist execution paths $\omega_1, \omega_2 \in \Omega$ such that ω_1 achieves $r \rightarrow_{cr} r'$ and ω_2 achieves $r' \rightarrow_{cr} r$.*

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

To measure the coverage achieved by a set of execution paths, 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 .

Another approach would be very precisely identifying each receive event by its happen-before relation. However, this approach is very expensive because identifying a receive event requires considering all the history of that receive event in each execution path. Moreover, the number of unmatched receive events and consequently interleaving goals would be very huge which makes it inappropriate for saturation-based testing [60].

To establish a balance between these two extreme approaches, we uniquely identify each receive event by a hash value which is computed from its *receiver actor* hash value and its *message* hash value. 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

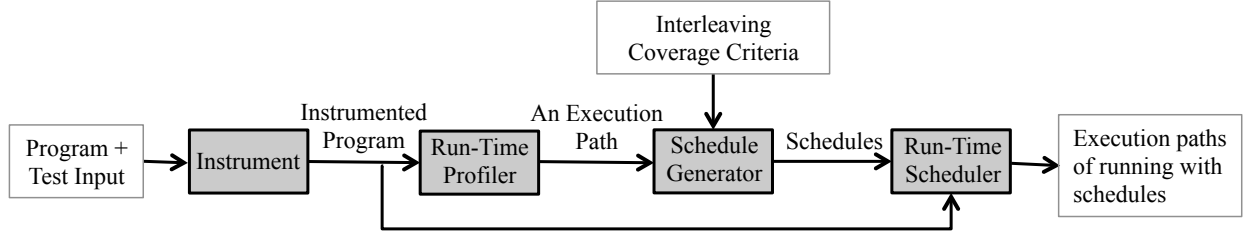


Figure 5.2: The architecture of Bitu.

event r that sends M , and the number of messages that r has sent before M . 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 path.

5.3 Coverage-guided Schedule Generation

Bitu leverages the coverage criteria from Section 5.2 to automatically explore potentially bug-revealing paths. The architecture of Bitu is shown in Figure 5.2. Bitu is given the program under test, its test input, and the interleaving coverage criteria. It first instruments and runs the program to obtain an execution path. The run-time profiler records the information of receive events in the execution path. Then, the schedule generator uses the execution path to generate schedules that increase the coverage with respect to the given coverage criteria. The last part is the run-time scheduler that runs the test with the generated schedules and outputs the execution paths.

In a nutshell, Bitu combines a technique for automatically generating schedules, presented in Section 5.3, with a run time scheduler that forces a test execution to satisfy a specified schedule, presented in Section 5.5. Before presenting the schedule generation algorithm, we present the notion of schedule in Bitu.

5.3.1 Schedule

The notion of schedule in Bitu is slightly different from Setac/Setak. Setac/Setak envision a schedule as a set of constraints on the order of message delivery by assuming that the mailboxes of actors sort the incoming messages in FIFO order. Therefore, it only schedules delivery of messages because the order of inspecting messages in each actor corresponds to the order of messages delivery.

However, in some actor systems, including the new versions of Akka, the mailboxes of actors do not necessarily sort the incoming messages in FIFO order; but they can be customized to sort messages with different policies. To handle such systems, Bitka generalizes the notion of schedule to a set of constraints on the order of *receive events*. The following definition gives a more precise description of a schedule:

Definition 16 (Schedule). *A schedule $s = \langle r_1, r_2, \dots, r_n \rangle$ is a finite sequence of receive events and indicates a set of constraints on the order of receive events. That is, for each r_i and r_j in s such that $i < j$, r_i must be executed before r_j .*

Executing an actor program with a given schedule means scheduling the concurrent receive events in the execution so that all the ordering constraints of the schedule are preserved, i.e., the execution path satisfies the schedule. More formally, let $s = \langle r_1, r_2, \dots, r_n \rangle$ be a schedule and $\omega = r'_1, r'_2, \dots, r'_n$ be an execution path of an actor program. We say ω satisfies s if and only if

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

A schedule is *feasible* if there exists at least one execution path of the program that satisfies the schedule. Note that a schedule does not need to contain all events of an execution path to be feasible but its events should be a subset of an execution path events and have the same order as they have in the execution path. As a result, *each execution path of an actor program is a feasible schedule*.

5.3.2 Overview of Schedule Generation

The algorithm for generating schedules takes an execution path and reorders receive events to create new schedules. It also uses the execution path to compute the ordering constraints on receive events, thereby preventing infeasible schedules. The schedule generation algorithm is shown in Algorithm 3. The inputs to the algorithm are an execution path ω and a coverage criterion cr . Each receive event r in ω contains the following information:

- $rec(r)$: the receiver actor of r ;

- $msg(r)$: the message of r ;
- $sender(r)$: the sender actor of $msg(r)$;
- $cmh(r)$: whether r changes $rec(r)$ message handlers;
- $sent(r)$: the set of messages sent by r ;
- $create(r)$: the set of actors created by r ; and
- $sync(r)$: whether r is a synchronous receive.

Moreover, $ind(r, \omega)$ for each receive event r and execution path ω stands for the index (position) of r in ω . The algorithm uses ω to construct feasible schedules that increase the coverage of cr .

The algorithm has three main variables:

- list S : the list of generated schedules;
- global set I : the set of interleaving goals that have been covered by ω and S ; and
- global set $mustHB$: the set of ordering constraints among receive events in ω ; indicates that a receive event must happen before another receive event.

First, the algorithm fills $mustHB$ by analyzing the ordering constraints of events in ω (Line 2). Each pair of receive events (r_i, r_j) in ω is added to $mustHB$ if swapping them may lead to an infeasible schedule, that is, they are in must-happen-before relation (details are explained in Section 5.3.4).

In addition, it initializes I to the set of interleaving goals achieved by ω (Line 3). The algorithm updates I whenever it adds a schedule to S —the algorithm for computing interleaving goals achieved by a schedule is similar to the algorithm for computing interleaving goals achieved by an execution path (explained in Section 5.2.1).

The algorithm looks at all pairs of receive events r_i and r_j in ω where $i < j$ (line 6). It checks if r_i and r_j are related by the coverage criteria— $isCrRelated(r_i, r_j, cr)$ —and if it is not the case that r_i must happen before r_j — $(r_i, r_j) \notin mustHB$. These two conditions are explained in detail in Section 5.3.3 and Section 5.3.4.

If these conditions hold, the algorithm tries to achieve both interleaving goals related to r_i and r_j . It checks whether any of the goals $r_i \rightarrow_{cr} r_j$ (Line 7) and $r_j \rightarrow_{cr} r_i$ (Line 12) has not yet been covered by the schedules in S and execution path ω , that is, whether the goal is not yet in I . Note that, although r_i appears before r_j in ω , the interleaving goal $r_i \rightarrow_{cr} r_j$ may not be achieved by ω if the criterion is *PCR* or *PMHR* and if the events are not consecutive.

For an interleaving goal that has not yet been covered, the algorithm calls *schedule* to generate a new schedule that can achieve the goal (details in Section 5.3.5). Thus, the number of schedules is bounded by twice the square of the length of the execution path, but usually will be smaller. 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 interleaving goals I .

Algorithm 3 *generateSchedules*(ω, cr)

Require: An execution path ω , coverage criterion cr

Ensure: List S of generated schedules

```

1:  $S \leftarrow \emptyset$ 
2:  $mustHB \leftarrow$  all pairs  $(r_i, r_j)$  such that  $r_i \xrightarrow{mhb}_{\omega} r_j$ 
3:  $I \leftarrow cr$ -interleaving goals achieved by  $\omega$ 
4: for all  $r_i$  in  $\omega$  so that  $0 < i < |\omega|$  do
5:   for all  $r_j$  in  $\omega$  so that  $i < j \leq |\omega|$  do
6:     if  $isCrRelated(r_i, r_j, cr)$  and  $(r_i, r_j) \notin mustHB$  then
7:       if  $r_i \rightarrow_{cr} r_j \notin I$  then
8:          $s' \leftarrow schedule(\omega, i, j, false, cr)$ 
9:          $S \leftarrow S \cup \{s'\}$ 
10:         $I \leftarrow I \cup cr$ -interleaving goals can be achieved by  $s'$ 
11:       end if
12:     if  $r_j \rightarrow_{cr} r_i \notin I$  then
13:        $s' \leftarrow schedule(\omega, i, j, true, cr)$ 
14:        $S \leftarrow S \cup \{s'\}$ 
15:        $I \leftarrow I \cup cr$ -interleaving goals can be achieved by  $s'$ 
16:     end if
17:   end if
18: end for
19: end for
20: return  $S$ 

```

5.3.3 Identifying Coverage-related Events

To ensure that each generated schedule increases coverage compared to the already generated schedules, Algorithm 3 checks whether two events r_i and r_j are related to coverage criterion cr . To contribute to the PR and PCR criteria, the events must have the same receiver; to contribute to the $PMHR$ criterion, the events must have the same receiver and at least one of them must change the receiver’s message handlers. 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 $cr = PMHR$, it returns *true* if and only if $rec(r_i) = rec(r_j)$ and if either $cmh(r_i)$ or $cmh(r_j)$.

5.3.4 Must-Happen-Before Relation

To avoid creating infeasible schedules, Algorithm 3 checks whether two events can be reordered or whether the first is not required to happen before the second. For this purpose, it computes the must-happen-before relation from ω by considering all ordering constraints that exist in actor programs.

The following explains three kinds of ordering constraints and how we compute must-happen-before relation from them. The first constraint—casual constraint—is drawn from the basics of the actor model and can be applied to any actor system. The other two constraints—sender-receiver and synchronous constraints—are established due to Akka infrastructure and constructs. Other actor systems may have more/different constraints which go beyond the basic actor model. However, Bitra can be extended for those systems by customizing the last two constraints presented in this section.

Causal Constraint

If one receive event causes the message of another receive to be sent, then these two events cannot be reordered. We exploit the message casual relation from Definition 11 to define a variant of casual relation which is between two receive events:

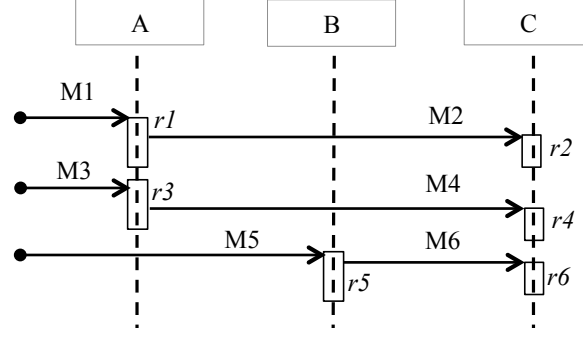


Figure 5.3: Sequence diagram of an actor program with three actors. Because of sender-receiver constraint, M_2 is always delivered before M_4 .

Definition 17 (Receive Causal Relation). *In an execution path ω , the receive casual relation $\xrightarrow{rca}_\omega$ is a binary relation on $r, r' \in \omega$ such that $r \xrightarrow{rca}_\omega r'$ if and only if $ind(r, \omega) \xrightarrow{mca}_\omega msg(r')$.*

If the causal relation $r \xrightarrow{rca}_\omega r'$ holds, we say r' is caused by r or r causes r' .

For the example in Figure 5.1, the execution path is $\omega = (ex, w, ad, fl, fd)$ and the receive causal relation includes:

$ex \xrightarrow{rca}_\omega w$, $ex \xrightarrow{rca}_\omega ad$, $ad \xrightarrow{rca}_\omega fl$, $fl \xrightarrow{rca}_\omega fd$, $ex \xrightarrow{rca}_\omega fl$, $ad \xrightarrow{rca}_\omega fd$, $w \xrightarrow{rca}_\omega fd$, and $ex \xrightarrow{rca}_\omega fd$.

Sender-Receiver Constraint

Some actor systems—including Akka which we use for the evaluation—implement sender-receiver constraint. As mentioned before, with this constraint, messages sent from the first to the second will not be delivered out-of-order. This constraint only concern the order of messages between a pair of actors. Messages delivered from different actors can be reordered unless this violates other constraints.

As an example, consider the sequence diagram in Figure 5.3. Actor A upon receiving M_1 and M_3 respectively sends M_2 and M_4 to actor C . When actor B receives M_5 , it sends M_6 to actor C . Because of sender-receiver constraint, message M_2 is always delivered before M_4 but M_6 may be delivered before or after M_2 and M_4 . Hence, although r_2 and r_4 are not in casual relation, a schedule like $\langle r_1, r_3, r_4, r_2, r_5, r_6 \rangle$ is infeasible² while a schedule like $\langle r_1, r_2, r_5, r_6, r_3, r_4 \rangle$ is

²By assuming that the mailbox of actor C sorts M_2 and M_4 with the order they are delivered.

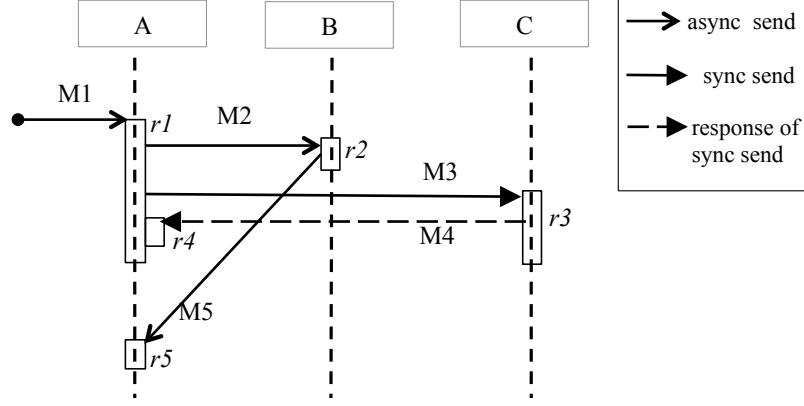


Figure 5.4: Sequence diagram of an an actor program with synchronous communication.

feasible.

We define sender-receiver relation to apply this constraint when generating schedules:

Definition 18 (Sender-Receiver Relation). *In an execution path ω , the sender-receiver relation $\xrightarrow{sr}_{\omega}$ is a binary relation on $r, r' \in \omega$ such that $r \xrightarrow{sr}_{\omega} r'$ if and only if $ind(r, \omega) < ind(r', \omega)$, $sender(r) = sender(r')$, $rec(r) = rec(r')$, and $msg(r)$ is sent before $msg(r')$.*

Note that if the mailbox of an actor does not sort the incoming messages with FIFO order, it might be possible that two events which are in sender-receiver relation to be reordered. However, if this situation does not happen in ω , we conservatively assume that reordering events in sender-receiver relation is not feasible.

Synchronous Constraint

In a synchronous communication the message sender is blocked until it receives the response. Because of this constraint, the causal and sender-receiver relations are not sufficient for generating feasible schedules.

To give an intuition, the sequence diagram in Figure 5.4 shows an execution of an actor program with a synchronous communication. When actor A receives message M_1 , it sends message M_2 to actor B and then, it synchronously sends message M_3 to actor C and waits until it receives the response in message M_4 . Actor B upon receiving M_2 sends message M_5 to A .

The corresponding execution path of this diagram is $\omega = (r_1, r_2, r_3, r_4, r_5)$. In this execution

path, r_4 and r_5 are not in causal or sender-receiver relation but they cannot be reordered in any feasible schedule. For example, a schedule like $\langle r_1, r_2, r_5, r_3, r_4 \rangle$ is not feasible because r_4 is a synchronous receive that happens as a part of asynchronous receive r_1 . In this case, we say r_1 and r_4 are in synchronous relation and since r_1 and r_5 cannot be reordered in any feasible schedule, r_4 and r_5 also cannot be reordered in any feasible schedule.

Definition 19 (Synchronous Relation). *In an execution path ω , the synchronous relation $\xrightarrow{\text{sync}}_\omega$ is a binary relation on $r, r' \in \omega$ such that $r \xrightarrow{\text{sync}}_\omega r'$ if and only if $\text{sync}(r')$ and $\text{ind}(r, \omega) = \max\{0 < k < \text{ind}(r', \omega) \mid \text{rec}(r_k) = \text{rec}(r') \text{ and } \text{sync}(r_k) = \text{false}\}$.*

We exploit synchronous relation to compute the ordering constraints imposed by synchronous communications. More precisely, any synchronous receive r' which is in a synchronous relation with an asynchronous receive r , $r \xrightarrow{\text{sync}}_\omega r'$, inherits the ordering constraints of r for any event appears after r' in ω .

Computing Must-Happen-Before Relation

The set of all must-happen-before relation is computed as the *transitive closure* of union of the constraints imposed by casual, sender-receiver, and synchronous relations.

Definition 20 (Must-Happen-Before Relation). *In an execution path $\omega = (r_1, r_2, \dots, r_n)$ the must-happen-before relation $\xrightarrow{\text{mhb}}_\omega$ is the smallest transitively-closed binary relation on $r, r' \in \omega$, $\text{ind}(r, \omega) < \text{ind}(r', \omega)$ such that $r \xrightarrow{\text{mhb}}_\omega r'$ if one of the following conditions holds:*

- $r \xrightarrow{\text{rca}}_\omega r'$; or
- $r \xrightarrow{\text{sr}}_\omega r'$; or
- there exists r'' such that $r'' \xrightarrow{\text{sync}}_\omega r$ and at least one of the $r'' \xrightarrow{\text{rca}}_\omega r'$ or $r'' \xrightarrow{\text{sr}}_\omega r'$ holds.

Algorithm 3 computes the must-happen-before relation from an arbitrary execution path, ω and keeps it in global variable *mustHB*. The sets $\text{mustHB}_{\text{rca}}$, $\text{mustHB}_{\text{sr}}$, and $\text{mustHB}_{\text{sync}}$ respectively stand for the pairs of (r, r') in *mustHB* which are in casual relation, $r \xrightarrow{\text{rca}}_\omega r'$, sender-receiver relation, $r \xrightarrow{\text{sr}}_\omega r'$, and synchronous relation $r \xrightarrow{\text{sync}}_\omega r'$.

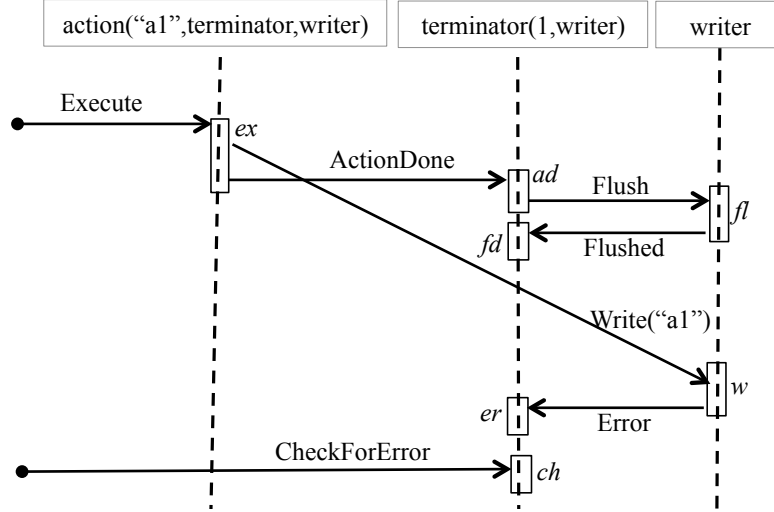


Figure 5.5: Sequence diagram that shows an execution of the code in Listing 5.1 with some extensions in which the terminator accepts a *CheckForError* message and the writer sends an *Error* message to the terminator if *results* is *null*.

5.3.5 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 covers a not yet covered interleaving goal, the schedule generator creates a new schedule to cover this goal (Lines 8 and 13 in Algorithm 3). Our approach to schedule generation addresses two important challenges. First, to create a schedule with enough information for the run-time scheduler to guarantee that it will succeed in realizing the schedule. Second, to create a schedule that can achieve not only a single new interleaving goal but multiple new interleaving goals.

To illustrate these challenges, consider the sequence diagram in Figure 5.5, which is an extended version of Figure 5.1. 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 5.5 shows an execution where the writer sends an *Error* message and where the check for errors occurs after the terminator receives this *Error* message.

Suppose that, based on the execution in Figure 5.5, the schedule generator tries to reorder *ch*

and er , so that terminator receives *CheckForError* before *Error*. A naïve approach would be to create a schedule that specifies only the two events $\langle ch, er \rangle$. Unfortunately, this schedule does not address the two challenges. First, the schedule does not contain enough information for the run-time scheduler to realize 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 [28, 29]. Second, the schedule achieves only a single additional interleaving goal. Instead, a longer schedule can achieve multiple additional goals at once. For example, a schedule $\langle ch, fd, er \rangle$ for PR covers the two interleaving goals $ch \rightarrow_{PR} fd$ and $ch \rightarrow_{PR} er$.

Our schedule generation approach addresses both challenges. First, to create a schedule that contains enough information for the run-time scheduler to realize 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 covered 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 4, 5, and 6 summarize our schedule generation approach. The main part is described in Algorithm 4. 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 . It 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 5.1 with two actions and a schedule $s = \langle ex_1, w_1, ex_2, w_2, ad_1, ad_2, fl, fd \rangle$ which is an execution path of the program. 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 interleaving goal. At first, the algorithm copies all the events before r_i to the generated schedule s' (Line 2), giving $s' = \langle ex_1 \rangle$. Next, it searches through all events between i and j in s and appends those events that must happen before r_j to s' (Lines 3 to 7), 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, based on the *swap* value the algorithm appends either r_i, r_j (Line 9) or r_j, r_i (Line 11) to s' , resulting in $s' = \langle ex_1, ex_2, w_2, w_1 \rangle$.

So far, the generated schedule, s' , is feasible and covers an uncovered interleaving goal. However,

Algorithm 4 $schedule(s, i, j, swap, cr)$

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

Ensure: 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: add all  $r_k$  in  $s$  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:      $s' \leftarrow s' \parallel r_k$ 
6:   end if
7: end for
8: if  $swap$  then
9:    $s' \leftarrow s' \parallel r_j, r_i$ 
10: else
11:    $s' \leftarrow s' \parallel r_i, r_j$ 
12: end if
13:  $tail \leftarrow getTail(s, s', i, j)$ 
14: return  $scheduleTail(s', tail, cr)$ 
```

Algorithm 5 $getTail(s, s', i, j)$

Require: feasible schedules s , new schedule s' , and indices i, j of the receive events rescheduled in s'

```
1:  $t \leftarrow$  empty list
2: if  $cr = PR$  then
3:   add the last event of  $s'$  to  $t$ 
4: end if
5:  $r_i \leftarrow s[i]$ 
6:  $r_j \leftarrow s[j]$ 
7: for  $k = i + 1$  to  $|s|$  do
8:    $r_k \leftarrow s[k]$ 
9:   if  $r_k \notin s'$  and  $(r_i, r_k) \notin mustHB_{rca}$  and  $(r_j, r_k) \notin mustHB_{rca}$  then
10:     $t \leftarrow t \parallel r_k$ 
11:   end if
12: end for
13: return  $t$ 
```

a longer schedule whose prefix is s' might be able to cover more uncovered interleaving goals.

Lines 13 and 14 invoke Algorithm 5 and Algorithm 6 to apply a heuristic for covering multiple interleaving goals in a schedule (the second challenge mentioned for schedule generation). These algorithms are explained in the following in more details.

A Heuristic for Reducing Generated Schedules

An observation is that the same coverage may be achieved by different sets of schedules. The set with fewer schedules is always desirable.

One of the solutions to obtain a smaller set is to cover multiple uncovered interleaving goals in a single generated schedule rather than covering one uncovered interleaving goal in each generated schedule. For this purpose, Algorithm 4 after creating schedule s' that brings two events in a particular order, it invokes Algorithm 5 and Algorithm 6 which search through the remaining events (events not included in the schedule) and try to schedule them to achieve additional interleaving goals. If it is revealed that scheduling remaining events would gain more uncovered interleaving goals, it considers s' as the prefix of a longer schedule whose tail contains the scheduled remaining events.

Nevertheless, not all remaining events can be included in the schedule. Algorithm 4 invokes Algorithm 5 to compute the remaining events that can be scheduled in variable *tail* (Line 13). The inputs of Algorithm 5 are the old schedule s , the new schedule s' , and two indices i and j indicating that s' has been obtained by scheduling two events at i and j in s . Depending on the coverage criterion, the tail contains the last event of the so far generated schedule s' . For *PCR* and *PMHR*, r_i and r_j must be consecutive and hence 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 as part of the tail (Line 3).

Then, the algorithm searches through all events in s after r_i and includes all events which are not yet scheduled, excluding those which are caused by r_i or r_j (Lines 7 to 12). It is crucial to exclude such events from the tail because the messages of these events may not be sent after changing the place of r_i and r_j . For example, in the execution of Figure 5.5, by reordering fl and w , message *Error* is not sent by the writer and *er* does not happen in the execution. Indeed, the algorithm needs to eliminate *er* from any schedule in which w happens before fl to make the schedule feasible.

Back to our example of a program with two actions, the tail after computing $s' = \langle ex_1, ex_2, w_2, w_1 \rangle$ is $t = \langle w_1, ad_1, ad_2, fl \rangle$. 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 $w_1 \xrightarrow{rca}_{\omega} fd$ and $w_2 \xrightarrow{rca}_{\omega} fd$.

After computing the tail, Algorithm 4 invokes Algorithm 6 (Line 14), to schedule events in the tail for gaining additional interleaving goals. Algorithm 6 takes a prefix p of scheduled events, the tail t , and criterion cr as its inputs. 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 3 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 4. The concatenation $p||t$ removes any event in p if it exists in t to handle the cases where the last event in prefix p is included in tail t , for example, when the criterion is PR .

Since Algorithm 4 generates a schedule in which all events before the first event to schedule have the same order as the given schedule, calling Algorithm 4 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 4 and 6 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 6 this recursion is guaranteed to terminate for every finite initial execution path.

For our example with $t = \langle w_1, ad_1, ad_2, fl \rangle$, the first call to Algorithm 6 selects w_1 and fl for reordering and calls Algorithm 4 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 4 returns the generated schedule. This schedule achieves two new interleaving goals $w_2 \rightarrow_{PR} w_1$ and $fl \rightarrow_{PR} w_1$.

5.4 Properties of Generated Schedules

Schedules generated by Algorithm 3 contain three important properties that make testing more efficient: (1) they are *feasible*; (2) each generated schedule *increases coverage* with respect to the currently generated schedules; and (3) they contain *sufficient detailed information* so that the run-time scheduler would be able to realize them in the execution.

In the rest of this section, these properties are explained in details.

Algorithm 6 *scheduleTail*(p, t, cr)

Require: schedule prefix p , and tail t , coverage criterion cr

Ensure: 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  $r_i \rightarrow_{cr} r_j \notin I$  then
5:          $s' \leftarrow schedule(p || t, i + |p|, j + |p|, false, cr)$ 
6:         return  $s'$ 
7:       else if  $r_j \rightarrow_{cr} r_i \notin I$  then
8:          $s' \leftarrow schedule(p || t, i + |p|, j + |p|, true, cr)$ 
9:         return  $s'$ 
10:      end if
11:    end if
12:  end for
13: end for
14: return  $p$ 
```

5.4.1 Feasible

The schedule generation algorithm guarantees that each generated schedule is feasible.

We prove this property by contradiction. Let $s = \langle r_1, \dots, r_i, r_j, \dots, r_n \rangle$ be an infeasible schedule generated by Algorithm 3. In other words, there exists a receive event r_j in s that cannot be executed after r_i in any of the execution paths that satisfy $\langle r_1, \dots, r_i \rangle$.

There are two possibilities that r_j cannot be executed:

(1) $msg(r_j)$ is not sent (violating casual constraint); or (2) $msg(r_j)$ is sent but at least one of the ordering constraints of s cannot be satisfied, that is, there exists a receive event $r_k, 0 < k < j$ in s such that r_j must happen before r_k (violating sender-receiver or synchronous constraint).

In the following, we show that none of these situations happen using Algorithm 3 for generating schedules.

Casual Constraint We show that Algorithm 3 never violates the casual constraint in the generated schedules. Receive event r_j is included in schedule s by Algorithm 3 either (1) in the call to Algorithm 4 from Algorithm 3; or (2) in the call to Algorithm 4 from Algorithm 6.

In the first case, Algorithm 4 before including r_j , it includes all the receive events that must happen before r_j in ω by preserving their orders in ω (Lines 2 and 3 of Algorithm 4). Therefore, any receive event r such that $r \xrightarrow{rca}_{\omega} r_j$ would be added to s with the same order as they have in

ω .

In the second case, schedule s is generated via reordering events in the tail of $p||t$ and r_j is included in tail t . Algorithm 5 includes a receive event in the tail if it is not caused by any receive event rescheduled in Algorithm 4 (Line 9 of Algorithm 5). Because of transitivity of casual relation, this implies that for each event r included in $p||t$, all events that cause r are also included in $p||t$ with the same order as they have in ω . Thus, r_j is included in the tail if all receive events that cause r_j remain intact, i.e., they exist in $p||t$ and have the same order as they have in ω .

On the other hand, as explained in (1), Algorithm 4 only includes an event r from its input s in the new schedule s' if it has included all events from s that must happen before r and consequently cause r . Hence, calling Algorithm 4 from Algorithm 6 (Lines 5 and 8) with $p||t$ given as its input, preserves the casual relation of the receive events in $p||t$ in the generated schedule.

From above discussions, it can be concluded that whenever the schedule generator includes r_j in a generated schedule, it also includes all the receive events that cause r_j by preserving their orders in ω . Indeed, $\langle r_1, \dots, r_i \rangle$ contains all orderings of receive events required for $msg(r_j)$ to be sent and the casual constraint will not be violated in schedule s .

Sender-Receiver or Synchronous Constraint In this case, the message of r_j is generated in at least one of the execution paths that satisfy $\langle r_1, \dots, r_i \rangle$ but there exists at least one receive event $r_k, 0 < k \leq i$ in s such that r_k must happen before r_j because of synchronous or sender-receiver constraints.

Since in r_j appears before r_k in ω , such schedule is generated if the schedule generator either (1) changes the order of r_j and r_k (Algorithm 4, Lines 9 and 11); or (2) includes r_k in the prefix of generated schedule and leaves r_j in the tail (Algorithm 4, Line 13).

The first case never happens since schedule generator only reorders two receive events if they are not in must-happen-before relation and hence do not have synchronous or sender-receiver constraint.

The second case also never happens because Algorithm 4 always adds r_k which must happen before r_j to prefix p before it adds r_j to the prefix.

Therefore, it is not possible that the generated schedules violate sender-receiver or synchronous constraint.

5.4.2 Increase Coverage

Each generated schedule by Algorithm 3 increases the coverage compared to the already generated schedules. This property holds because the algorithm only generates a new schedule for an interleaving goal if the goal has not been covered yet. After generating each new schedule, it adds all the interleaving goals covered in the schedule to the current covered interleaving goals. Therefore, it will never generate a schedule for an interleaving goal that has been already covered.

5.4.3 Sufficiently Detailed

Although a schedule is not required to contain all the receive events of an execution path, having more details of receive events in the generated schedules helps the run-time scheduler succeed in realizing schedules during execution. If sending a message requires a specific order of receive events and the schedule does not contain those receive events, the run-time scheduler may not be able to realize the schedule even if the schedule is feasible (the first challenge of schedule generation explained in Section 5.3.5).

The schedule generator in Section 5.3 as shown in Section 5.4.1 before including any receive event r in the schedule, it includes all the receive events that cause r by preserving their orders. This helps the run-time scheduler to properly schedule receive events which are in the race and succeed in realizing a schedule at run-time.

5.5 Run-Time Profiler/Scheduler

This section describes Bitá’s run-time profiler—that generates an arbitrary execution path—and run-time scheduler—that runs the program with a given schedule. We use the same infrastructure for run-time profiler and run-time scheduler. The inputs of this infrastructure are the instrumented program, the test input, and a list of feasible schedules. Bitá instruments the actor system to intercept all calls to the actor system for sending, receiving, actor creation, and message handlers change. For each such call, this common infrastructure updates the information about the program execution.

When the list of schedules given as input of this infrastructure is empty, it acts as run-time

profiler and generates an arbitrary execution path of the program. On the other hand, when the list of generated schedules is not empty, it acts as a run-time scheduler, that is, it iterates over the list and runs the program with each schedule and outputs the execution path. 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 delivered message is processed by the actor. In this way, it makes sure that the messages are processed with the same order as they are delivered.

Upon sending messages, the scheduler is called when the message is going to be placed in the mailbox. When the schedule is not empty, it checks two conditions and if one of them is passed then it allows the message to be placed in the receiver mailbox; otherwise, it keeps the message in a pool of held messages to be delivered later. Note that even if the message is held in the pool, the sender actor does not block for sending messages.

In the first condition, it compares the corresponding receive of the message with the receive event at the head of schedule. If it matches, then it returns true; otherwise, it checks the second condition. The second condition is intended for managing the cases where the event does not exist in the schedule. In this situation, it checks whether the message should be delivered to preserve synchronous or sender-receiver constraints. If the condition is satisfied, it returns true; otherwise, it returns false.

Upon executing each receive event, the scheduler compares the receive event with the head of schedule. If it matches, the scheduler updates the current schedule by removing the head of 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.

5.6 Implementation and Evaluation

To evaluate the effectiveness of Bitu, we have implemented it for Akka [4], a popular, commercially supported actor library for Scala, and applied it to five real-world actor programs and three smaller actor programs. In summary, we have the following results:

Program	LOC	Description
Gatling-v1.4 (Ga)	11,902	Stress testing tool for HTTP servers [53]
GeoTrellis-v0.9 (Geo)	20,706	Geographic data processing engine [63]
Fyrie Redis-v1.2 (FR1)	3,517	Redis client written in Scala [64]
Fyrie Redis-v2.0(FR2)	2,788	Redis client written in Scala [64]
SignalCollect-v2.1(SC)	6,315	Framework for scalable graph computing [65]
Barber (Ba)	329	Sleeping barber problem [61]
Messenger (Ms)	197	Instant messaging application
ProcessRegistry (PReg)	322	Translation of process registry in Erlang [62]

Table 5.1: Programs used in the experiments.

- 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.

5.6.1 Experimental Setup

Programs

Table 5.1 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 [61] and the translation of a program used in earlier work [62]. 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 receive 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, 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 Bitá so that for each external entity, we can define artificial actors as the senders of the external messages. As a result, Bitá 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.

In addition, a small source to source transformation is needed when using Bitá for programs with synchronous communication. The programmer needs to replace all calls to Akka library for synchronous messaging such as `akka.patterns.Pattern.ask`, with a call to Bitá which is `bita.akka.patterns.Pattern.ask`. However, this transformation can be easily automated.

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 [66] 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.

The Akka’s default scheduler uses Java thread pool for executing actors. That is, at each step of execution, it non-deterministically chooses an actor that has a message in its mailbox and executes that actor.

5.6.2 Bug Detection

Real-World Bugs

Applying Bitá to the programs in Table 5.1 reveals twelve bugs, as shown in the first column of Table 5.2. We experiment with four known bugs and Bitá finds all of them. For example, the

Bug	Issue	Bita			Random Scheduler						Default Scheduler	
		Tried Criteria	Time	Schedule	$d_{max}=100ms$ Time Execs	$d_{max}=200ms$ Time Execs	$d_{max}=300ms$ Time Execs	$d_{max}=300ms$ Time Execs	$d_{max}=300ms$ Time Execs	$d_{max}=300ms$ Time Execs	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, PMHR</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
PReg(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			10,939		371,543		339,622		335,903		419,020	
Total #of Bugs			120		20		30		30		7	
Avg. Detec. Time			91		18,577		11,320		11,196		59,860	
Slowdown			1x		203x		124x		122x		656x	

Table 5.2: 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.

developers of the known bug SC3 mention that “In rare cases the test fails in the following way [...]“ [67], 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.

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 execution path from the initial execution. To address this source of non-determinism, we run Bita (including initial execution path) 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 Bitá? We prioritize criteria based on their cost, which is the number of generated schedules. Based on the discussion in Section 5.2 and initial experiments, the number of generated schedules for *PR* and *PMHR* are usually smaller than for *PCR*. The number of generated schedules for *PR* and *PMHR* may not be comparable. We configure Bitá to obtain an arbitrary execution path and to use at first *PR*, then *PMHR*, and finally *PCR* until a bug is found. The bug detection time is the sum of the time for obtaining the initial execution path, 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 5.2 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 PReg and the default scheduler. The “Tried Criteria“ column shows the set of criteria tried by Bitá until it finds the bug. For programs where *PMHR* is not applicable because these programs do never change the message handlers of an actor, Bitá skips *PMHR* and uses *PCR* after *PR*. The “Schedule“ column indicates the number of schedules tested by Bitá. The “Execs“ column for the baselines shows the number of executions until the bug is detected or until the timeout is reached.

The results show that Bitá 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. Bitá finds ten of the twelve bugs with the first criterion, *PR*, and by running the program with at most three schedules. The *PCR* and *PMHR* criteria each detect one bug missed by *PR*. The small confidence intervals for Bitá and

the large confidence intervals for the baselines show the stability of Bita in detecting bugs.

The bottom of Table 5.2 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 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.

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 5.2 and how many schedules would be generated if it is configured to use only one criterion. For each bug, Bita obtains an initial execution path, 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 5.6 and Figure 5.7 compare pairs of criteria to each other. Each point (x, y) corresponds to one execution of each bug. Because most programs do not change actor message handlers at run-time, *PMHR* is not applicable for them, and there are fewer points in the plots for *PMHR*.

In Figure 5.6, 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 5.6 shows that both *PR* and *PMHR* perform much better than *PCR* in detecting bugs. For the programs that change actor message handlers at run-time, *PR* and *PMHR* perform similarly.

In Figure 5.7, the x and y values respectively show the number of schedules generated using the criterion at the X axis and the criterion at the Y axis. As expected, the number of generated schedules for *PR* and *PMHR* is fewer than *PCR*. Moreover, in most cases, the number of schedules

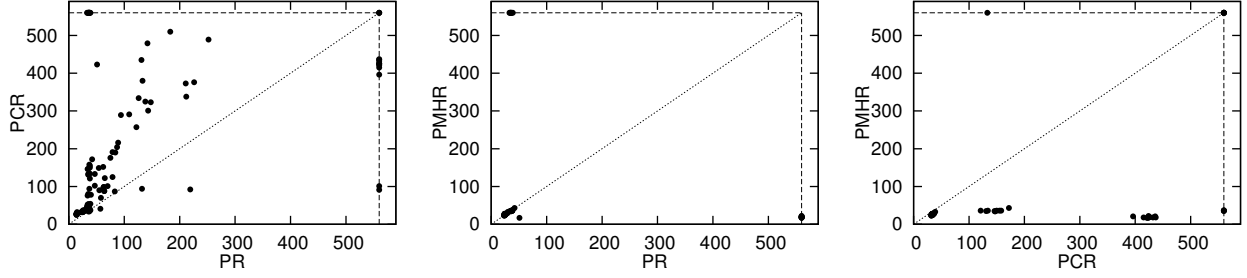


Figure 5.6: Pairwise comparison of the three criteria. The X and Y axes show the time (in seconds) that each criterion detects the bug.

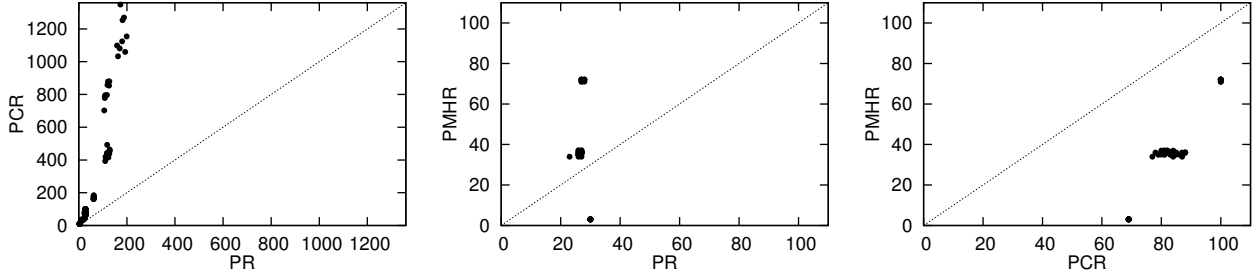


Figure 5.7: Pairwise comparison of the three criteria. The X and Y axes show the number of schedules generated for each criterion.

generated for PR is fewer than $PMHR$.

Table 5.3 shows the results of bug detection of the three criteria more accurately with numbers. Since some of the criteria do not detect the bug in all ten executions of each bug, we give the probability of bug detection in column “Prob.” which is computed by dividing the number of executions in which the bug is detected to the total number of executions. In addition, for each bug and each criterion, Table 5.3 shows the total number of schedules generated (column “Total Sched.”) as well as the number of schedules tested (column “Tested Sched.”) and the time taken until the bug is found (column “Time“). We measure these numbers from executions in which the bug is detected and give the arithmetic mean and confidence intervals (95% confidence level).

In summary, the results suggest that among all three criteria, PR is the most effective criterion for detecting bugs in that in most cases, it generates fewer schedules and detects bugs faster.

Bug	<i>PCR</i>				<i>PR</i>				<i>PMHR</i>			
	Prob.	Total Sched.	Tested Sched.	Time	Prob.	Total Sched.	Tested Sched.	Time	Prob.	Total Sched.	Tested Sched.	Time
Ga1	0.1	84	7	133	1.0	26	1	36±1	0.3	37±1	1	36±1
Ga2	0.9	81±2	8±1	146±10	1.0	26±1	1	37±1	1.0	35	1	36±2
Ga3	1.0	100	1	34±1	1.0	27	1	26±1	1.0	72	1	28±1
Ga4	1.0	84±2	1	34±1	1.0	27	1	25±1	1.0	36	1	25±1
SC1	1.0	821±36	821±36	244±34	1.0	117±5	117±5	102±15	NA	NA	NA	
SC2	1.0	439±17	439±17	103±12	1.0	120±4	120±4	86±32	NA	NA	NA	
SC3	1.0	1181±78	1181±78	404±43	1.0	177±8	177±8	176±29	NA	NA	NA	
FR11	1.0	10	10	52±15	1.0	3	3	43±6	NA	NA	NA	
FR12	1.0	30±2	30±2	59±8	1.0	10	10	36±1	NA	NA	NA	
Ba	1.0	69	69	424±7	0.1	30	30	51	1.0	3	3	19±1
Ms	1.0	171±5	171±5	28±1	1.0	62	62	14	NA	NA	NA	
PReg	0.7	48±7	48±7	113±17	0.5	24±3	24±3	62±15	NA	NA	NA	

Table 5.3: Comparison of three criteria when Bita is configured to use one criterion. “Prob.” is the probability of bug detection; “Total Sched.” is the total number of schedules generated; “Tested Sched.” is the number of schedules tested until the bug is found; “Time” is the time taken (in seconds) until the bug is found; “NA” means not applicable.

5.6.3 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 5.4 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 5.8 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 5.2) because Bita’s estimated coverage domain is smaller than the real coverage domain. There are two reasons. First, our approach is based on a single initial execution,

Program	<i>PR</i>				<i>PCR</i>				<i>PMHR</i>			
	Time	Bit	Rand	Impr.	Time	Bit	Rand	Impr.	Time	Bit	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

Table 5.4: 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.

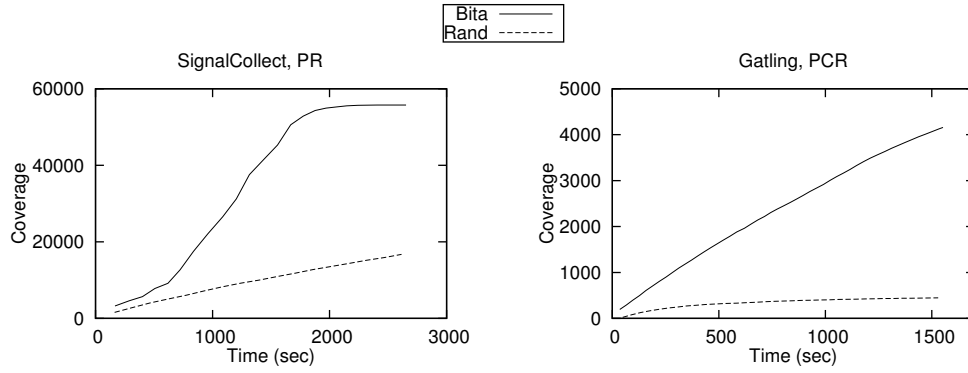


Figure 5.8: Comparison of coverage achieved by Bita and random scheduling with $d_{max} = 300ms$.

whereas random scheduling can discover additional interleaving goals in later executions. Second, the dependency relation and consequently the must-happen-before relation considered by Bita is very conservative. Because of that, it may miss feasible interleaving goals. Despite the smaller estimated coverage domain, Bita clearly outperforms random scheduling in both coverage and bug finding ability.

Chapter 6

Related Work

This chapter gives an overview of the previous work related to this dissertation. Section 6.1 and Section 6.2 discuss about systematic and random explorations which are related to TransDPOR and our baselines in Bitac. Section 6.3 is related to our work on Setac/Setak and gives an overview of testing frameworks that involve programmers in specifying interleavings. Section 6.4 and Section 6.5 walk through studies related to Bitac and present various coverage criteria proposed for concurrent programs and coverage-guided testing techniques. Finally, Section 6.6 presents previous studies on testing and debugging actor programs.

6.1 Exhaustive Exploration

Systematic testing approaches such as model checking and verification techniques for shared-memory [46, 68, 69] and message-passing programs [14, 70–72] exhaustively explore the entire state-space possibly optimized through applying partial-order reduction (POR) techniques.

McErlang [71] is a tool for model checking Erlang programs. The input of this tool is the abstract model of program in the form of Finite State Machine (FSM). Also, the programmer needs to annotate the program to trigger the appropriate events to map the current state of the program to a state in the FSM model. One of the problems with McErlang is that specifying a model of a system which is conformable to the system specification is a challenging task for the programmers. In contrast, Basset [14, 73] is a framework for state-space exploration of actor programs written in Scala actor [11] and ActorFoundry [3]. Basset directly works on the source code and explores (almost) all possible interleavings.

The problem with exhaustive exploration is lack of scalability which leads to state-space explosion. POR techniques aim for improving the exhaustive explorations by removing redundant

interleavings. One of the earliest POR approaches is based on computing persistent (or stubborn) sets [34, 35, 37] and the related technique of ample sets [36]. Persistent sets can be computed statically or dynamically. Using static analysis for computing persistent sets [35] suffers from conservative approximation, resulting in coarse persistent sets. Therefore, *dynamic POR* (DPOR) techniques [12], which compute the persistent sets at runtime, have been proposed to yield more accurate persistent sets. Our work on TransDPOR is inspired by the algorithm in DPOR but exploits the transitivity of race relation in actor systems to improve the exploration.

Another variation of persistent (or stubborn) sets is *weak persistent sets* [34, 37], which can generate smaller sets and lead to better reduction. This reduction needs additional knowledge about the transitions that enable and disable each other, which may not be easy to compute during the exploration.

Sen and Agha proposed a DPOR technique for testing multi-threaded programs [40], as well as for testing distributed message-passing programs [42]. Both papers present an operational definition of the set of transitions to be explored from a state, and the presented algorithms are conceptually similar to that in [12]. Kastenbergh and Rensink proposed a new DPOR which is based on *probe sets* for handling dynamic creation and destroying of processes and objects [74]. Probe sets relies on abstract enabling and disabling relations among actions, rather than associated sets of concurrent processes. The authors show that their technique leads to a better reduction in comparison to persistent sets.

Recently a new partial-order reduction technique called cartesian POR was proposed, which is based on *cartesian semantics* [43] and stateful exploration. The authors provide an operational definition, and present a dynamic algorithm that overcomes the acyclic state space restriction in stateless approaches. Since the formal definition of actor systems presented in this dissertation implies no cycle in the state-space, this feature is not useful for the actor systems used in our TransDPOR study. Cartesian POR is shown to improve over optimal persistent sets for some examples. The cartesian approach trades space for time since the approach requires storing program states precisely.

Lei and Carver [75] propose a technique for both message-passing and shared-memory programs that explores only one interleaving from each partial order. However, their technique for message-

passing assumes sender-receiver constraint and requires a non-trivial amount of memory for storing interleavings that are yet to be explored.

Message Passing Interface (MPI) [76, 77] is a popular environment for writing message-passing programs. MPI programs are more constrained than the basic actor programs. Specifically, MPI processes always assume *sender-receiver constraint* and usually have *matched sends and receives*. Vakkalanka et al. [44] proposed a stateless DPOR technique for MPI programs, called POE, which exploits these constraints. POE can produce only one interleaving for large MPI programs that do not have an MPI wild-card receive. Because of the differences between the actor system and MPI, this approach cannot be applied to the actor systems.

6.2 Bounded Explorations and Random Testing

Exploring all possible interleavings even when POR techniques is applied, does not scale. To overcome this problem, some heuristics have been proposed to limit exploration time and speedup bug detection [57, 78]. CHESS [57] bounds the exploration by limiting the preemptions. Wang et al. [78], propose a systematic exploration based on selective search. This approach prioritizes high-risk interleavings which expose orderings that have not been covered yet for exploration. We are not aware of a bounded model checker for actor programs.

An alternate, simple, but effective solution for state-space explosion problem of systematic testing is random testing. In this technique, the testing tool randomly chooses and explores a subset of all possible interleavings. Several variants of random testing have been proposed. ConTest [79] tries to explore different interleavings by randomly injecting delays in memory accesses and synchronization points. This work is similar to our baseline for evaluating Bitá in that our baseline injects random delays for message delivery.

Other random approaches apply some heuristics to make random testing more effective. Sen [80] proposed a random-based approach that schedules threads based on the partial order of events. The goal of this approach is to make sampling partial orders more uniformly than the naïve randomized algorithm. In PCT [56, 81], the scheduler assigns random priority to each thread. This priority is changed at some random points during the run time which increases the coverage with respect to pure random testing. Although random testing techniques make testing more scalable, if the

probability of bug exposing interleaving is low, they may not be able to detect the bug.

6.3 Testing Programmer-specified Interleavings

Among many explored interleavings in systematic or random approaches, a few of them might be sufficient to expose potential bugs. Thus, exploiting the knowledge of the programmers in selecting a subset of all possible interleavings in some cases might be more effective than automated selection approaches.

Several testing frameworks have been developed for shared-memory, multi-threaded Java code for this purpose [18–20, 82]. In ConAn [18], the programmer can specify the order of the events in the test by using a global clock value. A difficulty of writing test cases in ConAn is that the user needs to provide the order of *all* events in the test, but the order of some events might not be important. Another issue with ConAn is that tests are written as scripts in a language different from the language of the program under test. In MultithreadedTC [19], tests are written in Java, which is the same as the language of the program under test. The programmer can specify the order of some (not necessarily all) events with respect to a global clock. However, tracking the exact value of the global clock can make it complex and difficult to specify the order of events. Our design and implementation of Setac/Setak are inspired by these testing frameworks but we tried to address the mentioned drawbacks of these frameworks and also adapt the idea for actor programs. To the best of our knowledge, there was no similar testing framework for actor programs.

Recently, Jagannath et al. [20] developed a framework for multi-threaded Java programs. In their framework, the programmer annotates parts of the code as events and specifies the order of event executions in the test. The authors show that their testing framework makes it easier and more reliable to specify the interleaving of the events in the test execution. In addition, Park et al. [82] developed a testing framework for multi-threaded C programs. In this framework, the programmer can mark parts of the program as *concurrent breakpoints* and specify the order of their execution.

6.4 Coverage Criteria for 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. [24] are the first to propose to apply coverage criteria to concurrent programs. Yang et al. [25] adapt all-definition-use pair coverage to concurrent programs and show how to measure it. Synchronization coverage is a set of coverage criteria focused on synchronization primitives of shared memory programs [26]. For example, one metric requires that each lock acquisition must be observed as blocked and blocking at least once. The main focus of [26] are metrics that are usable for humans, whereas our approach in Bita automates the use of concurrency coverage metrics.

Lu et al. [27] propose a hierarchy of seven interleaving coverage criteria and theoretically analyze the cost of each criterion. One of the criteria proposed in Chapter 5 is inspired from this study. Krena et al. [60] propose a set of search-based and saturation-based coverage metrics that are derived from dynamic analyses to find concurrency errors. The authors also present how to measure the coverage of a set of executions. 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. However, in Bita, we identify actors by a hash value which is computed based on their dynamic type and the receive events that create the actors.

Location-pair (LP) [83] is a coverage metric for shared-memory multi-threaded programs that considers pairs of program locations that consecutively access the same shared variable but from different threads. The authors show that LP coverage is a better metric for evaluating the quality of test suites than other concurrency coverage metrics such as method pairs and def-use pairs. Since actors do not operate on shared states, this criterion cannot be directly applied for actor programs. However, it is somehow related to *PCR* criterion proposed in Chapter 5.

Souza et al. [84] propose structural coverage criteria for measuring the quality of test suites in MPI programs. The proposed coverage criteria are based on MPI synchronization primitives. Specifically, the criteria target all synchronized send-receive statements in MPI which correspond to all possible interleavings of receive events in actor programs. Therefore, exploiting those criteria for generating schedules is not a scalable solution. In contrast, for the coverage criteria presented in Chapter 5, we consider pairs of receives to keep the approach scalable. Deniz et al. [85] describe

mutation operators for MCAPI (Multicore Communications API) programs and measure coverage in terms of how many mutants a test kills. In contrast to both existing approaches in [84] and [85], Bitra leverages coverage criteria for schedule generation.

6.5 Coverage-guided and Active Testing

Active testing combines static or dynamic analysis that finds potential concurrency bugs with controlling schedule of execution aimed at exposing these potential bugs. These techniques consist of two phases. In the first phase, they analyze the program to predict bug exposing interleavings; and in the second phase they control the scheduler to guide the execution to bug exposing interleavings [86]. The idea has been applied to data races [87,88], deadlocks [86], atomicity violations [88–91], memory-related errors [92], concurrent access anomalies [58], and other errors [93,94]. Bitra is similar to active testing in that it contains the two phases of active testing. However, it is more general since it does not focus on concurrency bugs, but generates schedules to increase coverage.

There are few studies that similar to Bitra, adapt active testing to predict interleavings based on some coverage metrics. Hong et al. [95] propose a method to increase the synchronization construct-pair coverage in multi-threaded programs. Yu et al. [96] consider multiple interleaving idioms to construct a coverage domain and predict the interleavings that increase the coverage of those idioms. The authors show that the guided execution which increases the idioms coverage detects concurrency bugs faster than random testing techniques. Similar to Bitra, both approaches define interleaving goals based on one or more initial executions. Bitra differs from these solutions in that they do not generate the schedule with sufficient information that can increase the coverage. Therefore, in the second phase, as the results of the studies show, the tool may not be able to achieve the predicted coverage due to lack of guidance in the execution. Moreover, these solutions are proposed for shared-memory programs and not for actor programs.

Lei et al. [97] introduce random delays into message-passing programs to increase the coverage of send-receive pairs. Their coverage criterion considers all permutations of receive events and therefore may not scale to large programs. Moreover, introducing random delays does not guarantee increasing coverage. In contrast to [97], Bitra guarantees that each generated schedule increases

coverage.

Another stream of work forces execution of shared-memory programs into unusual interleavings under the assumption that these interleavings are not tested sufficiently [98, 99]. Adapting these approaches to actors is subject to future work.

6.6 Debugging Actor Programs

Claessen et al. [62] propose a solution to detect concurrency bugs in Erlang programs by automatically generating tests and by using the linearizations of a concurrent execution as an oracle for the concurrent execution. In contrast to our solutions that work on the programs source codes, their technique relies in finite state models that formally specify how the program under test should behave.

Christakis and Sagonas [59] describe a static analysis of Erlang programs to find message passing errors based on four common bug patterns. One of the interleaving coverage criteria proposed in Chapter 5 is inherited from bug patterns studied in this work. In contrast to their analysis, which suffers from false positives, our approach in Bita guarantees that each detected bug is a real bug. In addition, Bita does not search for particular bugs but increases coverage.

Chapter 7

Conclusions and Future Work

This dissertation presents three techniques for testing non-deterministic behavior of actor programs. The techniques specifically aim for reducing the number of explored interleavings among the huge number of possible interleavings. The following gives a summary of each technique and its related future work.

7.1 TransDPOR

Chapter 3 presents a dynamic POR technique that exploits the transitivity of race relation in actor systems and improves the state-space exploration over a well-known POR techniques called DPOR.

Although we applied TransDPOR for message-passing systems, we believe that the technique may be applicable to shared-memory multi-threaded programs if the race relation between the transitions is defined so that it is transitive. Making the race relation transitive may require defining more coarse dependency relation, for example, all accesses (read or writer) to a shared variable would be considered to be dependent. On the other hand, it has been shown that refining the dependency relation in shared-memory model also improves the POR techniques [34]. Investigating this trade-off in different concurrent systems would be an interesting line of research. In actor systems, according to our initial manual inspections, most receive events write to the local state of the actor. Hence, considering all receives with the same receiver actor to be dependent does not make significant difference with the case where the dependency is refined.

Moreover, exploring the possibility of combining TransDPOR with stateful exploration is another research topic that can be considered as the future work.

7.2 Setac/Setak

Chapter 4 presents two testing frameworks for Scala actor and Akka that allow programmers to specify the order of message delivery in the test execution. Moreover, these frameworks provide features that facilitate checking assertions in testing actor programs. These testing frameworks can be improved by extending the schedule language so that similar to [100], programmers can encode advanced patterns of message ordering rather than simple partial ordering of messages. Another improvement would be providing feedback to the programmer about her specified schedule. For example, currently Setac/Setak are agnostic about infeasible schedules. Recognizing the infeasible schedules and giving appropriate alert in response to them would help programmers to revise the schedule.

7.3 Bita

Chapter 5 presents a coverage-guided testing technique called Bita. Based on three coverage criteria, Bita generates schedules and executes the program under test with those schedules. Experimental results show that Bita is very effective in detecting concurrency bugs. However, Bita can be extended from several aspects.

Extending Criteria. The implemented coverage criteria target pairs of receive events; extending these criteria to the criteria that target three or four receive events may increase the bug detection capability of the technique. However, this extension should be performed by taking the cost of the criteria into account.

Improving Coverage Domain Prediction. Bita may miss testing some possible pairs of events in the coverage domain for two reasons. First, Bita relies on the first execution of program for generating schedules. If some events do not happen in the first execution, they would not be considered in the generated schedules. This problem can be alleviated if multiple executions of the program are used as the foundation for generating schedules. Second, the casual relation in Bita is very conservative. It assumes that all the receive events with the same receiver actor are dependent. As a result, some pairs estimated to be in must-happen-before relation might be possible to be reordered. The dependency relation and consequently

the casual relation can be refined, for example, if receive events that read from actor’s local state are discriminated from those that write to actor’s local state.

Reducing Generated Schedules. Applying heuristics to reduce the number of generated schedules for a given criterion makes testing more efficient, specially for the criteria with large coverage domain. One of the heuristics is presented in Section 5.3.5. For some criteria the problem of generating minimum number of schedules may not be solvable in polynomial time. For example, generating minimum number of schedules for *PR* reduces to the problem of finding the dimension of a partial order which is NP-complete [101]. However, investigating various heuristics that can obtain the sub-optimal answer would be beneficial.

Prioritizing Schedules. Prioritizing the generated schedules might also be helpful in reducing the bug detection time. Currently, Bita runs the program with the generated schedules in the order they were generated. However, using techniques such as [98, 99] for prioritizing schedules that exercise more untested interleaving goals may reduce the bug detection time.

Combining Schedule Generation with Input Generation. Combining schedule generation of Bita with test input generation would be an effective approach for detecting various kinds of bugs in actor programs. Sen and Agha [42] propose similar technique for model checking actor programs. The approach combines systematic exploration with symbolic execution. However, due to scalability problems of systematic exploration, it is not a practical solution for programs with large state-space. Our conjecture is that combining schedule generation with test input generation would be a scalable practical solution.

References

- [1] C. Hewitt, P. Bishop, and R. Steiger, “A universal modular actor formalism for artificial intelligence,” in *IJCAI*, 1973, pp. 235–245.
- [2] G. A. Agha, *ACTORS — A Model of Concurrent Computation in Distributed Systems*, ser. MIT Press series in artificial intelligence. MIT Press, 1986.
- [3] *The Actor Foundry: A Java-based Actor Programming Environment*, Open Systems Laboratory, University of Illinois at Urbana-Champaign, 1998-09. [Online]. Available: <http://osl.cs.uiuc.edu/af>
- [4] J. Bonér et al., “Akka library,” <http://akka.io>.
- [5] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D’Hondt, and W. De Meuter, “Ambient-oriented programming,” in *OOPSLA*, 2005, pp. 31–40.
- [6] “Charm++,” <http://ppl.cs.illinois.edu/research/charm>.
- [7] J. Armstrong, “Making reliable distributed systems in the presence of software errors,” Ph.D. dissertation, Kungl Tekniska Högskolan, December 2003, <http://www.erlang.org>.
- [8] Václav et al., “GPars,” <http://gpars.codehaus.org/>.
- [9] 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.
- [10] M. Sirjani and M. M. Jaghoori, “Formal modeling,” G. Agha, J. Meseguer, and O. Danvy, Eds., 2011, ch. Ten years of analyzing actors: Rebeca experience, pp. 20–56.
- [11] P. Haller and F. Sommers, *Actors in Scala - Concurrent programming for the multi-core era*. Artima, 2011.
- [12] C. Flanagan and P. Godefroid, “Dynamic partial-order reduction for model checking software,” in *POPL*, 2005, pp. 110–121.
- [13] S. Lauterburg, R. K. Karmani, D. Marinov, and G. Agha, “Evaluating ordering heuristics for dynamic partial-order reduction techniques,” in *FASE*, 2010, pp. 308–322.
- [14] S. Lauterburg, M. Dotta, D. Marinov, and G. Agha, “A framework for state-space exploration of java-based actor programs,” in *ASE*, 2009, pp. 468–479.
- [15] P. Godefroid, “Using partial orders to improve automatic verification methods,” in *CAV*, 1991, pp. 176–185.

- [16] S. Tasharofi, R. K. Karmani, S. Lauterburg, A. Legay, D. Marinov, and G. Agha, “Trans-DPOR: A novel dynamic partial-order reduction technique for testing actor programs,” in *FMOODS/FORTE’12*, pp. 219–234.
- [17] S. Tasharofi and R. E. Johnson, “Patterns in testing concurrent programs with non-deterministic behaviors,” University of Illinois at Urbana–Champaign, Tech. Rep., 2011, <http://hdl.handle.net/2142/35209>.
- [18] B. Long, D. Hoffman, and P. Strooper, “Tool support for testing concurrent Java components,” *IEEE Trans. Softw. Eng.*, vol. 29, pp. 555–566, 2003.
- [19] W. Pugh and N. Ayewah, “Unit testing concurrent software,” in *ASE*, 2007, pp. 513–516.
- [20] V. Jagannath, M. Gligoric, D. Jin, Q. Luo, G. Rosu, and D. Marinov, “Improved multi-threaded unit testing,” in *ESEC/FSE*, 2011, pp. 223–233.
- [21] A. Dantas, F. Brasileiro, and W. Cirne, “Improving automated testing of multi-threaded software,” in *ICST*, 2008, pp. 521–524.
- [22] M. Odersky et al., “Scala programming language,” <http://www.scala-lang.org>.
- [23] S. Tasharofi, M. Gligoric, D. Marinov, and R. E. Johnson, “Setac: A framework for phased deterministic testing of scala actor programs,” The Second Scala Workshop, 2011.
- [24] R. N. Taylor, D. L. Levine, and C. D. Kelly, “Structural testing of concurrent programs,” *IEEE Trans. Software Eng.*, vol. 18, no. 3, pp. 206–215, 1992.
- [25] C.-S. D. Yang, A. L. Souter, and L. L. Pollock, “All-du-path coverage for parallel programs,” in *ISSTA*, 1998, pp. 153–162.
- [26] A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur, “Applications of synchronization coverage,” in *PPoPP*, 2005, pp. 206–212.
- [27] S. Lu, W. Jiang, and Y. Zhou, “A study of interleaving coverage criteria,” in *ESEC/FSE*, 2007, pp. 533–536.
- [28] 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.
- [29] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam, “Maple: a coverage-driven testing tool for multithreaded programs,” in *OOPSLA*, 2012, pp. 485–502.
- [30] S. Tasharofi, M. Pradel, Y. Lin, and R. E. Johnson, “Bitac: Coverage-guided, automatic testing of actor programs,” in *ASE’13*, to Appear.
- [31] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott, “A foundation for actor computation,” *J. Funct. Program.*, vol. 7, no. 1, pp. 1–72, Jan. 1997.
- [32] R. K. Karmani and G. Agha, in *Encyclopedia of Parallel Computing*, D. A. Padua, Ed. Springer, 2011, pp. 1–11.
- [33] S. Tasharofi, P. Dinges, and R. E. Johnson, “Why do scala developers mix the actor model with other concurrency models?” in *ECOOP*, 2013, pp. 302–326.

- [34] P. Godefroid and D. Pirotin, “Refining dependencies improves partial-order verification methods,” in *CAV*, 1993, pp. 438–449.
- [35] P. Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems-An Approach to the State-Explosion Problem*, ser. LNCS, 1996.
- [36] D. Peled, “All from one, one for all: On model checking using representatives,” in *CAV*, 1993, pp. 409–423.
- [37] A. Valmari, “Stubborn sets for reduced state space generation,” in *ATPN*, 1991, pp. 491–515.
- [38] K. McMillan, “Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits,” in *CAV*, 1992, pp. 164–177.
- [39] J. Esparza, “Model checking using net unfoldings,” *Science of Computer Programming*, vol. 23, no. 2-3, pp. 151–195, 1994.
- [40] K. Sen and G. Agha, “A race-detection and flipping algorithm for automated testing of multi-threaded programs,” *Lecture Notes in Computer Science*, vol. 4383, p. 166, 2007.
- [41] C. Wang, Z. Yang, V. Kahlon, and A. Gupta, “Peephole partial order reduction,” in *TACAS*, 2008, pp. 382–396.
- [42] K. Sen and G. Agha, “Automated systematic testing of open distributed programs,” in *FASE*, 2006, pp. 339–356.
- [43] G. Gueta, C. Flanagan, E. Yahav, and M. Sagiv, “Cartesian partial-order reduction,” *Model Checking Software*, pp. 95–112, 2009.
- [44] S. S. Vakkalanka, G. Gopalakrishnan, and R. M. Kirby, “Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings,” in *CAV*, 2008, pp. 66–79.
- [45] G. Holzmann, “The model checker SPIN,” *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997.
- [46] 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.
- [47] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.
- [48] “Pi code,” www-unix.mcs.anl.gov/mpi/usingmpi/examples/simplempi/main.htm.
- [49] K. M. Chandy and J. Misra, “Distributed computation on graphs: Shortest path algorithms,” *ACM*, vol. 25, no. 11, pp. 833–837, 1982.
- [50] T. Arts, K. Claessen, and H. Svensson, “Semi-formal development of a fault-tolerant leader election protocol in Erlang,” *LNCS*, vol. 3395, pp. 140–154, 2005.
- [51] “ScalaTest,” <http://www.scalatest.org/>.
- [52] “JUnit,” <http://junit.org/>.

- [53] “Gatling stress tool,” <http://gatling-tool.org/>.
- [54] A. S. Namin and J. H. Andrews, “The influence of size and coverage on test suite effectiveness,” in *ISSTA*, 2009, pp. 57–68.
- [55] E. Trainin, Y. Nir-Buchbinder, R. Tzoref-Brill, A. Zlotnick, S. Ur, and E. Farchi, “Forcing small models of conditions on program interleaving for detection of concurrent bugs,” in *PADTAD*, 2009, pp. 7:1–7:6.
- [56] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte, “A randomized scheduler with probabilistic guarantees of finding bugs,” *SIGPLAN Not.*, vol. 45, no. 3, pp. 167–178, 2010.
- [57] 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.
- [58] J. Huang and C. Zhang, “Persuasive prediction of concurrency access anomalies,” in *ISSTA*, 2011, pp. 144–154.
- [59] M. Christakis and K. Sagonas, “Detection of asynchronous message passing errors using static analysis,” in *PADL*, 2011, pp. 5–18.
- [60] B. Křena, Z. Letko, and T. Vojnar, “Coverage metrics for saturation-based and search-based testing of concurrent software,” in *RV*, 2012, pp. 177–192.
- [61] http://en.wikipedia.org/wiki/Sleeping_barber_problem.
- [62] 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.
- [63] “Geotrellis,” <http://www.azavea.com/products/geotrellis/>.
- [64] “Fyrie redis,” <https://github.com/derekjw/fyrie-redis/tree/akka-1.2>.
- [65] “Signal/collect,” <http://uzh.github.io/signal-collect/>.
- [66] 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.
- [67] <https://github.com/uzh/signal-collect/issues/58>.
- [68] O. Shacham, M. Sagiv, and A. Schuster, “Scaling model checking of dataraces using dynamic information,” in *PPoPP*, 2005, pp. 107–118.
- [69] K. E. Coons, S. Burckhardt, and M. Musuvathi, “GAMBIT: effective unit testing for concurrency libraries,” in *PPOPP*, 2010, pp. 15–24.
- [70] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou, “Modist: transparent model checking of unmodified distributed systems,” in *NSDI*, 2009, pp. 213–228.
- [71] L.-A. Fredlund and H. Svensson, “McErlang: a model checker for a distributed functional programming language,” *SIGPLAN Not.*, vol. 42, no. 9, pp. 125–136, Oct. 2007.

- [72] E. Barlas and T. Bultan, “Netstub: a framework for verification of distributed java applications,” in *ASE*, 2007, pp. 24–33.
- [73] S. Lauterburg, R. K. Karmani, D. Marinov, and G. Agha, “Basset: a tool for systematic testing of actor programs,” in *FSE*, 2010, pp. 363–364.
- [74] H. Kastenberg and A. Rensink, “Dynamic partial order reduction using probe sets,” in *CONCUR*, 2008, pp. 233–247.
- [75] Y. Lei and R. H. Carver, “Reachability testing of concurrent programs,” *IEEE Transactions on Software Engineering*, vol. 32, pp. 382–403, 2006.
- [76] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, “MPI: The Complete Reference (Vol. 1),” *The MPI Core. Second edition*, 1998.
- [77] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir, “MPI: The Complete Reference: Vol 2, The MPI-2 Extensions,” 1998.
- [78] C. Wang, M. Said, and A. Gupta, “Coverage guided systematic concurrency testing,” in *ICSE*, 2011, pp. 221–230.
- [79] 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, Jan. 2002.
- [80] K. Sen, “Effective random testing of concurrent programs,” in *ASE*, 2007, pp. 323–332.
- [81] S. Nagarakatte, S. Burckhardt, M. M. Martin, and M. Musuvathi, “Multicore acceleration of priority-based schedulers for concurrency bug detection,” *SIGPLAN Not.*, vol. 47, no. 6, pp. 543–554, June 2012.
- [82] C.-S. Park and K. Sen, “Concurrent breakpoints,” in *PPoPP*, 2012, pp. 331–332.
- [83] S. Tasiran, M. E. Keremoglu, and K. Muslu, “Location pairs: a test coverage metric for shared-memory concurrent programs,” *Empirical Softw. Engg.*, vol. 17, no. 3, pp. 129–165, June 2012.
- [84] 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.
- [85] E. Deniz, A. Sen, and J. Holt, “Verification and coverage of message passing multicore applications,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 17, no. 3, pp. 23:1–23:31, July 2012.
- [86] P. Joshi, C.-S. Park, K. Sen, and M. Naik, “A randomized dynamic program analysis technique for detecting real deadlocks,” in *PLDI*, 2009, pp. 110–120.
- [87] K. Sen, “Race directed random testing of concurrent programs,” *SIGPLAN Not.*, vol. 43, no. 6, pp. 11–21, June 2008.
- [88] F. Chen, T. F. Şerbănuţă, and G. Roşu, “jPredictor: a predictive runtime analysis tool for Java,” in *ICSE*, 2008, pp. 221–230.

- [89] C.-S. Park and K. Sen, “Randomized active atomicity violation detection in concurrent programs,” in *FSE*, 2008, pp. 135–145.
- [90] F. Sorrentino, A. Farzan, and P. Madhusudan, “Penelope: weaving threads to expose atomicity violations,” in *FSE*, 2010, pp. 37–46.
- [91] 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.
- [92] W. Zhang, C. Sun, and S. Lu, “ConMem: detecting severe concurrency bugs through an effect-oriented approach,” in *ASPLOS*, 2010, pp. 179–192.
- [93] 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.
- [94] Q. Gao, W. Zhang, Z. Chen, M. Zheng, and F. Qin, “2ndstrike: toward manifesting hidden concurrency typestate bugs,” in *ASPLOS*, 2011, pp. 239–250.
- [95] 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.
- [96] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam, “Maple: a coverage-driven testing tool for multithreaded programs,” in *OOPSLA*, 2012, pp. 485–502.
- [97] Y. Lei and W. E. Wong, “A novel framework for non-deterministic testing of message-passing programs,” in *HASE*, 2005, pp. 66–75.
- [98] S. Park, S. Lu, and Y. Zhou, “Ctrigger: exposing atomicity violation bugs from their hiding places,” *SIGPLAN Not.*, vol. 44, no. 3, pp. 25–36, Mar. 2009.
- [99] C. Flanagan and S. N. Freund, “Adversarial memory for detecting destructive races,” in *PLDI*, 2010, pp. 244–254.
- [100] J. Cuny, G. Forman, A. Hough, J. Kundu, C. Lin, L. Snyder, and D. Stemple, “The ariadne debugger: scalable application of event-based abstraction,” in *WPDD*, 1993, pp. 85–95.
- [101] M. Yannakakis, “The Complexity of the Partial Order Dimension Problem,” *Siam Journal on Algebraic and Discrete Methods*, vol. 3, 1982.