State Exploration of Scala Actor Programs

by

Mirco Dotta

BSc., Computer Science
École Polytechnique Fédérale de Lausanne (2006)

Submitted to the School of Computer and Communication Sciences
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

at the

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

March 2009

Author .....
School of Computer and Communication Sciences
March 13, 2009

To my mother, Giusy.

State Exploration of Scala Actor Programs

by

Mirco Dotta

Submitted to the School of Computer and Communication Sciences
on March 13, 2009, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science

Abstract

The growing use of multicore and networked computing systems is increasing the importance of developing reliable parallel and distributed code. Unfortunately, developing and testing such code is notoriously hard, especially for shared-memory models of programming. The actor model offers a promising alternative paradigm based on message passing. Essentially, an actor is an autonomous concurrent object which interacts with other actors only by exchanging messages. In actor-based systems, the key source of non-determinism is the order in which messages are delivered to—and processed by—the actors. Bugs can still occur in actor programs as the interleaving of messages may be incorrect, or the sequential code within an actor can have bugs.

We developed a general framework, called SEJAP, for exploring possible message schedules in actor systems compiled to the Java bytecode. Specifically, in this dissertation, we present one instantiation of SEJAP for the actor library of the Scala programming language. To the best of our knowledge, this is the first framework that allows systematic exploration of Scala actor programs. We also present two optimizations that alleviate the state explosion problem typical for such exploration, and thus speed up the overall exploration of actor programs in SEJAP. We have implemented our framework, Scala instantiation, and optimizations in Java PathFinder, a widely used model checker for Java bytecode developed by NASA. Preliminary results show that SEJAP can effectively explore executions of actor programs. Further, our use of SEJAP already discovered a previously unknown bug in the sample code from a popular site for Scala.

Thesis Supervisor: Viktor Kuncak
Title: Assistant Professor

External Thesis Supervisor: Darko Marinov
Title: Assistant Professor

Acknowledgements

I would like to express my gratitude to my advisor here at UIUC, Darko Marinov, for accepting me as a visiting student in his group, for giving me the freedom of pursuing my personal research interests and for providing me tremendous support throughout this unique experience.

I'm also grateful to my advisor at EPFL, Viktor Kuncak, for giving me the opportunity to come to UIUC to carry out my research and for his continuous support from EPFL. I was very fortunate in meeting Viktor during my first year of Master studies at EPFL. I thank him for believing in me ever since we met and for pushing me to learn the many nuances of software verification.

I'm greatly thankful to Steven (Steve) Lauterburg—a PhD student in Darko's group—with whom I worked very closely during my entire visit at UIUC. Many of the ideas presented in this dissertation have in fact been the realization of this collaboration. Steve spent a lot of time in discussing various issues with me, pair programming SEJAP, and providing many valuable comments about several draft versions of this dissertation. I wish him the very best of luck with his work and life. I also would like to thank Yun-young Lee, with whom I shared an office, for making every working day enjoyable. Besides being a wonderful friend, her baking skills were a valuable source of energy during the night and weekends spent working on this research.

I would like to thank professor Gul Agha for the many fruitful discussions on the actor model, and for giving me the opportunity to audit his class which has greatly helped me in realizing this work. I also would like to sincerely express my gratitude to Philipp Haller for his constant availability and dedication in answering the many

questions I had on the semantics of actors in Scala. I'm also deeply grateful to Cédric Jeanneret for providing me with comments and criticisms that helped improve the quality of this dissertation.

I would like to thank professor Ralph Johnson for finding the time to personally discuss several software engineering topics and for giving me the opportunity to audit his class on object-oriented programming and design. I'm also grateful to Sen Koushik for meeting with me during my brief visit at UC Berkeley, and sharing his vision on the area of automatic software testing.

There are many extraordinary individuals whom I have met in my life and that have shaped me into the person I am today. Due to the lack of space I cannot mention everyone here but you all are in my heart and memories. Nicola Ermotti and Samuele Gantner have played fundamental roles in my growth, I will never be able to express the real extent of my gratitude. Samuele also provided an incredible number of comments on this dissertation, I owe him a big thanks. Also thanks to my fellow "blogger-friends" who supported me constantly during this adventure abroad. Thanks to Martina Börner, Claudia Comin, Stefano Giordano, Guy Müller, Gisella Pfund, Dario Poggiali, and Luca Sulmoni. Without you my life would not be as much fun.

Last but not least, I would like to thank my family for their unconditional love, support, encouragement, and patience. This work is dedicated to them.

Table of Contents

1 Introduction **1**

1.1 Problem Statement . . . 3

1.2 Proposed Solution and Contributions . . . 4

1.3 Organization of this Dissertation . . . 5

2 Background **7**

2.1 The Actor Model . . . 7

2.1.1 Actors in a Nutshell . . . 7

2.1.2 Actor Systems . . . 9

2.2 The Scala Programming Language . . . 10

2.2.1 Scala in Brief . . . 10

2.2.2 Pattern Matching . . . 10

2.2.3 Actors by Examples . . . 11

2.2.4 Impurities of the Library . . . 14

2.2.5 Actor Semantics . . . 15

2.3 Java PathFinder . . . 17

2.3.1 A Model Checker for Java Bytecode . . . 17

2.3.2 JPF Architecture and Design . . . 18

3 Example **21**

3.1 A Simple Client-Server Program . . . 21

3.2 Description of the Space Exploration . . . 24

4 Framework **27**

4.1 Actor States . . . 27

4.2 Actor Execution . . . 28

4.3 Message Management . . . 29

4.4 Exploration Algorithm . . . 30

4.5 Library Instantiation . . . 32

4.5.1 Binding a Concrete Actor Library to the Framework 32
4.5.2 Adapting the Library Interface 34

5 Optimizations **37**
5.1 State Comparison 38
5.2 Step Granularity 39

6 Implementation **41**
6.1 System Overview 42
6.2 Java PathFinder Modifications 42
6.3 Java PathFinder and Scala 44
6.4 Adapting the Scala Actor Library 46

7 Evaluation **49**
7.1 Optimized Thread Context Switching 50
7.2 Subjects 50
7.3 State Comparison 52
7.4 Step Granularity 54

8 Related Work **55**

9 Conclusions **59**
9.1 Future Work 60

A Original Client-Server ScalaWiki Code **63**

List of Figures

2-1 Representation of an actor 8

3-1 Buggy client-server code sample 22
3-2 Possible message schedules and final states 23
3-3 State space for the example program 25

4-1 UML class diagram: Core and Adapter layers 33
4-2 Skeleton of an adapter class for an actor library. 35

6-1 Framework's stack. 42
6-2 UML class diagram of the Scala actor library. 45

List of Tables

7.1 Evaluating different state comparison implementations 53
7.2 Comparing step granularity – *Big* vs. *Little* steps 54

List of Abbreviations

FIFO	First-In First-Out
IDE	Integrated Development Environment
JNI	Java Native Interface
JPF	Java PathFinder
JVM	Java Virtual Machine
JVM^{JPF}	Java PathFinder's Java Virtual Machine
MJI	Model Java Interface
OS	Operating System

Chapter 1

Introduction

Computing is undergoing a historic change as processor clock speed is not significantly increasing any more but instead processors shift to having multiple computing cores. To fully exploit multicores to improve the performance of applications, software developers need to write parallel code (either from scratch or by parallelizing existing sequential code). One category of parallel code is based on shared data, where multiple threads of computation communicate by reading and writing shared objects. For example, the Java programming language [43] provides support for threads in the language and libraries, with shared data residing on the heap. However, multithreaded code is notoriously hard to get right, with common concurrency bugs including dataraces, atomicity violations, and deadlocks. An alternative approach for writing parallel and distributed code is message passing, where multiple entities of computation communicate by exchanging messages.

In the *actor programming model* [3, 4], each computation entity (called an actor) has its own control and state, and communicates with other actors by explicitly sending messages. Existing actor-oriented programming systems include Charm++ [48], Erlang [69], E [19], Newspeak [12, 59], Ptolemy II [65], Revactor [67], SALSA [80], Scala [61, 71], and THAL [49], as well as a number of libraries for Java such as ActorFoundry [2], Jetlang [42], Jsasb [47] and Kilim [77]. An actor program can have different executions (even for the same input) based on the interleaving of messages exchanged between the actors, in much the same way as a multithreaded program can

have different executions based on the interleaving of accesses to the shared memory.

While actor programming avoids some of the bugs inherent in shared-memory programming, e.g., low-level dataraces involving access to shared data, actor programs can still have bugs due to incorrect interleaving of messages or incorrect sequential code within an actor. Identifying and eliminating these bugs becomes remarkably important as the use of the actor model increases for parallel and distributed programming. Systematic testing and model checking are approaches that help in improving reliability by automatically searching for potential bugs. A key to these approaches is *state-space exploration* [17] that searches through the possible executions of a program. Previous research on model checking for actor or distributed systems [7, 8, 10, 23, 74, 86] focuses on (stateless) checking of programs in one specific system and not on providing a general platform for checking and experimentation.

This dissertation describes SEJAP, a framework for State Exploration of Java-based Actor Programs. Instead of building a tool for one specific actor implementation, we *developed a general exploration framework* that can support several actor systems with a small adaption layer required for each system. The interface between the SEJAP core and the adaption layers is designed both to allow *direct exploration of unmodified application code* and to ensure that optimizations for exploration (such as customized actor state comparison or partial order reduction [17]) can be made available to multiple actor systems by changing only the SEJAP core. To prove the generality of our approach, we instantiated this framework for two actor-based systems: the ActorFoundry library [2] for the Java programming language, and the actor library for the Scala programming language [61]. Specifically, this dissertation will focus on the Scala instance.

Scala is a modern programming language that combines the object-oriented and functional paradigms. Scala blends together the best practices of these two worlds and provides the developer with several advanced abstractions [62] such as type members, explicit selftypes, and mixin composition, with the goal of providing better language support for creating modular, reusable components.

Scala enables actor concurrency through a flexible actor library. However, it is

important to note that we are concerned with the *checking (through state-space exploration) of Scala actor application code*, not of the actor library itself. The adaption layer that we developed replicates the behavior of the original actor library from the point of view of the application code. Therefore, SEJAP does not check the actual Scala library code but focuses instead on exposing potential bugs in the application code due to message scheduling, which is the source of non-determinism in actor-based programs.

Moreover, we choose to *build on an existing exploration tool*, to foster adoption and integration with related results. In particular, we build on the Java PathFinder (JPF) tool [44, 82]. JPF is an explicit-state model checker for Java bytecode [44, 82]. It was developed at NASA and has been used in numerous research projects (a partial list is available online [45]), primarily for checking programs written directly in the Java programming language. In contrast, we extended JPF and applied it to Scala, which also compiles to Java bytecode. JPF provides a specialized Java Virtual Machine (JVM^{JPF}) that supports state backtracking and control over non-deterministic choices such as thread scheduling. Prior to our work, JPF did not have any direct support for actors, e.g., for high-level choices such as message scheduling. We chose JPF as our implementation platform due to its popularity and because it can execute compiled Scala code, but our techniques would be applicable in other explicit-state model checkers for real code, e.g., BogorVM [68], CMC [57], SpecExplorer [81], or Zing [6].

1.1 Problem Statement

The actor library available in the Scala programming language offers a powerful abstraction to develop concurrent software. Testing is very useful for ensuring that the concurrent software behaves correctly. However, testing concurrent software is usually very hard because it is difficult to explore all possible interleavings that may affect the program execution. Software model checking is a technique used to *systematically explore* a large number of executions of a concurrent program, and it has been suc-

cessfully applied to the verification of concurrent, as well as distributed, applications. To date, no tool supports systematic exploration for Scala actor programs.

1.2 Proposed Solution and Contributions

This dissertation presents an approach to state-space exploration of Scala actor programs. Specifically, we describe a Scala instantiation of SEJAP [52], our generic, library-independent framework for checking actor programs.

This dissertation makes several contributions.

- The first contribution is the description of the design of SEJAP, the framework that we developed for *state-space exploration* of actor programs.
- The second contribution is the instantiation of SEJAP for the Scala actor library. To the best of our knowledge, this is the first tool that enables state-space exploration of Scala actor programs.
- The third contribution is the implementation of SEJAP as an extension to JPF. As part of the implementation we also integrated in the framework two global optimizations:

  1. A customized actor state comparison, which can help mitigating the state explosion problem and generally makes the overall exploration faster.
  2. Two different step granularity tactics, which impose a trade-off between faster straight-line execution and efficient overall exploration.

  Prior to this work, JPF did not have any special capability to efficiently explore Scala actor programs.
- The fourth contribution is an evaluation of our implementation on a number of programs, demonstrating the effectiveness of our approach.

Our use of SEJAP already discovered a previously unknown bug in a code sample available on the ScalaWiki website [72], which we have included in Appendix A.

1.3 Organization of this Dissertation

The rest of this dissertation is organized as follows: Chapter 2 gives a brief overview of the actor programming model, the Scala programming language, and the Java PathFinder model checker. Chapter 3 presents a real Scala application that had a previously unknown bug, and we describe how our verification framework helped us identify this bug. Chapter 4 covers the main aspects of the framework that we built. Chapter 5 presents some optimizations that we integrated to speed up the exploration. Chapter 6 describes how we modified the Scala actor library to integrate it in our framework. Chapter 7 evaluates our approach on five examples. Finally, Chapter 8 discusses related work, and Chapter 9 concludes and presents ideas for future work.

Chapter 2

Background

This chapter starts with a discussion on the foundation of the *Actor Model* in Section 2.1. Section 2.2 briefly presents the Scala programming language in general and, in particular, introduces the Scala *actor library* through a number of simple examples. Section 2.3 presents the key aspects of Java PathFinder, a widely used model checker for Java bytecode, on top of which we implemented our framework for state-space exploration of actor programs.

2.1 The Actor Model

2.1.1 Actors in a Nutshell

The actor model was originally proposed by Hewitt in the early 1970s [34] where the term *Actor* was used to describe the concept of an autonomous agent in Distributed Artificial Intelligence. Over the years, several contributions [3, 4, 27, 33] have refined the notion of actor into a rigorous mathematical model for concurrency. Essentially, an actor is an autonomous concurrent object which interacts with other actors only by exchanging messages. By default, these messages are *asynchronous*. Other forms of communication, e.g., synchronous remote-procedure-calls, are defined in terms of asynchronous messages [3]. Each actor has a unique *actor name* (its virtual address) and mailbox (or mail queue). Figure 2-1 depicts the different parts of an actor.

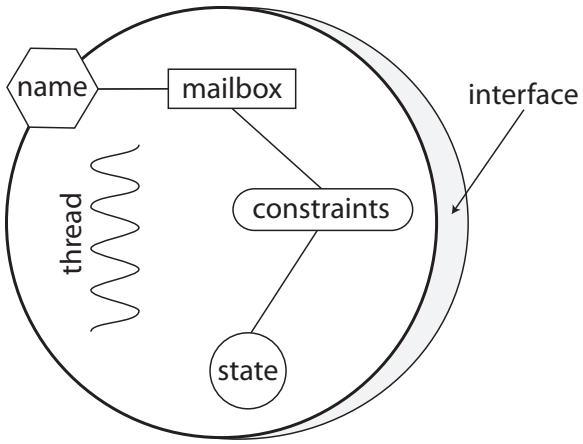

Figure 2-1: Representation of an actor

If an actor is busy, messages sent to it are queued in its mailbox. When an actor is done processing a message, it checks its mail queue for another message. In response to processing a message, an actor may do one or more of these three actions:

- Send messages to other actors or to itself.
- Create new actors that have their own unique names and mail queues.
- Update the local state to prepare for processing the next message.

Each actor operates asynchronously. Consequently, if two actors send independent messages (i.e., the sending events are not causally related), the order of arrival of these messages is indeterminate. Moreover, no assumption is made about the routing of messages; therefore, by default, two messages sent from the same actor may arrive in any order. On the other hand, when the message is actually delivered to the actor, its computation is completely deterministic. Specifically, actors have a *history-sensitive* behavior, meaning that the current behavior of an actor is determined by the sequence of messages that it has received and processed. Another interesting property of actors is their *isolation*: an actor is unaware whether it is executing alone or concurrently with other actors and, indeed, does not need to know. Since actors only update their local variables and do not share state, the behavior of an actor is completely independent of external events generated by the system where the actor resides. This

entails that actors can be executed atomically or, equivalently, that their execution respects a *macro-step semantic* [4, 74]. In the macro-step semantic, the execution of an actor from the receipt of a message up to the next receipt takes place consecutively without interleaving with any other actor.

A last interesting aspect of actors is that, by nature, they emphasize the independence of what is done (the *interface*) from how it is done (the *representation*). On the one hand, the interface of an actor consists in the shape of messages that the actor is able to receive and process. On the other hand, the representation determines how the local state of the actor evolves. Encapsulation and modularity are hence two fundamental characteristics of actors, which allow better reuse and composition of fine-grained actors into coarse-grained ones. Essentially, actors enable *separation of concerns* and simplify reasoning about complex, potentially concurrent, systems.

2.1.2 Actor Systems

Several actor-oriented programming systems exist nowadays, e.g., Charm++ [48], Erlang [69], E [19], Newspeak [12, 59], Ptolemy II [65], Revactor [67], SALSA [70], Scala [61, 71] and THAL [49], as well as a number of libraries for Java such as ActorFoundry [2], Jetlang [42], Jsasb [47] and Kilim [77]. Interestingly, these concrete implementations all differ from the theoretical actor model. For instance, actor languages typically provide higher level language constructs for synchronous (or request-reply) communication, where a value is returned by the actor receiving a message; the sending actor waits until this value is received before carrying out further computation. Also, in many actor languages, the developer may specify *synchronization constraints* [36], which allow the enforcement of an order on the sequence of messages processed by the actor. Nonetheless, these constructs can be translated into basic actor primitives [38], and hence are not part of the theoretical model.

2.2 The Scala Programming Language

2.2.1 Scala in Brief

Scala [60, 61, 71] is a general purpose programming language developed by Martin Odersky and his team at EPFL. The language's first official release dates back to November 2003 [60].

In the words of its author:

> *Scala is a general purpose programming language designed to express common programming patterns in a concise, elegant, and type-safe way. It smoothly integrates features of object-oriented and functional languages.*

Scala is tailored to both the Java Virtual Machine (JVM) and the Common Language Runtime[1](CLR), and shares with Java and C# a similar syntax for many language constructs (e.g., class definition and control structures). However, Scala's unified object model[2] enables a new style of programming which is both more concise and allows for greater reuse. An important addition to the language consisted of the integration of an *actor library* (since version 2.1.7[3]), which enabled Erlang-style concurrency [69].

In Section 2.2.3 we cover the basic building blocks of the Scala actor library. This is a recommended reading to understand the code example presented in Chapter 3. We now introduce *pattern matching*, a specific Scala programming language feature that we will extensively use in all the code examples described in this dissertation.

2.2.2 Pattern Matching

Pattern matching is a widely used conditional construct in programming languages, particularly in functional ones, e.g., Erlang [69], Haskell [79], ML [1], or OCaml [9]. It specifies a computation as a sequence of tests on the shape of a given value and

[1]The Common Language Runtime is part of the Microsoft .NET framework.
[2]Every value is an object and every operation is a method invocation.
[3]At the time of this writing the official Scala release is version 2.7.3.

binds variables to value components if the test succeeds, generalizing if-then-else and case statements.

A pattern matching expression with n patterns takes the following form in Scala:

---

s **match** {

  case $p_1 \Rightarrow expr_1$

  case $p_2 \Rightarrow expr_2$

  ...

  case $p_n \Rightarrow expr_n$

}

Patterns p_1, p_2, ..., p_n are tested against the selector s sequentially. As soon as a pattern matches the selector's shape, the pattern variables are bound and the entire match expression evaluates to what is on the right-hand side, which, in turn, can be any valid Scala expression. If no pattern matches the selector, a runtime exception is reported to the user. The patterns p_i allowed in a pattern matching expression vary a lot, ranging from simple (primitive[4]) values to complex compound objects.

Semantically, Scala treats blocks of pattern matching cases as instances of partial functions. Partial functions are functions that are defined only in some part of their domain. Actors use partial functions to concisely define their behavior, as we will show in the examples of the next subsection.

2.2.3 Actors by Examples

In this subsection we start exploring the Scala actor library. Specifically, we will present how actors can be created and started, along with the basic building blocks for actor communication (i.e., message send and receive).

In Scala, an actor is any object whose class realizes the `Actor` trait. A trait resembles a Java abstract class, in the sense that it can contains both concrete and abstract methods. However, unlike Java abstract classes, traits can be composed

[4]Primitive types in Scala are exactly the same as in Java i.e., integer, boolean, char, double, float, long, and byte.

together and they allow for broader class reuse (this language feature is called mixin-class composition, see [61] for more details).

A class that extends the `Actor` trait needs only to implement the `act` method. This method defines the initial behavior of the actor. The `Actor` trait also contains a method named `start` which is the entry point for the actor's execution. When an actor is started it implicitly calls its `act` method.

The simplest possible behavior that an actor can have is to do nothing or, as in the below example, output a message on the standard output.

```
class MyFirstActor extends Actor {
    def act = println("I'm an actor")
}

val anActor = new MyFirstActor
anActor.start // will implicitly call the actor's 'act' method
```

Despite its evident simplicity, this example already shows an interesting particularity of the library. We can observe that starting an actor is a two-steps operation involving a) the creation of the actor, and b) a call to `start` on the freshly created instance. When started, the actor begins its computation by executing the body of the `act` method. It is important to note that messages sent to a non-started actor are queued in its mailbox and will be processed only after the actor is started.

The library also offers some syntactic alternatives to create actors. For instance, the previous example is equivalent to the following one-line statement.

```
val anActor = actor { println("I'm an actor") }
```

The `actor` expression will create a new actor, define the body of `act` with the sequence of statements inside the block (which in this example consists only of printing on the standard output), and immediately start the actor.

The main characteristic of actors is that they can send and receive messages; the next example introduces the syntax for these operations.

```
val anActor = actor { receive { case msg ⇒ println(msg) } }
actor { anActor ! "Hello" //asynchronous send }
```

Here, we first create an actor that waits until a message is received. Then, a second actor sends message "Hello" to `anActor`, which will react by printing the value on the standard output. The example shows that in Scala receiving a message is an explicit operation. Hence, an actor will process only as many messages as the number of `receive` statements defined in its body (fortunately, the library provides combinators for creating an infinite chain of reactions). In this example, `anActor` terminates its life-cycle upon receipt of the first message. Moreover, the actor that sends the "Hello" message ends its computation immediately after having executed the send. This is an important variation from the theoretical model [4], where actors receive messages implicitly and stay idle when their mailboxes are empty.

The library also offers a second solution for receiving messages. Namely, instead of using `receive` we could have used `react` and obtained the very same execution. The difference between the semantics for these two operations consists in the returned values (note that `receive` and `react` are both standard function calls). While `receive` returns normally, `react` always throws an exception at the end of its execution[5]; entailing that any instruction that follows will never get executed. For example, the following definition of `anActor` has the very same behavior as the previous one.

```
val anActor = actor {
                react { case any ⇒ println(any) }
                assert(false) // unreachable code
              }
```

Because of the particular semantics of react, the failing assertion is never executed. At first glance, this might appear to be a surprising execution. However, the motivation is in the core of the Scala actor library. By now, it should be clear that

[5]This is done to detach the thread from the actor, as will be explained later in this section.

actors are entities which may (potentially) execute in parallel. Since Scala compiles to Java bytecode, the only way to execute two or more actors concurrently is by running them on distinct threads. However, threads in Java are costly (since they are mapped to heavy OS threads) and need to be used with care, since a large number of them can quickly deteriorate the overall performances of the system. To help mitigate this cost, the Scala actor library provides two alternatives for mapping actors to threads. When a developer uses `receive`, the library maps the actor to a unique thread, which is held even when the actor cannot progress, e.g., no message is available. Alternatively, when `react` is used, the actor frees (at the end of its computation) the thread that it is using, so that the thread can be reused. (The library contains a component that manages a dynamic pool of threads for executing actors.) This entails that the total number of threads is usually considerably lower than the overall number of actors in the system. The interested reader can find more insights concerning the library implementation in [29, 30].

2.2.4 Impurities of the Library

In Section 2.1.2 we briefly mentioned that concrete implementations of actor systems tend to differ in several aspects from the theoretical model. Scala is no exception, and it is important to know and understand these differences. The main reason being that those additional "language features" can result in subtle bugs in the application, when the developer is unaware of their existence.

First, in Section 2.1 we stated that, because actors execute atomically, multiple actors running in parallel do not interfere. In other words, actors are ensured to be *isolated* one from the other and, in general, from the rest of the system. However, this property is not currently ensured in Scala[6]. A developer is not prevented from creating an actor that exposes part of its state. This is not a problem in itself, as long as the part of the actor state that gets exposed is immutable. On the other hand, if it is mutable, classic shared-memory issues arise since multiple threads can potentially access, concurrently, the visible members (forcing the developer to resort

[6] A type system to ensure isolation is under development.

to lock/unlocking strategies to regulate access to the contended objects).

The lack of isolation is also responsible for a second critical issue, which is related to the cost of sending messages. Specifically, in many actor languages and libraries, the cost of message passing is minimized by transferring ownership rather than by copying data. Unfortunately, these actor languages and libraries generally require the programmer to ensure that passing messages by transferring data ownership is correct (i.e., that the sender does not subsequently attempt to access the data). Evidently, this requirement can be a source of bugs. An easy solution to this problem is to ensure (again) that only immutable messages are exchanged among actors. However, this might not always be possible. An alternative is to resort to static verification techniques to identify read/write operation on shared messages. Specifically, the Scala team is considering the integration of linear types [83], which would ensure that only one reference to any object exists at a time. Consequently, sending a message would also transfer its ownership.

Finally, in one of the examples in Section 2.2.3 we have already seen that because of the particular semantics of `react`, it is possible to create unreachable code. This might lead to hard-to-explain executions, since the developer has to be aware of the semantics of each actor's method.

2.2.5 Actor Semantics

In this subsection, we present the semantics of the actor's methods defined in the `Actor` trait. It is important to remark that the correct semantics of these methods is only ensured if the methods are called within the scope of another actor[7]. This is enforced by throwing a runtime exception in case of violations. The only exception to this rule are methods `start` and `act`, which are used to initialize the life-cycle of an actor and to define its initial behavior.

The `Actor` trait includes the following methods:

- `act()` Define the behavior of the actor.

[7]Therefore, the existence of a *this* value that refers to the executing actor is assumed.

- `start()` It is the starting point of the actor's execution. When an actor is started it implicitly calls its `act` method.

- `mailboxSize()` Return the number of messages queued in the actor's mailbox.

- `recv.send(msg, sender)` Send (asynchronously) a message to actor `recv`, and explicitly specify the sender actor for the sent message. From the perspective of the actor that receives the message, `sender` is the actor that has sent the message.

- `recv.!(msg)` Send asynchronously a message to actor `recv` and (implicitly) attach the sender's reference to the message payload (equivalent to `rcv.send(msg,this)`).

- `recv.!?(msg)` Send synchronously a message to actor `recv` and (implicitly) attach the sender's reference to the message payload. The actor that sent the message then pauses its execution until actor `rcv` sends back a reply.

- `recv.!?(msg, msec)` Same semantic as `!?`, with the important difference that the sender actor is forced to continue its computation if it doesn't get a reply within the given time span of `msec` milliseconds.

- `sender()` Return the reference to the actor that is associated with the message that is currently processed by *this* actor.

- `recv.forward(msg)` Send (asynchronously) a message to actor `recv` and (implicitly) attach to the message's payload the reference of the sender actor that is associated with the message currently processed by *this* actor (equivalent to `rcv.send(msg,sender())`).

- `receive(f)` Takes out of the actor's mailbox a message that is in the domain of function `f`, and applies `f` to the message. If no message is available, then pause the actor until a message is received.

- `receiveWithin(f, msec)` It has the exact same semantic as `receive`; with the important difference that if no message is delivered to the current actor within a time span (specified in milliseconds), the system generates a *timeout* message and the actor starts immediately processing it. *timeout* has to be in `f`'s domain or the actor will continue to wait.

- `react(f)` It has a semantics similar to `receive`. However, when function `f` completes its computation, an exception is always thrown.

- `reactWithin(f, msec)` Works in the very same way of `receiveWithin`; but when `f` terminates an exception is always thrown.

- `?()` Takes out of the current actor mailbox a message and returns it. If no message is available the actor blocks until a new message is enqueued.

- `link(to)` Add actor `to` to the list of actors known by current actor. Also add the current actor to the list of actors known by `to`. `link` can be used to coordinate an action among a group of (linked) actors.

- `unlink(from)` Inverse of calling `link`; mutually detach actor `from` from the current actor.

- `exit()` Immediately terminates the execution of the current actor. For each linked actor (with state variable `trapExit`[8] set to true) send (asynchronously) an *exit* message to inform them that the current actor is shutting down.

2.3 Java PathFinder

2.3.1 A Model Checker for Java Bytecode

Java PathFinder [44, 82] (JPF) is an explicit-state model checker for Java bytecode developed at NASA Ames since 1999 (the project became open source in 2005). In

[8]This is a member of the `Actor` trait, hence it is inherited by all actors.

its first incarnation, JPF was implemented as a translator from Java to the Promela language of the SPIN model checker [32]. This strategy soon proved to be inappropriate since each feature in Java needed to be encoded into a semantically equivalent Promela feature (and this was not always feasible). Thus, the decision was made to completely rewrite JPF as a customized model checker able to directly execute Java bytecode. We next describe this new version.

2.3.2 JPF Architecture and Design

JPF is implemented in Java and at its core is a specialized Java Virtual Machine (JVM^{JPF}) that can execute Java bytecode. This enables the model checker to have multiple hooks in the runtime, which allow it to obtain control over the program execution. Specifically, the JVM^{JPF} is invoked from the model checker to interpret the bytecode while the different traces of the program are being explored. The fact that JPF is executed on top of a standard JVM (hence the program that is verified in JPF is interpreted) results in a large execution overhead. However, JPF aims to explore *all possible executions of a concurrent application*, with the intent of making the *overall exploration fast*. To this end, the JVM^{JPF} allows state backtracking and the entire state of the application is saved each time that a non-deterministic choice (e.g., thread scheduling) is faced. Specifically, each state consists of all full stack frames for every thread, the entire heap, and the static information about the loaded classes. JPF uses a "collapse" algorithm to transform a concrete state into a list of indexes indicating which components compose the state itself. A fundamental aspect of the transformation is that the decomposition must be unique, so that by comparing the indexes it is possible to determine state equality. This special representation used to store the program's state allows JPF to achieve fast backtracking. Lastly, to alleviate the state space explosion problem, JPF provides [82] several verification techniques such as partial order reduction, symmetry reduction, slicing, abstraction, and runtime analysis.

JPF is able to execute any *pure* Java program. However, not all programs consist of only pure Java code. Java classes that contain *native* (to the system) methods

cannot always be handled directly by JPF. The issue is that these native methods are usually implemented in languages such a C/C++, and thus their body cannot be interpreted as is by JPF. Consequently, JPF cannot keep track of the state information, and JPF will fail when attempting to execute the code. Unfortunately, any interesting Java application is likely to be impure, since many Java library classes contain native methods[9]. Therefore a solution to this problem is mandatory. Fortunately, in many cases it is not too difficult to create Java code that emulates the behavior of the native operation. For instance, wrappers can be built to match the native operations.

We already mentioned that one of the most important drawbacks of JPF is that the program is interpreted, which imposes a significant overhead to the verification task. To alleviate this problem, JPF allows portions of the program to execute directly on the host JVM. This mechanism is implemented through the Model Java Interface (MJI). The MJI is a powerful component of the JPF architecture that enables communication between the JVM^{JPF} and the underlying host JVM. The design goals of the MJI strictly resemble the ones of the well known Java Native Interface [55] (JNI), which allows Java code running in the JVM to call and be called by native applications and libraries written in other languages, e.g., C/C++. When executing parts of the program on the host JVM, JPF cannot keep track of (potential) modifications to the state of the program; therefore the developer must ensure that no relevant information is lost.

The interested reader can find more information about JPF in [31, 44].

[9]For instance, many of the Java library classes that provide file I/O (java.io and java.nio), user interface (java.awt and java.swing), and networking communication (java.net) functionalities contain several native methods.

Chapter 3

Example

This chapter presents a simplified version of a sample actor program available on ScalaWiki [72], a popular website that provides a number of widely used resources for Scala, including code samples contributed by some developers of the Scala programming language. Using our framework, we discovered a bug in one of these samples. Section 3.1 presents the example and describes the possible executions that lead the system to an inconsistent state. Section 3.2 provides some insights into the framework that we built and describes how it achieves systematic exploration of actor program executions.

3.1 A Simple Client-Server Program

Figure 3-1 shows the code for a simplified version of the real sample code. The *server* actor accepts three kinds of messages: `Set`, `Get`, and `Kill`. This actor simply stores and retrieves an integer value[1]. In addition to storing and retrieving the value, the server actor can also process a `Kill` message, which instructs the actor to terminate itself by invoking the `exit` actor library method.

The program starts from the `main` method, where a *client* actor is created and automatically starts executing. The client first asynchronously sends (using the bang '`!`'

[1]The actual server from the ScalaWiki code was keeping track of an inventory of items with specified prices and quantities.

```
1  object ClientServer {
2    case class Set(v: Int)
3    case object Get
4    case object Kill
5
6    def main(args: Array[String]) = {
7      val client = actor {
8        server ! Set(1)
9        val v1 = (server !? Get).asInstanceOf[Int]
10       val v2 = (server !? Get).asInstanceOf[Int]
11       // assert (v1 == v2)
12       server ! Kill
13     }
14   }
15
16   val server = actor {
17     var value = 0
18     loop {
19       react {
20         case Set(v) ⇒ value = v
21         case Get ⇒ reply(value)
22         case Kill ⇒ exit()
23       }
24     }
25   }
26 }
```

Figure 3-1: Buggy client-server code sample

operator) a `Set` message to the server to store value 1. The client then queries (twice) the server and retrieves the current integer `value` maintained by the server. (Note the use of the `!?`, which enables synchronous—blocking—actor communication.) The original source assumed `v1` and `v2` to be identical[2]; we could have made this explicit by inserting an assertion. Finally, the client sends a `Kill` message to the server and terminates its execution.

The cause of problems in this example is the *order of message deliveries*. Actor systems, including Scala, usually do not guarantee in-order delivery of the messages.

[2]The actual code on ScalaWiki was also retrieving the inventory of items twice but performing two different computations on the inventory.

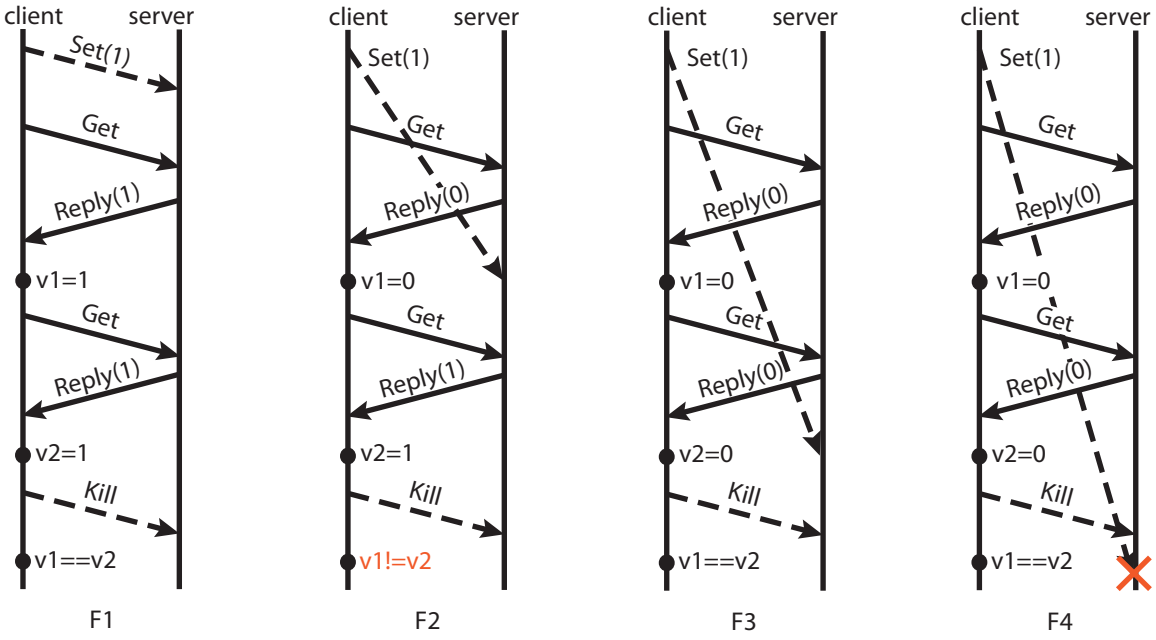

Figure 3-2: Possible message schedules and final states: correct, incorrect, correct, warning

In other words, the default communication channels among the actors are *not FIFO*. However, anecdotal experience shows that assuming in-order delivery is a common cause of programming errors in many actor programs. In our example, for instance, the server need not process the message `Set` before the first `Get`.

Figure 3-2 shows some possible executions for our example program. Specifically, if the server interleaves processing `Get` between the two `Set` messages, it will return inconsistent values for `v1` and `v2`. This faulty execution can be observed in the execution trace labeled `F2` in Figure 3-2. While this execution is not very likely[3], it is possible. The program therefore may expose an *atomicity violation*. (The Scala developers confirmed the bug that we reported for their code on ScalaWiki). Also observe that in Figure 3-2 we used a `Reply` message to carry the returned payload of a synchronous call; however, this type of message does not get created in the real program (the value is simply returned with no message wrapper) and is included here to clarify the description of the example.

[3]We ran our example code on the standard Scala run-time 1000 times, and it always processed message `Set` before `Get`.

3.2 Description of the Space Exploration

In the framework that we developed, the program exploration starts from the `main` method and explores all non-deterministic choices that arise when several messages can be delivered. We assume for this example a depth-first strategy for driving the exploration, but other strategies are also possible. Figure 3-3 shows the state space that is explored for the above example code. Each state of an actor program consists of the *actors' state* (in our example, the field `value` in the server and the variables `v1` and `v2` in the client) and a *message cloud* (a multiset of all messages that have been sent but not yet delivered). In Figure 3-3, a slash denotes an undefined state variable. Each arrow corresponds to the delivery of a message to an actor, and a dashed arrow is used to emphasize that the new state has been already visited (which entails that state comparison is enabled). Thick states (labeled with F_N) are final states.

In this example, our framework explores 20 states and finds two potential bugs. In the state labeled `E` (for "error") in Figure 3-3, the values of `v1` and `v2` differ, being 0 and 1. Figure 3-2 helps visualize this fact. If the `assert` statement in Figure 3-1 was uncommented, we would have reported an error and stopped the exploration for that trace. Additionally, in the state labeled `F4`, the framework reports a *warning* because the server actor is not alive even though there are undelivered messages for that actor. This case occurs when the server actor processes `Kill` before `Get`, as shown in Figure 3-2.

Interestingly, this example already demonstrates the importance of identifying states that have been visited previously. In fact, equivalent states lead to the same execution and thus, they are of no interest if they have already been executed once. For example, in Figure 3-3, `B` and `D` can be visited through various executions. To effectively spot these similarities we implemented a customized *state comparison* for actor programs. We describe this first optimization in Section 5.1. Without this, we would have explored the executions from `B` to `F2` and from `D` to `F3` twice.

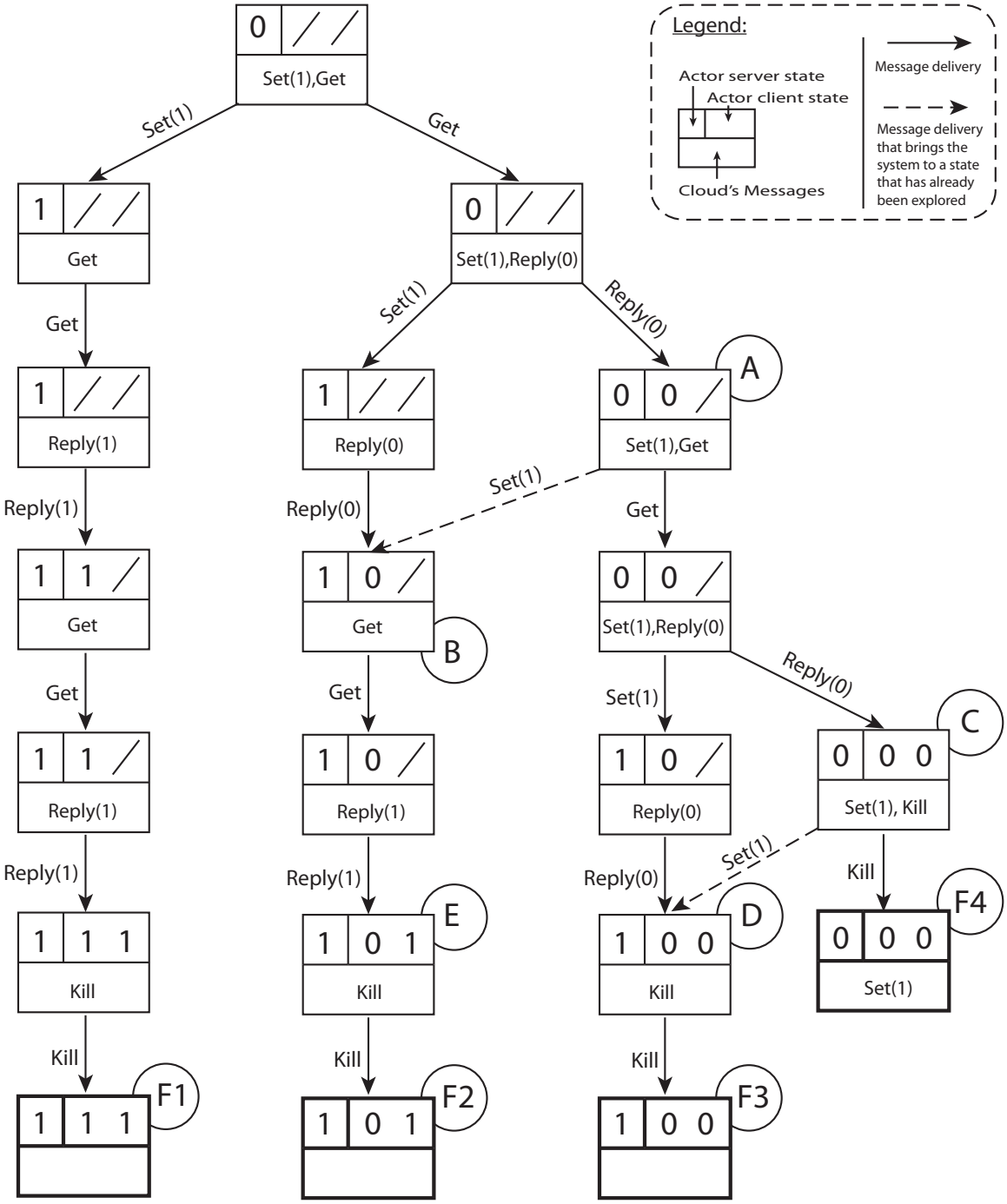

Figure 3-3: State space for the example program

Chapter 4

Framework

This chapter presents SEJAP, our general framework for state-space exploration of actor-based programs. SEJAP's design goal is to provide common, library-independent functionality for running actor programs and exploring different schedules of messages.

SEJAP aims to be a platform *efficient for stateful exploration* but *not necessarily efficient for straight-line execution* of actor programs. This goal affects the design decisions we made for the execution of actors. The three main modules of SEJAP handle *actor states* (keeping track of created and destroyed actors, and comparing states), *actor execution* (managing actor threads and the granularity of execution steps), and *message management* (scheduling and delivering messages). This chapter first discusses the responsibilities of these three modules and presents a high-level algorithm for state exploration. Lastly, Section 4.5 presents the concept of *library instantiation*, a design element that is key to SEJAP's flexibility.

4.1 Actor States

Each actor program creates several actors that compute and exchange messages. Various actor libraries provide different specialized operators for actors and, specifically, actor creation and termination are handled differently for each particular implementation. SEJAP therefore does not create the concrete actors by itself, but delegates that task to particular instantiations. On the other hand, SEJAP maintains generic

information about actors, e.g., keeping track of all created and destroyed actors during an execution. Through our experience with SEJAP, and the two current library instantiations that are tailored to it, we identified the following list of states that are common to any actor, independent of the library to which it belongs:

- `SUSPENDED` An actor that has been created (hence a name or a reference for the actor object has been produced) but is not yet able to process a message.
- `RUNNING` An actor that is making progress.
- `IDLE` An actor that is idle and waiting to receive (and process) a message.
- `BLOCKED` An actor that is blocked waiting. Typically this is the state of an actor that has sent a message synchronously and is now waiting an answer from the receiver of the message.
- `TERMINATED` An actor whose execution terminated but that may be started again in the future (not all libraries allows this i.e., this is not possible in ActorFoundry, but it is in Scala).
- `DESTROYED` An actor that has been completely removed from the application and hence it cannot be resurrected.

SEJAP currently uses this state information for state comparison (as discussed in Section 5.1), statistics about the exploration, and logging. Furthermore, additional features such as *deadlock detection*, could be implemented by collecting (at every program step) the list of actors in the state `BLOCKED`. If a circular dependency is found, a deadlock exists, and an error could be reported.

4.2 Actor Execution

A crucial part of an actor system is how to execute the actor code that processes each message. Semantically, each actor has its own thread of control. However, efficient implementations of actor libraries [61, 77] typically do not assign one native

thread/process per actor and do not create a new thread every time a new actor is created, because these operations are expensive. Instead, they employ thread pools to reuse threads for new actors, migrate actors among threads, and/or use more lightweight parallel constructs, such as the Java Fork/Join Framework [53].

Exploring all possible fine-grain interleaving of instructions from these threads would be very costly and is not necessary when actors have no shared state[1]. Instead, SEJAP uses the *macro-step semantic* [4, 74] for actor execution: after SEJAP delivers a message to an actor, the actor executes atomically until the next blocking point (e.g., a synchronous send or when the actor waits for the next message). In our initial design, we even considered avoiding threads altogether, since we execute actors using the macro-step semantic. However, when an actor blocks in the middle of its computation (i.e., synchronous send) we need to save the stack frame of the current execution (or, alternatively, its continuation) before switching to a different actor, so that context information about the current execution is not lost. Considering that when using threads we obtain (for free) a separate stack frame for each thread, and because SEJAP aims for efficient *exploration* (not *execution*) of actor programs, we decided to attach every actor to a separate object/Thread (named `SEJAPActThread`). Similarly as for the actors, SEJAP does not itself create actor threads but instead delegates that task to particular instantiations. The actor thread provides the main control for processing of one message. Section 5.2 discusses the granularity at which SEJAP combines processing of several messages.

4.3 Message Management

Actors communicate by exchanging messages. SEJAP again delegates the creation of concrete messages to instantiations, but SEJAP maintains a *message cloud* of all messages that were sent but not yet delivered to actors. The main loop in SEJAP

[1]Many actor libraries allow state sharing but some avoid it, say, by using functional languages (e.g., Erlang [69]), pass-by-copy rather than pass-by-reference messages (e.g., ActorFoundry [2]), or a type system based on linear types to statically check for absence of sharing (a feature that may be integrated in Scala in a near future).

controls the delivery schedule of these messages. Whenever the cloud has more than one deliverable message, SEJAP non-deterministically chooses to deliver one of these messages to the receiver actor. SEJAP then systematically explores all the program states that arise from the delivery of the cloud's messages.

It is important to point out that not all messages are deliverable at all times. One reason is that an actor can terminate itself while there might still be sent but undelivered messages for that actor. SEJAP is able to detect this situation and warns the user of a possible unwanted execution. Another reason is that most actor libraries allow the use of *synchronization constraints* [36] to narrow the shape of messages that an actor can receive (and hence process). Scala, for instance, expresses these constraints by using *guards* on the cases of the pattern matching expression [61]. If a message sent to some actor does not match any of its patterns, the message cannot be delivered until the behavior of the actor changes (so that its pattern matching accepts the message).

4.4 Exploration Algorithm

Algorithm 1 shows pseudo code for the algorithm that the main controller uses to explore all possible orders of message delivery for an actor program. The algorithm maintains two sets of program states: states that need to be explored (`StatesToExplore`), and states that have been already visited (`VisitedStates`). The set of states that still have to be explored initially contains only the starting state. The algorithm first selects the next state to explore and removes it from the set `StatesToExplore`. The search strategy[2] determines which message gets selected. Every state consists of the set of messages (stored in the *cloud*) and the set of actors. From the cloud, the algorithm selects all messages that can be delivered to the actors in their current state. Then, the algorithm non-deterministically selects a message from this set, removes it from the cloud, and delivers it to the receiver actor. At this point, the main controller gives control to the receiver actor so that it can process

[2]Common heuristics are depth-first search (DFS) and breadth-first search (BFS).

the message. When the actor blocks its execution (either because of a synchronous call or because it finishes processing the current message), the actor gives control back to the exploration loop. Any modifications to the program state performed by the executing actor (e.g., new messages sent, new actors created, the actor's state changed, or the actor's self destruction) are reflected in the current cloud and actors, which together form the new state. If this new state has not already been visited, it is added to both the set of visited states and the set of states that still need to be explored. The exploration then continues until there are no states to be explored.

Algorithm 1 Pseudo-code for exploration in SEJAP

```
1:  StatesToExplore ← { initial program state }                    ▷ singleton set
2:  VisitedStates ← ∅                                              ▷ empty set of states
3:  while StatesToExplore ≠ ∅ do
4:      State ← choose a state from StatesToExplore
5:      remove State from StatesToExplore
6:      DeliverableMsgs ← filter deliverable messages from State.Cloud
7:      for all Msg ∈ DeliverableMsgs do
8:          Cloud ← State.Cloud                                    ▷ set of message objects
9:          remove Msg from Cloud
10:         Actors ← State.Actors                                  ▷ set of actor objects
11:         (NewMsgs, NewActors) ← process Msg by the Msg.Receiver actor   ▷
        processing can send new messages, create new actors, and change the local state
        of the receiver actor including terminates the actor itself
12:         Cloud ← Cloud ∪ NewMsgs
13:         Actors ← Actors ∪ NewActors
14:         NewState ← (Cloud, Actors)
15:         if NewState ∉ VisitedStates then
16:             VisitedStates ← VisitedStates ∪ NewState
17:             StatesToExplore ← StatesToExplore ∪ NewState
18:         end if
19:     end for
20: end while
```

We should point out that the algorithm we have just presented is an ideal representation of the engine that drives the exploration. Specifically, when comparing states (line 15 in the algorithm), it is generally not possible to always exactly identify whether two states are equivalent. Furthermore, this is an involved operation which can be very costly. Therefore, in concrete implementation, there exists a trade-off

between the accuracy and the overall cost of this operation. Section 5.1 discusses the challenges that we faced while implementing state comparison in our framework.

4.5 Library Instantiation

The concept of *library instantiation* is a key element of SEJAP, because it enables the reuse of SEJAP's core for *different concrete actor libraries*.

This section presents the necessary steps needed to plug into SEJAP a new actor library. Based on our experience with SEJAP and the two actor libraries (Scala and ActorFoundry) that we have integrated into the framework, we identified best practices for carrying out this particular task. In each case, we always started from the existing library source code and we modified it with the following goals:

- Preserve the API of the library toward actor programs (such that we can check unmodified actor applications).
- Simplify the library, considering that we want fast exploration (for relatively small program states) and not necessarily fast executions (for relatively large program states, e.g., scaling up to thousands of actors). These modifications are likely to remove some parts of the library (e.g., distribution of actors across various systems), and replace other parts (e.g., when a message is sent to an actor, the message has to be rerouted into SEJAP).
- Hook the library into the SEJAP framework to enable exploration. We describe this process in the rest of this section.

While these modifications to the library may appear time consuming from the description, they were actually much easier to perform than building the framework.

4.5.1 Binding a Concrete Actor Library to the Framework

During our experience with SEJAP, we identified that the actor (plus the related actor thread object) and the message, are library-dependent elements that need to

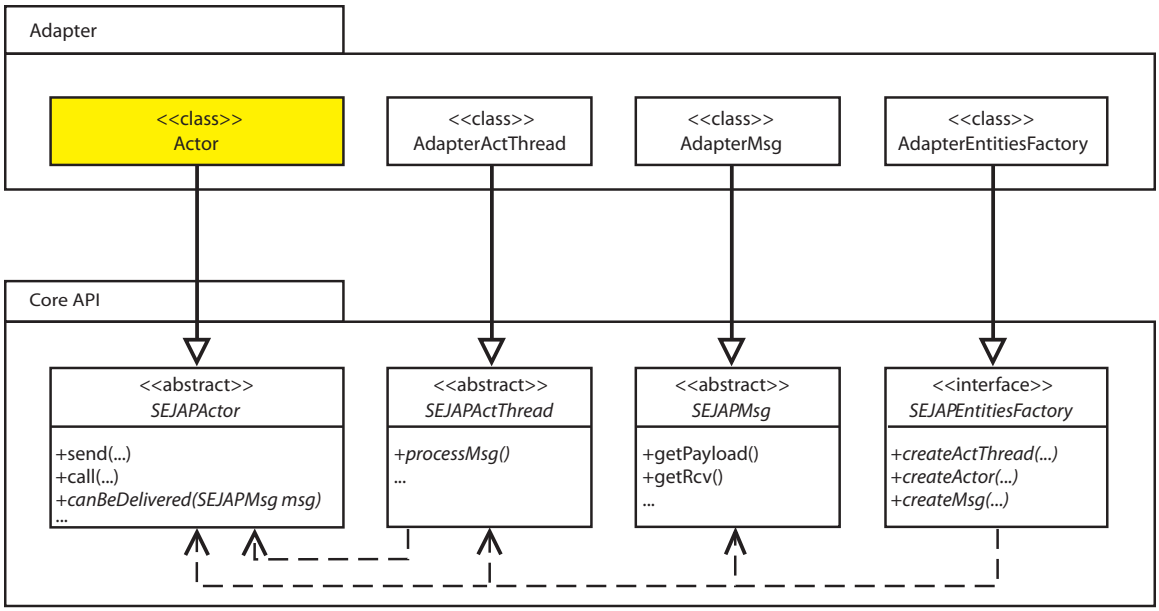

Figure 4-1: UML class diagram: Core and Adapter layers

be provided separately for each actor language. Nevertheless, the SEJAP core needs to manipulate actors, actor threads and messages, in order to explore the state space of an application; hence the need for a general and uniform representation of actors and messages. We defined three (abstract) classes: `SEJAPActor`, `SEJAPActThread` and `SEJAPMsg`. They represent: an actor, an actor thread and a message respectively, these classes are part of SEJAP's *Core API*. Developers that wish to hook a new language to SEJAP need to extend the Core API's classes by providing a set of *Adapter* classes that realize the Core's interfaces. The UML diagram in Figure 4-1 depicts this situation. The `Actor` class (shaded in the UML diagram) plays a decisive role, since it is the element connecting the concrete actor library to SEJAP's core. Specifically, we assume that every actor library provides a class `Actor` (the name is irrelevant) that clients of the library use to create actors in the application. By modifying the interface of this class, all actors that get created by the user application will conform to the `SEJAPActor` type that the framework expects. Furthermore, this change does not break the user code, meaning that the user application does not need to be modified or recompiled when run on our verification framework.

This layered structure permits SEJAP to manipulate generic instances of actors,

actor threads and messages. However, SEJAP also requires the ability to create concrete instances of these objects. This is problematic because SEJAP cannot know in advance which concrete set of classes needs to be instantiated, since actors, actor threads and messages are different for each library instantiation. Consequently, SEJAP does not directly create these entities but instead uses the *Abstract Factory* design pattern [24] to carry out this task. The framework itself refers to the `SEJAPEntitiesFactory` interface, and each library instantiation provides a concrete factory class that can create appropriate objects (the `AdapterEntitiesFactory` class in Figure 4-1).

Ultimately, the Core API's classes also provide *common behavior* that can be reused. For example, `send` (send a message asynchronously) and `call` (send a message synchronously) are already implemented in the `SEJAPActor` class and can hence be reused (freeing the developer from maintaining the logic of these common methods). Naturally, if the semantics of one or more of these operators does not match the one in use in the library, the developer can always provide a customized version. Finally, there are only a few methods—which are used internally by SEJAP—that are asked to be implemented by the developer e.g., `canBeDelivered(SEJAPMsg msg)` (which is used to determine whether an actor is in a state that allows the reception of the specific message `msg`).

4.5.2 Adapting the Library Interface

Above we described how an actor library can be plugged into SEJAP. Specifically, the class' interface that enables actors to be created in the original library (`Actor` in Figure 4-1) needs to be modified; since it should extend the `SEJAPActor` class. This strategy cannot be pursued if the language does not support multiple inheritance and the `Actor` class already extends some other class (this is a common issue for actor libraries written in the Java programming language). Fortunately, we can overcome this issue using the *Adapter* design pattern [24]. The idea is the following: if the interface of the original actor class cannot be modified, the developer can create a concrete `AdapterActor` class that wraps the reference to the original actor instance,

```
public class AdapterActor extends SEJAPActor {
  private Actor adaptee; //"'Actor" is the library's type for an actor
  public AdapterActor(Actor ref) {
    this.adaptee = ref;
  }
  ...
}
```

Figure 4-2: Skeleton of an adapter class for an actor library.

the *adaptee* (the snippet of code in Figure 4-2 provides a sample of how this can be implemented in Java). Furthermore, the adaptee needs to be able to call the adapter's methods. Two solutions exist. The simpler one is to add a new member to the original `Actor` class, so that the adaptee knows about the adapter. However, this solution might not always work, e.g., because of Scala compilation model, members added to the `Actor` trait are not visible to previously compiled subclasses. The alternative is to maintain the correspondence between the adaptee and the adapter in a map, outside of the `Actor` class. Each actor operation that needs to communicate with SEJAP will hence use this map. In Section 6.4 we discuss how we used this solution to adapt the Scala actor library and integrate it into our framework.

Chapter 5

Optimizations

One of the major concerns of software model checking is the combinatorial blow-up of the state-space (also known as the state-explosion problem) that it is faced when dealing with real applications. Researchers have been working on this problem for a long time, and several solutions have been considered to mitigate this issue. Examples are symbolic execution [46] (where a symbolic state represents a set of concrete states[1]), bounded model checking [16] (where a system property is checked only up to a certain program depth), partial order reduction [17](works by exploiting commutativity of concurrently executed transitions), symmetry reduction [40] (works by identifying equivalence classes of executions). Although these analyses are not a definitive answer to the problem, they all are interesting techniques that can improve the scalability of a tool.

As we will see in Chapter 6, our framework for state exploration of actor programs has been implemented on top of Java PathFinder (JPF), a stateful model checker for Java bytecode. Due to JPF's nature, states are compared at every thread scheduling relevant program step. Therefore, it is important to spot equivalent classes of states to prevent exploration of previously seen traces. To achieve this, we used model abstraction and symmetry reduction to effectively reduce the state-space of an actor program; this is presented in Section 5.1. However, comparing states is a costly oper-

[1]The downside of this approach is that executing a program symbolically is hard since the constraint on the values need to be solved, and this can quickly become impossible or very time-consuming.

ation and might not always be the best strategy, e.g., stateless model checkers explore different traces through program re-execution, see [63] for example. Section 5.2 describes two different step granularity tactics that impose a trade-off between faster straight-line execution and efficient overall exploration.

5.1 State Comparison

The algorithm that drives the program exploration (discussed in Section 4.4) checks whether a new state has been already visited previously, effectively comparing one state against a set of states. This is a common operation in explicit-state model checking [17]. A challenge for object-oriented programs (whose state include heaps with connected objects) is that states need to be compared for *isomorphism* [11, 39, 56]. Namely, two states are equivalent when their heaps have the same shape among connected objects and the same primitive values, even if they have different object identities. Typical comparison of states for isomorphism involves linearizing the *entire* state into an integer sequence that normalizes object identities, so that isomorphic states have equal sequences [39, 56].

An additional challenge we faced when comparing the state of actor programs is that the top-level (roots) state items—actors and message clouds—are sets. The usual JFP linearization does not specially treat sets but simply compares them at the concrete level at which they are implemented, say, as arrays or lists. Therefore, two sets that have the same elements may be deemed different because of the order of their elements. To compare states of actor programs in SEJAP, we use a heuristic that first sorts set elements and then linearizes them as usual. This provides more opportunity to identify equivalent sets and states, but does not guarantee that all equivalent states will be found. Specifically, the sorting is done only for the elements, without following any pointers from these elements and, therefore, does not handle arbitrary graph isomorphism [76].

Additional care has to be taken with actor names/references, since they are not relevant for state comparison, i.e., the states are equivalent up to α-conversion. More-

over, the usual JPF linearization considers entire states, while in SEJAP we want to consider only the application state (i.e., the actors and message cloud) and ignore the irrelevant parts of the library state and the SEJAP framework: even if these irrelevant parts do differ, the application states are equivalent.

5.2 Step Granularity

The primary source of non-determinism for actor programs is the order in which messages are delivered to the actors. When exploring different message schedules, the standard step used by SEJAP consists of all instructions starting from some actor receiving a message up to the point where the actor blocks, either because it is waiting to receive the next message or because it executed a synchronous send. This step is called a *macro step* [4, 74]. Provided that an actor program is limited to message passing only, no interleaving of different actor threads is necessary, i.e., all the behaviors of the program with fine-grained thread interleaving can be obtained with the macro step interleaving. SEJAP always explores macro steps of actor programs.

An additional consideration is whether to merge macro steps from several actors when there is only one message in the cloud. We refer to such steps that *combine several deterministic macro steps* from different actors as *Big* steps. Recall the state space from Figure 3-3. After the `Set(1)` message is processed from the initial state, the rest of the execution (until the `F1` state) is deterministic: there are several macro steps alternatively executed by the server and client actors, but there is always one message in the cloud. A Big step would thus execute the program until `F1`, without performing state comparison for the intermediate states shown in the figure. In contrast, a *Little* step executes *only one macro step at a time* and performs state comparison for each intermediate state. In other words, Figure 3-3 shows the exploration that SEJAP performs for Little step.

Whether Big or Little steps provide faster exploration is not immediately clear. The trade-off is that Big steps are faster for straight-line execution since they perform longer executions without stopping to compare states. Big steps, however, can miss

the opportunity to find equivalent states and thus end up re-executing a number of transitions, which may result in a slower overall exploration. For example, in the state space from Figure 3-3, Big steps would miss that states `B` and `D` are equal along two different execution paths, and would thus end up re-executing twice the code from `B` to `F2` and from `D` to `F3`. On the other hand, Little steps have slower straight-line execution due to more frequent state comparison. This increases opportunities to avoid execution of already explored traces, but also runs the risk that frequent comparisons unnecessarily slow down the exploration if (most of) the states are different. Whether program re-execution or aggressive state comparison is more costly depends on the particular execution platform and the frequency with which states are repeated during state exploration. Our experiments with Java PathFinder and the subject actor programs described in Chapter 7 show that Little step performs better than Big step.

Chapter 6

Implementation

We implemented SEJAP on top of Java PathFinder, an extensible, explicit-state model checker for Java bytecode [44, 82]. We decided to use JPF because of the following considerations:

- JPF executes Java bytecode and hence represents a natural candidate for executing Scala code (since Scala compiles to Java bytecode).
- JPF has been designed with the intent of being extensible.
- JPF is open source and it implements many state-of-the-art verification techniques to efficiently explore the state space of concurrent applications.

In this chapter we discuss some of the challenges that we faced while implementing SEJAP. Section 6.1 provides a high-level overview of how SEJAP interacts with JPF, the modified Scala actor library, and the compiled user code. Section 6.2 briefly introduces JPF and how we modified it to control thread switching for actors. Section 6.3 motivates why the standard JPF's distribution cannot be used to verify Scala actor programs. Ultimately, in Section 6.4, we present the relevant modifications performed on the original library to enable execution of Scala actors into SEJAP.

Prior to this work, JPF did not have any special capability for efficiently explore actor programs. Moreover, this is the first framework that enables state exploration of actors for the Scala programming language.

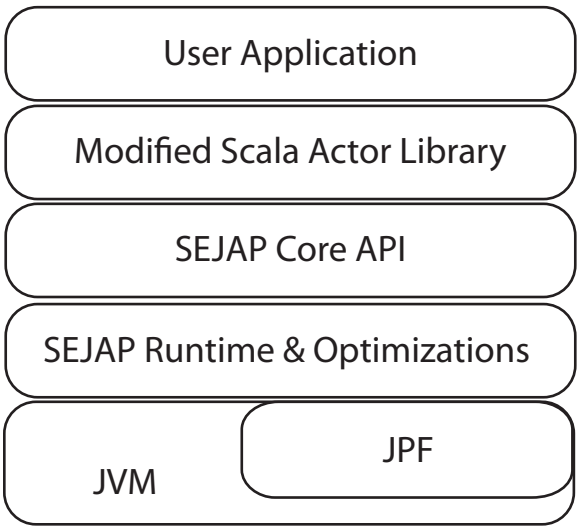

Figure 6-1: Framework's stack.

6.1 System Overview

Figure 6-1 depicts a stack view of the system. On the bottom there is the host JVM together with JPF. One layer up we find the modules that allows execution and state exploration of actor programs (we described these in Chapter 4). At this level we also implemented the optimizations presented in Chapter 5, which are de facto available to any library instantiations tailored to SEJAP. One layer up there are the *Core API* classes described in Section 4.5 and, on top of them, there is the adapted Scala actor library (the modifications that we made are described in Section 6.4). Finally, at the top is the compiled user application. Note that the user code only knows about the actor library. Because we maintained the interfaces of the library, the user application does not need to be modified or recompiled when executed in SEJAP.

6.2 Java PathFinder Modifications

JPF is written in Java and provides a specialized Java Virtual Machine (JVM^{JPF}) that supports state backtracking and control over non-deterministic choices. The default non-deterministic choices in JPF are thread scheduling and explicit choices made in the code using JPF library calls such as `Verify.getInt` [31]. The JVM^{JPF} is

an interpreter that executes the bytecode of the application under exploration. JPF itself runs on top of a host, native JVM. JPF provides an interface, called Model Java Interface (MJI), for communication between the JVM^{JPF} and the host JVM (we described the MJI in Section 2.3.2).

The key change we made to JPF is in the control of thread scheduling. Recall that SEJAP architecture puts each actor in its own thread. The actor code itself is in Java (more precisely, compiled to Java bytecode). Additionally, SEJAP has a main controller thread that decides which actor(s) should be executed at which point. We wrote the main controller itself in Java so that it runs in the JVM^{JPF} (and not on the host JVM). All these threads are actual JPF/Java threads[1]. Based on the macro-step semantics (discussed in Section 2.1.1), it is not necessary to explore all fine-grained interleaving of these threads; the non-determinism in actor programs is due to the order in which messages are processed by the actors. Therefore, SEJAP makes only one of these threads enabled at a time.

Note that the thread switch could not be implemented in pure Java executing in the JVM^{JPF}. Namely, the main loop in SEJAP proceeds as follows: the main controller chooses one actor to execute (more specifically, one message to deliver to an actor that then starts processing the message), and when that actor *blocks* (waiting to receive a new message or because it synchronously send a message), the main controller should take control back and execute to schedule the next actor (Section 4.4 for further details on the exploration algorithm). However, once the actor blocks, it cannot return the control to the main controller at the Java level because Java doesn't offer this kind of fine-grained control over the runtime execution. So, we used the MJI to implement thread switches, effectively replacing JPF's thread scheduler.

Besides thread switching, we also modified JPF to implement two further optimizations (which have been discussed in Chapter 5). First, we optimized the core of SEJAP to eliminate JPF backtracking points when switching back and forth between the actor threads and the main controller thread. Second, our implementation of the

[1]JPF provides a customized version of thread that emulates the behavior of the java.lang.Thread Java library class. This is needed to enable JPF to have fine-grained control over threads scheduling.

Big step requires executing several threads as one transition in JPF. Since the JPF main loop is structured around executing only one thread in a transition, we had to modify the JPF code to enable longer transitions. To the best of our knowledge, this is the first JPF extension that considered state transitions with bytecode executed by more than one thread.

6.3 Java PathFinder and Scala

One question that we may ask is why not just run the user program with the standard JPF distribution, considering that Scala compiles to Java bytecode. The downside of doing this is that the actor library is model checked together with the user application. The Scala actor library is a complex multi-threaded, highly concurrent, software component. This entails that even the simplest application, e.g., a single actor that prints "Hello, World!" to the terminal, when run on the standard JPF distribution, generates a huge number of different states that need to be explored, making it impractical to perform an exhaustive exploration. We even tried to simplify the original library as much as possible (i.e., forced the pool thread to contain exactly one thread, removed the unused timer thread that is automatically started when an actor is created, disabled the actor garbage collector and some other minor modifications). With all these modifications, JPF was able to finish the exploration in about 7 minutes. In contrast, SEJAP needs less than one second. Moreover, any attempt at checking a more involved actor application using directly JPF failed miserably.

In our approach we assume the library to be correct and focus on the task of verifying the correctness of the user application. Ascertaining the correctness of the library is an important aspect that can be done as a separate effort, since it is orthogonal to our work.

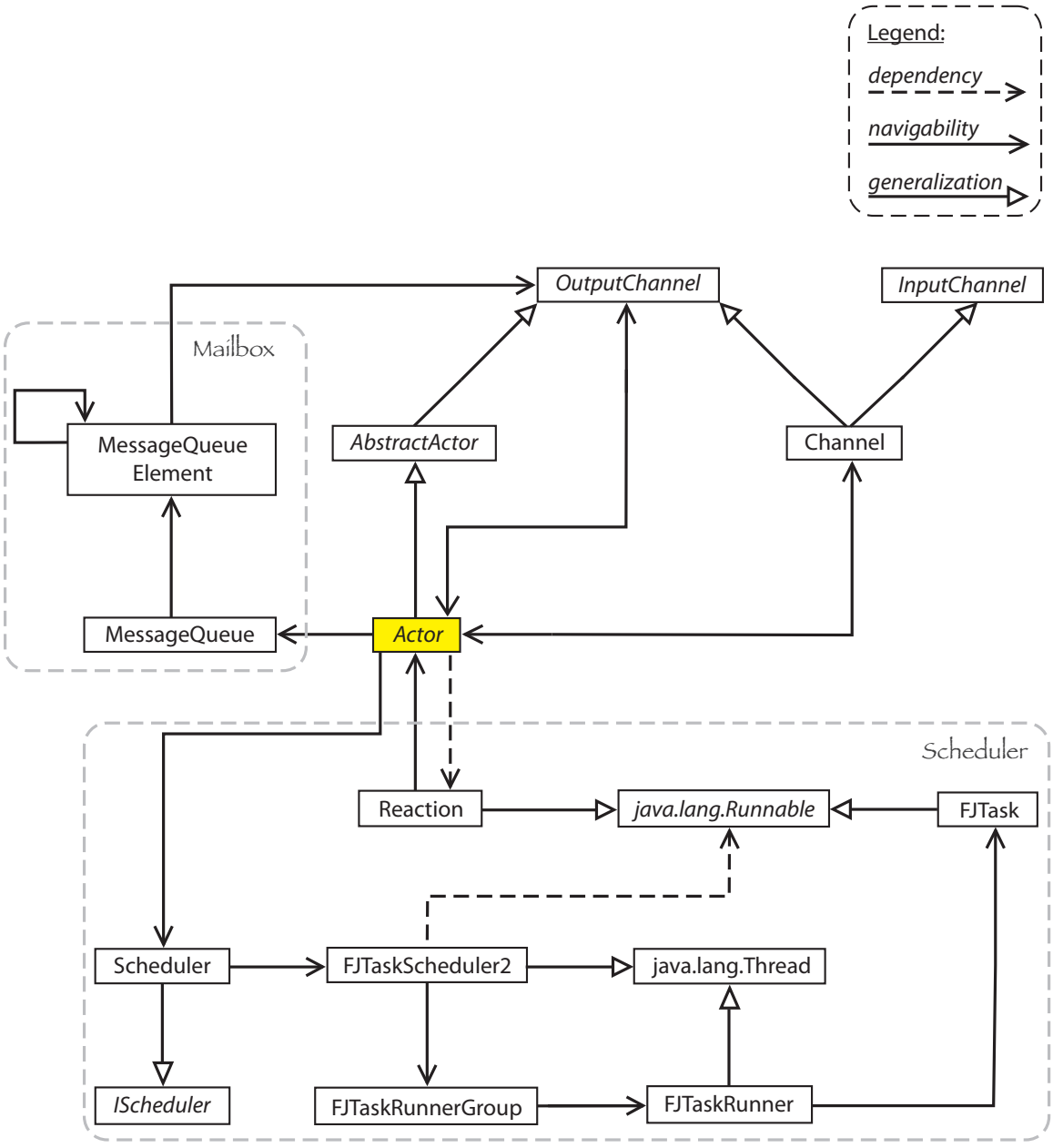

Figure 6-2: UML class diagram of the Scala actor library.

6.4 Adapting the Scala Actor Library

In Section 4.5, we provided general guidelines for integrating a new actor library in SEJAP. In this section, we describe our experience with the Scala actor library.

The very first step was to identify which parts of the library are exposed to the developer and how the different components of the library relate to each other. Therefore, we started from a (simplified) sketch of the library's design (depicted in Figure 6-2). The class diagram is not complete, but it aims to provide an overview of most of the elements that constitute the library. The class diagram clearly shows that the classes that manage actor execution (*Scheduler* dashed box in the diagram) are loosely coupled to the classes that contain the actor logic (basically, the `Actor` trait). This has greatly simplified our job since SEJAP's core takes care of the program's exploration, and hence there is no need for the library to be linked to a scheduler. Consequently, we removed from the `Actor` trait all interaction with the scheduling side of the library. We should point out that clients of the library are allowed to provide a customized scheduler, and hence potentially change the semantic of actor execution (e.g., the scheduler could implement *FIFO* queues for message delivery, or even filter some kind of messages depending on the system state/configuration). We restrained our implementation to handle the current default scheduler provided with the library, where no order on the message delivery can be assumed.

In Scala, each actor has access to the list of messages that have been delivered to the actor (the *Mailbox* dashed box of Figure 6-2). This is a potential source of problems because side-effects on the mailbox are allowed (e.g., messages may be deleted/added without calling the usual actor send operator). In the current implementation we decided to follow the theoretical model where actors do not have control over their mailbox, because the receipt of a message is an implicit operation. As a matter of fact, in SEJAP, we have a centralized container for yet to be delivered messages, the *message cloud* (described in Section 4.3), and actors only have access to the currently processed message. At present, if an actor tries to access its mailbox, an error message is delivered and the exploration stops.

After having unlinked the *Scheduler* and the *Mailbox* classes from the library, what remains is essentially the logic that allows actors to send, receive and process messages. Specifically, the `Actor` trait is the only visible element that we modified[2] to interface the library to SEJAP. However, this step has some nuances that are worth discussing. The problem is the following: the `Actor` trait needs to preserve the actual interface (so that the user code does not need to be re-compiled), and also needs to subclass the core class `SEJAPActor` (which enables the communication with the internal of the framework). Despite Scala permitting multiple inheritance, extending the `Actor` trait with the abstract `SEJAPActor` class doesn't yield the desired result. This is due to the way that Scala compiles its sources and, specifically, how Scala *trait*s are transformed into several Java interfaces and abstract classes when the source code is compiled [73]. Hence we used the strategy described in Section 4.5.2 to hook up the Scala actor library in our framework; namely, we created an adapter class `ScalaActor` that contains a pointer to the original `Actor` object and maintained an external map to allow the Scala actor instance to call the adapter actor. Lastly, we modified the body of methods for sending and receiving messages (and a few other methods that are not visible to clients) in the `Actor` trait, so that they use the corresponding methods (e.g., `send`, `call`) of the adapter `ScalaActor` class, which is responsible for managing the communication to and from the SEJAP's core.

[2] We also had to edit one line in `ActorProxy`, a class used internally to the library to unify actors and threads.

Chapter 7

Evaluation

This chapter present our evaluation of the performance of the SEJAP framework. Section 7.1 discusses the implications of one, JPF related, global optimization that we integrated in the framework. Section 7.2 briefly introduces the subjects that we used to assess how well SEJAP performs. We evaluate the results of executing these tests in Section 7.3. Section 7.4 concludes by presenting how Little and Big steps (two optimizations that we have implemented) compares for the analyzed programs.

All experiments were performed using Sun's JVM 1.6.0_01-b06 on a 3.4GHz Pentium 4 with 2GB of RAM workstation, running RedHat Enterprise Linux 4.

We would like to remind the reader that using the original Scala actor library together with the standard JPF distribution would not work for any of the subjects that we use to evaluate SEJAP. In Section 6.3, we described a number of modification that we made on the original JPF and Scala actor library distribution, which allowed to terminate the exploration of a trivial (helloworld) actor in about 7 minutes (while in SEJAP this takes less than a second). Using this same configuration, we also tried to explore the state-space of the subject programs described in this chapter. However, for none of them JPF was able to terminate the exploration (not even of one execution-trace) within the time limit of one hour.

7.1 Optimized Thread Context Switching

In Section 6.2, we mentioned that we eliminated JPF backtracking points when switching back and forth between the actor threads, and the main controller thread. This change is simply a consequence of our implementation in JPF, and not a general characteristic of SEJAP. Specifically, saving the state when executing a context switch in SEJAP is completely useless because the program's state[1] can change only while an actor is making progress. In Section 4.4, we presented the algorithm that drives the program exploration and is executed by the main controller thread. The algorithm highlights two facts: (1) saving the state before the actor starts processing a message is useless, since the program's state can change only when an actor makes progress, and the actor cannot make progress if we did not yet context switch to it; and (2) storing the state when the actor stops and gives control back to the main controller is also useless, since the program's state is already saved by the algorithm (at line 16), and hence there is no need to store the exact same state twice.

We ran some experiments[2] and, as expected, the results showed that avoiding these state operations at context switches resulted in JPF exploring fewer states and transitions, and also produced less state backtracking, thus speeding up the exploration. Hence, we integrated this optimization as part of the framework's core and all the discussed results already take advantage of this enhancement.

7.2 Subjects

To evaluate the performances of SEJAP, we use five actor program examples. We have implemented all the examples but the $\texttt{server}_{real}$ one. This is the original client-server application available on the ScalaWiki website[3] [72], which exposed an atomicity violation bug (we spotted this error thanks to SEJAP and we reported it to the community, which confirmed it). The $\texttt{server}_{simple}$ subject is the simplified version

[1] Which is composed of the set of actors and the set of messages (as mentioned in Section 4.4).

[2] Numbers for these experiments have not been reported because they do not provide any interesting source of discussion, besides supporting our intuition.

[3] Appendix A provides the original source code of the example.

of the $\texttt{server}_{real}$ that we used in Chapter 3 to describe the main characteristics of SEJAP. All tests can be run in both the standard Scala run-time and SEJAP with no modifications. The remaining four subjects implement more complex algorithms and were previously used in a study on testing actor applications [74]. We will now briefly describe the main characteristics of each of the implemented programs.

Leader Election is an implementation of the famous Peterson algorithm [64], generally used in distributed computing to select a single process as the organizer (leader) of some task. Before running the algorithm, the nodes are unaware of which node will serve as the coordinator of the task. However, after the algorithm is run, every single node recognizes the same unique node as the task leader. We evaluated our leader election implementation for a system with three nodes.

Spin Sort is an implementation of a parallel sorting algorithm similar to systolic sorting [84]. The system has N nodes and every node has an identity so that a strict total order exists among the different nodes. Each single node only knows about its closest neighbor (defined as the node with the smallest identity higher than the current node's identity). The algorithm works as follows: initially, the node with smallest identity receives a set of messages, each of them containing a value. The node keeps the smallest value and forwards to its neighbor the other values; the neighbor, in turn, keeps the smallest value among the received messages and passes the values left to its neighbors (and so forth for each of the remaining nodes). At the end of the computation, we can retrieve the sorted list by looking at the value contained in each node, starting from the node with smallest identity and then by looking at the neighbor of each subsequent node. We evaluated our spin sort implementation for a list of four elements.

Shortest Path is an implementation of the Chandy-Misra's shortest path algorithm [15] used to find a path between two nodes, such that the sum of the weights of its constituent edges is minimum. The implemented algorithm computes the minimum distance that separates a particular node n_i from all the other nodes n_j of the

graph. We evaluated our implementation for a network topology with three nodes fully connected.

Fibonacci is an example frequently used to illustrate and evaluate actor implementations. The algorithm computes the N-th element in the Fibonacci sequence and follows a standard divide and conquer strategy. We evaluated our implementation for the input $N = 4$.

All the implemented subjects have been instantiated for a relatively small input size. The rationale is that a high proportion of bugs in concurrent programs can be found by testing the program with a small test input. This assumption is sometimes called the *small scope hypothesis* [41] in the literature.

7.3 State Comparison

In Section 5.1, we discussed how we improved state comparison within JPF, with the intent of increasing the chances of identifying previously visited states. The abstraction that we use for state comparison allows for more aggressive pruning of redundant message schedules, which results in faster state-space exploration. Table 7.1 shows the results of experiments comparing the JPF's standard state comparison (State Comp = JPF) against our optimized actor state comparison (State Comp = Actor). For reference purposes, the table also provides results with the state comparison disabled altogether (State Comp = None), which shows how important it is to identify, as soon as possible, equivalent states in JPF. Each experiment was performed using the Little step granularity (since for these programs it is faster than Big step, as shown by the results in the next section).

For each type of state comparison, we tabulate the total exploration time in minutes and seconds, memory usage in MB, the number of states identified during the entire exploration, the maximum exploration depth (i.e., the length of the longest trace of Little steps), and the total number of messages (across all execution traces) that were delivered during the exploration. Effectively, the number of states and

Table 7.1: Evaluating different state comparison implementations

Experiment		Resources		Statistics		
Subject	State Comp.	Exploration Time (mm:ss)	Memory (MB)	# of States	Max Depth	# of Msgs Delivered
fib	None	0:22	25	768	9	767
	JPF	0:06	24	80	9	144
	Actor	0:05	24	43	9	75
leader	None	0:36	25	1467	11	1466
	JPF	0:11	26	255	11	341
	Actor	0:08	26	187	11	237
server_{real}	None	8:13	97	22522	19	22521
	JPF	2:38	96	6513	19	7303
	Actor	2:20	100	5456	19	6267
server_{simple}	None	0:03	16	26	5	25
	JPF	0:03	18	23	5	23
	Actor	0:03	23	20	5	21
shortpath	None	4:00	27	10000	10	9999
	JPF	0:32	28	534	10	1160
	Actor	0:18	28	230	10	556
spinsort	None	1:55	26	5045	9	5044
	JPF	0:15	26	317	9	508
	Actor	0:13	27	269	9	432

messages are the number of nodes and edges, respectively, in the state-space graph that SEJAP explores for these programs (Figure 3-3 depicts the state-space graph for the `server` example). Also, for all the tests we used a depth-first strategy to explore the program execution.

Exploration time typically improves as we progress through the three types of state comparison, from None to JPF to our Actor comparison. The only seeming exception is for the the `server` subject, where the time difference is due to the imprecision of the measurements. In all cases, the Actor comparison results in the fastest exploration. Memory utilization remains reasonable across all of the experiments. Similar to reducing exploration time, the Actor comparison also reduces the number of explored states and the number of delivered messages. As the abstraction used by the state comparison is refined to consider only relevant state differences, the number of states and executions that can be pruned increases.

Table 7.2: Comparing step granularity – *Big* vs. *Little* steps

Experiment		Resources		Statistics		
Subject	Step Size	Exploration Time (mm:ss)	Memory (MB)	# of States	Max Depth	# of Msgs Delivered
fib	Big	0:05	25	41	6	109
	Little	0:05	24	43	9	75
leader	Big	0:10	28	132	7	308
	Little	0:08	26	187	11	237
server_{real}	Big	3:26	100	2872	18	8715
	Little	2:20	100	5456	19	6267
server_{simple}	Big	0:03	23	11	4	25
	Little	0:03	23	20	5	21
shortpath	Big	0:20	28	275	8	639
	Little	0:18	28	230	10	556
spinsort	Big	0:16	27	221	7	552
	Little	0:13	27	269	9	432

7.4 Step Granularity

In Section 5.2, we discussed the trade-off associated with performing explorations using *Big* steps versus *Little* steps. We performed experiments to compare the performance of state-space explorations that enforced Little steps with explorations that allowed combining multiple Little steps into Big steps when appropriate. Each experiment was performed using our optimized *Actor state comparison*, since the previous sections shows it to be the fastest. Table 7.2 shows the results of these experiments. For both Little and Big step granularity, we tabulate the same information as for the state comparison experiments.

The table shows that Little step granularity results in explorations that are faster than explorations using Big step granularity. This is due to increased opportunities for state pruning (exposed by using the smaller step size for our subject programs). This is not to say, however, that Little step granularity is always faster. Subjects may exist where Big step granularity outperforms Little step.

Chapter 8

Related Work

The most related work to SEJAP is on model checking actor programs. Sen and Agha present dCUTE [74] which checks a given actor program, written in a simplified library, by re-executing the program for various message schedules. dCUTE uses a dynamic partial-order reduction [17] to avoid exploring equivalent schedules. dCUTE combines the happens-before relation [50] with a mixed concrete and symbolic execution for test generation [14, 26, 75]. In contrast, SEJAP provides a common framework for *stateful* exploration of Java-based actor libraries, handles *full actor libraries* for Scala and ActorFoundry (including dynamic creation and destruction of actors), and provides *state comparison* and varying *granularity of execution steps*. Also, SEJAP is built on top of JPF and can reuse its functionality for state-space exploration (e.g., heuristics for ordering exploration). The partial order reduction discussed in dCUTE is an orthogonal optimization that could further improve the quality of our framework. This is discussed in further detail in Chapter 9.1.

Fredlund and Svensson present McErlang [23], a stateful model checker for actor programs written in the Erlang programming language [69]. McErlang is itself written in Erlang and modifies the concurrency system of the Erlang run-time library. A previous model checker for Erlang, *etomcrl* [8], checked Erlang programs by translating them into μCRL [28] and using off-the-shelf model checkers, similarly as the very first version of JPF [32] checked Java programs by translating them into Promela and using SPIN [35]. Again, SEJAP does not focus on one language/library

but provides a general framework, builds on an existing tool (JPF), and incorporates several optimizations for exploration.

Related work is also on checking distributed systems [7, 10, 37, 78, 85]. In particular, Artho and Garoche [7] and Barlas and Bultan [10] provide frameworks for executing distributed Java code in JPF. A key problem is that such code uses network calls that JPF does not support as they depend on native code from the Java standard libraries. These two projects solve this problem by instrumenting the bytecode [7] or providing stub classes [10]. These solutions are conceptually similar to SEJAP in that they replace/avoid the standard Java network library as SEJAP replaces actor libraries. However, both solutions focus on low-level communication, whereas SEJAP focuses on high-level exploration of possible behaviors for actor programs.

Yabandeh et al. [85] describe CrystalBall, which is a lightweight framework for online model checking of loosely coupled distributed applications. Specifically, each node in the system executes CrystalBall in parallel with the application, and explores several possible message interleavings with the intent of preventing the system from reaching an inconsistent state. Since CrystalBall is run in parallel with the real application (on each node), the exploration algorithm needs to detect potential inconsistencies before it is too late, e.g., before the system has already passed the execution depth at which the violation occurs. Therefore, a heuristic is used to drive the exploration. The key difference between CrystalBall and SEJAP is that while CrystalBall focuses on preventing a deployed system from reaching an inconsistent state (CrystalBall might alter the execution of the application to achieve this goal), SEJAP is intended to be used as a traditional testing framework that can help the developer identify bugs before deployment. In short, the two frameworks are orthogonal to each other. A framework similar to CrystalBall could improve the reliability of distributed actor programs when tailored to the runtime of an actor system.

Partial-order reduction is an important optimization for alleviating the state-space explosion in model checking [5, 17, 22, 25, 85, 86]. It uses static or dynamic analysis to avoid exploring certain message schedules altogether. In contrast, SEJAP executes the schedules and uses state comparison to determine when to prune exploration. In

effect, partial-order reduction for actor programs is complementary to our work, and we plan to integrate it as an optimization in SEJAP.

Chapter 9

Conclusions

This dissertation presented a tool for systematic exploration of actor programs written in the Scala programming language. To the best of our knowledge, this is the first tool that allows systematic exploration of the state space of Scala actor programs. We developed this tool as an instantiation of SEJAP, a general framework for systematic exploration of actor programs based on Java bytecode.

A common issue for verification frameworks such as SEJAP is the combinatorial explosion of the state-space when dealing with large applications. We presented two optimizations, based on the granularity of execution steps and actor state comparison, which alleviate the state-space explosion problem and thus, speed up the overall exploration of actor programs in SEJAP.

We implemented our framework, instantiations, and optimizations on top of Java PathFinder, a widely used model checker for Java bytecode developed by the NASA. The experiments presented show that SEJAP can effectively explore executions of actor programs.

SEJAP already helped in discovering a previously unknown error (which was confirmed by the authors after we reported it) in a sample Scala code from the ScalaWiki website [72].

9.1 Future Work

Future work includes supporting actor distribution for the Scala actor library, optimizing SEJAP further and using it for testing larger applications. Further questions of interest include unit testing and debugging actor programs.

Currently, we do not support actor distribution for the Scala actor library. One problem with distribution is that actors may reside on different machines and the library uses network protocols (e.g., TCP) to enable standard actor communication. This is an issue because model checkers usually work on closed (or self-contained) systems, which means that the program execution does not depend on interaction with its environment. Two alternatives are possible to address this problem: (1) We could implement in SEJAP a set of stub classes that replace low-level network communication (Java) classes, so that it would be possible to explore the state of distributed actor programs on a single machine. Barlans and Bultan [10] presented this idea in their framework for verification of distributed applications (we have discussed their work in Chapter 8). (2) Since the source code of the Scala actor library is publicly available, we could directly modify library code to remove low-level network communication. The advantage of the first solution is that it is more general, since different actor libraries are likely to use similar network protocols for implementing actors distribution. On the other hand, the second solution is easier to implement and would also allow faster exploration of Scala actor programs, since we would considerably simplify the library.

An important optimization that we are planning to implement is a variation of the partial order reduction algorithm presented in dCUTE [74]. dCUTE uses vector clocks [21] to track a partial order, through the *happens-before relation* [50], among the exchanged messages, and it uses this knowledge to avoid exploring traces that would have equivalent execution. Vector clocks require that the number of processes in the distributed system has to be constant and known in advance, therefore only static actor programs (actors are neither created nor destroyed during the program execution) are currently handled by dCUTE. SEJAP could explore a similar solution;

however, in order to enable actors to be dynamically created and removed during execution, it would need a more flexible abstraction than vector clocks. Our preliminary study indicates that *tree clocks* [51] and *hierarchical clocks* [66] may represent interesting alternatives for implementing a generic partial order reduction for actor programs.

A second possible optimization is improving the actual state comparison (presented in Section 5.1). Currently, we do not handle arbitrary graph isomorphism and, therefore, we might miss opportunities for pruning the program's state-space. Spermann and Leuschel [76] presented an efficient approach for identifying state symmetries in the ProB [54] model checker. The idea is to convert the concrete state into a vertex-colored graph and then use NAUTY [58] to obtain the *canonical form* of the graph[1]. The resulting graph can be used to effectively compare the original concrete states. The trade-off of this solution is the cost that such translation requires, versus the higher opportunities for identifying state symmetries. It would be of interest to implement a similar approach in our framework and to evaluate its performance. Note that this refined state comparison could also be integrated within the standard JPF distribution, since it is not specific to actor systems.

Despite all optimizations, the combinatorial explosion of the state space still remains the major limitation of verification frameworks that aim to systematically explores all possible executions of an application. Because of this, it is important that the algorithm that drives the exploration is tuned to the goal of prioritizing execution paths that are more likely to lead to bugs. Finding heuristics that can quickly expose bugs in actor programs is an interesting area where there is still room for contributions. SEJAP can currently use generic search strategies that are provided by Java PathFinder (depth-first search, breadth-first search, heuristic search), but the heuristics in JPF are not specifically tailored to actors and messages.

Orthogonal to our work is the question of how actor programs should be unit tested. In fact, model checking is usually time-consuming and cannot often be applied

[1]NAUTY is an efficient library that can be used to obtain canonically-labeled isomorph of the input graph.

as a part of the software development process. A common problem when unit testing concurrent software is that for the same input, very different executions may be possible. As a consequence, what appears to be correct most of the time, might manifest a failure for some sporadic execution. These type of software bugs are usually described as *heisenbugs*. Furthermore, when unit testing a concurrent application, it is usually not possible to have a fine-grained control over the threads' execution. Common strategies are the use of sleeping timeouts to create a particular interleaving. For instance, the ConTest tool [18, 20] uses instrumentation of the Java bytecode to force different timing scenarios to happen in tests. However, this is an empirical solution that lacks precision. Some initial work on unit testing actor programs has been done by Burmeister [13].

It would also be interesting to provide better debugging support for actors in future debugging tools. For instance, the Eclipse IDE allows the user (when debugging Java code) to manually control the next thread's statement to execute. We think that a similar feature that would enable the developer to have fine-grained control over the order in which messages are delivered to actors, would considerably improve user productivity and simplify the task of identifying and correcting bugs in actor programs.

Appendix A

Original Client-Server ScalaWiki Code

This appendix presents the full (untouched) code for the original client-server sample, publicly available on the ScalaWiki website [72], which expose an atomicity violation bug. The data inconsistency appears if the `SafeServer` actor interleaves the processing of the `"items"` messages (generated by the `Server` actor in the expressions at line 170 and 173) with the processing of the `Update` message (sent by the `Server` actor at line 94). In Chapter 3 we presented a simplified version of this example where the `SafeServer` and `Server` actors have been renamed *server* and *client*, respectively. Furthermore, in the simplified version, the overall number of messages exchanged between the actors has been reduced, so that the atomicity violation bug stands out. We also renamed the original messages `Update`, `"items"` and `"exit"` (used to shutdown the `SafeServer`), into `Set`, `Get` and `Kill`, respectively. It is important to remark that the communication pattern that leads to the data inconsistency in the simplified code sample is exactly the same of the one exposed by the original version, whose source code is provided here.

In Chapter 7 we used the below code as part of SEJAP's evaluation (we named this code example $\texttt{server}_{real}$ in the Table 7.1 and 7.2).

```
1  // we use an immutable TreeMap to store our items
2  import scala.collection.immutable.TreeMap
3  import scala.xml.{Node, NodeSeq, Elem, Text}
4
5  // We use actors to manage our inventory
6  // to allow for concurrency and remote access
7  import scala.actors.Actor
8  import scala.actors.Actor._
9
10 // define a holder for items
11 // It differs from Step 1's Item in that this implementation is non-mutable.
12 // It's like a Java string, once you create it
13 // you cannot change it. To get a new one, you make a copy.
14 case class Item(name: String, pn: String, price: double, qnty: int) {
15   // add to the quantity, return a new instance
16   // (like "123".substring(2) == "3")
17   def add(toAdd: int) = {
18     itemWithQnty(qnty + toAdd) // create a new, immutable instance
19   }
20
21   // take a certain number out of the item
22   // return the number that was taken and the new item
23   def take(howMany: int) = {
24     val toTake = Math.max(howMany, qnty)
25     // return the number "taken" and the rebuilt item
26     Pair(toTake, itemWithQnty(qnty - toTake))
27   }
28   // a private function that creates a new Item with the new quantity
29   private def itemWithQnty(newQnty: int) = Item(name, pn, price, newQnty)
```

```
30
31    // convert this item to XML
32    def toXml = <item name={name} pn={pn} price={price.toString}
33                  qnty={qnty.toString} />
34  }
35
36  // an "object" (singleton) that extends the "Application"
37  // class will get run like a Java class with public static void main(String[])
38  object Sample2 extends Application {
39    // create mock XML inventory updates
40    val inv1 = <update>
41                    <item pn='a' qnty='25' name='Apple' price='0.25' />
42                </update>
43    val inv2 = <update>
44                    <item pn='o' qnty='45' name="Orange" price='0.4' />
45                    <item pn='b' qnty='4' name="Banana" price="0.15" />
46                </update>
47
48    // Send them to the server
49    Console.println(Server.update(inv1))
50    Console.println(Server.update(inv2))
51
52    // create a mock XML purchase order (note the embedded XML)
53    val purchase = <order>
54                       <item pn='a' qnty='20'/>
55                       <item pn='b' qnty='10'/>
56                       <item pn='o' qnty='35'/>
57                       <item pn='na' qnty='23'/>
58                   </order>
59
```

```
60    // send the XML to the server and print the response
61    Console.println(Server.order(purchase))
62
63    // shut down the server so we can exit gracefully
64    Server.shutdown
65  }
66
67  /*
68   The singleton "inventory server"
69   Implemented as an "Actor"
70   Actors are "share nothing" constructs that receive messages and
71   process them one by one. Actors provide a concurrency model
72   that (1) is deadlock free (or at least deadlock reduced) (2)
73   has been proven in Erlang over 10+ years and (3) does not
74   require "synchronizes" or other keywords
75  */
76  object Server {
77    // the server actor.
78    // vals are created the first time they are used... singletons
79    private val server = {
80      val ret = new SafeServer
81      ret.start
82      ret
83    }
84    private var v1 = 0
85    private var v2 = 0
86
87    // shut the server down by sending it "exit"
88    def shutdown = server ! "exit"
89
```

```
90    def update(in: Elem) = {
91      eachItem(in) {
92        (e, pn, qnty) ⇒
93          // send an update message to our server
94          server ! Update(pn, e.attribute("name").get.text,
95                          java.lang.Double.parseDouble(e.attribute("price").
96                                                       get.text), qnty)
97        // no reason to return anything meaningful because we're not using
98        // the XML return block
99        Text("")
100     }
101     currentInventory // return the current inventory
102   }
103
104   // place an order and return an <order>...</order> XML block
105   def order(in: Elem) = {
106     <order>
107     {
108       eachItem(in) { // for each item
109         (e, pn, qnty) ⇒
110           // send the server an "Order" and wait for the response
111           server !? Order(pn, qnty) match {
112
113             // if there's no matching part, generate a <not_found/> tag
114             case None ⇒ <not_found pn={pn}/>
115
116             case s: Some[Pair[int, Item]] ⇒ {// deconstruct the return value
117               val item = s.get._2 // the item
118               val took = s.get._1 // the number of items taken
119
```

```
120                 // generate the XML tag
121                 <shipped pn={pn} ordered={qnty.toString} shipped={took.toString}
122                     cost={(item.price * took).toString}/>
123               }
124             }
125       }
126     }
127     </order>
128   }
129
130   // iterate over each "item" node and call f
131   private def eachItem(in: Elem)(f : (Node, String, int) ⇒ Node): NodeSeq = {
132     in.child.map { // for each node
133       node ⇒
134         node match {
135           // we've got an <item> tag and an 'pn' attribute
136           case n @ <item/> if (!n.attribute("pn").isEmpty &&
137                                // and a 'qnty' attribute
138                                !n.attribute("qnty").isEmpty) ⇒
139                                  // call the function
140                                  f(n, n.attribute("pn").get.text,
141                                    Integer.parseInt(n.attribute("qnty").get.
142                                                     text))
143
144            // if there's no match, return a "no op" Node
145           case _ ⇒ Text("")
146         }
147     }
148   }
149
```

```
150   // get a list of items from the server
151   private def items: Iterator[Item] = {
152     /*
153       send an "items" message (note that you can pass Any object
154       to an actor server and the server can response with Any object.
155       This is a lot like "duck typing" or dynamic typing. Actors
156       either respond to the message or ignore it.
157     */
158     server !? "items" match {
159       // if the response is an Iterotor[Item] return it
160       case i: Iterator[Item] ⇒ i
161
162       // otherwise, return a zero length iterator
163       case _ ⇒ new Array[Item](0).elements
164     }
165   }
166
167   // generate the XML of the current inventory
168   def currentInventory = {
169     // the inventory value is the sum of price * qnty
170     val invValue = items.foldRight(0.0){(i,sum) ⇒ sum + i.price * i.qnty}
171     // return the XML including the inventory value and the XML of each node
172     <inventory value={invValue.toString}>{
173       items.filter{i ⇒ i.qnty > 0}.
174                     map {i ⇒ Text("\n ") concat i.toXml}.
175                     toList
176     }
177     </inventory>
178   }
179 }
```

```
180
181 /*
182  Scala Actors are stand−alone units of computation. They get messages
183  sent to them and they do something (or nothing) with the messages.
184  They may send an asynchronous response, or not.
185
186  Erlang has a time proven, very successful model of distrubuted computing
187  based on Actors.
188
189  You can read more about Actors in 'Event−Based Programming without Inversion
190  of Control' @ http://lampwww.epfl.ch/~odersky/papers/jmlc06.pdf
191 */
192 class SafeServer extends Actor {
193   // start the message receive loop
194   // with an empty map of our inventory
195   def act = loop(new TreeMap[String, Item])
196
197   // this function continues to receive
198   // messages and processes the messages
199   def loop(info: TreeMap[String, Item]) {
200
201     // receive a message that matches
202     // one of the cases
203     receive {
204
205       // if it's "items" reply to the sender with
206       // the values of our inventory and then loop
207       case "items" ⇒ {reply(info.values) ; loop(info)}
208
209
```

```
210         // exit from listening if we get "exit"
211         case "exit" ⇒ exit("done")
212
213         // If the message is an "Update" object
214         // process the update without responding
215         // to the sender
216         case Update(pn, name, price, qnty) ⇒
217           {
218             /* If the Item is in our inventory,
219                update the quantity (remember that
220                Item is immutable, so an inventory
221                update creates a new instance of Item.
222
223                If the Item is not in inventory, create
224                a new one */
225             val tmpItem = info.get(pn) match {
226               case None⇒ Item(name, pn, price, qnty)
227               case Some(item) ⇒ item.add(qnty)
228             }
229
230             // loop back on ourself with an updated inventory
231             // tree
232             loop(info.update(tmpItem.pn, tmpItem))
233           }
234
235         // Process an "order" which requires a reply
236         case Order(pn, qnty) ⇒
237           {
238             // create a new info tree based on
239             // matching the inventory level
```

```
240             val tmpInfo = info.get(pn) match {
241               // the part number is not found, respond with
242               // None and the info tree does not change
243               case None ⇒ {reply( None); info}
244
245               // we've got a match
246               case Some(item) ⇒ {
247                 // update the item
248                 val res = item.take(qnty)
249                 // reply
250                 reply( Some(res))
251                 // update the info tree
252                 info.update(res._2.pn, res._2)
253               }
254             }
255             // continue looping
256             loop(tmpInfo)
257           }
258       }
259     }
260 }
261
262 // an order has a part number and a quantity
263 case class Order(pn: String, qnty: int)
264
265 // update has more
266 case class Update(pn: String, name: String, price: double, qnty: int)
```

Bibliography

[1] W. AAke. *Functional programming using standard ML.* Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1987.

[2] ActorFoundry webpage. http://osl.cs.uiuc.edu/af/. March 11, 2009.

[3] G. Agha. *Actors: A model of concurrent computation in distributed systems.* MIT Press, Cambridge, MA, USA, 1986.

[4] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1997.

[5] R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Partial-order reduction in symbolic state space exploration. In *Computer Aided Verification*, 1997.

[6] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: A model checker for concurrent software. In *Proc. of the International Conference on Computer Aided Verification (CAV)*, 2004.

[7] C. Artho and P.-L. Garoche. Accurate centralization for applying model checking on networked applications. In *Proceedings of the IEEE International Conference on Automated Software Engineering (ASE)*, pages 177–188, Washington, DC, USA, 2006.

[8] T. Arts and C. B. Earle. Development of a verified Erlang program for resource locking. In *Formal Methods in Industrial Critical Systems*, 2001.

[9] S. J. B. *Practical OCaml*. Apress, Berkely, CA, USA, 2006.

[10] E. Barlas and T. Bultan. NetStub: A framework for verification of distributed Java applications. In *International Conference on Automated Software Engineering (ASE)*, pages 24–33, 2007.

[11] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *Proc. of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2002.

[12] G. Bracha, P. Ahe, V. Bykov, Y. Kashai, and E. Miranda. The Newspeak Programming Platform. http://bracha.org/newspeak.pdf. March 6, 2009.

[13] R. Burmeister. Quality assurance for concurrent software: An actor-based approach. In *Proceedings of the 8th International Workshop on Autonomous Systems—Self-Organization, Management, and Control*, pages 119–126, 2008.

[14] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[15] K. M. Chandy and J. Misra. Distributed computation on graphs: Shortest path algorithms. *Communications of the ACM*, 25(11):833–837, 1982.

[16] E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[17] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, MA, 1999.

[18] ConTest webpage. http://www.alphaworks.ibm.com/tech/contest. 11 March, 2009.

[19] E webpage. http://www.erights.org/elang/index.html. March 11, 2009.

[20] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded Java program test generation. *IBM Systems Journal*, 41(1):111–125, 2002.

[21] C. J. Fidge. Timestamps in Message-Passing Systems that Preserve the Partial Ordering. In *Proceedings of the 11th Australian Computer Science Conference*, pages 56–66, 1988.

[22] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2005.

[23] L.-Å. Fredlund and H. Svensson. McErlang: A model checker for a distributed functional programming language. In *The ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 125–136, 2007.

[24] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. Design patterns: elements of reusable object-oriented software, 1995.

[25] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[26] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Procedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 40, pages 213–223, New York, NY, USA, 2005.

[27] I. Grief and I. Greif. Semantics of communicating parallel processes. Technical report, Cambridge, MA, USA, 1975.

[28] J. Groote and A. Ponse. The syntax and semantics of μCRL. In *Algebra of Communicating Processes, Workshops in Computing, Utrecht, The Netherlands*, 1994.

[29] P. Haller and M. Odersky. Event-based programming without inversion of control. In *Proc. Joint Modular Languages Conference*, 2006.

[30] P. Haller and M. Odersky. Actors that unify threads and events. In *COORDINATION*, 2007.

[31] K. Havelund. Java PathFinder User Guide. *NASA Ames Research*, 1999.

[32] K. Havelund. Java Pathfinder, a translator from Java to Promela. In *Proc. of the International SPIN Workshop on Model Checking of Software (SPIN)*, 1999.

[33] C. Hewitt and H. G. Baker. Laws for communicating parallel processes. In *IFIP Congress*, pages 987–992, 1977.

[34] C. Hewitt, P. Bishop, and R. Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *IJCAI*, pages 235–245, 1973.

[35] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[36] C. Houck and G. Agha. Hal: A high-level actor language and its distributed implementation. In *21st International Conference on Parallel Processing (ICPP '92), vol. II, pp 158-165, St. Charles, IL, August*, 1992.

[37] D. Hughes. A framework for testing distributed systems. In *Proceedings of the 4th IEEE International Conference on Peer-to-Peer computing (P2P'04)*, pages 262–263, 2004.

[38] Ian A. Mason and Carolyn L. Talcott. A semantically sound actor tranlsation. In *ICALP*, pages 369–378, 1997.

[39] R. Iosif. Exploiting heap symmetries in explicit-state model checking of software. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, page 254, Washington, DC, USA, 2001. IEEE Computer Society.

[40] R. Iosif. Symmetry reduction criteria for software model checking. In *Proceedings of the SPIN Workshop on Software Model Checking (SPIN)*, volume 2318 of *LNCS*, pages 22–41, July 2002.

[41] D. Jackson and C. Damon. Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector. In *ISSTA*, pages 239–249, 1996.

[42] Jetlang webpage. http://code.google.com/p/jetlang/. March 11, 2009.

[43] B. Joshua. *Effective Java (2nd Edition) (The Java Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2008.

[44] Java PathFinder webpage. http://javapathfinder.sourceforge.net. March 11, 2009.

[45] JPF related papers. http://javapathfinder.sourceforge.net/JPF_Related_Papers.html. March 11, 2009.

[46] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[47] Jsasb webpage. https://jsasb.dev.java.net/. March 11, 2009.

[48] L. V. Kale and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based On C. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, pages 91–108, 1993.

[49] W. Kim. *THAL: An actor system for efficient and scalable concurrent computing.* Ph.D., University of Illinois at Urbana-Champaign, May 1997.

[50] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[51] T. Landes. Tree clocks: An efficient and entirely dynamic logical time system. In *Parallel and Distributed Computing and Networks*, pages 349–354, 2007.

[52] S. Lauterburg, M. Dotta, D. Marinov, and G. Agha. A Framework for State-Space Exploration of Java-based Actor Programs. (in submission), 2009.

[53] D. Lea. A Java fork/join framework. In *Java Grande*, pages 36–43, 2000.

[54] M. Leuschel and M. Butler. ProB: A model checker for B. In *FME 2003: Formal Methods, LNCS 2805*, pages 855–874. Springer-Verlag, 2003.

[55] S. Liang. *Java Native Interface: Programmer's Guide and Specification*. 1999.

[56] M. Musuvathi and D. L. Dill. An incremental heap canonicalization algorithm. In *Proc. of the International SPIN Workshop on Model Checking of Software (SPIN)*, 2005.

[57] M. Musuvathi, D. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *Proc. of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[58] NAUTY webpage. http://cs.anu.edu.au/~bdm/nauty/. March 11, 2009.

[59] Newspeak webpage. http://newspeaklanguage.org/. March 11, 2009.

[60] M. Odersky. Scala Language Specification - the Scala language reference document. http://www.scala-lang.org/docu/files/ScalaReference.pdf. February 5, 2009.

[61] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima, 2008.

[62] M. Odersky and M. Zenger. Scalable component abstractions. In *OOPSLA*, pages 41–57, 2005.

[63] G. Patrice. Software Model Checking: The VeriSoft Approach. *Form. Methods Syst. Des.*, 26(2):77–101, 2005.

[64] G. L. Peterson. An $O(n \log n)$ Unidirectional Algorithm for the Circular Extrema Problem. *ACM Trans. Program. Lang. Syst.*, 4(4):758–762, 1982.

[65] Ptolemy II webpage. http://ptolemy.berkeley.edu/ptolemyII/. March 11, 2009.

[66] P. Ravi and S. Mukesh. Dependency sequences and hierarchical clocks: efficient alternatives to vector clocks for mobile computing systems. *Wirel. Netw.*, 3(5):349–360, 1997.

[67] Revactor webpage. http://revactor.org/. March 11, 2009.

[68] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular software model checking framework. In *Proc. of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2003.

[69] V. Robert, W. Claes, and W. Mike. *Concurrent programming in ERLANG (2nd ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996.

[70] SALSA webpage. http://www.cs.rpi.edu/research/groups/wwc/salsa/. March 11, 2009.

[71] Scala webpage. http://www.scala-lang.org/. March 11, 2009.

[72] ScalaWiki website. http://scala.sygneca.com/. March 11, 2009.

[73] M. Schinz. *Compiling scala for the Java virtual machine*. PhD thesis, 2005.

[74] K. Sen and G. Agha. Automated systematic testing of open distributed programs. In *Fundamental Approaches to Software Engineering (FASE)*, pages 339–356, 2006.

[75] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the European Software Engineering Conference and the International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 263–272, 2005.

[76] C. Spermann and M. Leuschel. ProB gets Nauty: Effective Symmetry Reduction for B and Z Models. In *TASE '08: Proceedings of the 2008 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering*, pages 15–22, Washington, DC, USA, 2008. IEEE Computer Society.

[77] S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for Java. In *European Conference on Object-Oriented Programming (ECOOP)*, 2008.

[78] S. D. Stoller. Model-checking multi-threaded distributed Java programs. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 224–244, London, UK, 2000. Springer-Verlag.

[79] S. Thompson. *Haskell: The Craft of Functional Programming, Second Edition*. Addison-Wesley Longman, 1999.

[80] C. Varela and G. Agha. Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices. OOPSLA'2001 Intriguing Technology Track Proceedings*, 36(12):20–34, 2001.

[81] M. Veanes, C. Campbell, W. Schulte, and N. Tillmann. Online testing with model programs. In *Proc. of the European Software Engineering Conference and the International Symposium on Foundations of Software Engineering (ESEC/FSE)*.

[82] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, April 2003.

[83] P. Wadler. Linear types can change the world. In *Programming Concepts and Methods*, pages 347–359. North, 1990.

[84] F. S. Wong and M. R. Ito. Parallel Sorting on a Re-Circulating Systolic Sorter. *Comput. J.*, 27(3):260–269, 1984.

[85] M. Yabandeh, N. Knežević, D. Kostić, and V. Kuncak. CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems. In *NSDI*, 2009. (to appear).

[86] J. Yang, T. Chen, M. Wu, Z. Xu, H. L. Xuezheng Liu, M. Yang, F. Long, L. Zhang, and L. Zhou. MODIST: Transparent model checking of unmodified distributed systems. In *NSDI*, 2009. (to appear).