Why Do Scala Developers Mix the Actor Model with Other Concurrency Models?*

Samira Tasharofi, Peter Dinges, and Ralph Johnson

Department of Computer Science, University of Illinois at Urbana–Champaign, USA
tasharo1@illinois.edu, pdinges@acm.org, rjohnson@illinois.edu

Abstract. Mixing the actor model with other concurrency models in a single program can break the actor abstraction. This increases the chance of creating deadlocks and data races—two mistakes that are hard to make with actors. Furthermore, it prevents the use of many advanced testing, modeling, and verification tools for actors, as these require *pure* actor programs. This study is the first to point out the phenomenon of mixing concurrency models by Scala developers and to systematically identify the factors leading to it. We studied 15 large, mature, and actively maintained actor programs written in Scala and found that 80% of them mix the actor model with another concurrency model. Consequently, a large part of real-world actor programs does not use actors to their fullest advantage. Inspection of the programs and discussion with the developers reveal two reasons for mixing that can be influenced by researchers and library-builders: weaknesses in the actor library implementations, and shortcomings of the actor model itself.

1 Introduction

The actor model [1] for concurrent and parallel programming is gaining popularity as multi-core architectures and computing clouds become common platforms. The model's restriction of communication to asynchronous message-passing simplifies reasoning about concurrency, guarantees scalability, allows distributing the program over the network, and enables efficient tools for testing [17,32], modeling [30] and verifying [33] actor programs.

These advantages, however, depend on an intact actor abstraction. Programmers mixing the actor model with other concurrency models can easily break the abstraction. It increases their chance of committing mistakes that the actor semantics carefully avoid: shared state between actors breaks transparent distribution and can introduce fine-grained data races; and blocking and synchronous communication can lead to deadlocks. Furthermore, most of the tools for testing actor programs lose their bug detection capability and their efficiency when used on programs that mix concurrency models.

When examining Scala [23] programs available from public github[1] repositories that use either the original Scala actor library [11] or Akka [5], we discovered

* The original publication is available at http://www.springerlink.com

[1] https://github.com

that many of the programs mix actors with other kinds of concurrent entities such as Java threads.

This raised the question why programmers gave up the advantages of actors and mixed them with threads. Was it a temporary measure, as the programmers converted thread-based parallelism to actors? Does this indicate problems with the actor model, with the implementation of the actor libraries for Scala, or in the education of Scala programmers?

In this paper, we formulate three research questions to study the phenomenon of mixing concurrency models:

RQ1. *How often do Scala programmers mix actors with other kinds of concurrent entities?* This question obviously goes far beyond Scala, but we decided to look first at Scala before looking at other languages.

RQ2. *How many of the programs are distributed over the network, and does distribution influence the way programmers mix concurrency models?* Our motivation for this question is that the actor model can be used to exploit multiple local processors, as well as to distribute the program over the network. Hence, one reason for mixing concurrency models could be that some models are better for particular kinds of programming than others.

RQ3. *How often do the actors in the programs use communication mechanisms other than asynchronous messaging?* Communication through asynchronous messaging reduces the possibility of deadlock and data races, which are common problems in the shared-memory model. However, in Scala, actors can also communicate via other mechanisms such as shared locks. The motivation for this research question is to find out if mixing the actor model with other concurrency models is related to the advantages of asynchronous communication, that is, whether developers use actors for those parts of the program that have high risk of data races or deadlocks.

This paper describes how we selected programs to study (Section 3), the way we measured them, the resulting measurements (Section 4), and the conclusions we drew. We also contacted the developers, and they provided many insights into the meaning of our observations (Section 5). Our findings (Section 6) reveal that the reasons for mixing the actor model with other concurrency models are mostly due to weaknesses in the implementations of the libraries. However, they also show weaknesses in the actor model itself, as well as in the experience of developers.

In summary, this work makes the following contributions:

1. It is the first to point out the phenomenon of Scala developers mixing the actor model with other concurrency models. This phenomenon is at odds with the accepted wisdom about the actor model, which says that its benefits from no shared state and asynchronous communication outweigh its drawbacks.
2. It gives statistics about mixing actors with other kinds of concurrent entities in real-world Scala programs.
3. It gives recommendations for researchers and actor-library designers.

2 Background: Concurrent Programming with Actors

Actors [1,13] are a model of concurrent programming. They are concurrently executing objects that communicate exclusively via asynchronous messages. Each actor buffers the messages it receives in a mailbox and processes them sequentially, one at a time. Upon processing a message, an actor can change its state, send messages, or create other actors. The event-based computation model avoids blocking waits for specific messages, which helps to keep clear of deadlocks in the system. Each message is processed in an atomic step [2]. This reduces nondeterminism in actor programs and makes reasoning about the program easier.

Actor semantics furthermore mandate that each actor is perfectly encapsulated, that is, there is no shared state between actors. This greatly reduces the potential for data races. In combination with asynchronous execution, the lack of shared state allows actor programs to fully exploit the processing cores of current—and future—multi-core processors. Hence, actors offer strong local scalability, which makes them an attractive programming model for modern architectures.

Another trait of actor semantics is location transparent addressing: actors know each other only by unique, opaque addresses. Not having to specify the (physical) location of a message recipient allows the run-time system to distribute the actors that constitute a program across a computing cluster. Consequently, the actor model provides scalability beyond single machines.

Actor libraries. Obtaining the scalability benefits does not require a language that enforces *strict* actor semantics; it is sufficient to have a library providing asynchronous messaging between concurrent objects, and to adhere to coding conventions for avoiding shared state. This allows programmers to reap the scalability-benefits of the actor model, and to break the abstraction if desired—for example by introducing shared state between actors or using non-actor concurrency constructs.

Scala [23]—an object–functional language that runs on the Java virtual machine—is one of the most popular languages that follow this path. Its standard library provides non-strict actors as the default concurrency mechanism. We refer to the actors of this implementation as *Scala actors* [11]. Building upon experience from Scala actors, the Akka library [5] supplies another implementation of non-strict actors for Scala. Besides offering better performance, it adds automatic load-balancing, improves the Erlang-style [3] resilience and fault-tolerance, and introduces opaque actor references for better encapsulation.

While making programming more convenient, breaking the actor abstraction has severe drawbacks. For example, most of the tools for testing [17,32], modeling [30], and verification [33] lose their efficiency when used on programs that mix the actor model with other concurrency models. Mixing concurrency models furthermore re-introduces the potential for fine-grained data races and reduces the readability and maintainability of programs.

3 Methodology

This section describes our methodology for compiling the corpus of Scala programs that use actors. It also explains how we gather statistics about these programs. We use these statistics to answer our research questions in Section 4 and complement them with discussions with the developers in Section 5.

3.1 The Corpus of Actor Programs

The foundation of our program corpus is the list of publicly available Scala programs on the github repository web site that import the Scala actor library or the Akka library. We ignore programs with less than 500 lines of code, which reduces the initial set of around 750 programs to a list of 270 programs[2]. Since the goal of the corpus is to characterize *real-world actor programs*, we further filter the list of 270 programs, reducing it to the 16 programs shown in Table 1. Our criteria for real-world actor programs are as follows:

(1) **Library Usage:** The program not only imports the Scala or Akka actor library, but also uses the library to implement a portion of its functionality. Note that this does not mean that the program uses the Actor class from the library. We merely require that it uses functionality provided by the library, for example futures or remote actors.
(2) **Size:** Scala makes it easy to import Java libraries and write programs containing a mixture of Java and Scala code. Since our analysis tool is agnostic to the difference, we require that the program consists of at least 3000 lines of code in total. Of these, at least 500 lines must be Scala code, which is more compact than Java code.
(3) **Eco-System:** At least two developers contribute to the project, and these provide a way to contact them. Intuitively, having two developers reduces the chance of completely random design mistakes; the agreement of two developers is more likely to represent a systematic mistake. Furthermore, the source code must compile, and the program must have documentation.

3.2 Data Collection

To collect the data underlying our statistics, we wrote a custom tool for analyzing the selected programs. The tool employs the WALA static analysis framework [10] and accepts the compiled bytecode of a program as input. It scans each class in the class hierarchy of the application, looking for indicators of relevant properties. For example, if the application contains a class that implements the actor base trait, it flags the application as making use of actors. Moreover, the tool detects the use of different communication mechanisms by scanning for the signatures of the relevant methods. For asynchronous communication in Scala actor programs, for example, it looks for the signature of `!` in `scala.actors.OutputChannel`.

[2] `http://actor-applications.cs.illinois.edu/index.html`

Table 1. *The corpus of actor applications studied in this paper.* The *Library* column denotes which actor library in which version the application uses. The *kLoC* columns list the number of thousand lines of source code in the application, without comments or empty lines, and the figures in the *Dev.* column shows the number of people contributing to the project. Detailed information about the selected programs, including their versions, is available at: `http://actor-applications.cs.illinois.edu/selected.html`.

			kLoC		
Library	*Program*	*Description*	*Scala*	*Java*	*Dev.*
Akka 2.0	BlueEyes	Web framework	28.6	–	3
Akka 2.0	Diffa	Real-time data differencing	29.6	5.2	8
Akka 2.1	Evactor	Event processing framework	4.6	–	2
Akka 2.0	Gatling	Stress test tool	8.2	0.6	19
Akka 2.0	GeoTrellis	Geographic data engine	10.1	–	2
Akka 2.0	Scalatron	Multi-player programming game	10.5	–	2
Akka 2.1	SignalCollect	Graph processing framework	4.6	–	4
Akka 2.0	Socko	Web server	5.7	1.6	5
Akka 2.0	Spray	RESTful web services library	15.8	–	8
Scala 2.9	BigBlueButton	Web conferencing system	0.8	52.5	30
Scala 2.7	CIMTool	Modeling tool	3.6	26.5	3
Scala 2.10	ENSIME	Scala interaction mode for emacs	8.0	–	19
Scala 2.9	Kevoree	Distributed systems platform	31.5	39.8	9
Scala 2.9	SCADS	Distributed storage system	26.3	1.0	15
Scala 2.9	Spark	Cluster computing system	12.2	–	17
Scala 2.9	ThingML	Modeling language for distributed systems	8.9	61.1	6

We chose this approach to overcome the lack of type and inheritance information in a purely string-based source code analysis. It allows us to gather data with higher precision in the following two situations: First, while programmers typically adhere to the convention of giving actor classes a name ending in `Actor`, this is not enforced by the compiler. Hence, a class `B extends A` that inherits from a class `A extends Actor` is an actor class that cannot be discovered through string matching. Second, because Scala employs type inference, programmers may—and often do—supply only a limited number of type annotations. This makes it hard to discover, for instance, whether an object or class uses a `Lock` for synchronization of concurrent operations. Aside from inheritance, other reasons for preferring Java bytecode over source code were the available tool set, and the ability to easily include mixed Scala–Java programs in our study.

The drawback of analyzing bytecode is the reduced precision of the results: the compiler discards (from its perspective) superfluous static information, which is thus no longer available to our tool. Consequently, some cases that could have been detected during compilation may now go unnoticed. However, the property detection methods we use are sound; our results are therefore lower bounds. In Section 4, we explain the detection mechanisms in detail.

4 Results

In this section, we answer our research questions with statistical data gathered from the programs in our corpus.

4.1 RQ1: How often do Scala programmers mix actors with other kinds of concurrent entities?

The Scala and Akka actor libraries provide two main constructs for implementing concurrent entities: Actor and Future. Futures are place-holders for asynchronously computed values; they block the current thread of execution when it tries to access a yet unresolved value. This way, futures provide a light form of synchronization between the producer and the consumer of the value. With their blocking result-resolution semantics, futures provide a natural way of adding partial synchrony to actor programs. Furthermore, Scala supports access to the Java standard library, which provides more conventional constructs for concurrent computation such as Runnable and Future (from java.util.concurrent). The programs in our corpus can therefore employ a mixture of actors, futures, and threads (via Runnable) to implement concurrent entities.

As an example, Listing 1 shows a code snippet from ENSIME, one of the programs in our corpus. In this code, DebugManager is an actor that has an inner class, MonitorOutput, which inherits from Runnable (Thread). When an instance of DebugManager is created, it initializes a list of MonitorOutputs, passing appropriate input streams to their constructors (Line 4). When DebugManager is started, it also starts the MonitorOutput threads in the list (Line 8), which leads to the execution of the run method in the MonitorOutputs (Line 24). Once a MonitorOutput is started, in a while loop, it reads data from the input stream and sends a message to project, which is itself an actor (Line 30). The DebugManager actor uses the finished variable (Line 23) to interrupt MonitorOutputs and break the while loop if desired (Line 29).

This example shows the mix of actors and threads in a single program. Our tool detects the elements of such a mixture of concurrent entities by checking for usage of Actor, Future, and Runnable. Recall that our tool does not count the usage of concurrent entities in the actor libraries, but only in the program itself. The use of the actor concurrency paradigm is detected via subclassing: if the application contains a class that implements the actor base trait, the tool marks the application as making use of actors[3]. Similarly, it detects thread-based concurrency through subclasses of java.lang.Runnable. Since application classes rarely inherit from futures, the tool detects the use of this concurrent entity by looking for fields, parameters, or local variables whose type is (a subtype of) Future.

[3] We furthermore include a detector for actors created by one of Scala's actor factory functions.

```
class DebugManager(project: Project, indexer: Actor,[...]) extends Actor {
  [...]

  private val monitor = List(new MonitorOutput([...]),new MonitorOutput([...]))

  def start() {
     [...]
     monitor.map { _.start() }
     [...]
  }
  def act() {
     loop {
     [...]
       receive {
         [...]
       }
     }
  }
  [...]

  private class MonitorOutput(val inStream: InputStream) extends Thread {
    val in = new InputStreamReader(inStream)
    @volatile var finished = false
    override def run() {
      try {
        var i = 0
        val buf = new Array[Char](512)
        i = in.read(buf, 0, buf.length)
        while (!finished && i >= 0) {
          project ! AsyncEvent(toWF(DebugOutputEvent(new String(buf, 0, i))))
          i = in.read(buf, 0, buf.length)
        }
      } catch {
        case t: Throwable => {
          t.printStackTrace()
        }
      }
    }
  }
}
```

Listing 1. A code snippet from ENSIME; see `https://github.com/aemoncannon/ensime/blob/d96f4e61ee85a07665348cb3933db7423082b428/src/main/scala/org/ensime/server/DebugManager.scala`

Table 2. *The usage of concurrency constructs.* A bullet (•) in the respective column means that the program contains a class that is derived from Actor; that is derived from Runnable; or that contains a field, parameter, or local variable whose type is (a subtype of) Future.

Program	*Actor*	*Runnable*	*Future*	*Program*	*Actor*	*Runnable*	*Future*
BlueEyes	•	•	•	BigBlueButton	•	•	–
Diffa	•	•	•	CIMTool	•	•	–
Evactor	•	–	–	ENSIME	•	•	–
Gatling	•	–	–	Kevoree	•	•	•
GeoTrellis	•	–	•	SCADS	–	•	•
Scalatron	•	•	•	Spark	•	•	•
SignalCollect	•	–	•	ThingML	•	–	–
Socko	•	•	–				
Spray	•	•	•				

Observations. The results in Table 2 show that 13 of the 16 programs (81%) mix concurrent entities and 12 of the 15 programs (80%) mix Actor with Runnable or Future. Specifically, the results indicate that futures alone seem to be insufficient to handle the concurrency related tasks of the programs: none of the programs relies solely on futures.

The use of futures together with actors has been long established and can be found in actor languages as early as ABCL [35]. Following this tradition, the Scala and Akka actor libraries support a special asynchronous messaging primitive for actors that returns a future. It is therefore not surprising to find that 8 out of 15 programs (53%) that use actors also use futures.

In Table 2, 10 out of 15 programs (66%) use both Actor and Runnable. The reasons for mixing Actor and Runnable are unclear. For example, while in Listing 1, the MonitorOutput could be an actor, which brings consistency to the concurrency model, it inherits from Runnable. One hypothesis would be that the program development started with thread-based concurrency, and later on shifted towards actors. However, by manually inspecting the programs and asking the developers for clarification, we discovered that this hypothesis is wrong. We discuss the details of our findings in Section 5.

4.2 RQ2: How many of the programs are distributed over the network, and does distribution influence the way programmers mix concurrency models?

The previous section shows that mixing the actor model with other concurrency models is common, and that the reason is not historical evolution. Another explanation could be that some models excel at a particular kind of programming. While the actor model allows both exploiting local processing resources, and distributing the program over the network, threads and shared-memory communication are limited to exploiting local processing resources.

As our second question, we ask whether this difference in support for distributed programming could be the reason for mixing concurrency models. Maybe programmers use actors for distributing the program over the network, but prefer threads for using the locally available processing resources on a multi-core machine.

Both actor libraries enforce the use of a special remote actor API in the case of network distribution. Our analysis tool can therefore distinguish between local and remote actor usage. Table 3 shows the results of searching the application code for invocations of the remote actor API.

Table 3. *The usage of actors for distributed programming.* Distributed programs spread their computation across a network; a bullet (•) in the *Distributed* column marks these. The *Remote* column shows which of the programs contain a class that uses the remote actor facilities provided by the actor library.

Program	*Remote*	*Distributed*	*Program*	*Remote*	*Distributed*
BlueEyes	–	–	BigBlueButton	–	•
Diffa	–	–	CIMTool	–	–
Evactor	–	–	ENSIME	–	–
Gatling	–	–	Kevoree	–	•
GeoTrellis	•	•	SCADS	–	•
Scalatron	–	–	Spark	•	•
SignalCollect	•	•	ThingML	–	–
Socko	–	•			
Spray	–	–			

Observations. Only 3 out of 16 programs use actors for remote deployment. This indicates that most developers use actors to address the local scalability problem, that is, they use actors as a solution for local concurrent programming.

However, we expected more of the applications to be distributed. To identify which of the applications are actually distributed (not necessarily using remote actors), we inspected the program code and contacted developers for confirmation. We found that 7 out of the 16 programs are distributed. This implies that *developers tend to use other ways than remote actors for implementing distributed computations.*

In Section 5 we discuss the reasons the developers gave for preferring other methods of distribution.

4.3 RQ3: How often do the actors in the programs use communication mechanisms other than asynchronous messaging?

The results of the previous section indicate that distributed computing is not the reason for mixing the actor model with other concurrency models. Consequently,

programmers seem to use actors to exploit local computing resources and achieve local scalability. If scalability is the goal, then, as motivated in Section 2, actors should communicate solely via asynchronous messages.

The limitation to asynchronous message-passing helps maintain scalability and avoid data races and deadlocks, but it adds complexity to coordination.

Instead of implementing asynchronous distributed protocols to achieve a coordination task, programmers can follow a different route to solve the coordination problems with Scala and Akka actors. In simple cases, programmers can use the provided synchronous messaging operations for blocking *remote procedure call* operations. Another option for synchronization are futures (see Section 4.1). Finally, Scala and Akka allow programmers to take a third route: programmers can choose to break the actor abstraction and rely on customary coordination methods from the shared-memory model, for example shared locks or latches.

An example of communication via a shared variable is shown in Listing 1. As mentioned in Section 4.1, the DebugManager actor uses a shared variable finished to communicate with each OutputMonitor thread. Listing 2 shows an example of a different way of communication: via latches. The code snippet is taken from Gatling. In the code, the Terminator actor coordinates several DataWriter actors—which are stored in the registeredDataWriters variable—to flush their data to the output stream when all of the users have finished their execution. The Terminator contains a latch which is initialized to the latch object sent in the Initialize message. The message's userCount argument determines the number of users in the program. The Terminator starts in the uninitialized state (Line 41), in which it can only accept Initialize messages (Line 13). After receiving an Initialize message, it changes to the initialized state, in which it can accept RegisterDataWriter and EndUser messages (Line 22). Upon receiving an EndUser message, the Terminator checks if the number of messages received from the users has reached the expected number. If that is the case, it creates futures that ask the DataWriter actors to flush their data to the output stream (Line 31). After all futures have completed their tasks, the Terminator counts down the latch (Line 35) to notify the entity that waits on the latch about the completion of its task. Therefore, the latch serves as communication channel between the Terminator actor and other entities in the program.

While using communication mechanisms other than asynchronous messaging can solve coordination and memory-limitation problems, breaking the actor abstraction re-introduces problems that actor semantics carefully avoid: shared state between actors allows fine-grained data races and breaks transparent distribution; blocking and synchronous operations can lead to deadlocks and, for older versions of the Java standard library, exhaust the available threads in the threadpool implementations of Scala and Akka.

Hence, using actors with communication mechanisms other than asynchronous messaging is a trade-off decision. Our third question asks how common these trade-offs are. We consider three main categories of communication:

```
class Terminator extends BaseActor {

  import context._

  /**
   * The countdown latch that will be decreased when all message are written and all
       scenarios ended
   */
  private var latch: CountDownLatch = _
  private var userCount: Int = _

  private var registeredDataWriters: List[ActorRef] = Nil

  def uninitialized: Receive = {

    case Initialize(latch, userCount) =>
      this.latch = latch
      this.userCount = userCount
      registeredDataWriters = Nil
      context.become(initialized)
  }

  def initialized: Receive = {

    case RegisterDataWriter(dataWriter: ActorRef) =>
      registeredDataWriters = dataWriter :: registeredDataWriters

    case EndUser =>
      userCount = userCount - 1
      if (userCount == 0) {
        implicit val timeout = Timeout(configuration.timeOut.actor seconds)
        Future.sequence(registeredDataWriters.map(_.ask(FlushDataWriter).mapTo[
            Boolean]))
          .onComplete {
            case Left(e) => error(e)
            case Right(_) =>
              latch.countDown
              context.unbecome
          }
      }
  }

  def receive = uninitialized
}
```

Listing 2. A code snippet from Gatling; see `https://github.com/excilys/gatling/blob/974299f78c433dcc7f4a1a46501127a41c37e11c/gatling-core/src/main/scala/com/excilys/ebi/gatling/core/result/terminator/Terminator.scala`

(1) *Non-blocking operations* like sending asynchronous messages (sm); resolving a future (rf); and signaling a synchronization construct (ss), for example counting down a latch or releasing a lock.
(2) *Blocking operations* like waiting to receive a message from a channel (wm); waiting for a future to be resolved (wf); and waiting for a synchronization construct to be signaled (ws), for example waiting on a latch.
(3) *Other operations* that do not fit in either of the above categories, for example communication via external resources like files or shared objects that are not synchronization constructs.

To answer RQ3, our tool searches through all Actor classes in each program, detecting if a class—or any of its super-classes—uses the non-blocking or blocking communication operations described in categories (1) or (2). The tool finds instances of non-blocking or blocking communications by looking for the signatures of the relevant methods. For example, for asynchronous communication in Scala actor programs it looks for invocations of library-defined methods like that of `!` in `scala.actors.OutputChannel`.

If the tool finds an actor class that does not use any of the blocking or non-blocking communication methods, it puts it in the *other* category. However, because of the static nature of our analysis, our tool may not be able to detect indirect use of blocking or non-blocking operations. For example, suppose class ServiceActor is an actor class that has a field, printer, instantiated from class Printer. The Printer class is not an actor class but has a method print that performs asynchronous messaging (non-blocking operation). If ServiceActor *only* communicates by invoking printer.print, then our tool does not diagnose ServiceActor as using non-blocking operations and hence marks it as belonging to the *other* category. Therefore, actor classes in the *other* category may indirectly use blocking or non-blocking operations. To address this problem, we manually inspected the classes reported as using *other* category to confirm that they do not use any of the communication mechanisms from the blocking or non-blocking categories.

By using the above method, as mentioned in Section 3, we obtain a lower bound on the usage of each kind of communication operation. Note that a more precise detection of each kind of communication operation requires a more complex analysis [22] and is beyond the capability of our tool. We consequently leave this task for future work.

The results are shown in Table 4. We removed SCADS because it does not use Actors from the libraries. For every program in Table 4, we mark a communication mechanism with a bullet (•) if we find at least one Actor in the program that uses that mechanism. Otherwise, we mark it with an en-dash (–). Consequently, we mark a program with a bullet for *other* if we find at least one Actor that does not use any of the six blocking or non-blocking communication operations in category (1) or (2).

Observations. As the results show, 2 out of 15 programs (ENSIME and Kevoree) use blocking operations to receive a message. Moreover, two programs (Gatling

Table 4. *Communication of actors with other entities.* A bullet (•) denotes that the program contains an actor that uses a communication mechanism in the respective category. The operations in the *Non-blocking* column are sending asynchronous messages (sm); resolving a future (rf); and signaling a synchronization construct (ss) like a latch. *Blocking* operations are waiting on a message from a channel (wm); waiting for a future to be resolved (wf); and waiting for a synchronization construct (ws). *Other* operations do not fit either of these categories.

Program	*Non-block.*			*Blocking*			*Other*	*Program*	*Non-block.*			*Blocking*			*Other*
	sm	rf	ss	wm	wf	ws			sm	rf	ss	wm	wf	ws	
BlueEyes	•	•	–	–	–	–	–	BBButton	•	–	–	–	–	–	–
Diffa	•	–	–	–	–	–	•	CIMTool	•	–	–	–	–	–	–
Evactor	•	–	–	–	–	–	•	ENSIME	•	–	–	•	–	–	–
Gatling	•	–	•	–	–	–	–	Kevoree	•	–	–	•	–	–	•
GeoTrellis	•	–	–	–	–	–	–	Spark	•	–	–	–	–	–	–
Scalatron	•	–	–	–	–	–	–	ThingML	–	–	–	–	–	–	•
SignalCollect	•	–	–	–	–	–	•								
Socko	–	–	–	–	–	–	•								
Spray	•	–	–	–	–	–	–								

and BlueEyes) contain an Actor that communicates through non-blocking operations on futures or synchronization constructs. However, 6 of the 15 programs (40%) contain an Actor that *only* communicates via an operation of the *other* category. Manual inspection of the actors using *other* communication reveals that in two programs, the actors perform I/O, and in four programs the actors operate on a shared object. In these cases, developers were willing to accept the potential drawbacks of data races and deadlocks to solve the problem at hand.

Recall that these numbers might be lower than the actual values. For example, an actor like DebugManager in Listing 1 uses asynchronous messaging as well as operations of the *other* category (shared object). However, since the tool finds that the actor uses asynchronous messaging, it does not report the actor as using communication from *other* category.

To summarize, *in at least 9 out of 15 programs (60%), actors use communication mechanisms other than asynchronous messaging.*

5 The Reasons for Mixing Concurrency Models

The results presented in Section 4 show that around 68% (11 out of 16) of the real-world Scala actor programs in our corpus mix Runnable with actor library constructs like Actor and Future. To investigate the reasons, we manually inspected these programs and contacted the developers, asking them about the details of their design decisions. In order to avoid a bias toward a specific reason, we omitted potential answers. Instead, we posed open-ended questions of the form: *Why did you implement module X with Runnable and not with Actor?*

We received answers from the developers of 10 programs (all 11 programs except CIMTool). After receiving the answers, we dismissed the initial hypothesis that the programs started with thread-based concurrency that was later (partially) replaced with actors or futures: only for 3 of the 11 programs did the answers indicate such a motivation. In the cases of the other eight programs, the developers desired to have pure actor programs. However, they faced problems and as a consequence decided to replace Actor with Runnable. We categorize the reasons into three groups:

- *Actor library inadequacies*: The reasons in this category are lack of library support for efficient I/O, as well as problems implementing low-end systems, managing many blocking operations, and customizing the default features of the actor implementation.
- *Actor model inadequacies*: Certain protocols are easier to implement using shared-memory than using asynchronous communication mechanisms without shared state.
- *Inadequate developer experience*: Developers lack enough knowledge about the library, or reuse legacy code.

5.1 Actor Library Inadequacies

Efficient I/O. The developers of four programs mention efficient input and output (I/O) management as a reason for using Runnable. They either decided to perform I/O in dedicated threads (and not to use threads from the actor library thread pool) to avoid deadlock cases, or they claimed efficiency and performance benefits through dedicated I/O threads.

BigBlueButton: The developers of BigBlueButton use Runnables for reading and writing I/O streams to avoid blocking actors which are executed on the library thread pool. However, they agree that it is possible to refactor the Runnables into actors running on a specific thread[4]. While the Scala actor documentation describes a pattern for this use case [11], it seems that the pattern is either too obscure or inconvenient to implement.

Spark: Spark is a distributed computation framework that needs to exchange large blocks of data over the network. Because the developers are unsure about the actor library's performance regarding large data transfer, they spawn dedicated threads for handling this task.

> "[...] in ParallelShuffleFetcher, we are receiving large blocks of data from multiple machines. Most actor libraries don't deal well with that – they are optimized for transferring small messages (up to a few hundred bytes) [...], and they might have a small number of IO threads that block when you're sending something bigger. In this case, instead of worrying

[4] `https://groups.google.com/forum/?fromgroups=#!topic/bigbluebutton-dev/2ad-HBeNQeY`

about whether the actor library will handle the transfer well [...] and whether it will affect other messages being sent by other actors, we chose to explicitly spawn threads. I'd love an actor library that also handles large IOs, or exposes asynchronous IO primitives, but I haven't found one."

The above message by one of the developers shows that there is demand for an API that gives programmers control over the I/O operations. Moreover, it shows the lack of documentation of the Scala actor library's capabilities: because the capabilities of the library are *unknown*, the developers chose a *known* solution using Runnables, accepting the design drawbacks.

Spray: The Spray framework builds upon the Akka library, which, unlike the Scala actor library, provides an API for managing I/O. Despite this, the Spray developers implement a custom module for asynchronous network I/O using Runnable. As motivation, the Spray developers explain[5] that tailoring the implementation to the specific use case yields performance benefits over using the (more abstract) API of Akka. This is confirmed by one of the Akka developers.

ENSIME: In ENSIME, multiple Runnables are created and executed to read and write from I/O streams. One of the developers expressed that, since there is no need for the actor mail box, using Runnable has less overhead.

Low-End Systems. Mobile phones and low-end systems are among the target platforms of the Kevoree framework for dynamically reconfigurable distributed systems. While Kevoree uses Scala actors, it uses threads to implement the core components that are shared among all platforms. The developers state performance considerations as their motivation:

"[...] JVM ForkAndJoin implementation and other implementation of Thread are very slow and switching context cost a lot of computational power. Again part of Core section of Kevoree are now write with thread to avoid such limitations. [...] More globally this is true for the whole Scala library which is now growing more and more.[...] Porting such a library of more than 10 mb is now challenging for limited environment (RaspberryPI) and especially for Android, it was a nightmare [...]"

Although there are some actor libraries specialized for writing embedded systems [8], Scala actors are not a proper solution for embedded systems.

Managing and Debugging Many Blocking Operations. The Kevoree middleware also contains parts with many blocking operations. According to explanations by the developers, some operations define atomic actions which should block the calling thread and wait for the completion of the operation. Initially,

[5] `https://groups.google.com/d/msg/spray-user/b4YwS5XUsB8/8q_88qs2Gu0J`

the developers started with a pure actor-based solution in which actors used blocking operations for receiving messages. However, they faced deadlock problems and decided to replace the blocking actors with threads:

> "In earlier versions of Kevoree we used Actor everywhere [...]. That was a mistake because we faced a lot of deadlock use cases. Some deadlock was issues from bug in the ForkAndJoin implementation in JVM and some others went from OS limitation (for example using VPS hosting which limited the number of process). For those reasons critical section of Kevoree now start with some dedicated threads, which costs a little more but is far easier to manage in case of blocking actors."

The developers also mention that Runnable helped them to manage blocking operations:

> "[...] we use plain old thread when dealing with third party library which do some waiting operation internally. Using thread let us to control such blocking operation and allow use to start a sibling watchdog thread when something goes wrong. We could also use ThreadedActor but in this case the benefit is not so important."

As noted by the developers, it is possible to execute an actor in a dedicated thread (ThreadedActor) and manage such blocking cases. However, the developers decided to follow the old way of programming. In fact, when facing problems with actors, the developers replaced some of them with Runnables to debug the program. After finding the root cause of the problems, they decided to stay with Runnables so that they can handle similar problems more easily in the future.

The explanations given by the developers indicates that the abstraction that actor libraries provide over threads complicates conventional debugging approaches.

Customized Actors. The developers of SCADS and BlueEyes implemented their own actor-like entities using Runnable and Future.

SCADS: The SCADS distributed database system aims at improving performance with their customized actors. The problem faced by the developers was that the Scala actor library uses a hard-coded serializaion mechanism (the default Java serializer) when sending messages over the network. To make use of a more efficient serialization mechanism, the developers implemented custom actor-like concurrent objects. These objects are furthermore optimized towards processing the key–value messages customary to the program. While Akka provides an API to customize the serialization of messages, this library was not tried by the developers of SCADS.

BlueEyes: The developers of BlueEyes are interested in having *typed actors*. Neither Scala, nor default Akka actors use type information to characterize the messages that an actor accepts or sends. Hence, the compiler cannot discover

whether an actor sends the wrong type of message to another actor. To have support for this kind of static composition checks, the BlueEyes developers implemented their own actor-like class hierarchy. The classes incorporate in their signature the types of messages that are acceptable for the actor, and the types of messages sent by the actor.

5.2 Actor Model Inadequacies

The developers of BlueEyes found using actors to implement the coordination protocol in their HTTP server harder than implementing the protocol with threads.

> "Now let's look at Actors. They address concurrency and mutual exclusion, but they conflate the two (you either get both or none). They don't address coordination at all – you have to build your own protocols for coordination. This code [...] is all about coordination, so using a lock is much simpler way to implement it than using an Actor."

The problem pointed out by the developers concerns purely asynchronous systems in general and is not restricted to the Akka or Scala actor libraries. To give an intuition of this problem, consider the example shown in Listing 3. The DataProcessor processes an array of data supplied to its processData method and returns an array of results. The results should be ordered such that the value in results(i) corresponds to dataArray(i).

For the sake of performance, data processing is implemented in parallel using the fork–join pattern [20]: for each element in dataArray, a future is created that processes the element and puts the result in the results array. Since the results data structure is shared between the futures, it is protected by a synchronized block. The algorithm waits for the results to become ready by calling awaitAll on the futures and returns the results to the caller.

An alternative implementation of the same algorithm using Scala actors and purely asynchronous communication is shown in Listing 4. In this code, the DataProcessor is an actor that, upon receiving ProcessData message (Line 10), stores administrative information in its local variables and then delegates the processing to worker actors. The administrative information consists of the results array, the customer reference to the sender of the message, the totalCount of data elements in dataArray, and curCount, which is the number of results received so far. For each data element in the dataArray, the DataProcessor creates a Worker actor and sends it a Process message.

Unlike in the previous implementation, to preserve encapsulation, the results array is not shared between the worker actors. Instead, workers send the results to the DataProcessor via ProcessResult messages (Line 36) and let the DataProcessor put them in the results array (Line 22).

There are two issues that need to be resolved with this solution. The first issue is that, because of asynchrony, the DataProcessor may receive the results in an arbitrary order from the worker actors. To address this issue, each Process

```
1  class DataProcessor {
2
3    def processData(dataArray: Array[Data]): Array[Result] = {
4
5      var results = new Array[Result](data.length)
6
7      val workers = for (i <- 0 to data.length - 1) yield
8        future {
9          var r = process(data(i)) //process data
10         synchronized {
11           results(i) = r
12         }
13       }
14
15     awaitAll(20000L, workers: _*)
16     return results
17   }
18
19 }
```

Listing 3. Implementation of a parallel data processor in the shared-memory model.

message not only contains the data element to process, but also its index. The worker actors send the results using the same index in ProcessResult messages. This allows the DataProcessor actor to order the results.

The second issue is that the DataProcessor must know when the results are ready to be sent to the customer. This is addressed through the totalCount and curCount variables. The variable curCount is incremented after each ProcessResult message (Line 23). When it reaches totalCount, the DataProcessor knows that processing is complete and sends the results to the customer (Line 24). Note that the customer variable, which records the sender of the ProcessData message, is needed to return the results to the right client. Unlike in synchronous method invocation, there is no implicit return address.

The example shows that implementing some coordination protocols in the actor model can be more complex than using a shared-memory model. The developers may need to add extra variables and implement more complex logic to handle the asynchrony in the actor model that is not present in the shared-memory model. Specifically, for developers who are new to the actor model, understanding and managing coordination in an asynchronous and no-shared state model might be harder than in the shared-memory model.

To address this problem, prior work has extended the Scala actor library with coordination patterns used in parallel programming, for example joins [12] and divide-and-conquer tasks [15]. More advanced coordination mechanisms for actor systems have also been proposed [4,31,9,27]. However,to the best of our knowledge, none has been integrated with a widely used actor library.

```
class DataProcessor extends Actor {

  var curCount = 0
  var totalCount = 0
  var results: Array[Result] = _
  var customer: OutputChannel[Any] = _

  def act() = loop {
    react {
      case ProcessData(dataArray: Array[Data]) => {
        results = new Array[Result](dataArray.length)
        customer = sender
        totalCount = dataArray.length
        curCount = 0

        for (i <- 0 to totalCount - 1) {
          var worker = new Worker().start()
          worker ! Process(dataArray(i), i)
        }
      }
      case ProcessResult(result: Result, index: Int) => {
        results(index) = result
        curCount += 1
        if (curCount == totalCount) customer ! results
      }
    }
  }
}

class Worker extends Actor {

  def act() = loop {
    react {
      case Process(data: Data, index: Int) => {
        var r = process(data(i)) //process data
        sender ! ProcessResult(r, index)
      }
    }
  }

 }
```

Listing 4. Implementation of a parallel data processor in the actor model.

5.3 Inadequate Developer Experience

In three programs, Socko, Scalatron, and Diffa, the developers did not have any special objection to the actor library. They used Runnable because (1) they used to their traditional style of programming; (2) they had some legacy code and wanted to reuse it; or (3) they did not want to trust a new technology when using Runnable would be enough for implementing the required functionality. They use actors when it is necessary to handle concurrent accesses to an object. In these cases, having automated tools that can detect such inconsistencies and help developers to transform the Runnable to Actor would be helpful.

6 Implications and Discussion

In this section, we combine our analysis results with feedback from the developers to give recommendations to researchers and library designers.

Implications for Researchers. The analysis results show that mixing concurrency models is common in real-world Scala programs that use actor libraries. Each model has its strengths, and developers tend to use the model that best fits the problem. However, the current implementations of actors in the Scala standard library and Akka force developers to use models other than actors to meet the application requirements.

On the one hand, research on modeling, testing, and analysis tools for actor programs should take this into account. Specifically, mixtures of Actor and Future are common, as they help implementing coordination between the purely asynchronous actors. Therefore, unless the proposed tools and approaches for actor programs can handle a mixture of actors with other concurrent entities, only few real-world programs can benefit from them.

On the other hand, the results show that in three cases, mixing actors with threads is unnecessary. Automated tools that can detect such cases and help developers refactor threads to actors in their programs would alleviate the problem of mixing concurrency models.

The actor model itself also puts a burden on developers. The property of no shared state and asynchronous communication can make implementing coordination protocols harder than using established constructs like locks. However, providing a language for coordination protocols would alleviate this problem.

Implications for Library Designers. The library APIs can help developers comply with the best practices of a concurrency model in two ways:

- First, the API can provide commonly required features like modules for efficiently handling or customizing I/O. This would address one of the main problems that Scala developers currently have in pure actor programs.

– Second, it can prevent developers from misusing the library constructs and violating best practices. For example, if messages were restricted to immutable types, actors could not easily share objects by exchanging references through messages. While libraries cannot completely prevent shared state in actors, such a limitation would push developers towards using a proper design.

Apart from the API, library-specific tools for debugging and testing would be beneficial for developers. In particular, the high-level abstraction of actors makes it hard for developers to trust, test, and debug low-level execution. A way to get insight into the execution mechanism would reduce these worries.

Finally, clarifying the limitations and capabilities of the libraries also helps developers make the right decisions during the design and development of their programs.

7 Threats to Validity

Internal threats to the validity of this study concern the accuracy of our data collection tool. An inherent limitation is its use of static analysis: the detected method invocations and usage of concurrency constructs may not represent the usage at run time. Moreover, since our tool cannot detect indirect method invocations, the reported statistics about the communication operations may be lower than the actual values. To alleviate this problem, we supplemented the analysis with manual inspection when the tool could not detect any kind of our targeted communication operations. For the cases that the tool reported the usage of concurrency constructs or communication operations, we randomly selected some reported instances and confirmed the results by manual inspection.

To ensure that the concluded reasons for mixing the actor model with other concurrency models are aligned with the real reasons, we eliminated any bias towards specific answers in our questions to the developers. Moreover, we validated the reasons supplied by the developers against the library documentation and other related resources. We discuss our findings after each developer answer in Section 5.

The external threats are related to how much our results are generalizable. To ensure external validity regarding other Scala actor programs, we (1) obtained our programs from github, which we found to be the most common repository site for Scala programs by surveying the Akka and Scala mailing lists; and (2) target the two most popular actor libraries for Scala.

Our selection criteria (Section 3.1) exclude the majority of programs from the initial list, which greatly shrinks the sample size. However, the criteria ensure high-quality specimens by preventing the inclusion of programs with overly idiosyncratic styles of single programmers and test projects. The criteria also exclude large enough programs that we could not compile; however, only four programs were excluded on this ground. Finally, we compiled our initial list of

programs one year ago. Consequently, it will exclude programs hosted only recently on github. Since we demand a certain maturity of projects, we do not expect this to be problematic.

The actor libraries we target have features similar to many other actor libraries for imperative languages [26,7,16], which also allow mixing threads and actors. We therefore believe that our results hold for actor programs written with these libraries. However, the results may not hold for actor languages like Erlang [3] that put more restrictions on the language constructs to force programmers to comply with the foundations of the actor model.

8 Related Work

To the best of our knowledge, this is the first systematic study of the phenomenon of mixing concurrency models in a single program.

The work most closely related to our study are comparisons between different libraries and paradigms for multi-core programming. They are controlled user studies that aim to determine the productivity of programmers. Nanz et al. [21] compare two object-oriented languages, multi-threaded Java and SCOOP, for concurrent programming. Besides productivity, the comparison also focuses on the correctness of the programs written by the participants. Luff [19] compares three concurrent programming paradigms: the actor model, transactional memory, and standard shared-memory threading with locks, in Java. Pankratius et al. [25] compare Scala as an imperative and functional language with Java as an imperative language for concurrent programming. None of these studies considers the mixing of concurrency models.

Several empirical studies investigate the usage of the concurrency constructs from a single library. Naturally, these studies are confined to the concurrency model of the library and do not discuss mixed models. Weslley et al. [34] study 2000 Java projects to determine the most commonly used Java concurrent library constructs. They also analyze usage trends over time. Similarly, Okur and Dig [24] study programs using the Microsoft parallel libraries. By analyzing programs semantically, they achieve higher precision than the syntactic analysis of Weslley et al. The study of Hochstein et al. [14] concerns the productivity of developers using MPI in a large-scale project.

Other studies [18,6] collect and document common mistakes in the usage of concurrent constructs in a single library that lead to concurrency bugs. These collections help developers and researchers to prevent them in the future. However, they are also confined to the concurrency model of the library.

Another line of work integrates the actor model with task parallelism. Haller et al. [12] augment the Scala actor library with join patterns. PAM [29] adds parallel execution of messages inside of actors to achieve better performance. JCoBox [28] combines actors and futures to implement parallel execution of tasks and synchronous messaging. Immam et al. [15] propose a unified parallel programming model for Scala and Java that integrates the actor model with the divide-and-conquer task parallel model. These works use small benchmarks to

show that implementing certain protocols with their proposed model is easier and can provide better performance than the pure actor model. However, none of these works conducts any study on real-world programs to show the weaknesses of the actor libraries or the actor model. Our study complements these works by supplying the empirical evidence for these weaknesses.

9 Conclusion and Future Work

This study is the first to investigate how often and why developers mix the actor model with other concurrency models, which has severe drawbacks (Section 2). The study uses a corpus of real-world Scala programs collected from public github repositories (Section 3). Statically analyzing the programs reveals (Section 4) that most of the programs (80%) mix actors with other concurrent entities. 66% of the programs combine actors and threads, and 53% combine actors and futures. Moreover, at least 60% of all programs contain an actor that does not communicate via asynchronous messaging. Thus, in some situations, other factors than the advantages of asynchronous message-passing dominate the decisions of developers. Through discussion with the developers (Section 5), we find that the reasons for mixing concurrency models and avoiding asynchronous communication lie in inadequacies of the actor libraries and the actor model itself. In Section 6, we discuss the implications of our findings for researchers and library designers.

A direction for future work is to correlate the phenomenon of mixing concurrency models with bug rates and types. This would reveal whether the phenomenon we observed is actually a problem, and if so, which concurrency constructs may help to remedy it. A related question is whether mixing occurs across different layers of abstraction. For example, mixing may occur only on the lower, more concrete layers of the program while actors prevail on the higher, more abstract layers. Finally, it would be interesting to see how different actor libraries for the same language, for example Scala, affect the design decisions of programmers. Results from such investigation would provide guidance for the library developers.

Acknowledgments

The authors would like to thank Nicholas Chen, Stas Negara, Marjan Sirjani, Yun Young Lee, Minas Charalambides, Philipp Haller, Viktor Klang, and Madan Musuvathi for their comments on the earlier versions of this paper. This paper is based upon work partially supported by the U.S. Department of Energy under Grant No. DOE DE-FG02-06ER25752, and the Army Research Office under award W911NF-09-1-0273. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

1. G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.
2. G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *J. Funct. Program.*, 7(1):1–72, Jan. 1997.
3. J. Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, Kungl Tekniska Högskolan, 2003. `http://www.erlang.org`.
4. R. Atkinson and C. Hewitt. Synchronization in actor systems. In *Proc. of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 267–280, 1977.
5. J. Bonér, V. Klang, R. Kuhn, et al. Akka library. `http://akka.io`.
6. J. S. Bradbury and K. Jalbert. Defining a catalog of programming anti-patterns for concurrent Java. In *Proc. of the 3rd International Workshop on Software Patterns and Quality*, SPAQu'09, pages 6–11, Oct. 2009.
7. S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin. Orleans: cloud computing for everyone. In *Proc. of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 16:1–16:14, 2011.
8. J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. Ambient-oriented programming. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 31–40, 2005.
9. P. Dinges and G. Agha. Scoped synchronization constraints for large scale actor systems. In *Proc. of the 14th international conference on Coordination Models and Languages*, COORDINATION'12, pages 89–103, 2012.
10. J. Dolby, S. J. Fink, and M. Sridharan. T. J. Watson libraries for analysis (WALA). `http://wala.sf.net`.
11. P. Haller and F. Sommers. *Actors in Scala*. Artima Series. 2012.
12. P. Haller and T. Van Cutsem. Implementing joins using extensible pattern matching. In *Proc. of the 10th international conference on Coordination models and languages*, COORDINATION'08, pages 135–152, 2008.
13. C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proc. of the 3rd international joint conference on Artificial intelligence*, IJCAI'73, pages 235–245, 1973.
14. L. Hochstein, F. Shull, and L. B. Reid. The role of MPI in development time: a case study. In *Proc. of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 34:1–34:10, 2008.
15. S. M. Imam and V. Sarkar. Integrating task parallelism with actors. In *Proc. of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12, pages 753–772, 2012.
16. R. K. Karmani, A. Shali, and G. Agha. Actor frameworks for the JVM platform: a comparative analysis. In *Proc. of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 11–20, 2009.
17. S. Lauterburg, M. Dotta, D. Marinov, and G. Agha. A framework for state-space exploration of Java-based actor programs. In *Proc. of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 468–479, 2009.
18. S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. *SIGPLAN Not.*, 43(3):329–339, Mar. 2008.

19. M. Luff. Empirically investigating parallel programming paradigms: A null result. PLATEAU at the ACM Onward! Conference, 2009.
20. T. Mattson, B. Sanders, and B. Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, first edition, 2004.
21. S. Nanz, F. Torshizi, M. Pedroni, and B. Meyer. Design of an empirical study for comparing the usability of concurrent programming languages. In *Proc. of the 2011 International Symposium on Empirical Software Engineering and Measurement*, ESEM '11, pages 325–334, 2011.
22. S. Negara, R. K. Karmani, and G. Agha. Inferring ownership transfer for efficient message passing. In *Proc. of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 81–90, 2011.
23. M. Odersky, L. Spoon, and B. Venners. *Programming in Scala, 2/e*. Artima Series. Artima Press, 2010.
24. S. Okur and D. Dig. How do developers use parallel libraries? In *Proc. of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 54:1–54:11, 2012.
25. V. Pankratius, F. Schmidt, and G. Garretón. Combining functional and imperative programming for multicore software: an empirical study evaluating Scala and Java. In *Proc. of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 123–133, 2012.
26. V. Pech, D. König, R. Winder, et al. GPars. `http://gpars.codehaus.org/`.
27. J. Proença, D. Clarke, E. de Vink, and F. Arbab. Dreams: a framework for distributed synchronous coordination. In *Proc. of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1510–1515, 2012.
28. J. Schäfer and A. Poetzsch-Heffter. JCoBox: generalizing active objects to concurrent components. In *Proc. of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 275–299, 2010.
29. C. Scholliers, . Tanter, and W. D. Meuter. Parallel actor monitors. In *14th Brazilian Symposium on Programming Languages*, 2010.
30. M. Sirjani and M. M. Jaghoori. Formal modeling. chapter Ten years of analyzing actors: Rebeca experience, pages 20–56. 2011.
31. M. Song and S. Ren. Coordination operators and their composition under the actor-role-coordinator (ARC) model. *SIGBED Rev.*, 8(1):14–21, Mar. 2011.
32. S. Tasharofi, M. Gligoric, D. Marinov, and R. Johnson. Setac: A framework for phased deterministic testing of Scala actor programs. The Second Scala Workshop, 2011.
33. S. Tasharofi, R. K. Karmani, S. Lauterburg, A. Legay, D. Marinov, and G. Agha. Transdpor: a novel dynamic partial-order reduction technique for testing actor programs. In *Proc. of the 14th joint IFIP WG 6.1 international conference and the 32nd IFIP WG 6.1 international conference on Formal Techniques for Distributed Systems*, FMOODS'12/FORTE'12, pages 219–234, 2012.
34. W. Torres, G. Pinto, B. Fernandes, J. a. P. Oliveira, F. A. Ximenes, and F. Castor. Are Java programmers transitioning to multicore?: a large scale study of Java FLOSS. In *SPLASH Workshops*, SPLASH '11 Workshops, pages 123–128, 2011.
35. A. Yonezawa, editor. *ABCL: an object-oriented concurrent system*. MIT Press, Cambridge, MA, USA, 1990.