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 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 [11,22], modeling [20] and verifying [23] 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 efficiency when used on programs that mix concurrency models.

When examining Scala [16] programs available from public github[1] repositories that use either the original Scala actor library or Akka [10], we discovered

[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, both, 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?* The communication through asynchronous messaging reduces the possibility of deadlock and data races, which is a common problem 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 the experience of developers.

In summary, this work makes the following contributions: (1) it is the first to point out a phenomenon that is at odds with the accepted wisdom about actors; (2) it gives statistics about the phenomenon of mixing actors with other kinds of concurrent entities in Scala programs; and (3) it gives recommendations for researchers and actor-library designers.

2 Background: Concurrent Programming with Actors

Actors [1] 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. This reduces non-determinism 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 race conditions. In combination with asynchronous execution, the lack of shared state allows actor programs to fully exploit the processing cores of current—and future—mutli-core processors. Hence, actors offer strong local scalability, which makes them attractive 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 [16]—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*. Building upon experience from Scala actors, the Akka library [10] supplies another implementation of non-strict actors for Scala. Besides offering better performance, it adds automatic load-balancing, as well as Erlang-style [2] resilience and fault-tolerance.

While making programming more convenient, breaking the actor abstraction has severe drawbacks. For example, most of the tools for testing [11,22], modeling [20], and verification [23] 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.

Table 1. The corpus of actor applications

Program	Description	LoC (Scala+Java)	# of Developers
BlueEyes (Akka)	A web framework for Scala	28,638+0	3
Evactor (Akka)	Complex event processing	4,652+0	2
Diffa (Akka)	Real-time data differencing	29,613+5,207	8
Gatling (Akka)	A stress test tool	8,252+632	19
GeoTrellis (Akka)	Geographic data engine	10,131+0	2
Scalatron (Akka)	A multi-player programming game	10,498+0	2
SignalCollect (Akka)	Parallel graph processing framework	4,590+0	4
Spray (Akka)	RESTful web services library	15,795+0	8
Socko (Akka)	A Scala web server	5,683+1,619	5
BigBlueButton (Scala)	Web conferencing system	799+52,513	30
CIMTool (Scala)	A modeling tool	3,598+26,524	3
ENSIME (Scala)	ENhanced Scala Interaction Mode for Emacs	7,975+0	19
SCADS (Scala)	Distributed storage system	26,308+963	15
Kevoree (Scala)	A tool for modeling distributed systems	31,548+39,787	9
Spark (Scala)	Cluster computing system	12,193+0	17
ThingML (Scala)	Modeling language distributed systems	8,938+61,122	6

3.1 The Corpus of Actor Programs

The list of publicly available Scala programs on the github repository web site serves as the foundation of our program corpus. We ignore programs with less than 500 lines of code from the initial set of around 750 programs, which results in 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 [3]. Our criteria for real-world actor programs are as follows:

(1) **Actor Library Usage:** The program uses the Scala or Akka actor libraries to implement a portion of its functionality. Note that this does not mean that the program uses actors. We merely require that it uses functionality provided by the library, for example futures or remote actors.
(2) **Size:** Since Scala makes it easy to import Java libraries and 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. Furthermore, the source code must compile, and the program must have a manual or documentation web site.

3.2 Data Collection

To collect the data underlying our statistics, we analyze each program using a custom-written tool. The tool employs the WALA static analysis framework [7] and accepts the compiled bytecode of a program as input. To increase completeness and accuracy, the libraries used by the program are supplied alongside the

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

actual application code. However, the libraries are specifically marked as such and do not contribute to the statistics.

We chose this approach to overcome the lack of type and inheritance information in a purely syntactic 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 purely syntactic analysis. 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.

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`. Furthermore, Scala supports access to the Java 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.

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.

As explained in section 2, maintaining a consistent model of concurrency is good software design practice, helps reducing errors, and allows the usage of advanced tools. Using our static analysis tool, we check each program for usage of `Actor`, `Future`, and `Runnable`.

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. Following this tradition, the Scala and Akka actor libraries support a special asynchronous messaging primitive for

Table 2. The usage of concurrent constructs

Program	Actor	Runnable	Future	Program	Actor	Runnable	Future
BigBlueButton	$\surd$	$\surd$	$\times$	Kevoree	$\surd$	$\surd$	$\surd$
BlueEyes	$\surd$	$\surd$	$\surd$	SCADS	$\times$	$\surd$	$\surd$
CIMTool	$\surd$	$\surd$	$\times$	Scalatron	$\surd$	$\surd$	$\surd$
Diffa	$\surd$	$\surd$	$\surd$	SignalCollect	$\surd$	$\times$	$\surd$
ENSIME	$\surd$	$\surd$	$\times$	Socko	$\surd$	$\surd$	$\times$
Evactor	$\surd$	$\times$	$\times$	Spark	$\surd$	$\surd$	$\surd$
Gatling	$\surd$	$\times$	$\times$	Spray	$\surd$	$\surd$	$\surd$
GeoTrellis	$\surd$	$\times$	$\surd$	ThingML	$\surd$	$\times$	$\times$

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 this are unclear. 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 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.

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

Table 3. The usage of actors for distributed programming

Program	Uses remote actor	Is distributed	Program	Uses remote actor	Is distributed
BigBlueButton	×	√	Kevoree	×	√
BlueEyes	×	×	SCADS	×	√
CIMTool	×	×	Scalatron	×	×
Diffa	×	×	SignalCollect	√	√
ENSIME	×	×	Socko	×	√
Evactor	×	×	Spark	√	√
Gatling	×	×	Spray	×	×
GeoTrellis	√	√	ThingML	×	×

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 devlopers 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 subsection 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 world, for example shared locks or latches. Breaking the abstraction by sharing state between actors can also avoid memory limitations when actors operate on large (global) data structures.

While the above methods 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 exhaust the available threads in the threadpool implementations of Scala and Akka.

Table 4. Communication of actors with other entities

Program	Non-block sm,rf,ss	Block wm,wf,ws	Other	Program	Non-block sm,rf,ss	Block wm,wf,ws	Other
BigBlueButton	√,×,×	×,×,×	×	Kevoree	√,×,×	√,×,×	√
BlueEyes	√,√,×	×,×,×	×	Scalatron	√,×,×	×,×,×	×
CIMTool	√,×,×	×,×,×	×	SignalCollect	√,×,×	×,×,×	√
Diffa	√,×,×	×,×,×	√	Socko	×,×,×	×,×,×	√
ENSIME	√,×,×	√,×,×	×	Spark	√,×,×	×,×,×	×
Evactor	√,×,×	×,×,×	√	Spray	√,×,×	×,×,×	×
Gatling	√,×,√	×,×,×	×	ThingML	×,×,×	×,×,×	√
GeoTrellis	√,×,×	×,×,×	×				

Hence, using actors with communication mechanims 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:

(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 signalled (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. Detecting *other* communication operations requires a more complex analysis [15] 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 `Actor`s from the libraries. For every program in Table 4, we mark a communication mechanism with √ if we find at least one `Actor` in the program that uses that mechanism. Otherwise, we mark it with ×. Consequently, we mark a program with √ for *other* if we find at least one `Actor` that does not use any of the six blocking or non-blocking communication operations in 1 or 2.

Observations. As the results show, only 2 out of 15 programs (ENSIME and Kevoree) use blocking operations to receive a message. Moreover, only two programs (Gatling and BlueEyes) contains an `Actor` that communicates through non-blocking operations on futures or synchronization constructs. However, *at least* 6 of the 15 programs (40%) contain an `Actor` that communicates via an operation of the *other* category (Note that it is possible that an actor that uses asynchronous messaging also uses operations of the *other* category.). In these cases, developers were willing to accept the potential drawbacks of data races and deadlocks to solve the problem at hand. Manual inspection of some of the actors using *other* communication reveal that in two programs, the actors performed I/O, and in four programs the actors operate on a shared object. 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`.

The developers of BigBlueButton use `Runnable`s 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 `Runnable`s into actors running on a specific thread[4]. While the Scala actor documentation describes a

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

pattern for this use case [8], it seems that the pattern is either too obscure or inconvenient to implement.

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) [...] In this case, instead of worrying about whether the actor library will handle the transfer well [...] we chose to explicitly spawn threads. I'd love an actor library that also handles large I/Os, or exposes asynchronous IO primitives, but I haven't found one."

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 `Runnable`s, accepting the design drawbacks.

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.

In ENSIME, multiple `Runnable`s 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 [...]"

Managing and Debugging Many Blocking Operations. The Kevoree middlware also contains parts with many blocking operations.

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

According to explanations by the developers, some operations define atomic actions which should block the calling thread and await the completion of the operation. Initially, the developers started with a pure actor-based solution in which actors use 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 `Runnable`s to debug the program. After finding the root cause of the problems, they decided to stay with `Runnable`s so that they can handle similar problems easier in the future.

The explanations given by the developers show 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 is a distributed database system which aims at improving performance for exchanging data over the network. Developers implemented custom actor-like concurrent objects to use a more efficient serialization mechanism than the hard-coded Java object serializaion of the Scala actor library.

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.

5.2 Actor Model Inadequacies

The developers of BlueEyes found using actors to implement the coordination protocol in their HTTP server harder than implementing it 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. Coordination mechanisms for actor systems have been proposed in prior research [3,21,6], but to the best of our knowledge, none have been integrated with widely used actor libraries.

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.

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 even 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 synchronization 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.

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 calls and usage of concurrency constructs may not represent the usage at run time. To reduce the probability of errors in our tool, we wrote simple programs as analysis test cases. Results from these tests were confirmed manually.

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 is the most common repository site for Scala programs; and (2) target the two most popular actor libraries for Scala. Our selection criteria (subsection 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 [19,5], 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 translate to pure functional languages like Erlang [2].

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. [14] 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 [13] compares three concurrent programming paradigms: the actor model, transactional memory, and standard shared-memory threading with locks, in Java. Pankratius et al. [18] 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. [24] 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 [17] study programs using the Microsoft parallel libraries. By analyzing programs semantically, they achieve higher precision than the semantic analysis of Weslley et al. The study of Hochstein et al. [9] concerns the productivity of developers using MPI in a large-scale project.

Other studies [12,4] collect and document common mistakes in the usage of concurrent constructs in a single library. These collections help developers and researchers to prevent them in the future. However, they are also confined to the concurrency model of the library.

9 Conclusion

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.

References

1. G. A. Agha. *ACTORS — A Model of Concurrent Computation in Distributed Systems.* MIT Press series in artificial intelligence. MIT Press, 1986.
2. J. Armstrong. *Making reliable distributed systems in the presence of sodware errors.* PhD thesis, Kungl Tekniska Högskolan, December 2003. `http://www.erlang.org`.
3. R. Atkinson and C. Hewitt. Synchronization in actor systems. POPL '77, pages 267–280, 1977.
4. J. S. Bradbury and K. Jalbert. Defining a catalog of programming anti-patterns for concurrent java. SPAQu'09, pages 6–11, 2009.
5. S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin. Orleans: cloud computing for everyone. SOCC '11, pages 16:1–16:14, 2011.
6. P. Dinges and G. Agha. Scoped synchronization constraints for large scale actor systems. COORDINATION'12, pages 89–103, 2012.
7. J. Dolby, S. J. Fink, and M. Sridharan. T. J. Watson libraries for analysis (WALA). `http://wala.sf.net`.
8. P. Haller and F. Sommers. *Actors in Scala.* Artima Series. 2012.
9. L. Hochstein, F. Shull, and L. B. Reid. The role of MPI in development time: a case study. SC '08, pages 34:1–34:10, 2008.
10. V. Klang, R. Kuhn, J. Bonér, et al. Akka library. `http://akka.io`.
11. S. Lauterburg, M. Dotta, D. Marinov, and G. Agha. A framework for state-space exploration of java-based actor programs. ASE '09, pages 468–479, 2009.
12. 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, 2008.
13. M. Luff. Empirically investigating parallel programming paradigms: A null result. PLATEAU at the ACM Onward! Conference, 2009.
14. S. Nanz, F. Torshizi, M. Pedroni, and B. Meyer. Design of an empirical study for comparing the usability of concurrent programming languages. ESEM '11, pages 325–334, 2011.
15. S. Negara, R. K. Karmani, and G. Agha. Inferring ownership transfer for efficient message passing. PPoPP '11, pages 81–90.
16. M. Odersky et al. Scala programming language. `http://www.scala-lang.org`.
17. S. Okur and D. Dig. How do developers use parallel libraries? FSE '12, 2012.
18. V. Pankratius, F. Schmidt, and G. Garretón. Combining functional and imperative programming for multicore software: an empirical study evaluating scala and java. ICSE 2012, pages 123–133, 2012.
19. V. Pech, D. König, R. Winder, et al. GPars. `http://gpars.codehaus.org/`.
20. M. Sirjani and M. M. Jaghoori. Formal modeling. chapter Ten years of analyzing actors: Rebeca experience, pages 20–56. 2011.
21. 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.
22. S. Tasharofi, M. Gligoric, D. Marinov, and R. Johnson. Setac: A framework for phased deterministic testing of scala actor programs. Second Scala Workshop, 2011.
23. 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. FMOODS/FORTE'12, pages 219–234.
24. 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. SPLASH '11 Workshops, 2011.