Actor Concurrency Bugs: A Comprehensive Study on Symptoms, Root Causes, API Usages, and Differences

MEHDI BAGHERZADEH, Oakland University, USA
NICHOLAS FIREMAN, Oakland University, USA
ANAS SHAWESH, Oakland University, USA
RAFFI KHATCHADOURIAN, City University of New York (CUNY) Hunter College, USA

Actor concurrency is becoming increasingly important in the development of real-world software systems. Although actor concurrency may be less susceptible to some multithreaded concurrency bugs, such as low-level data races and deadlocks, it comes with its own bugs that may be different. However, the fundamental characteristics of actor concurrency bugs, including their symptoms, root causes, API usages, examples, and differences when they come from different sources are still largely unknown. Actor software development can significantly benefit from a comprehensive qualitative and quantitative understanding of these characteristics, which is the focus of this work, to foster better API documentation, development practices, testing, debugging, repairing, and verification frameworks. To conduct this study, we take the following major steps. First, we construct a set of 186 real-world Akka actor bugs from Stack Overflow and GitHub via manual analysis of 3,924 Stack Overflow questions, answers, and comments and 3,315 GitHub commits, messages, original and modified code snippets, issues, and pull requests. Second, we manually study these actor bugs and their fixes to understand and classify their symptoms, root causes, and API usages. Third, we study the differences between the commonalities and distributions of symptoms, root causes, and API usages of our Stack Overflow and GitHub actor bugs. Fourth, we discuss real-world examples of our actor bugs with these symptoms and root causes. Finally, we investigate the relation of our findings with those of previous work and discuss their implications. A few findings of our study are: ❶ symptoms of our actor bugs can be classified into five categories, with *Error* as the most common symptom and *Incorrect Exceptions* as the least common, ❷ root causes of our actor bugs can be classified into ten categories, with *Logic* as the most common root cause and *Untyped Communication* as the least common, ❸ a small number of Akka API packages are responsible for most of API usages by our actor bugs, and ❹ our Stack Overflow and GitHub actor bugs can differ significantly in commonalities and distributions of their symptoms, root causes, and API usages. While some of our findings agree with those of previous work, others sharply contrast.

CCS Concepts: • **Theory of computation → Concurrency**; • **General and reference → Empirical studies**.

Additional Key Words and Phrases: Akka actor bugs, Actor bug symptoms, Actor bug root causes, Actor bug API usages, Actor bug differences, Stack Overflow, GitHub.

ACM Reference Format:
Mehdi Bagherzadeh, Nicholas Fireman, Anas Shawesh, and Raffi Khatchadourian. 2020. Actor Concurrency Bugs: A Comprehensive Study on Symptoms, Root Causes, API Usages, and Differences. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 214 (November 2020), 32 pages. https://doi.org/10.1145/3428282

Authors' addresses: Mehdi Bagherzadeh, Oakland University, USA, mbagherzadeh@oakland.edu; Nicholas Fireman, Oakland University, USA, nfireman@oakland.edu; Anas Shawesh, Oakland University, USA, ashaweshn@oakland.edu; Raffi Khatchadourian, City University of New York (CUNY) Hunter College, USA, raffi.khatchadourian@hunter.cuny.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2020 Copyright held by the owner/author(s).
2475-1421/2020/11-ART214
https://doi.org/10.1145/3428282

1 INTRODUCTION

Actor concurrency is becoming increasingly important in the development of real-world software systems, which are built using industrial-strength actor programming frameworks and languages, such as Akka [Lightbend 2019a], Orleans [Bernstein et al. 2014], and Erlang [Armstrong 2007]. For example, Akka actors allow PayPal to serve more than a billion financial transactions per day [Lightbend 2020d], the Spark big data ecosystem to shuffle hundreds of terabytes of data [Apache 2015], and Groupon to provide real-time personalized coupons to 48 million customers [Lightbend 2020c]. Twitter, LinkedIn, HP, Samsung, Walmart, Verizon, CapitalOne, and Weight Watchers are among other users of Akka actor concurrency [Lightbend 2019b]. Unlike multithreaded concurrency, in which threads communicate using shared memory and locks, in actor concurrency, actors communicate using asynchronous message [Agha 1986; Agha and Hewitt 1985]. The use of higher-level actors and messages—instead of lower-level threads and locks—makes actor concurrency less susceptible to some of the standard bugs in multithreaded concurrency, such as low-level data races and deadlocks [Lee 2006]. However, actor concurrency comes with its own bugs that are different from multithreaded concurrency bugs.

There is previous work on the classification of actor bugs [Hedden and Zhao 2018; Torres Lopez et al. 2018], as well as testing [Li et al. 2018; Sen and Agha 2006], debugging [Caballero et al. 2019; Li et al. 2014; Lopez et al. 2019; Torres Lopez et al. 2017], and verification [Bagherzadeh and Rajan 2015, 2017; Charalambides et al. 2019; Clebsch et al. 2015; Colaço et al. 1997; D'Osualdo et al. 2013; Gordon 2019; Haller and Loiko 2016; Lauterburg et al. 2009; Negara et al. 2011; Rajan 2015; Stiévenart et al. 2017; Tasharofi et al. 2012; Tasharofi et al. 2013] of actor software. Although this work advances our knowledge of actor bugs, the fundamental characteristics of actor concurrency bugs, including their symptoms, root causes, API usages, examples, and differences when they come from different sources are still largely unknown. Actor software development can significantly benefit from a comprehensive quantitative and qualitative understanding of these characteristics of actor bugs to foster better API documentation, development practices, testing, debugging, repairing, and verification frameworks. For example, static bug mitigation tools tend to focus on bugs that are classified by their root causes and can use actor bugs root causes and their classification [Hanam et al. 2016]. The same is true for dynamic bug mitigation tools that target bugs that are classified by their symptoms. Symptoms, root causes, and fixes are the main criteria for defining bug classes that bug mitigation tools target [Ball et al. 2003; Monperrus 2014]. In this work, we present the first comprehensive study on these characteristic of real-world actor concurrency bugs in Akka and answer the following research questions:

- ***RQ1: Symptoms of actor concurrency bugs in Akka*** What are the symptoms of real-world Akka actor bugs and their classification? What are the real-world examples of actor bugs with these symptoms? How common are these symptoms?
- ***RQ2: Root causes of actor concurrency bugs in Akka*** What are the root causes of Akka actor bugs and their classification? What are the real-world examples of actor bugs with these root causes? How common are these root causes?
- ***RQ3: API usages of actor concurrency bugs in Akka*** What APIs do Akka actor bugs use? How common are these APIs?
- ***RQ4: Differences of actor concurrency bugs in Akka*** How different are the symptoms, root causes, and API usages for Akka actor bugs in Stack Overflow and GitHub? How different are their commonalities? How different are their distributions?

We conduct our study on Akka actor bugs that we construct using Stack Overflow and GitHub. Stack Overflow is the most common question & answer website, with more than 18 million questions, 28 million answers, 72 million comments, and 4 million developer participants [Stack Exchange

2019]. Akka actor bugs in Stack Overflow give us insight about bugs for which developers may ask questions and receive help in fixing. Similarly, GitHub is the most common code repository for open-source software, with more than 100 million projects, 900 million commits, and 40 million developers [GHTorrent 2020; GitHub 2019; Khari Johnson 2018]. Akka actor bugs in GitHub give us insight about bugs that developers leave in their code and later find and fix. We focus on Akka as it is growing faster than other popular industrial-strength actor frameworks and languages, such as Orleans [Bernstein et al. 2014] and Erlang [Armstrong 2007]. For the past five years, there are 7,291 Akka questions in Stack Overflow, 1.6 and 48 times more than 4,544 and 152 Erlang and Orleans questions, respectively. Similarly, there are 14,098 Akka projects in GitHub, 1.2 and 10.6 times more than 11,312 and 1,325 Erlang and Orleans projects, respectively.

To conduct this study, we take the following major steps. First, we construct a set of 186 real-world Akka actor from Stack Overflow and GitHub via manual analysis of 3,924 Stack Overflow questions, answers, and comments and 3,315 GitHub commits, messages, original and modified code snippets, issues, and pull requests. Second, we manually study these actor bugs and their fixes to understand and classify their symptoms, root causes, and API usages. Third, we study the differences between the commonalities and distributions of symptoms, root causes, and API usages of our actor bugs in Stack Overflow and GitHub. Fourth, we discuss real-world examples of actor bugs with these symptoms and root causes. Finally, we investigate the relation of our findings with previous work and discuss their implications.

A few findings of our study are: ❶ symptoms of our actor bugs can be classified into five categories *Error*, *Unexpected Behavior*, *Incorrect Messaging*, *Incorrect Termination*, and *Incorrect Exceptions*, where *Error* is the most common symptom (36.1%) and *Incorrect Exceptions* is the least common (2.2%), ❷ root causes of our actor bugs can be classified into 10 categories *Logic*, *Race*, *API Confusion*, *Explicit Life Cycle*, *Programming*, *Messaging Patterns*, *Model Confusion*, *Misnaming*, *Misconfiguration*, and *Untyped Communication*, where *Logic* is the most common root cause (19.4%) and *Untyped Communication* and is least common (1.1%), ❸ a small number of Akka API packages (2.7%) are responsible for most of API usages (97.7%) by our actor bugs, and ❹ our Stack Overflow and GitHub actor bugs can differ significantly in the commonality and distribution of their symptoms, root causes, and API usages. While some of our findings agree with those of previous work, others sharply contrast. For example, ❺ our finding that *Logic* is the most common root cause for our actor bugs agrees with that of Hedden and Zhao [2018]. In contrast, ❻ our symptoms *Incorrect Messaging* and *Incorrect Exceptions*, that are symptoms for 27.5% of our actor bugs, and our root causes *API Confusion*, *Model Confusion*, *Misnaming*, *Misconfiguration*, and *Untyped Communication*, that are root causes for 31.9% of our actor bugs, are new and cannot be found in previous work. Finally, we show the implications of our findings, e.g., ❼ Akka bug finding tools and techniques can be enhanced by our new symptoms and root causes.

All the data used in this study are publicly available at *https://tinyurl.com/y4rkwhco* under a Creative Commons Attribution 4.0 International license.

2 METHODOLOGY

In this section, we discuss our methodology for the collection and analysis of our Akka actor bugs from Stack Overflow and GitHub.

2.1 Data Collection

Akka actor bugs in Stack Overflow In Stack Overflow, an "asker" developer asks a question and assigns one to five tags to the question to better specify its topic. "Answerer" developers provide answers or comments for the question, and the asker selects one of these answers as the accepted answer. Questions and answers may include code in their body. There are 18,947,469 questions,

28,717,692 answers, and 72,782,793 comments in Stack Overflow [Stack Exchange 2019], as of December 2019, that are asked and answered by 4,697,218 developer participants.

To identify Akka actor bugs in Stack Overflow, we take the following steps. First, we identify 1,001 Akka actor questions—those with tags [Akka] and [Actor]. Second, we identify 704 actor coding questions—those that include code. Third, we manually analyze these 704 coding questions, all their 926 answers, and 2,294 comments to identify actor bug questions and fix answers. In total, we analyze 3,922 questions, answers, and comments. An actor coding question is considered to be a "bug" question if its asker explicitly identifies a diversion from the expected behavior of its code. The accepted answer for this bug question is considered to be a "bug fix" answer if its answerer explicitly identifies a solution for the unexpected behavior of the code in the question. For this manual analysis, the second and third authors individually analyze actor coding questions, answers, and comments, and reiterate and refine until they agree on the set $\mathscr{S}$ of actor bug questions and fix answers. The first and fourth authors individually analyze, reiterate and refine until they agree and verify $\mathscr{S}$. Our dataset $\mathscr{S}$ includes 130 Akka actor bugs and their fixes from Stack Overflow.

The first author is a Software Engineer and Programming Languages professor with extensive expertise in both actor and multithreaded concurrent systems [Ahmed and Bagherzadeh 2018; Bagherzadeh and Khatchadourian 2019; Bagherzadeh and Rajan 2015, 2017; Khatchadourian et al. 2018; Long et al. 2016]. The second and third authors are Ph.D. students with extensive coursework in actor, multithreaded and cloud systems. The fourth author is a Software Engineer professor with extensive expertise in parallel and streaming systems [Khatchadourian et al. 2019, 2020]. All authors have several years of industrial work experience.

Our data collection steps are in line with the best practices of previous work. Previous work uses Stack Overflow tags, coding questions, and manual analysis often to identify big data [Bagherzadeh and Khatchadourian 2019], concurrency [Ahmed and Bagherzadeh 2018], security [Meng et al. 2018; Yang et al. 2016], and deep learning [Islam et al. 2019, 2020] bugs and questions and answers. We write our code to process Stack Exchange Data Dump [Stack Exchange 2019] and its XML files to identify Akka actor and coding questions. Code snippets are marked with XML tags ⟨*code*⟩⟨/*code*⟩ in the body of questions and answers. Stack Exchange Data Dump is publicly available and covers a time span of over 11 years, from August 2008 to December 2019.

Akka actor bugs in GitHub In GitHub, a developer creates a repository to host a project and saves changes to the project using commits. A commit includes a message that specifies its purpose and changes to the original code. Another developer can open an issue to report a bug or ask for a new feature or open a pull request to ask for the integration of their code changes into the project. There are 83,624,114 projects, 930,401,807 commits and 67,442,279 issues in GitHub [GHTorrent 2020], as of April 2020, that are written by 24,154,883 developers.

To identify Akka actor bugs and fixes in GitHub, we take the following steps. First, we identify 10,832 Scala and Java Akka repositories—those with the keyword "Akka" in their names or descriptions and main languages Scala or Java. We focus on Scala and Java because they are the most common programming languages for Akka. There are 8,741 Scala and 2,498 Java Akka repositories in comparison to 964 C# and 12 C++ Akka repositories. Second, we identify 121 mature Akka repositories that have at least five stars and five contributors [Biswas et al. 2019; Gu et al. 2016; Nadi et al. 2016] and code that includes the import statement `import akka.actor*`. An Akka project must import the classes in the package `akka.actor` to work with actors. Third, we extract all 39,442 commits of Akka repositories and stem the words in their messages. Stemming reduces a word to its base and allows for grouping and similar treatment of words with the same base. For example, the words "fixing", "fixes", and "fixed" all stem from the base "fix". Fourth, we identify 3,315 candidate bug commits whose messages include keywords "error," "bug," "fix," "issue," "mistake," "incorrect," "fault," "defect," and "flaw" [Casalnuovo et al. 2015; Kim and Whitehead 2006; Ray et al. 2016]. Fifth,

we manually analyze these 3,315 commits, their messages, original and modified code snippets, issues, and pull requests to identify actor bugs and fixes.

A GitHub commit is an actor "bug and fix" commit if its message, issues, or pull requests describe a bug in its original code and describe a fix in its modified code where the modified code makes changes to either Akka actor (sub)classes or usages of actor instances. For this manual analysis, the second and third authors individually analyze commits, their messages, original and modified code, pull requests, and issues and reiterate and refine until they agree on the set $\mathscr{G}$ of bug and fix commits. The first and fourth authors individually analyze, reiterate and refine until they agree and verify $\mathscr{G}$.

Our dataset $\mathscr{G}$ of Akka actor bugs includes 56 actor bug commits and their fixes from GitHub. These bugs belong to 12 repositories for Gatling, Spray, PayPal/squbs, akka-persistence-mongo, oracle/wookiee, NewMotion/akka-rabbitmq, kamon-io/kamon-akka, horta-hell, dn4s, amient/affinity, gearpump, MessageClassifier, and akka-cluster-etcd projects. These projects cover a broad spectrum of ranging from load testing to web service development to cloud-based messaging.

Our data collection steps are in line with the best practices of previous work. Previous work uses keywords, maturity criteria, and manual analysis often to identify actor [Hedden and Zhao 2018] and non-actor bug and fix commits [Casalnuovo et al. 2015; Islam et al. 2019, 2020; Kim and Whitehead 2006; Ray et al. 2016] in mature GitHub projects [Biswas et al. 2019; Gu et al. 2016; Nadi et al. 2016]. We use Natural Language ToolKit stemming algorithm [NLTK Project 2020], GitHub Search, GitHub Code Search, and its REST API during the steps above.

Akka actor bug dataset Our Akka actor bug dataset $\mathscr{B}$ includes 186 bugs and their fixes, which is the union of 130 Stack Overflow actor bugs in $\mathscr{S}$ and 56 GitHub actor bugs in $\mathscr{G}$. The scale of our bug dataset is in line with those of previous work. For example, Hedden and Zhao [2018] study 126 Akka actor bugs selected from 12 projects, Torres Lopez et al. [2018] study 23 actor bugs from 11 literary works, and Zhang et al. [2018] study 87 Stack Overflow and 88 GitHub TensorFlow deep learning bugs. We include both run-time and compile-time bugs in $\mathscr{B}$. The characteristics of these bugs are complementary, and the inclusion of both allows us to obtain a comprehensive understanding of the characteristics of our actor bugs. 89.2% of our bugs occur during the execution and 10.8% during the compilation. This inclusion is in line with the practices of previous work. For example, Islam et al. [2019, 2020] include both run-time and compile-time bugs in their dataset of deep learning bugs.

2.2 Data Analysis

RQ1 and RQ2: Symptoms and root causes To answer ***RQ1*** and ***RQ2***, we manually analyze each actor bug and fix in $\mathscr{B}$. For a Stack Overflow bug and fix question and answer, we analyze the question, including its text and code snippets, to identify and classify the symptom of the bug. Similarly, we analyze all the answers of the question, including its accepted answer, and their comments to identify the root cause of the bug. For a GitHub bug and fix commit, we analyze the commit, its message, original code, and all its issues and pull requests to identify and classify the symptom of the bug. Similarly, we analyze the commit, its message, original and modified code, issues, and pull requests to identify the root cause of the bug. We use the open card sort [Fincher and Tenenberg 2005] to classify symptoms and root causes. In the open card sort, there are no predefined symptom and root cause categories; instead, the categories are developed during the sorting process. During the sorting process, the first three authors individually assign categories to and classify symptoms and root causes of actor bugs and reiterate and refine until they all agree.

Our data analysis steps are in line with the best practices of previous work. Previous work uses manual analysis and card sort often to classify concurrency [Ahmed and Bagherzadeh 2018] and big data [Bagherzadeh and Khatchadourian 2019] Stack Overflow questions and answers, symptoms,

root causes, and fixes for deep learning bugs [Islam et al. 2019, 2020], and root causes for actor bugs [Hedden and Zhao 2018; Torres Lopez et al. 2018] and multiple reopenings of bugs [Zimmermann et al. 2012].

RQ3: API usages To answer ***RQ3***, we manually analyze the code snippets for each actor bug and fix in $\mathscr{B}$ and collect the qualified names of Akka API classes for objects that are used in method invocations as receivers. In Akka, a message send is implemented as a method invocation. For a Stack Overflow bug and fix question and answer, we analyze all the code snippets in the question and its accepted answer. Similarly, for a GitHub bug and fix commit, we analyze the original code of the bug and modified code of the fix. To disambiguate classes that have the same name but can exist in different packages, such as `Failure` in `akka.actor.Status` and `scala.util` packages, we study the context around the usage of the class in the code. The first three authors individually analyze the code snippets to find API usages and reiterate and refine until they agree.

RQ4: Differences To answer ***RQ4***, we investigate the differences of our actor bugs in Stack Overflow and GitHub for commonalities and distributions of their symptoms, root causes, and API usages. For commonality, we compare the percentages of symptoms, root causes, and API usages between Stack Overflow and GitHub bugs. For distribution, we use a statistical t-test and a Mann-Whitney U test to investigate if distributions of symptoms, root causes, and API usages are the same for Stack Overflow and GitHub bugs. We report the results that are confirmed by both of these tests.

3 BACKGROUND

In this section, we discuss the basics of the actor concurrency model and its implementation with additional features in Akka actor framework.

3.1 Basic Actor Concurrency

Unlike multithreaded concurrency, where a program is modeled as a set of threads that communicate using shared memory and locks, in basic actor concurrency [Agha 1986; Agha and Hewitt 1985], a program is modeled as a set of actors that communicate by sending, receiving, and processing asynchronous messages. An actor has its thread of execution and behavior and makes its state accessible only through its messages. To send a *fire-and-forget* message, a “sender” actor sends the message without waiting or blocking for its response. To receive a message, a “receiver” actor enqueues the message by adding it to the end of its *mailbox*. Similarly, to process a message, the actor dequeues a message by removing it from the beginning of its mailbox and executes it sequentially to the end before processing the next message in the mailbox. Messages are processed one by one and in the order they are received. During the processing of a message, an actor can change state, change behavior, send a message, or create a new actor.

3.2 Akka Actor Concurrency

To allow for the development of real-world actor software, Akka implements and adds several necessary features to the basic actor model. These features include additional message sending and receiving patterns, parental supervision, failure management, life cycle management, serialization, remoting, clustering, scheduling, dispatching, and testing.

Messaging patterns Using messaging patterns, in addition to an asynchronous fire-and-forget message, a sender actor can send a *request-response* message and wait and block for its response in a *future* variable [Halstead 1985] for a *timeout* period. A “future” is a placeholder for an incomplete task with a result that is not yet ready; it will be ready when the future is “complete.” Similarly, a receiver can *forward* a message it receives, or *route* it using different strategies, such as *round-robin* and *broadcast*. If a “receiver” actor receives a message that it cannot process using its current

behavior, it could *stash* the message in a temporary buffer and processes it when the actor changes to the appropriate behavior. If a message cannot be delivered to its original receiver actor, such as a receiver that is terminated, Akka delivers the message to a special receiver actor `/deadLetters`.

Parental supervision With parental supervision, a *child* actor can only be created by a *parent* actor. The parent is responsible to *supervise* its children and manage their failures. An actor fails when it throws an exception during its creation or message processing. The *supervising strategy* of a supervisor specifies if the supervisor should *resume*, *restart*, or *stop* the child or *escalate* the failure to the supervisor of the supervisor. An actor is created in a parental hierarchy and resides in an *actor system* that provides services, such as scheduling, dispatching, and configuration.

Actor life cycle With life cycles, an actor can be programmatically created and shut down. An actor can *watch* the life cycle of another actor and receive messages, such as a termination message, when the watched actor goes through different stages of its life cycle.

Remoting and clustering With remoting, actors that are in separate Java Virtual Machines (JVMs) can send and receive *serialized* messages. An actor resides in an actor system, which—in turn—reside in a JVM. Several actor systems can reside in one JVM. With clustering, individual actor systems can form a cluster.

Schedulers and dispatchers A “scheduler” schedules the execution of a message send or a task to occur at or during a specific time or time period. A dispatcher assigns a thread to an actor for its execution and processing of its messages. The dispatcher has an executor that provides the thread from a thread pool that it maintains.

4 SYMPTOMS

In this section, we answer ***RQ1*** and discuss the symptoms of our actor bugs, their classification and commonalities, and provide real-world examples of actor bugs with these symptoms and how developers discuss these symptoms. We also compare our symptoms with those of actor bugs from previous work by Bianchi et al. [2018]. For space reasons, we adapt our bug examples from Stack Overflow bugs, which are more likely to include minimal code examples [Stack Overflow 2020]. Our examples are written in Scala.

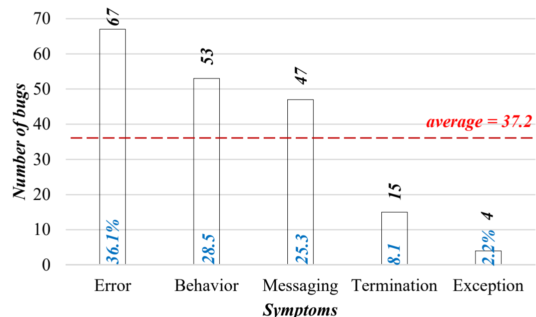


Fig. 1. Symptoms of our actor bugs.

Figure 1 shows the symptoms of our actor bugs, their numbers, and commonalities in terms of percentages. According to this figure, the symptoms of our actor bugs can be classified into the following 5 categories, with decreasing commonalities: *Error*, *Unexpected Behavior*, *Incorrect Messaging*, *Incorrect Termination*, and *Incorrect Exceptions*. Among these, *Error* is the most common symptom (36.1%) and *Incorrect Exceptions* is the least common (2.2%).

4.1 Symptom 1: *Error*

Error is the most common symptom (36.1%) for actor our bugs. The impact of a bug with this symptom is that an actor or its enclosing application throws an error or exception during its execution or compilation. Errors cover a broad spectrum and range from out of memory and timeout errors to non-unique actor name errors.

Figure 2 shows a buggy actor program [Stack Overflow 2017a] with an *Error* symptom. Here, the developer intends to calculate the number Pi, using `Master` and `Worker` actors. `Master` divides the calculation of Pi between its `worker` actors, of type `Worker`, and accumulates their partial calculations.

Master receives a Calc message and to process it creates numWorker number of workers and sends numMessages number of Work messages to each worker. In Akka, an actor is a class that extends the trait Actor. The receive method specifies the messages that an actor receives and processes. The ActorSystem method creates an actor system with a given name and configuration. The actorOf() method creates an actor with a specified name using a Props configuration object. The invocation of actorOf() on an actor system creates an actor that resides in the actor system. The ! operator sends a fire-and-forget message. In Scala, the construct case allows for case analysis using pattern matching. A case class allows for the construction of immutable objects that can be constructed without new. The val keyword declares an immutable value.

However, this program is buggy with an *Error* symptom. The impact of this symptom is that Master actor cannot calculate Pi and instead throws a runtime exception, of type InvalidActorNameException. An InvalidActorNameException is thrown when the name of an actor is invalid. This is because Master attempts to create multiple worker actors with the same name "worker." However, in Akka, actor names must be unique. The developer explains this symptom and its impact as:

```
class Master extends Actor { ..
 val sys = ActorSystem(..) ..
 def receive = {
  case Calc(numWorkers, numMessages, numElements) => {
   for(i <- 0 until numWorkers){
    val worker = sys.actorOf(Props[Worker], "worker")
    for (j <- 0 until numMessages)
     worker ! Work(0, numElements) } } } .. }
class Worker extends Actor { .. }
case class Work(..) { .. }
```

Fig. 2. Actor bug with an *Error* symptom.

"I am getting an error when trying to create a .. Master Akka actor. I am not sure why I am getting this error: [InvalidActorNameException: actor name worker is not unique!]"

An application throwing an out-of-memory exception after it uses all of its available memory, a future throwing a timeout exception after the future is not complete during its timeout period, an actor throwing a transport exception after it sends a message to a remote actor it cannot lookup, and an actor throwing an exception after it cannot process the termination message of the actor it is watching are more examples of our *Error* symptom.

Our *Error* symptom overlaps with Bianchi et al.'s [2018] *Crash* and *Assertion Violation* symptoms. They classify the existing actor testing techniques [Lauterburg et al. 2010; Sen and Agha 2006; Tasharofi et al. 2013] and identify 3 symptoms *Crash*, *Deadlock*, and *Assertion Violation* that these techniques use to identify buggy executions at runtime. Their *Crash* symptom "identifies executions that lead to system crashes." Similarly, their *Assertion Violation* symptom is about the violation of "assertions about the correct behavior of a system." An assertion violation throws an exception.

4.2 Symptom 2: *Unexpected Behavior*

Unexpected Behavior is the second most common symptom (28.5%) for our actor bugs. The impact of a bug with this symptom is that an actor or its enclosing application does not behave in a way that the developer expects from its implementation. Unexpected behaviors cover a broad spectrum and range from misbehaving schedulers, dispatchers, and supervisor actors to misbehaving loggers and method invocations.

Figure 3 shows a buggy actor program [Stack Overflow 2017c] with an *Unexpected Behavior* symptom. Here, the developer intends to write to two files concurrently using a FileWriter actor, which receives a Save message. To process it, the actor writes its string str to a file with the name file. The Main application creates fileWriter and sends two Save messages to it for writing strings str1 and st2 to files file1 and file2, respectively. Main wraps these message sends in future tasks, of type Future, to allow for their concurrent execution. In Scala, an application is a class that extends the App class, which provides the main() entry method.

However, this program is buggy with an *Unexpected Behavior* symptom. The impact of this symptom is that the `fileWriter` actor does not write to `file1` and `file2` concurrently but instead sequentially. This is because, in Akka, an actor processes its messages sequentially. The second `Save` message is processed only after the processing of the first one is finished. The developer explains this symptom and its impact as (punctuation added for clarity):

"I am trying to write to multiple files concurrently using the Akka framework. First, I create a[n] [actor] class called `FileWriter` that writes to a file. Then, using futures, I [send messages to] the [`fileWriter` actor] twice, hoping that 2 files will be created and [written into] for me [concurrently]. But, when I monitor the execution of the program, it first populates [and writes into] the first file and then the second one [sequentially]."

```
class Main extends App { ..
 val sys = ActorSystem(..)
 val fileWriter = sys.actorOf(Props(new FileWriter),..)
 Future { fileWriter ! Save (file1, str1) }
 Future { fileWriter ! Save (file2, str2) } }
class FileWriter extends Actor {
 def receive = {
  case Save(file, str) => saveToFile(file, str) } }
case class Save(file: String, str:String)
```

Fig. 3. Actor bug with a *Behavior* symptom.

A scheduler not scheduling its message sends after the application increases the number of its actors, an actor resetting its state unexpectedly after processing a message, the `sender` method returning the wrong value of the `self` method, and an actor leaking its database connections are more examples of our *Unexpected Behavior* symptom. `sender` returns the sender of the current message of an actor. `self` returns the current actor instance.

Our *Unexpected Behavior* symptom overlaps with Bianchi et al.'s [2018] *Crash* symptom.

4.3 Symptom 3: *Incorrect Messaging*

Incorrect Messaging is the third most common symptom (25.3%) for our actor bugs. The impact of a bug with this symptom is that an expected message is not sent by an intended sender of the message or not received, stashed, or processed by its intended receiver. The opposite holds for an unexpected message.

```
object Starbucks extends App { ..
 val employees =
  List(sys.actorOf(Props[Employee], "Penny"),..)
 val customers = List(("Raj", "Machiato"), ..)
 val router = sys.actorOf(Props.empty.withRouter
  (SmallestMailboxRouter(routees = employees))
 customers foreach { ..
  customer => router ! CanIHave(..) } }
class Employee extends Actor {
 def receive = {
  case CanIHave(coffee, name) => ..
   sender.tell(MakeCofee(cofee, name), ..) .. } }
```

Fig. 4. Actor bug with a *Messaging* symptom.

Figure 4 shows a buggy actor program [Stack Overflow 2019] with an *Incorrect Messaging* symptom. Here, the developer intends to route the requests of the customers of the `Starbucks` coffee shop application to the employees of the shop. `Starbucks` creates the `router` actor, of type `SmallestMailboxRouter`, with a list of employee actors as routees. A router is an actor that receives a message and forwards it to its routees without changing the sender of the message to itself. For each customer in the `customers` list, `Starbucks` sends a `CanIHave` message to `router`, and `router` forwards this message to employees, of type `Employee`. The `Employee` receives a `CanIHave` message, and—to process it—sends a `MakeCoffee` message back to its `sender` for the final preparation of the coffee. The developer assumes that the sender of `CanIHave` is `router`; therefore, `MakeCoffee` is sent to `router`, which forwards this message back to employees. In Akka, the `withRouter()` method configures a router actor with a given routing strategy. A `SmallestMailboxRouter` forwards its message to routees with fewer messages in their inboxes. The `tell()` method sends a fire-and-forget message to its receiver object.

However, this program is buggy with an *Incorrect Messaging* symptom. The impact of this symptom is that the `MakeCoffee` message is not sent to `router`; thus, it is not routed to employees.

Instead, `MakeCoffee` is sent to `/deadLetters`. This is because `router` does not change the sender of the `CanIHave` message that it forwards from `Starbucks` to `Employee`. Therefore, `Starbucks` is the sender of `CanIHave` and not `router`. However, `Starbucks` is an application and not an actor; thus, `sender` in `Employee` evaluates to `/deadletters` instead of `router`. The developer explains this symptom and its impact as:

> "I can't send a [`MakeCoffee` message] reply back to the router so that another [`Employee`] actor in the routing list [`employees`] can pick this [`MakeCoffee` message] up ... when replying [to the router], it will give the following message in the console: ... Message [`router.MakeCoffee$`] from Actor[`akka://actorSystem//user//Penny/#-847662818`] to Actor[`akka://actorSystem/deadLetters`] was not delivered.. dead letters encountered."

The actor name `akka://actorSystem//user//Penny/#-847662818` denotes the `Penny/#-847662818` actor instance for the employee `Penny` in `employees`, which is a `user` actor in the `actorSystem` actor system. Similarly, `/deadLetters` is an actor in `actorSystem`. Akka distinguishes between user-created and system-created actors.

A parent actor not receiving the termination message of its child when it terminates, a mailbox receiving and processing more messages than its specified capacity, an actor losing its stashed messages, and a server actor ignoring messages from its client when they are sent too fast and too frequently are more examples of our *Incorrect Messaging* symptom.

Our *Incorrect Messaging* symptom is new and cannot be found in Bianchi et al.'s [Bianchi et al. 2018] symptoms for actor bugs.

4.4 Symptom 4: *Incorrect Termination*

Incorrect Termination is the fourth common symptom (8.1%) for our actor bugs. The impact of a bug with this symptom is that an actor or its enclosing application does not terminate, terminates prematurely, terminates and restarts infinitely, or hangs.

```
object PureAkka {
 def main(argv : Array[String]) = {
  val actorSystem = ActorSystem(..)
  val myActor = actorSystem.actorOf(Props(new MyActor {
   override def receive = { .. }
   override def preStart() = println("prestart")
   override def postStop() = println("poststop") }))
   /* work with myActor */
  myActor ! PoisonPill } }
class MyActor extends Actor { .. }
```

Fig. 5. Actor bug with a *Termination* symptom.

Figure 5 shows a buggy actor program [Stack Overflow 2012b] with an *Incorrect Termination* symptom. Here, the developer intends to create the actor `myActor`, of type `MyActor`, work with it for some time, and then terminate the application. The `PureAkka` application creates `myActor` and overrides its `receive()`, `preStart()`, and `postStop()` methods. After working with `myActor`, `PureAkka` shuts down the actor. The `preStart()` and `postStop()` life cycle methods are invoked by Akka automatically before an actor starts its execution and after it terminates, respectively. The `PoisonPill` message terminates the actor that receives and subsequently processes it.

However, this program is buggy with an *Incorrect Termination* symptom. The impact of this symptom is that the `PureAkka` application continues its execution and does not terminate even after the termination of the `myActor` actor. This is because terminating `myActor` does not terminate either its enclosing actor system `actorSystem` nor the `PureAkka` application in which it executes. The invocation of `preStart()`—before `myActor` starts its execution—prints "`preStart`" on the console. Similarly, the invocation of `postStop()`—after `myActor` stops its execution—prints "`postStop`." However, both `actorSystem` and `PureAkka` continue their executions after the termination of `myActor`. The developer explains this symptom and its impact as:

"I've written ... code that starts [myActor], kills it and finishes execution. This code prints [to the console]: '[info] prestart' '[info] poststop.' But, [the PureAkka application] refuses to stop until I kill the process with CTRL-C. What does the application wait for?"

An actor not terminating after processing a message that throws an exception, an application hanging after increasing the number of messages that its actors should process, a parent actor waiting on the termination of its children and never terminating, and an application deadlocking when shutting down its actor system are more examples of our *Incorrect Termination* symptom.

Our *Incorrect Termination* symptom overlaps with Bianchi et al.'s [2018] *Deadlock* symptom. Their *Deadlock* symptom "identifies [buggy] executions that lead to deadlocks."

4.5 Symptom 5: *Incorrect Exceptions*

Incorrect Exceptions is the least common symptom (2.2%) for our actor bugs. The impact of a bug with this symptom is that an intended actor does not throw, catch, or properly handle an expected exception. The opposite holds for an unexpected exception.

Figure 6 shows a buggy actor program [Stack Overflow 2013] with an *Incorrect Exceptions* symptom. Here, the developer intends to create the act actor, of type MyActor, and catch and handle the exception that its creation may throw. The app application creates act and surrounds its construction with a try-catch to catch and handle its UserExc exception by printing "failed!" to the console. MyActor defines the constructor this() that throws UserExc. In Akka, the context variable is the context of the current actor. The invocation of actorOf() on context creates an actor as the child of the current actor.

```
object app extends App {
 try {
 var act =
  context.actorOf(Props(classOf[MyActor],..)..)}
 catch { case e: UserExc => println("failed!") } }
class MyActor(..) extends Actor {
 def this(..) { throw new UserExc(..) }
 def receive = { .. } .. }
class UserExc extends Exception { .. }
```

Fig. 6. Actor bug with an *Exceptions* symptom.

However, this program is buggy with an *Incorrect Exceptions* symptom. The impact of this symptom is that the exception UseExc is not caught in the application app. This is because, in Akka, an exception that an actor throws—during its creation or message processing—is caught and handled by its parent actor. Here, the parent of act is the actor /user and not app. In Akka, /user is the parent of all user actors. The developer explains this symptom and its impact as:

"The problem I'm having is that it seems like, in Akka, the context.actorOf() call [in the application app] isn't actually creating the MyActor object [act] itself [in the same thread that app runs on], but deferring it to another thread. So when the constructor [of MyActor] throws an exception, the try-catch block that I put in has no effect."

A sender actor not being able to catch the exception that its receiver actor throws when processing its messages, a child actor not being able to log the exception it throws after its parent actor is terminated, and a test application not being able to catch the exception that its actor under test throws are more examples of our *Incorrect Exceptions* symptom.

Our *Incorrect Exceptions* symptom is new and cannot be found in Bianchi et al.'s [2018] symptoms for actor bugs.

4.6 Implications

Using our new symptoms to extend the set of buggy executions to test for and identify Bianchi et al. [2018] classify Bita [Tasharofi et al. 2013], Basset [Lauterburg et al. 2010] and the work by Sen and Agha [2006] as bug finding tools and techniques for actor software. These works generate inputs, messages and their orders to run the code and use *Crash*, *Deadlock*, and *Assertion Violation* symptoms—as testing oracles—to identify buggy executions and bugs. For example, Bita

finds Akka bugs with *Crash* and *Assertion Violations* symptoms. Our *Error*, *Unexpected Behavior*, and *Incorrect Termination* symptoms overlap with Bianchi et al.'s symptoms. However, our *Incorrect Messaging* and *Incorrect Exceptions* symptoms, accounting for 27.5% of our actor bugs, are new. Previous and future actor bug finding tools and techniques can use our new symptoms to extend the set of executions that they consider as buggy and therefore increase the number of bugs that they may find.

5 ROOT CAUSES

In this section, we answer ***RQ2*** and discuss the root causes of our actor bugs, their classification and commonalities, and provide real-world examples of actor bugs with these root causes and their fixes. We also compare our root causes with those of actor bugs from previous work by Hedden and Zhao [2018] and Torres Lopez et al. [2018].

Figure 7 shows the root causes of our actor bugs, their numbers, and commonalities in terms of percentages. According to this figure, the root causes of our actor bugs can be classified into the following ten categories, with decreasing commonalities: *Logic*, *Race*, *API Confusion*, *Explicit Life Cycle*, *Programming*, *Messaging Patterns*, *Model Confusion*, *Misnaming*, *Misconfiguration*, and *Untyped Communication*. Among these, *Logic* is the most common root cause (19.4%) and *Untyped Communication* is the least common (1.1%).

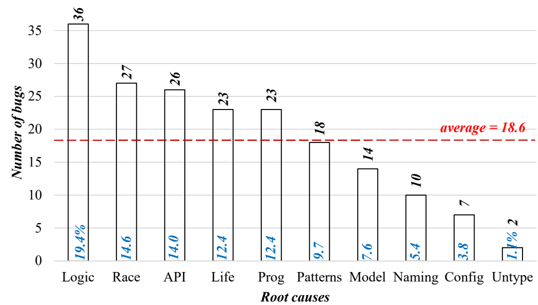


Fig. 7. Root causes of our actor bugs.

5.1 Root Cause 1: *Logic*

Logic is the most common root cause (19.4%) for our actor bugs. A program is a set of steps to implement a logic that transforms the input of the program to its desired output and effects. Akka developers are responsible for the correct implementation of the logic of their programs. Otherwise, incorrect implementation of the logic can cause bugs.

Figure 8 shows a buggy actor program [GitHub 2013a] with a *Logic* root cause. Here, the developer intends to accept a TCP connection—in the `TcpConnection` actor—and transfer data over this connection by writing it to a network socket `channel` of type `SocketChannel`. `TcpConnection` receives a message `Write` and, to process it, writes its `data` to `channel`. Writing to a `SocketChannel` requires a buffer to sit between the channel and the `TcpConnection`. The data is copied to the buffer and then written to the channel from the buffer. Depending on the sizes of the buffer, data, and channel, there are no guarantees that the data can be copied to the buffer or that the buffer can be written into the channel fully at once. Therefore, a loop that attempts to copy and write some data to and from the buffer to the channel, track uncopied and unwritten data, and copy and write them in the next attempt is needed. The

bug

```
class TcpConnection(..) extends Actor .. { ..
 val channel:SocketChannel = ..
 def receive: Receive {
  case Write(data,..) =>
   /* write data into channel */}
 class Pend(rem:ByteString, buffer:ByteBuffer,..)..{
  def writeToChannel(data:ByteString): Pend = { ..
     channel.write(buffer) ..
     if (buffer.hasRemaining) {
      if (data eq rem) this
      else new Pend(data, buffer, ..) }
     else if (data.nonEmpty) { ..
      val copied = rem.copyToBuffer(buffer) ..
      writeToChannel(rem drop copied) } .. } .. }
```

fix

```
      val copied = data.copyToBuffer(buffer) ..
      writeToChannel(data drop copied) } .. } .. }
```

Fig. 8. Actor bug with a *Logic* root cause.

writeToChannel() recursive method of the pend class copy data to buffer, write buffer to channel, and track the unwritten data rem of the buffer. To write data to channel, writeToChannel() first attempts to write the unwritten data of buffer, from its previous invocation, to channel. Afterwards, it checks if there is any unwritten data remaining in the buffer. If there is, writeToChannel() returns the receiver this of its current invocation if data is old and is the same as rem that was not written in its previous invocation. Otherwise, writeToChannel() creates a new Pend object to for writing data that is new and pending to be written to the channel later. If there is no unwritten data remaining in the buffer and data is not empty, writeToChannel() attempts to copy rem to the buffer and invokes itself with any data in rem that cannot be copied.

However, this program is buggy with an *Unexpected Behavior* symptom and a *Logic* root cause. The impact of this symptom is that larger data is broken into smaller parts for copying and writing; however, these parts become mixed and garbled such that the original and transferred data differ. The developer explains this symptom as, "Tcp.Writes get garbled ... if a write [to channel] is [larger] than 300k". The root cause of this bug is that the developer implements the logic for writing the data to the channel incorrectly and copies rem to the buffer, instead of data, although rem will be written to the channel using this. Similarly, the developer invokes writeToChannel() recursively with any uncopied part of rem instead of data. This causes rem—or part of it—to be written into the channel more than once or data—or part of it—not to be written at all. The fix for this bug, as shown in Figure 8, suggests copying data—and not rem—to the buffer and invoking writeToChannel() using the uncopied part of data—and not rem. In Scala, copyToBuffer() copies data into a buffer and returns the number of copied bytes. The drop() method removes the first n bytes of a ByteString.

Our *Logic* root cause overlaps with Hedden and Zhao's [2018] *Logic* root cause. They study 126 Akka actor bugs in 12 projects from the ScalaIndex website and classify their root causes into three categories and ten subcategories, with subcategories inside parentheses: *Logic*, *Communication* (*Response*, *Connection*, *Error Handling*, *Message Order*), and *Coordination* (*Cooperation*, *Shutdown*, *Recovery*, *Workload*, *Operation Order*, *Creation*). Their *Logic* bugs "range from common null pointer errors, optimization issues for buffers ... and any other number of bugs a program must solve during development." In addition, our observation that *Logic* is the most common root cause for our actor bugs agrees with Hedden and Zhao's observation that "*Communication* and *Coordination* [bugs together] occur less than common *Logic* bugs." However, their *Logic* bugs are 57.9% of their bugs, whereas our *Logic* bugs are only 19.4% of our actor bugs.

5.2 Root Cause 2: *Race*

Race is the second most common root cause (14.6%) for our actor bugs. A race happens when two conflicting computations have different execution and program orders. Two computations conflict if they access the same memory region concurrently and at least one of them writes to the region. Execution and program orders specify the orders in which computations execute at run time and occur in the program code at compile time. Akka developers are responsible to write code that is free from races. Otherwise, races can cause bugs.

Figure 9 shows a buggy actor program [Stack Overflow 2018b] with a *Race* root cause. Here, in the actor DownloadImageActor, the developer intends to asynchronously download an image and inform the actor that has requested this download of its success or failure. DownloadImageActor receives a DownloadImage message from a sender actor, say A, and to process it invokes the asynchronous method downloadImage() of imageDownloadService. This asynchronous invocation means that DownloadImageActor does not block and wait for the completion of the invocation result. Instead, the invocation returns immediately with an incomplete future as the placeholder for its result. The future is complete when the invocation finishes its execution. To process the result of the invocation, DownloadImageActor registers an onComplete() callback to be invoked when the future

is complete. The callback sends a `ImageDownloadSuccess` message back to the sender *A* if the value of the future is a `Success`.

However, this program is buggy with an *Incorrect Messaging* symptom and a *Race* root cause. The impact of this symptom is that the sender *A* does not receive the `ImageDownloadSuccess` message, and instead this message is sent to `/deadLetters` actor. The developer explains this symptom as "deadletters encountered." The root cause of this bug is that there is a race between accesses to the value of the sender actor. `downloadImage` and `DownloadImageActor` could run concurrently and on two different threads. By the time `downloadImage` is complete, `DownloadImageActor` could have received a new message from another sender, say *B*, that is different from the original sender *A*. Therefore, `sender()` evaluates to *B* and `ImageDownloadSuccess` is sent to *B*, instead of *A*. Here, not only *B* is the wrong sender but also by the time `ImageDownloadSuccess` is sent to it, *B* does not exists anymore to receive the message, maybe due to termination. Therefore, `ImageDownloadSuccess` cannot be delivered to *B* and instead is delivered to `/deadLetters`. The fix for this bug, as shown in Figure 9, suggests invoking `sender()` outside `onComplete()`, instead of inside, assigning its value to the variable `client`, and send `ImageDownloadSuccess` to `client`, instead of `sender()`.

bug

```
class DownloadImageActor(..) extends Actor .. { ..
 override def receive: Receive = {
  case DownloadImage(jobId, imageUrl) =>
   imageDownloadService.downloadImage().onComplete {
    case Success(image) =>
    sender() ! ImageDownloadSuccess(..)
    case Failure (..) => .. } } }
```

fix

```
   val client = sender()
   imageDownloadService.downloadImage().onComplete {
    case Success(image) =>
    client ! ImageDownloadSuccess(..)
```

Fig. 9. Actor bug with a *Race* root cause.

This bug and its incorrect way of using the mixture of actors and futures is a good example of the misuse of the mixture of concurrency models [Swalens et al. 2014]. These mixtures are often necessary for the implementation of real-world actor software [Swalens et al. 2014; Tasharofi et al. 2013], however their misuse can cause bugs. Tasharofi et al. [2013] study 15 large and mature Scala Akka software and observe that "80% of them mix actor model with another concurrency model [such as multithreaded concurrency]" where "mixtures of Actor[s] and Future[s] are common."

The race in Figure 9 is a simple bug with a well-known anti-pattern of closing over the mutable `sender()` in the callback of a future. This anti-pattern is well-documented in several places, such as Akka documentation [Lightbend 2020b], books [Khot 2018; Lewis and Lacher 2016], and blogs [Manuel Bernhardt 2020]. However, both Stack Overflow and GitHub developers still write buggy actor code with this race as the root cause. In fact, 29.6% of our *Race* bugs close over `sender()`, in the call back of a future or a scheduled message or a task. The anti-pattern behind these bugs is syntactic and can be found using a simple static analysis.

Races between messages that are sent concurrently to the same actor [Bagherzadeh and Rajan 2017; Tasharofi et al. 2013], between asynchronous life cycle actions of the actor, such as creation, initialization, lookup, restart, and termination, and between messages and life cycle actions are more examples of our *Race* root cause. There are actor frameworks and languages that are less susceptible to our *Race* bugs. For example, Orleans [Bernstein et al. 2014] supports more implicit life cycle management and is less susceptible to races between life cycle actions and messages. Similarly, Erlang [Armstrong 2007] prevents sharing among its actors and is less susceptible to races between accesses to shared data.

Our *Race* root cause overlaps with Hedden and Zhao's [2018] *Message Order*, *Operation Order*, and *Cooperation* root causes. Their *Message Order* bugs occur when a "developer expects messages to arrive in a certain order instead of planning around the non-deterministic nature of actor messaging." Similarly, the reason for their *Operation Order* bugs is that "much like message ordering, there are sometimes issues that occur as a result of the [incorrect] order of a job being carried out on a system."

Finally, their *Cooperation* bugs occur because of the "problems occurring from actors performing or existing simultaneously [concurrently]." In addition, our *Race* overlaps with Torres Lopez et al.'s [2018] *Bad Message Interleaving* and *Memory Inconsistency* root causes. They study 23 actor bugs in various actor models from 11 previous works and classify their root causes into 2 categories and 6 subcategories, with subcategories inside parentheses: *Lack of Progress* (*Communication Deadlock*, *Behavioral Deadlock*, and *Livelock*) and *Message Protocol Violation* (*Message Order Violation*, *Bad Message Interleaving*, and *Memory Inconsistency*). Their *Bad Message Interleaving* bugs occur when "a message is processed between two messages which are expected to be processed one after the other." Similarly, their *Memory Inconsistency* bugs occur when "different actors have inconsistent views of shared resources."

5.3 Root Cause 3: *API Confusion*

API Confusion is the third most common root cause (14.0%) for our actor bugs. Akka API provides a large number of 1,438 public classes with 41,554 methods [Akka 2.6.5 API 2020]. In comparison, the general-purpose programming language Scala provides only 491 classes with 38,334 methods [Scala 2.13.2 API 2020]. Akka API classes support a broad spectrum of actor functionalities ranging from untyped to typed actors, remoting to clustering, and supervision to routing. The syntax, semantics, and usage constraints for these classes could differ significantly, even for similar functionalities. For example, unlike any other supervisor actor that by default restarts its children on failure, `BackoffSupervisor` stops and starts its children. In addition, these differences can exist for similar functionalities in different versions of Akka. Akka developers are responsible to understand and satisfy syntax, semantics, and usage constraints of Akka API classes. Otherwise, confusions can cause bugs.

Figure 10 shows a buggy actor program [Stack Overflow 2018c] with an *API Confusion* root cause. Here, the developer intends for the supervisor actor `BackoffSupervisor` to restart its child actor `SenderActor` when it throws an exception and fails. `SenderActor` overrides its `preRestart()` method to send a failure message to its supervisor. Afterwards, it receives a `cmd` message and to process it throws a `MsgExc` exception. Throwing this exception should cause the parent `BackoffSupervisor` to restart its child `SenderActor`. The `Main` application creates the configuration objects `childProps` and `supProps`. A configuration object specifies options that are used in the creation of an actor. `childProps` configures a child actor, of type `SenderActor`, and `supProps` configures a `BackoffSupervisor` with a supervision strategy `OneForOneStrategy` for exception handling. `Main` creates the supervisor actor `sup` with the configuration `supProps` and sends a `cmd` message to it. `sup` forwards this message to its child which fails during its processing. In Akka, a `BackoffSupervisor` restarts its child whenever it throws an exception, but each time with an increasing delay. `OneForOneStrategy` of a parent applies only to the child that fails and leaves other children intact. `PreRestart` is invoked by Akka before an actor restarts.

bug

```
class SenderActor() extends Actor {
 override preRestart(..): Unit = { ..
  /* send a message to supervisor*/ }
 override def receive: Receive = {
  case cmd: .. => throw new MsgExc(..) } .. }
class Main extends App {
 val childProp = Props(new SenderActor())
 val supProp =
  BackoffSupervisor.props(Backoff.onFailure(childProp,..)
  .withSupervisorStrategy(OneForOneStrategy(..) {
   case m: MsgExc => { SupervisorStrategy.Restart}..}))
 val sup = context.actorOf(supProps)
 sup ! cmd }
```

fix

```
  BackoffSupervisor.props(Backoff.onStop(childProp,..)
```

Fig. 10. Actor bug with an *API* root cause.

However, this program is buggy with an *Unexpected Behavior* symptom and an *API Confusion* root cause. The impact of this symptom is that the failed child `SenderActor` does not restart and

its preRestart() is not invoked. The developer explains this symptom as "I attempted to resend a message to the [BackoffSupervisor] actor on preRestart() hook [of SenderActor], but somehow the hook is not being triggered." The root cause of this bug is that the developer does not know that BackoffSupervisor and its Backoff.OnFailure method are significantly different from other supervisors in Akka. Backoff.OnFailure does not restart SenderActor and instead stops and starts it and thus its preRestart() is never invoked. The fix for this bug, as shown in Figure 10, suggests using Backoff.OnStop to restart SenderActor, instead of Backoff.OnFailure.

This bug, similar to the bug in Figure 9, is well-documented in Akka API documentation [Lightbend 2020a]. The documentation says "note that this supervision strategy [onFailure()] does not restart the actor but rather stops and starts it. The preRestart() hook will not be executed if the supervised actor fails or stops." However, we observe that the developer still writes this buggy actor code with this API confusion as the root cause.

Our *API Confusion* root cause is new and cannot be found in Hedden and Zhao's [2018] or Torres Lopez et al.'s [2018] root causes for actor bugs.

5.4 Root Cause 4: *Explicit Life Cycle*

Explicit Life Cycle is the fourth most common root cause (12.4%) for our actor bugs. In Akka, an actor goes through different life cycle phases, such as creation, initialization, lookup, monitoring, termination, and restart. The life cycle of the actor should be managed explicitly and manually. In addition, the explicit life cycle management is combined with the implicit life cycle management. Unlike explicit life cycle management that is implemented by and is visible to the developer, the implicit and automatic life cycle management is implemented by Akka and is invisible to the developer. For example, Akka invokes life cycle methods preStop and postStop of an actor implicitly right before and after it shuts down the actor. Finally, explicit and implicit life cycle managements are combined with features such as parental supervision and actor systems. For example, a parent actor terminates only after all of its children terminate. Akka developers are responsible to understand and correctly manage these life cycles explicitly and manually. Otherwise mismanagements can cause bugs.

Figure 11 shows a buggy actor program [Stack Overflow 2017b] with an *Explicit Life Cycle* root cause. Here, the developer intends to perform some cleanup in the child actor Child before the termination of its parent actor Parent stops and terminates the child. Parent creates its child c. Afterwards, it overrides its postStop() method to send the message "doCleanup" to c and blocks and waits for its result future. Parent continues by stopping and terminating c and then terminating itself. In Akka, the ? operator sends a request-reply message. The result() method blocks on a future for a timeout period and returns the value of the future if it completes during this period. Otherwise, it times out and throws an exception. The stop() method terminates an actor. The self variable refers to the current actor instance.

bug

```
class Parent extends Actor { ..
 val c = context.actorOf(Props[Child])
 override def postStop {
  log.info("Shutdown .. Sending message to child..")
  val future = c ? "doCleanup"
  log.info("Waiting for child to complete task..")
  Await.result(future, Duration.Inf)
  context.stop(c)
  log.info("Child stopped. Stopping self. Bye!")
  context.stop(self) } }
class Child extends Actor { .. }
```

fix

```
class Child extends Actor { ..
 override def postStop {
  //move child cleanup from ParentActor to here }
```

Fig. 11. Actor bug with a *Life Cycle* root cause.

However, this program is buggy with an *Error* symptom and an *Explicit Life Cycle* root cause. The impact of this symptom is that the child c does not do any cleanup. The developer explains this symptom as, "[ERROR] Recipient Actor[akka:..Parent/Child#70868360] had already been

terminated." The root cause of this bug is that the developer misunderstands the explicit and implicit life cycle managements for `Parent` and `c` and their combination with the parental supervision of `c` by `Parent`. A parent actor stops only after all its children terminate and not before. Therefore, when Akka invokes `postStop()` of `Parent` the child `c` is already stopped and cannot receive and process "`doCleanup()`". The fix for this bug suggests overriding `postStop()` of `Child` and perform the cleanup of the child in there, instead of in `postStop()` of `Parent`. In addition to this bug, there is another bug in this program that causes `Parent` to block forever. This is because `Parent` sends a request-reply message to `c` and waits for its reply for the infinity duration of `Duration.Inf`. However, `c` is already stopped and never sends a reply back. Finally, the program in Figure 5 is another example of an actor bug with an *Explicit Life Cycle* root cause. Here, the developer explicitly terminates the actor `myActor` but forgets to explicitly terminate its enclosing actor system `actorSystem`.

Missing the explicit starting of remote actors before messaging them, explicit creation of actors before looking them up, and explicit termination of actors and actor systems before termination of their enclosing applications, are more examples of our *Explicit Life Cycle* root cause. There are actor frameworks and languages that are less susceptible to our *Explicit Life Cycle* bugs. For example, Orleans [Bernstein et al. 2014] supports more implicit and automatic actor life cycle management and is less susceptible to bugs that are due to explicit and manual mismanagements of life cycles.

Our *Explicit Life Cycle* root cause overlaps with Hedden and Zhao's [2018] *Creation*, *Shutdown*, and *Recovery* root causes. Their *Creation* bugs "occur as a result of this [incorrect] actor creation ... [that] can lead to errors if improperly implemented." Their *Shutdown* bugs are bugs "involving a problematic shutdown process [since] every system must shut down at some point and should do so gracefully." Their *Recovery* bugs occur due to "unexpected variables or events involved during this [actor] recreation process [that] if not anticipated would lead to errors [because] the actor model is built to allow actors that fail to recover ... by having its master [supervisor] recreate it."

5.5 Root Cause 5: *Programming*

Programming is the fifth most common root cause (12.4%) for our actor bugs. A program should satisfy the syntactic and semantics requirements of the programming language that is used to write it. Otherwise, violations could cause bugs.

Incorrect dependencies and imports, recreation of singleton objects, matching against erased type variables, and confusing classes with identical unqualified names that belong to different packages are examples of our *Programming* root cause.

Most (60.9%) of our *Programming* bugs are compile-time bugs and can be detected by compilers statically whereas the rest (39.1%) are run-time bugs.

Our *Programming* root cause overlaps with Hedden and Zhao's [2018] *Logic* root cause.

5.6 Root Cause 6: *Messaging Patterns*

Messaging Patterns is the sixth most common root cause (9.7%) for our actor bugs. In Akka, an actor can use several messaging patterns for sending and receiving messages. These patterns, such as request-response, forward, and route, can have complex semantics. For example, for a request-response message Akka creates an additional actor, that is invisible to the developer, to process the future reply of the message. In addition, the complex semantics of these patterns can be combined with the semantics of other features of Akka, such as parental supervision and exception handling. Akka developers are responsible to understand these complex semantics to use these patterns correctly. Otherwise, misuses can cause bugs.

Figure 12 shows a buggy actor program [Stack Overflow 2016] with a *Messaging Patterns* root cause. Here, the developer intends for the parent actor `Deletion` to send a delete message to its child `ES` and either receive a reply for the successful deletion or restart the child if it throws an

exception and fails. Deletion overrides its supervision strategy to restart its child when it throws an exception and creates the child actor es, of type ES. Afterwards, it sends a request-reply message DeleteFromES to es and blocks on its future reply f for the duration of timeout. ES overrides its preRestart() and postRestart() methods. Afterwards, it receives the message DeleteFromES and to process it throws an exception, of type Exception. Throwing this exception should cause the parent Deletion to restart the child ES. In Akka, the ask method sends a request-reply message.

However, this program is buggy with an *Unexpected Behavior* symptom and a *Messaging Patterns* root cause. The impact of this symptom is that the child es does not restart and its preRestart() and postRestart() methods are not invoked. The developer explains this symptom as, "postRestart() and preRestart() methods [of es] are not getting invoked." The root cause of this bug is that the developer misunderstands the semantics of the request-response pattern and its combinations with the semantics of parental supervision and exception handling. The parent Deletion sends a request-response message and blocks and waits for some time for the reply from the child es. However, es does not send the reply and instead throws an exception, that is stored in f, and fails. es may throw its exception before or after Deletion's wait is over. For before, Deletion is still blocked and cannot restart es. For after, Deletion throws a timeout exception and fails itself and cannot restart es. The fix for this bug, as show in Figure 12, suggests using the callback onComplete() on f, to identify if the result of the request-reply message is a success or a failure.

bug

```
class Deletion extends Actor { ..
 override val supervisorStrategy =
  OneForOneStrategy(..){case _:Exception => Restart}
 val es = context.actorOf(Props[ES]..) ..
  val f = ask(es, DeleteFromES(..))
  val isDel = Await.result(f, timeout.duration) ..}}
class ES extends Actor { ..
 override def preRestart() { .. } ..
 override def postRestart() { .. } ..
 def receive = {
  case DeleteFromES(..) =>
   throw new Exception("..") ..} }
```

fix

```
  f.onComplete {
   case Success (..) => ..
   case Failure(..) => .. }
```

Fig. 12. Actor bug with a *Patterns* root cause.

Our *Messaging Patterns* root cause overlaps with Hedden and Zhao's [2018] *Response* root cause. Their *Response* bugs occur due to "improper responses to different communication-based operations [messages]."

5.7 Root Cause 7: *Model Confusion*

Model Confusion is the seventh common root cause (7.6%) for our actor bugs. The computation model of Akka actor concurrency is significantly different from multithreaded concurrency, as discussed in Section 3. In addition, well-known and basic functionalities may have significantly different semantics in these models. For example, parental supervision and exception handling for Akka actors are significantly different from standard exception handling for Scala threads. Akka developers are responsible to clearly understand Akka and its underlying computational model. Otherwise, confusions can cause bugs.

Figure 13 shows a buggy actor program [Stack Overflow 2012a] with a *Model Confusion* root cause. Here, the developer intends to evaluate the performance of a LoadWorker router. The LoadGenerator actor creates the router actor loadRouter, of type LoadWorker, with a round robin routing strategy. Afterwards, it receives a "start" message and to process it sends an infinite number of random messages to loadRouter. A RoundRobinRouter forwards its messages to its routees one after another.

However, this program is buggy with an *Incorrect Termination* symptom and a *Model Confusion* root cause. The impact of this symptom is that the LoadGenerator actor cannot evaluate the performance of the LoadWorker router and instead hangs. The developer explains this symptom as, "actor app hangs under high volume." The root cause of this bug is that the developer is confused about the

basics of Akka actor model and its message processing semantics. To process "start," LoadGenertor sends messages to loadRouter in an infinite loop. Therefore, to process its first "start" and the messages it receives afterwards, LoadGenertor should be able to process its messages concurrently and more than one message at a time. However, in Akka, an actor processes its messages sequentially, one after another. Therefore, LoadGenerator falls into an infinite loop when processing its first "start," never finishes its processing, and never starts the processing of its next messages. The fix for this bug, as shown in Figure 13, suggests using self-messaging to send the "start" messages in batches with the size batchSize, instead of an infinite loop. In addition, the program in Figure 3 is another example of a bug with a *Model Confusion* root cause. Here, the developer is similarly confused about the sequential processing of messages in the FileWriter actor.

Sending a message from a non-actor entity to an actor, invoking a method of an actor instead of sending it a message, and sending a message to an actor Actor instead of its reference ActorRef, are more examples of our *Model Confusion* bugs. Akka separates an actor, of type Actor, and its reference, of type ActorRef, and makes the actor accessible only through its reference to make states of the actor accessible by its messages. Note that this separation still cannot guarantee that the state of the actor is not shared with other actors. There are actor models that are less susceptible to our *Model Confusion* bugs. For example, academic actor models that support Parallel Actor Monitors [Scholliers et al. 2014] and transactional message processing [Hayduk et al. 2015], allow for concurrent message processing and are less susceptible to bugs related to sequential message processing. However, industrial-strength actor frameworks and languages, such as Akka, Orleans [Bernstein et al. 2014], and Erlang [Armstrong 2007], processes their messages sequentially.

bug

```
class LoadGenerator(..) extends Actor { ..
 val loadRouter = context.actorOf(Props[LoadWorker]
  .withRouter(RoundRobinRouter(..)..))
 def receive = {
  case "start" => ..
   while(true)
    loadRouter ! r.getRandomCommand() } } ..}
class LoadWorker extends Actor { .. }
```

fix

```
  (1 to batchSize) foreach {
   loadRouter ! r.getRandomCommand() }
   self ! "start"
```

Fig. 13. Actor bug with a *Model* root cause.

Our *Model Confusion* root cause is new and cannot be found in Hedden and Zhao's [2018] or Torres Lopez et al.'s [2018] root causes for actor bugs.

5.8 Root Cause 8: *Misnaming*

Misnaming is the eighth most common symptom (5.4%) for our actor bugs. An actor name includes several properties of the actor, such as its proper name, enclosing actor system, user or system actor, supervision hierarchy, network protocol, and port number. Akka actor developers are responsible to manually provide and maintain correct actor names. Otherwise, misnamings can cause bugs.

Figure 14 shows a buggy actor program [Stack Overflow 2018a] with a *Misnaming* root cause. Here, the developer intends to configure the server application Server and its server actor db for the remote communication with the client application Client. Server configures and creates the system actor system with the name IMDB and the configuration config. IMDB and its actors are configured to run on a machine with the IP address 127.0.0.1 and at the port 2253. IMDB uses TCP protocol from Netty's client-server framework [Netty 2020] for remote messaging. Server creates the server actor db with the name ImdbActor to reside in the IMDB actor system. Client looks up dbs using the name AkkaIMDB@127.0.0.1:2552/user/ImdbActor and sends it a message. In this name, ImdbActor is the name of the server actor that resides in the AkkaIMDB actor system running on a machine with the IP address 127.0.0.1 and at the port 2552. AkkaIMDB is a user actor. In Akka, the actorSelection method looks up an actor with a specific name. In Scala, the parseString method parses a configuration string.

```
bug
class DB extends Actor { .. }
object Server extends App {
 val config = ConfigFactory.parseString("..remote { ..
   netty.tcp{ hostname="127.0.0.1" port=2253 }..}..")
 val data = ConfigFactory.load(config)
 val system = ActorSystem("IMDB", data)
 val db = system.actorOf(Props(new DB),"ImdbActor") }
object Client extends App {
  val dbs = system.actorSelection
   (s"IMDB@127.0.0.1:2552/user/ImdbActor")
 (dbs ? messages.Set(key , value)).map(..) .. }
fix
  val dbs = system.actorSelection
   (s"IMDB@127.0.0.1:2553/user/ImdbActor")
```

Fig. 14. Actor bug with *Misnaming* root cause.

However, this program is buggy with an *Incorrect Messaging* symptom and a *Misnaming* root cause. The impact of this symptom is that the `Client` application cannot communicate with the remote `Server` and its actor `db`. The developer explains this symptom as, "when I put the database actor [db] on a remote actor system . . . , I have the error *deadletters encountered*." The root cause of this bug is that the developer uses the incorrect name for `db` with the wrong port number `2252`, instead of `2253`. `db` is not running at the port `2253` and thus cannot be looked up and the messages sent to it are delivered to `/deadLetters`. The fix for this bug, as shown in Figure 14, suggests using the name `AkkaIMDB@127.0.0.1:2553/user/ImdbActor` for `dbs`, instead of `AkkaIMDB@127.0.0.1:2552/user/ImdbActor`. In addition, the program in Figure 2 is another example of a bug with a *Misnaming* root cause. Here, the developer attempts to create multiple `Worker` actors with the same non-unique name "`worker`," which is not allowed.

Actor names with incorrect parental hierarchies, incorrect letter cases for characters, and incorrect network protocols are more examples of our *Misnaming* root cause.

Our *Misnaming* root cause is new and cannot be found in Hedden and Zhao's [2018] or Torres Lopez et al.'s [2018] root causes for actor bugs.

5.9 Root Cause 9: *Misconfiguration*

Misconfiguration is the ninth most common root cause (3.8%) for our actor bugs. The default Akka configuration [Akka Actor Reference Config 2020] provides a large number of 255 parameters to configure actors and their dispatching, supervision, shutdown, routing, mailbox, clustering, remoting, logging, serialization, and deployment. About 42.4% of these parameters need values of complex data types, such as API class names, values of other configuration parameters, and arrays of these values. In addition to the default configuration, developers can declare and use their own custom configurations parameters. There are consistency constraints on the values of these parameters. For example, a `PinnedDispatcher` cannot be configured to use an executor that is not a `thread-pool-executor` [Akka Actor Reference Config 2020]. Akka developers are responsible to manually discover and satisfy these constraints. Otherwise, misconfigurations can cause bugs.

Figure 15 shows a buggy actor configuration [Stack Overflow 2014] with a *Misconfiguration* root cause. Here, the developer intends to configure the custom dispatcher `blocking-dispatcher` and create an actor that uses this dispatcher, in the `Main` application. In Akka, all actors share the same default dispatcher, however, a developer can configure and use their own dispatcher. The configuration file `application.conf` declares `blocking-dispatcher` to be a dispatcher of type `PinnedDispatcher` with an executor of type `thread-pool-executor`. The executor maintains a thread pool with a minimum of 2*2 threads (`core-pool-size-min * core-pool-size-factor`) and a maximum of 10*2 (`core-pool-size-max * core-pool-size-factor`).

However, this configuration is buggy with an *Error* symptom and a *Misconfiguration* root cause. The impact of this symptom is that an actor with the configuration `blocking-dispatcher` cannot be created. The developer explains this symptom as, "when I create [an] actor [with the configuration `blocking-dispatcher`] I'm getting the exception: Exception in thread 'main' `akka.ConfigurationException`: Dispatcher [`blocking-dispatcher`]." A `ConfigurationException` is

thrown when there is a problem with a configuration. The root cause of this bug is that the developer is not satisfying the consistency constraints for the configuration of `PinnedDispatcher`. `PinnedDispatcher` assigns a single thread to each actor and thus each actor has its own thread pool with *only one thread* in it. Therefore, `core-pool-size-min`, `core-pool-size-max`, and `core-pool-size-factor` should be 1, and not 2, 10, and 2.0. The fix for this bug, as shown in Figure 15, suggests setting these values to 1.

The configuration in Figure 15 is only a few lines long, however, it includes 2 consistency constraints that could be violated by the developer, doubling the chances of a *Misconfiguration* bug. The first is the constraint on the numbers of threads in `PinnedDispatcher`, which is already violated. The second is the constraint that a `PinnedDispatcher` must be used with a `thread-pool-executor`, which is not violated.

bug

```
/* part of application.conf */
blocking-dispatcher {
 type = PinnedDispatcher
 executor = "thread-pool-executor"
 thread-pool-executor {
  core-pool-size-min = 2
  core-pool-size-factor = 2.0
  core-pool-size-max = 10 } .. }
object Main extends App { ..
 ..actorOf(..withDispatcher("blocking-dispatcher")..)..}
```

fix

```
  core-pool-size-min = 1
  core-pool-size-factor = 1
  core-pool-size-max = 1
```

Fig. 15. Actor with a *Misconfig* root cause.

Incorrect configurations of actor systems, actor mailboxes, and remoting are a few more examples of our *Misconfiguration* root cause.

Our *Misconfiguration* root cause is new and cannot be found in Hedden and Zhao's [2018] or Torres Lopez et al.'s [2018] root causes for actor bugs.

5.10 Root Cause 10: *Untyped Communication*

Untyped Communication is the least common (1.1%) root cause for our actor bugs. Correct actor communication requires that a receiver actor knows about the messages that its senders can send. Classic Akka is untyped and its actors do not know about the type of messages that they may receive. There is nothing that prevents the sender from sending a message that its receiver cannot receiver and process [De Koster et al. 2016]. Akka developers are responsible to manually discover these message types and guarantee that the receiver can receive all the messages that its senders can send to it. Otherwise, miscommunications can cause bugs.

Figure 16 shows a buggy actor program [GitHub 2013b] with an *Untyped Communication* root cause. Here, the developer intends to receive and process all the messages that can be sent to the `HttpHostConnector` actor. `HttpHostConnector` receives messages of types `HttpRequest`, `Disconnected`, and `DemandIdleShutdown` to accept a connection from a client, disconnect the client, and shut down the service.

bug

```
class HttpHostConnector(..) extends Actor .. { ..
 def receive: Receive = {
  case request: HttpRequest => ..
  case Disconnected(..) => ..
  case DemandIdleShutdown => .. } }
```

fix

```
  case Terminated(..) => ..
```

Fig. 16. Actor bug with an *Untyped* root cause.

However, this program is buggy with an *Error* symptom and an *Untyped Communication* root cause. The impact of this symptom is that `HttpHostConnector` does not receive and process all the messages that can be sent to it. The developer explains this symptom as, "`DeathPactException` in `HttpHostConnector`." An actor throws a `DeathPactException` when it receives a `Terminated` message but cannot process it. A message `Terminated` is sent to a watcher actor by its watched actor when the watched actor terminates. `HttpHostConnector` does not receive and process messages of type

`Terminated` that can be sent to it. The fix for this bug, as shown in Figure 16, suggests receiving and processing `Terminated`.

There are actor languages and frameworks that are less susceptible to our *Untyped Communication* bugs. For example, Akka Typed [2020] is a recent variation of Classic Akka that requires typed communications and can statically guarantee the absence of our *Untyped Communication* bugs. In Classic Akka, the bug in Figure 16 is detected at run time, however, in Akka Typed, this bug can be detected at compile time.

Our *Untyped Communication* root cause is new and cannot be found in Hedden and Zhao's [2018] or Torres Lopez et al.'s [2018] root causes for actor bugs.

5.11 Implications

Using our new root causes to extend the set of bugs to test for and find Tasharofi et al. [2013] identify the bug pattern *Flexible Interfaces* [Torres Lopez et al. 2018] and use it to generate test schedules for actor software. In this pattern, an actor can dynamically change the set of messages it receives and processes. The root cause of a bug with this pattern is that a new behavior cannot processes a message that the previous behaviors of the actor could process. Some of our root causes overlap with Hedden and Zhao's [2018] and Torres Lopez et al.'s [2018] root causes for actor bugs. However, our *API Confusion*, *Model Confusion*, *Misnaming*, *Misconfiguration*, and *Untyped Communication*, that are the root causes for 31.9% of our actor bugs, are new. Both previous and future actor testing and bug finding tools and techniques can discover the patterns of our new root causes and use these patterns to extend the set of bugs that they test for to increase the number of bugs they find. These patterns could include both syntactic and semantic patterns. For example, 29.6% of our *Race* bugs follow a syntactic pattern in which an entity, such as a future or a scheduled task, that can run outside and concurrent to an actor, closes over the mutable value of the `sender()` of the actor. *Flexible Interfaces* pattern is an example of a semantic pattern. Previous work, such as FindBugs [Hovemeyer and Pugh 2004] for non-actor Java bugs, shows that finding bugs does not "require sophisticated or extensive forms of analysis" and "many errors can be found with trivial static examination [of the code]."

Supporting implicit life cycle management and typed communication to prevent bugs Orleans actor framework [Bernstein et al. 2014] supports more implicit and automatic life cycle management and is less susceptible to our *Explicit Life Cycle* bugs. Similarly, Akka Typed [2020] supports typed communications and is less susceptible to our *Untyped Communication* bugs. New variations of Akka can prevent our *Explicit Life Cycle* and *Untyped Communication* bugs, that are 13.5% of our actor, by supporting more implicit life cycle and typed communications.

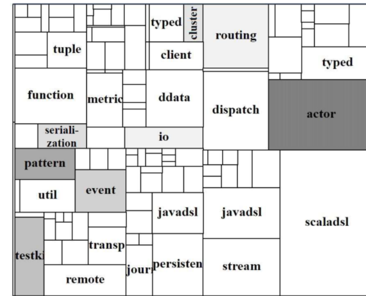
Using the commonality of our bugs to make tradeoffs between different features Designers of new variations of Akka can use the commonality of our actor bugs, as one of many factors that they may consider, to make tradeoffs between features that these new variations may support. For example, the designer can make a tradeoff and choose to support the implicit life cycle management, that can prevent as much as 12.4% of our actor bugs, over typed communications, that can prevent only as much as 1.1%.

6 API USAGES

In this section, we answer ***RQ3*** and discuss the usages of Akka API by our actor bugs.

Figure 17 shows the usages of Akka API packages, in a heatmap, where the darker the gray is, the more the usage is. At the bottom, the figure lists the usage numbers and percentages for used packages and their classes. The usage for packages and classes that are not listed is zero. For space reasons, in the heatmap, we shorten the name of a package to its second part. For example, *actor* is short for *akka.actor*.

According to Figure 17, actor bugs use the following 8 Akka API packages, with decreasing commonalities: `akka.actor`, `akka.pattern`, `akka.testkit`, `akka.event`, `akka.serialization`, `akka.cluster`, `akka.io`, and `akka.routing`. Among these packages, `akka.actor` is the most common package (81.2%) and `akka.routing` is the least common (0.2%). The usages of packages by actor bugs are different and this difference can be significant for some API packages. For example, `akka.actor` is used 406 times more than `akka.routing`. In addition, 3 most common packages `akka.actor`, `akka.pattern`, and `akka.testkit`, that are only 2.7% of Akka API packages, are responsible for 97.7% of usages. These packages provide the basics to configure, create, lookup, reference, schedule, send and receive messages, supervise, shutdown, and test actors. Packages for untyped actors, such as `akka.actor` (81.2%) and `akka.cluster` (0.4%), are used more than packages for typed actors, such as `akka.actor.typed` (0.0%) and `akka.cluster.typed` (0.0%). Similarly, packages for local actors, such as `akka.actor` (81.2%) and `akka.routing` (0.2%), are used more than packages for remote actors, such as `akka.serialization` (0.5%) and `akka.remote.routing` (0.0%). The testing package `akka.testkit` is the third most common package (2.5%).



akka.actor(492)[81.2%]: ActorRef(188) ActorContext(119) ActorSystem(115) Props(19) Actor(14) AbstractActor(13) Stash(8) Scheduler(6) AbstractFSM(4) ActorRefFactory(2) TypedActor(2) ChildRestartStats(1) TypedActorFactory(1) ***akka.pattern(52)[9.1%]:*** AskableActorRef(40) BackoffSupervisor(6) Backoff(3) PipeableFuture(3) ***akka.testkit(14)[2.5%]:*** TestActorRef(4) TestKit(4) TestProbe(4) TestFSMRef(2) ***akka.event(6)[1.1%]:*** EventStream(6) ***akka.serialization(3)[0.5%]:*** Serialization(3) ***akka.cluster(2)[0.4%]:*** Cluster(2) ***akka.io(1)[0.2%]:*** Tcp(1) ***akka.routing(1)[0.2%]:*** Routing(1)

Fig. 17. Akka API usages of actor bugs

Our observation about the uncommonality of Akka remote API agrees with Tasharofi et al.'s [2013] observation that, "most developers use actors to address the local scalability problem, that is, they use actors as a solution for local concurrent programming [instead of remote programming]." Tasharofi et al. study fifteen large and mature Scala Akka actor software.

6.1 Implications

Targeting less, untyped, and local Akka API packages to scale bug pattern mining to large APIs in large-scale code bases Previous work [Li and Zhou 2005; Liang et al. 2016] mines non-actor software code bases to discover their API usage patterns and use these patterns to find bugs that violate them. For example, for the class `ReentrantLock` in Java, the invocation of method `unlock()` after its `lock()` is a pattern that if violated can cause multithreaded bugs. One challenge in these works is to scale the pattern mining to large APIs in large-scale code bases. Similar tools and techniques can be developed for Akka software, however, they should address a similar challenge. Designers of these tools and techniques can use our observations about usages of Akka API to address this challenge. To scale to the large Akka API in large-scale code bases of Akka software, the designer can focus on only 2.7% of Akka packages, instead of all, that are responsible for 97.7% of Akka API usages by our actor bugs. Similarly, they can focus on API for untyped actors, instead of typed actors, and local actors, instead of remote actors.

7 DIFFERENCES

In this section, we answer ***RQ4*** and discuss the differences of our Stack Overflow and GitHub actor bugs for commonalities and distributions of their characteristics.

7.1 Symptoms

Figure 18 shows the symptoms of our actor bugs in Stack Overflow and GitHub. According to this figure, the common symptoms of actor bugs in Stack Overflow and GitHub are different. For our Stack Overflow actor bugs, *Error* is the most common symptom (40.0%) and *Incorrect Exceptions* is the least common (0.0%). In contrast, for our GitHub actor bugs, *Unexpected Behavior* is the most common symptom (53.6%) and *Incorrect Exceptions* is the least common (3.1%). In addition, the commonalities of some symptoms are significantly different among our Stack Overflow and GitHub actor bugs. *Incorrect Messaging* and *Error* are 3.6 and 1.5 times more common in Stack Overflow and there is no *Incorrect Exceptions* in GitHub. In contrast, *Unexpected Behavior* and *Incorrect Termination* are 3.1 and 1.6 times more common in GitHub. Finally, although there is a significant difference between commonalities of individual symptoms between our Stack Overflow and GitHub actor bugs, there is no statistically significant difference among their distributions.

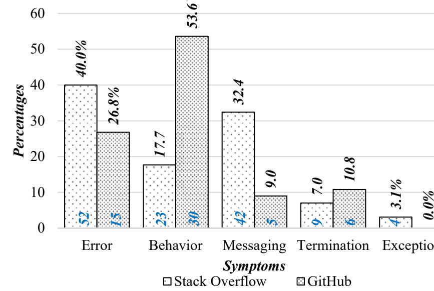


Fig. 18. Symptoms in Stack Overflow and GitHub.

7.2 Root Causes

Figure 19 shows the root causes of actor bugs in Stack Overflow and GitHub. According to this figure, the common root causes of actor bugs in Stack Overflow and GitHub are different. For our Stack Overflow actor bugs, *API Confusion* is the most common root cause (18.5%) and *Untyped Communication* is the least common (0.0%). In contrast, for our GitHub actor bugs, *Logic* is the most common root cause (57.2%) and *Programming* and *Model Confusion* are the least common (0.0%). In addition, the commonalities of some root causes are significantly different among Stack Overflow and GitHub actor bugs. *Logic* and *Explicit Life Cycle* are 18.5 and 1.3 times more common in GitHub. In contrast, *API Confusion*, *Misnaming*, *Misconfiguration Messaging Patterns*, and *Race* are 5.2, 3.9, 2.7, 2.2, and 1.3 times more common in Stack Overflow and there is no *Programming* and *Model Confusion* in GitHub. Finally, in addition to the significant differences between commonalities of individual root causes between Stack Overflow and GitHub actor bugs, there is a statically significant difference between their distributions.

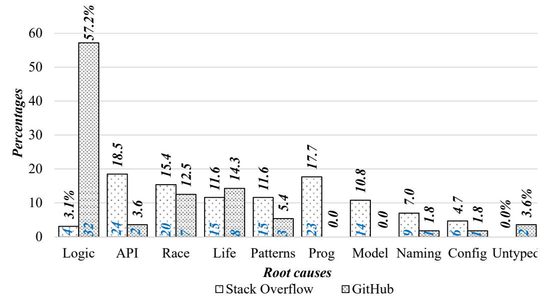


Fig. 19. Root causes in Stack Overflow and GitHub.

7.3 API Usages

Figure 20 shows the usages of Akka API by our actor bugs in Stack Overflow and GitHub. According to this figure, 3 packages `akka.actor`, `akka.pattern`, and `akka.testkit` are responsible for most usages in both Stack Overflow (98.6%) and GitHub (94.0%). However, the usages of individual API packages by Stack Overflow and GitHub actor bugs are different and this difference can be significant for some packages. Packages `akka.pattern` and `akka.actor` are used 2 and 1.1 times more in Stack Overflow and there is no usage of `akka.routing` in GitHub. In contrast, `akka.serialization`, `akka.event`, and `akka.testkit`, are used 9, 4, and 2 times more in GitHub and there is no usage of

akka.actor(396)[87.0%]: ActorRef(139) ActorSystem(108) ActorContext(89) Props(19) AbstractActor(12) Actor(11) Stash(6) AbstractFSM(4) Scheduler(3) ActorRefFactory(2) TypedActor(2) TypedActorFactory(1) ***akka.pattern(46)[10.1%]:*** AskableActorRef(34) BackoffSupervisor(6) Backoff(3) PipeableFuture(3) ***akka.testkit(7)[1.5%]***: TestActorRef(3) TestKit(2) TestProbe(2) ***akka.event(4)[0.9%]:*** EventStream(4) ***akka.serialization(1)[0.2%]:*** Serialization(1) ***akka.routing(1)[0.2%]:*** Router(1)

akka.actor(96)[82.8%]: ActorRef(49) ActorContext(30) ActorSystem(7) Actor(3) Scheduler(3) Stash(2) AbstractActor(1) ChildRestartStats(1) ***akka.testkit(7)[6.0%]:*** TestFSMRef(2) TestKit(2) TestProbe(2) TestActorRef(1) ***akka.pattern(6)[5.2%]:*** AskableActorRef(6) ***akka.serialization(2)[1.8%]:*** Serialization(2) ***akka.cluster(2)[1.8%]:*** Cluster(2) ***akka.event(2)[1.8%]:*** EventStream(2) ***akka.io(1)[0.9%]:*** Tcp(1)

Fig. 20. API usages in Stack Overflow (left) and GitHub (right).

`akka.cluster` and `akka.io` in Stack Overflow. In addition, although there is a significant difference between usages of individual packages between Stack Overflow and GitHub actor bugs, there is no statistically significant difference between their distributions.

7.4 Implications

Using characteristics of our Stack Overflow and GitHub actor bugs for predictive bug finding Previous work uses the characteristics of bugs as features to build and train models that can predict bugs. For example, Zhou et al. [2015] use characteristics, such as root causes and files involved in the bug, for multithreaded bugs in Mozilla, KDE, and Apache to train a model that can predict type, locations and quantities of similar bugs in multithreaded software. New bug prediction tools and techniques can use actor bugs and their characteristics in terms of their root causes, symptoms, API usages, and their differences as features to build and train models that can predict similar Akka actor bugs. In addition, our observations about the differences between the commonalities and distributions of these characteristics can be used to decide which characteristics should be used as features for a specific dataset. For example, root causes of our Stack Overflow actor bugs may not be a good feature to train a model that predicts GitHub actor bugs. This is because of the significant differences between both commonalities and distributions of our Stack Overflow and GitHub root causes.

8 THREATS TO VALIDITY

Using tags, keywords, and maturity criteria to identify Akka questions and answers in Stack Overflow and mature projects in GitHub could be a threat. This is because these may not identify a complete set of Akka questions and answers and commits or identify irrelevant ones. To minimize this threat, we closely follow the best practices of previous work in using tags [Ahmed and Bagherzadeh 2018; Bagherzadeh and Khatchadourian 2019; Islam et al. 2019, 2020; Meng et al. 2018; Yang et al. 2016], keywords [Casalnuovo et al. 2015; Islam et al. 2019, 2020; Kim and Whitehead 2006; Ray et al. 2016], and maturity criteria [Biswas et al. 2019; Gu et al. 2016; Nadi et al. 2016]. Additionally, we manually analyze each candidate question and answer and commit and discard irrelevant ones.

Using Stack Overflow and GitHub as the sources for actor bugs is another threat. This is because they may not include a representative set of actor bugs. However, the large numbers of questions and answers and developers in Stack Overflow and projects, commits, and developers in GitHub

helps to mitigate this threat. Furthermore, unlike some previous studies that use only Stack Overflow [Ahmed and Bagherzadeh 2018; Bagherzadeh and Khatchadourian 2019; Barua et al. 2014; Rosen and Shihab 2016; Yang et al. 2016] or GitHub [Hedden and Zhao 2018], we use both. The manual identification of actor bugs and fixes, their symptoms, root causes, API usages, and classifications could be another threat. To minimize this threat, we closely follow the best practices of previous work in using open card sort [Ahmed and Bagherzadeh 2018; Bagherzadeh and Khatchadourian 2019; Islam et al. 2019; Nadi et al. 2016]. In addition, our manual analyses use all the available information for an Stack Overflow candidate question bug, including its question, all its answers and comments, and for a GitHub candidate commit bug, including its message, original code, modified code, issue reports, and pull requests. Multiple authors with extensive expertise in actor and multithreaded concurrency performed the manual analysis, where they iterated and refined their results until in agreement.

The complexity of concurrency bugs—and its impact on their understanding, reporting, finding, and fixing—may be considered a threat. Simpler bugs are more likely to be understood and reported and easier to find and fix [Bron et al. 2005; Yu 2013]. Moreover, there could be bugs that are rarely or never reported, found, and fixed [Zhou et al. 2015]. To mitigate this threat, we closely follow the best practices of previous work [Bhattacharya et al. 2012; Hedden and Zhao 2018; Lu et al. 2008; Tu et al. 2019] in studying only the bugs that developers have found and fixed. Like other empirical studies, our findings should be interpreted with our methodology in mind and understood to hold only for Akka actor bugs written in Scala or Java from Stack Overflow and GitHub.

9 RELATED WORK

Actor concurrency bugs The most related to our work are the works by Hedden and Zhao [2018] and Torres Lopez et al. [2018]. Section 5 discusses these works and their overlaps with our work in detail. To summarize, Hedden and Zhao study 126 Akka actor bugs in twelve projects from ScalaIndex, classify their root causes into three categories and ten subcategories, and compare their actor bugs with cloud bugs. Our following root causes overlap with several of their root causes, with their root causes in parentheses: *Logic* (*Logic*), *Race* (*Message Order*, *Operation Order*, *Cooperation*), *Explicit Life Cycle* (*Creation*, *Shutdown*, *Recovery*), *Programming* (*Logic*), and *Messaging Patterns* (*Response*). However, our four root causes *API Confusion*, *Model Confusion*, *Misnaming*, *Misconfiguration*, and *Untyped Communication*, that are root causes for 31.9% of our actor bugs, are new. Similarly, Torres Lopez et al. study twenty-three actor bugs in various actor models from eleven previous works and classify their root causes into two categories and six subcategories, and provide a list of static analyses, testing, debugging, and visualization tools that address these bugs. Our following root causes overlap with several of their root causes, with their root causes in parentheses: *Race* (*Bad Message Interleaving*, *Memory Inconsistency*). However, our nine root causes, i.e., *Logic*, *API Confusion*, *Explicit Life Cycle*, *Programming*, *Model Confusion*, *Misnaming*, *Untyped Communication*, accounting for 85.4% of our actor bugs, are new.

Verification of actor concurrency For verification, Gordon [2019] proposes a program logic with modal assertions for deductive reasoning and verification of safety properties of actor programs in a core actor calculus. Charalambides et al. [2019] propose a typestate system for static reasoning of liveness properties in a restricted class of actor programs written in a simple actor language. Khatchadourian et al. [2019] use typestate to efficiently and safely parallelize streams, another form of reactive programming. Bagherzadeh and Rajan [2017] propose order types for static reasoning and detection of message race bugs. Haller and Loiko [2016] propose a type system to control aliasing in Scala actor software. Bagherzadeh and Rajan [2015] propose an interference-aware programming model to allow for modular reasoning and guarantee the absence of data races in Panini asynchronous message passing concurrency [Rajan 2015]. Khatchadourian et al. [2008]

also allow modular reasoning but for Aspect-Oriented Programming (AOP) using rely-guarantee clauses similar to those used for concurrent systems. Negara et al. [2011] proposes a static analysis to infer and guarantee the single ownership property for actors. Clebsch et al. [2015] use reference capabilities to guarantee the absence of data races in the combination of actor and multithreaded concurrency. Colaço et al. [1997] proposes a type inference system to prevent orphan messages in a primitive actor calculus. D'Osualdo et al. [2013] propose an infinite-state model checker for a core fragment of Erlang actor concurrency. Stiévenart et al. [2017] propose a mailbox abstraction to statically reason about the bounds on the sizes for mailboxes.

Testing and debugging of actor concurrency Lauterburg et al. [2009] propose Basset for systematic exploration of message schedules for given inputs. Tasharofi et al. [2013] propose Bita to explore message schedules for given inputs using different message coverage criteria. Tasharofi et al. [2012] also propose TransDPOR, a dynamic partial order reduction technique to reduce the state-space of message schedules. Li et al. [2018] propose Tap, a technique to generate system-level test cases for a given code location in Akka software. Sen and Agha [2006] propose dCute, a full coverage testing system to find deadlocks. For debugging, Lopez et al. [2019] propose Multiverse Debugging to allow for observation and debugging of all concurrent execution paths in AmbientTalk, an actor language for mobile adhoc networks.

Multithreaded concurrency bugs Lu et al. [2008] study 105 local concurrency bugs in 4 server and client applications and classify their patterns, manifestation, fixing, and avoidance strategies. Leesatapornwongsa et al. [2016] study 104 distributed concurrency bugs in 4 popular data center systems and classify their triggers, behaviors, and fixes. Gunawi et al. [2014] study 3,655 distributed concurrency issues in 6 popular cloud systems and classify their vitality, aspects, hardware, hardware failure mode, software, implications, and scope. Wang et al. [2017] study 57 concurrency bugs in 53 open-source Node.js software and classify their root causes, patterns, impacts, manifestation, and fix strategies. Khatchadourian et al. [2020] also report on API misuse for Java streams.

Unlike works that focus on the classification of root causes of actor bugs alone, verification, testing, and debugging of actor software, or analysis and classification of multithreaded concurrency bugs, our work focuses on classifying symptoms, root causes, API usages, and differences for 186 real-world Akka actor bugs from Stack Overflow and GitHub and discussing real-world examples of bugs with these root causes and symptoms and how developers discuss them.

10 CONCLUSIONS AND FUTURE WORK

In this work, we construct and study a set of 186 real-world Akka actor bugs and their fixes from Stack Overflow and GitHub, understand and classify their symptoms, root causes, API usages, and differences, discuss real-world examples of actor bugs with these symptoms and root causes, investigate the relation of our findings with the findings of previous work, and discuss the implications of our findings. One avenue of the future work is to analyze and classify fixes of our actor bugs. Another avenue is to analyze our actor bugs to identify the challenges in their detection and fixing.

REFERENCES

Stack Overflow. 2012a. Akka Actors app hangs under high volume. https://stackoverflow.com/questions/11141311.

Stack Overflow. 2012b. correctly terminate akka actors in scala. https://stackoverflow.com/questions/12324055/.

GitHub. 2013a. fix garbling of big Tcp.Write. https://github.com/spray/spray/commit/332ba626b193794fd4d753839e664aacd4d302a5.

Stack Overflow. 2013. How should an akka actor be created that might throw an exception? https://stackoverflow.com/questions/18648390.

GitHub. 2013b. Spray Project. https://github.com/spray/spray/commit/e34da115fa43d4d46db0e7ae06eea7fbcbc4fdfd.

Stack Overflow. 2014. akka.io dispatcher configuration Exception. https://stackoverflow.com/questions/21686327.

Stack Overflow. 2016. postRestart and preRestart methods are not getting invoked in akka actors. https://stackoverflow.com/questions/37608915.

Stack Overflow. 2017a. Actor Name is not Unique InvalidActorNameException. https://stackoverflow.com/questions/11693562/.

Stack Overflow. 2017b. Doesn't immediately stop child Akka actors when parent actor stopped. https://stackoverflow.com/questions/45574085.

Stack Overflow. 2017c. How to use futures with Akka for asynchronous results. https://stackoverflow.com/questions/11693562/.

Stack Overflow. 2018a. Cannot establish remote communication with Akka. https://stackoverflow.com/questions/53188945.

Stack Overflow. 2018b. Scala Akka Actor - Dead Letters encountered. https://stackoverflow.com/questions/53200356.

Stack Overflow. 2018c. Send message to actor after restart from Supervisor. https://stackoverflow.com/questions/48446194.

Stack Overflow. 2019. Akka Routing: Reply's send to router ends up as dead letters. https://stackoverflow.com/questions/20564381/.

Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA.

Gul Agha and Carl Hewitt. 1985. Concurrent Programming Using Actors: Exploiting Large-Scale Parallelism. In *Proceedings of the Fifth Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer-Verlag, Berlin, Heidelberg, 19–41.

Syed Ahmed and Mehdi Bagherzadeh. 2018. What Do Concurrency Developers Ask about? A Large-Scale Study Using Stack Overflow. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (Oulu, Finland) *(ESEM '18)*. Association for Computing Machinery, New York, NY, USA, Article 30, 10 pages. https://doi.org/10.1145/3239235.3239524

Akka 2.6.5 API. May 2020. https://doc.akka.io/api/akka/current/akka/index.html.

Akka Actor Reference Config. April 2020. https://github.com/akka/akka/blob/master/akka-actor/src/main/resources/reference.conf.

Akka Typed. May 2020. https://doc.akka.io/docs/akka/current/typed/index.html.

Apache. 2015. Akka Actors in Spark. https://issues.apache.org/jira/browse/SPARK-5293.

Joe Armstrong. 2007. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf.

Mehdi Bagherzadeh and Raffi Khatchadourian. 2019. Going Big: A Large-Scale Study on What Big Data Developers Ask. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 432–442. https://doi.org/10.1145/3338906.3338939

Mehdi Bagherzadeh and Hridesh Rajan. 2015. Panini: A Concurrent Programming Model for Solving Pervasive and Oblivious Interference. In *Proceedings of the 14th International Conference on Modularity* (Fort Collins, CO, USA) *(MODULARITY 2015)*. ACM, New York, NY, USA, 93–108. https://doi.org/10.1145/2724525.2724568

Mehdi Bagherzadeh and Hridesh Rajan. 2017. Order Types: Static Reasoning About Message Races in Asynchronous Message Passing Concurrency. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control* (Vancouver, BC, Canada) *(AGERE 2017)*. ACM, New York, NY, USA, 21–30. https://doi.org/10.1145/3141834.3141837

Thomas Ball, Mayur Naik, and Sriram K. Rajamani. 2003. From Symptom to Cause: Localizing Errors in Counterexample Traces. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New Orleans, Louisiana, USA) *(POPL '03)*. Association for Computing Machinery, New York, NY, USA, 97–105. https://doi.org/10.1145/604131.604140

Anton Barua, Stephen W. Thomas, and Ahmed E. Hassan. 2014. What Are Developers Talking About? An Analysis of Topics and Trends in Stack Overflow. *Empirical Softw. Engg.* 19, 3 (June 2014), 619–654. https://doi.org/10.1007/s10664-012-9231-y

Phil Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. 2014. *Orleans: Distributed Virtual Actors for Programmability and Scalability.* Technical Report MSR-TR-2014-41. https://www.microsoft.com/en-us/research/publication/orleans-distributed-virtual-actors-for-programmability-and-scalability/

Pamela Bhattacharya, Iulian Neamtiu, and Christian R. Shelton. 2012. Automated, Highly-Accurate, Bug Assignment Using Machine Learning and Tossing Graphs. *J. Syst. Softw.* 85, 10 (Oct. 2012), 2275–2292. https://doi.org/10.1016/j.jss.2012.04.053

Francesco Adalberto Bianchi, Alessandro Margara, and Mauro Pezze. 2018. A Survey of Recent Trends in Testing Concurrent Software Systems. *IEEE Trans. Softw. Eng.* 44, 8 (Aug. 2018), 747–783. https://doi.org/10.1109/TSE.2017.2707089

Sumon Biswas, Md Johirul Islam, Yijia Huang, and Hridesh Rajan. 2019. Boa Meets Python: A Boa Dataset of Data Science Software in Python Language. In *Proceedings of the 16th International Conference on Mining Software Repositories* (Montreal, Quebec, Canada) *(MSR '19)*. IEEE Press, 577–581. https://doi.org/10.1109/MSR.2019.00086

Arkady Bron, Eitan Farchi, Yonit Magid, Yarden Nir, and Shmuel Ur. 2005. Applications of Synchronization Coverage. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Chicago, IL, USA) *(PPoPP '05)*. Association for Computing Machinery, New York, NY, USA, 206–212. https://doi.org/10.1145/1065944.1065972

Rafael Caballero, Enrique Martin-Martin, Adrián Riesco, and Salvador Tamarit. 2019. A core Erlang semantics for declarative debugging. *J. Log. Algebraic Methods Program.* 107 (2019), 1–37. https://doi.org/10.1016/j.jlamp.2019.05.002

Casey Casalnuovo, Prem Devanbu, Abilio Oliveira, Vladimir Filkov, and Baishakhi Ray. 2015. Assert Use in GitHub Projects. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. IEEE Press, 755–766.

Minas Charalambides, Karl Palmskog, and Gul Agha. 2019. *Types for Progress in Actor Programs.* Springer International Publishing, Cham, 315–339. https://doi.org/10.1007/978-3-030-21485-2_18

Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny Capabilities for Safe, Fast Actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control* (Pittsburgh, PA, USA) *(AGERE! 2015)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/2824815.2824816

J-L. Colaço, M. Pantel, and P. Sallé. 1997. *A Set-Constraint-based analysis of Actors.* Springer US, Boston, MA, 107–122. https://doi.org/10.1007/978-0-387-35261-9_8

Joeri De Koster, Tom Van Cutsem, and Wolfgang De Meuter. 2016. 43 Years of Actors: A Taxonomy of Actor Models and Their Key Properties. In *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control* (Amsterdam, Netherlands) *(AGERE 2016)*. Association for Computing Machinery, New York, NY, USA, 31–40. https://doi.org/10.1145/3001886.3001890

Emanuele D'Osualdo, Jonathan Kochems, and C. H. Luke Ong. 2013. Automatic Verification of Erlang-Style Concurrency. In *Static Analysis*, Francesco Logozzo and Manuel Fähndrich (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 454–476.

Sally Fincher and Josh Tenenberg. 2005. Making sense of card sorting data. *Expert Systems* 22, 3 (2005), 89–93. https://doi.org/10.1111/j.1468-0394.2005.00299.x arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1468-0394.2005.00299.x

GHTorrent. April 2020. https://ghtorrent.org/.

GitHub. 2019. The State of Octoverse. https://octoverse.github.com/.

Colin S. Gordon. 2019. Modal Assertions for Actor Correctness. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control* (Athens, Greece) *(AGERE 2019)*. ACM, New York, NY, USA, 11–20. https://doi.org/10.1145/3358499.3361221

Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API Learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 631–642. https://doi.org/10.1145/2950290.2950334

Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. 2014. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SOCC '14)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/2670979.2670986

Philipp Haller and Alex Loiko. 2016. LaCasa: Lightweight Affinity and Object Capabilities in Scala. *SIGPLAN Not.* 51, 10 (Oct. 2016), 272–291. https://doi.org/10.1145/3022671.2984042

Robert H. Halstead. 1985. MULTILISP: A Language for Concurrent Symbolic Computation. *ACM Trans. Program. Lang. Syst.* 7, 4 (Oct. 1985), 501–538. https://doi.org/10.1145/4472.4478

Quinn Hanam, Fernando S. de M. Brito, and Ali Mesbah. 2016. Discovering Bug Patterns in JavaScript. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 144–156. https://doi.org/10.1145/2950290.2950308

Yaroslav Hayduk, Anita Sobe, and Pascal Felber. 2015. Dynamic Message Processing and Transactional Memory in the Actor Model. In *Distributed Applications and Interoperable Systems*, Alysson Bessani and Sara Bouchenak (Eds.). Springer International Publishing, Cham, 94–107.

Brandon Hedden and Xinghui Zhao. 2018. A Comprehensive Study on Bugs in Actor Systems. In *Proceedings of the 47th International Conference on Parallel Processing* (Eugene, OR, USA) *(ICPP 2018)*. ACM, New York, NY, USA, Article 56, 9 pages. https://doi.org/10.1145/3225058.3225139

David Hovemeyer and William Pugh. 2004. Finding Bugs is Easy. *SIGPLAN Not.* 39, 12 (Dec. 2004), 92–106. https://doi.org/10.1145/1052883.1052895

Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A Comprehensive Study on Deep Learning Bug Characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. ACM, New York, NY, USA, 510–520. https://doi.org/10.1145/3338906.3338955

Md Johirul Islam, Rangeet Pan, Giang Nguyen, and Hridesh Rajan. 2020. Repairing Deep Neural Networks Fix Patterns and Challenges. In *Proceedings of the 42nd International Conference on Software Engineering - Volume 2* (Seoul, South Korea)

(ICSE '20). IEEE Press.

Khari Johnson. 2018. GitHub passes 100 million repositories. https://venturebeat.com/2018/11/08/github-passes-100-million-repositories/.

Raffi Khatchadourian, Johan Dovland, and Neelam Soundarajan. 2008. Enforcing Behavioral Constraints in Evolving Aspect-Oriented Programs. In *Proceedings of the 7th Workshop on Foundations of Aspect-Oriented Languages* (Brussels, Belgium) *(FOAL '08)*. Association for Computing Machinery, New York, NY, USA, 19–28. https://doi.org/10.1145/1394496.1394499

Raffi Khatchadourian, Yiming Tang, Mehdi Bagherzadeh, and Syed Ahmed. 2018. A Tool for Optimizing Java 8 Stream Software via Automated Refactoring. In *International Working Conference on Source Code Analysis and Manipulation* (Madrid, Spain) *(SCAM '18)*. IEEE, IEEE Press, 34–39. https://doi.org/10.1109/SCAM.2018.00011

Raffi Khatchadourian, Yiming Tang, Mehdi Bagherzadeh, and Syed Ahmed. 2019. Safe Automated Refactoring for Intelligent Parallelization of Java 8 Streams. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 619–630. https://doi.org/10.1109/ICSE.2019.00072

Raffi Khatchadourian, Yiming Tang, Mehdi Bagherzadeh, and Baishakhi Ray. 2020. An Empirical Study on the Use and Misuse of Java 8 Streams. In *International Conference on Fundamental Approaches to Software Engineering (FASE 2020)*. ETAPS, Springer, 97–118. https://doi.org/10.1007/978-3-030-45234-6_5

Atul S. Khot. 2018. *Concurrent Patterns and Best Practices: Build Scalable Apps with Patterns in Multithreading, Synchronization, and Functional Programming*. Packt Publishing.

Sunghun Kim and E. James Whitehead. 2006. How Long Did It Take to Fix Bugs?. In *Proceedings of the 2006 International Workshop on Mining Software Repositories* (Shanghai, China) *(MSR '06)*. Association for Computing Machinery, New York, NY, USA, 173–174. https://doi.org/10.1145/1137983.1138027

Steven Lauterburg, Mirco Dotta, Darko Marinov, and Gul Agha. 2009. A Framework for State-Space Exploration of Java-Based Actor Programs. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*. IEEE Computer Society, Washington, DC, USA, 468–479. https://doi.org/10.1109/ASE.2009.88

Steven Lauterburg, Rajesh K. Karmani, Darko Marinov, and Gul Agha. 2010. Evaluating Ordering Heuristics for Dynamic Partial-order Reduction Techniques. In *Proceedings of the 13th International Conference on Fundamental Approaches to Software Engineering* (Paphos, Cyprus) *(FASE'10)*. Springer-Verlag, Berlin, Heidelberg, 308–322. https://doi.org/10.1007/978-3-642-12029-9_22

Edward A. Lee. 2006. The Problem with Threads. *Computer* 39, 5 (May 2006), 33–42. https://doi.org/10.1109/MC.2006.180

Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. 2016. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. ACM, New York, NY, USA, 517–530. https://doi.org/10.1145/2872362.2872374

Mark C. Lewis and Lisa L. Lacher. 2016. *Object-Orientation, Abstraction, and Data Structures Using Scala, Second Edition* (2nd ed.). Chapman & Hall/CRC.

He Li, Jie Luo, and Wei Li. 2014. A formal semantics for debugging synchronous message passing-based concurrent programs. *Science China Information Sciences* 57, 12 (2014), 1–18.

Sihan Li, Farah Hariri, and Gul Agha. 2018. Targeted Test Generation for Actor Systems. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018) (Leibniz International Proceedings in Informatics (LIPIcs))*, Todd Millstein (Ed.), Vol. 109. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 8:1–8:31. https://doi.org/10.4230/LIPIcs.ECOOP.2018.8

Zhenmin Li and Yuanyuan Zhou. 2005. PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Lisbon, Portugal) *(ESEC/FSE-13)*. ACM, New York, NY, USA, 306–315. https://doi.org/10.1145/1081706.1081755

B. Liang, P. Bian, Y. Zhang, W. Shi, W. You, and Y. Cai. 2016. AntMiner: Mining More Bugs by Reducing Noise Interference. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 333–344. https://doi.org/10.1145/2884781.2884870

Lightbend. 2019a. Akka. https://www.lightbend.com/akka-part-of-lightbend-platform.

Lightbend. 2019b. Customer Case Studies. https://www.lightbend.com/case-studies#filter:akka.

Lightbend. 2020a. Akka Documentation: Supervision and Monitoring. https://doc.akka.io/docs/akka/2.5/general/supervision.html.

Lightbend. 2020b. Akka.actor Documentation. https://doc.akka.io/api/akka/current/akka/actor/Actor.html.

Lightbend. 2020c. How Groupon Scales Personalized Offers To 48 Million Customers On Time. https://www.lightbend.com/case-studies/groupon-scalability-personalized-offers-to-48-million-customers.

Lightbend. 2020d. PayPal Blows Past 1 Billion Transactions Per Day Using Just 8 VMs With Akka, Scala, Kafka and Akka Streams. https://www.lightbend.com/case-studies/paypal-blows-past-1-billion-transactions-per-day-using-just-8-vms-and-akka-scala-kafka-and-akka-streams.

Yuheng Long, Mehdi Bagherzadeh, Eric Lin, Ganesha Upadhyaya, and Hridesh Rajan. 2016. On Ordering Problems in Message Passing Software. In *Proceedings of the 15th International Conference on Modularity* (Málaga, Spain) *(MODULARITY 2016)*. ACM, New York, NY, USA, 54–65. https://doi.org/10.1145/2889443.2889444

Carmen Torres Lopez, Robbert Gurdeep Singh, Stefan Marr, Elisa Gonzalez Boix, and Christophe Scholliers. 2019. Multiverse Debugging: Non-Deterministic Debugging for Non-Deterministic Programs (Brave New Idea Paper). In *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom (LIPIcs)*, Alastair F. Donaldson (Ed.), Vol. 134. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 27:1–27:30. https://doi.org/10.4230/LIPIcs.ECOOP.2019.27

Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (Seattle, WA, USA) *(ASPLOS XIII)*. ACM, New York, NY, USA, 329–339. https://doi.org/10.1145/1346281.1346323

Manuel Bernhardt. 2020. Akka anti-patterns: race condition. https://manuel.bernhardt.io/2016/08/16/akka-anti-patterns-race-conditions/.

Na Meng, Stefan Nagy, Danfeng (Daphne) Yao, Wenjie Zhuang, and Gustavo Arango Argoty. 2018. Secure Coding Practices in Java: Challenges and Vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 372–383. https://doi.org/10.1145/3180155.3180201

Martin Monperrus. 2014. A Critical Review of "Automatic Patch Generation Learned from Human-Written Patches": Essay on the Problem Statement and the Evaluation of Automatic Software Repair. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 234–242. https://doi.org/10.1145/2568225.2568324

Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. 2016. Jumping through Hoops: Why Do Java Developers Struggle with Cryptography APIs?. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 935–946. https://doi.org/10.1145/2884781.2884790

Stas Negara, Rajesh K. Karmani, and Gul Agha. 2011. Inferring Ownership Transfer for Efficient Message Passing. *SIGPLAN Not.* 46, 8 (Feb. 2011), 81–90. https://doi.org/10.1145/2038037.1941566

Netty. May 2020. https://netty.io/.

NLTK Project. 2020. Porter Stemming Algorithm in NLTK 3.4.5. https://www.nltk.org/api/nltk.stem.html.

Hridesh Rajan. 2015. Capsule-Oriented Programming. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2* (Florence, Italy) *(ICSE '15)*. IEEE Press, 611–614.

Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the "Naturalness" of Buggy Code. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 428–439. https://doi.org/10.1145/2884781.2884848

Christoffer Rosen and Emad Shihab. 2016. What Are Mobile Developers Asking about? A Large Scale Study Using Stack Overflow. *Empirical Softw. Engg.* 21, 3 (June 2016), 1192–1223. https://doi.org/10.1007/s10664-015-9379-3

Scala 2.13.2 API. May 2020. https://www.scala-lang.org/api/current/scala/index.html.

Christophe Scholliers, Eric Tanter, and Wolfgang De Meuter. 2014. Parallel actor monitors: Disentangling task-level parallelism from data partitioning in the actor model. *Science of Computer Programming* 80 (2014), 52 – 64. https://doi.org/10.1016/j.scico.2013.03.011 Special section on foundations of coordination languages and software architectures (selected papers from FOCLASA'10), Special section - Brazilian Symposium on Programming Languages (SBLP 2010) and Special section on formal methods for industrial critical systems (Selected papers from FMICS'11).

Koushik Sen and Gul Agha. 2006. Automated Systematic Testing of Open Distributed Programs. In *Proceedings of the 9th International Conference on Fundamental Approaches to Software Engineering* (Vienna, Austria) *(FASE'06)*. Springer-Verlag, Berlin, Heidelberg, 339–356. https://doi.org/10.1007/11693017_25

Stack Exchange. December 2019. https://archive.org/details/stackexchange.

Stack Overflow. 2020. How to create a Minimal, Reproducible Example. https://stackoverflow.com/help/minimal-reproducible-example.

Quentin Stiévenart, Jens Nicolay, Wolfgang De Meuter, and Coen De Roover. 2017. Mailbox Abstractions for Static Analysis of Actor Programs. In *31st European Conference on Object-Oriented Programming (ECOOP 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Peter Müller (Ed.), Vol. 74. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 25:1–25:30. https://doi.org/10.4230/LIPIcs.ECOOP.2017.25

Janwillem Swalens, Stefan Marr, Joeri De Koster, and Tom Van Cutsem. 2014. Towards Composable Concurrency Abstractions, In Proceedings of the Workshop on Programming Language Approaches to Concurrency and communication-cEntric Software (PLACES). *EPTCS* 155, 54–60. https://doi.org/10.4204/EPTCS.155.8

Samira Tasharofi, Peter Dinges, and Ralph E. Johnson. 2013. Why Do Scala Developers Mix the Actor Model with Other Concurrency Models?. In *Proceedings of the 27th European Conference on Object-Oriented Programming* (Montpellier,

France) *(ECOOP'13)*. Springer-Verlag, Berlin, Heidelberg, 302–326. https://doi.org/10.1007/978-3-642-39038-8_13

Samira Tasharofi, Rajesh K. Karmani, Steven Lauterburg, Axel Legay, Darko Marinov, and Gul Agha. 2012. TransDPOR: A Novel Dynamic Partial-order Reduction Technique for Testing Actor Programs. In *Proceedings of the 14th Joint IFIP WG 6.1 International Conference and Proceedings of the 32Nd IFIP WG 6.1 International Conference on Formal Techniques for Distributed Systems* (Stockholm, Sweden) *(FMOODS'12/FORTE'12)*. Springer-Verlag, Berlin, Heidelberg, 219–234. https://doi.org/10.1007/978-3-642-30793-5_14

S. Tasharofi, M. Pradel, Y. Lin, and R. Johnson. 2013. Bita: Coverage-guided, automatic testing of actor programs. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 114–124. https://doi.org/10.1109/ASE.2013.6693072

Carmen Torres Lopez, Elisa Gonzalez Boix, Christophe Scholliers, Stefan Marr, and Hanspeter Mössenböck. 2017. A Principled Approach towards Debugging Communicating Event-Loops. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control* (Vancouver, BC, Canada) *(AGERE 2017)*. Association for Computing Machinery, New York, NY, USA, 41–49. https://doi.org/10.1145/3141834.3141839

Carmen Torres Lopez, Stefan Marr, Elisa Gonzalez Boix, and Hanspeter Mössenböck. 2018. A Study of Concurrency Bugs and Advanced Development Support for Actor-based Programs. In *Programming with Actors - State-of-the-Art and Research Perspectives*. 155–185. https://doi.org/10.1007/978-3-030-00302-9_6

Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. 2019. Understanding Real-World Concurrency Bugs in Go. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. ACM, New York, NY, USA, 865–878. https://doi.org/10.1145/3297858.3304069

Jie Wang, Wensheng Dou, Yu Gao, Chushu Gao, Feng Qin, Kang Yin, and Jun Wei. 2017. A Comprehensive Study on Real World Concurrency Bugs in Node.Js. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 520–531. http://dl.acm.org/citation.cfm?id=3155562.3155628

Xin-Li Yang, David Lo, Xin Xia, Zhi-Yuan Wan, and Jian-Ling Sun. 2016. What Security Questions Do Developers Ask? A Large-Scale Study of Stack Overflow Posts. *Journal of Computer Science and Technology* 31, 5 (01 Sep 2016), 910–924. https://doi.org/10.1007/s11390-016-1672-0

Jie Yu. 2013. *Finding and Tolerating Concurrency Bugs*. Ph.D. Dissertation. The University of Michigan.

Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An Empirical Study on TensorFlow Program Bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. ACM, New York, NY, USA, 129–140. https://doi.org/10.1145/3213846.3213866

Bo Zhou, Iulian Neamtiu, and Rajiv Gupta. 2015. Predicting Concurrency Bugs: How Many, What Kind and Where Are They?. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering* (Nanjing, China) *(EASE '15)*. Association for Computing Machinery, New York, NY, USA, Article 6, 10 pages. https://doi.org/10.1145/2745802.2745807

Thomas Zimmermann, Nachiappan Nagappan, Philip J. Guo, and Brendan Murphy. 2012. Characterizing and Predicting Which Bugs Get Reopened. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, 1074–1083.