**μAkka: Mutation Testing for Actor Concurrency in Akka using Real-World Bugs**

Mohsen Moradi Moghadam  
Oakland University  
USA  
moradimoghadam@oakland.edu  

Raffi Khatchadourian  
Hunter College, City University of New York  
USA  
raffi.khatchadourian@hunter.cuny.edu  

Mehdi Bagherzadeh  
Oakland University  
USA  
magherdezeh@oakland.edu  

Hamid Bagheri  
University of Nebraska, Lincoln  
USA  
bagheri@unl.edu

### ABSTRACT

Actor concurrency is becoming increasingly important in the real-world and mission-critical software. This requires these applications to be free from actor bugs, that occur in the real world, and have tests that are effective in finding these bugs. Mutation testing is a well-established technique that transforms an application to induce its likely bugs and evaluate the effectiveness of its tests in finding these bugs. Mutation testing is available for a broad spectrum of applications and their bugs, ranging from web to mobile to machine learning, and is used at scale in companies like Google and Facebook. However, there still is no mutation testing for actor concurrency that uses real-world actor bugs. In this paper, we propose **μAkka**, a framework for mutation testing of Akka actor concurrency using real actor bugs. Akka is a popular industrial-strength implementation of actor concurrency. To design, implement, and evaluate **μAkka**, we take the following major steps: (1) manually analyze a recent set of 186 real Akka bugs from Stack Overflow and GitHub to understand their causes; (2) design a set of 32 mutation operators, with 138 source code changes in Akka API, to emulate these causes and induce their bugs; (3) implement these operators in an Eclipse plugin for Java Akka; (4) use the plugin to generate 11.7k mutants of 10 real GitHub applications, with 446.4k lines of code and 7.9k tests; (5) run these tests on these mutants to measure the quality of mutants and effectiveness of tests; (6) use PIT to generate 26.2k mutants to compare **μAkka** and PIT mutant quality and test effectiveness. PIT is a popular mutation testing tool with traditional operators; (7) manually analyze the bug coverage and overlap of **μAkka**, PIT, and actor operators in a previous work; and (8) discuss a few implications of our findings. Among others, we find that **μAkka** mutants are higher quality, cover more bugs, and tests are less effective in detecting them.

### CCS CONCEPTS

- Software and its engineering → Software testing and debugging;  
- Theory of computation → Concurrency.

### KEYWORDS

Mutation testing, Actor concurrency, Mutation operators, Mutant quality, Test effectiveness, **μAkka**, Akka.

### ACM Reference Format:


---

**1 INTRODUCTION**

Actor concurrency is becoming increasingly important in the real-world and mission-critical software. For example, PayPal uses actor concurrency to serve more than a billion financial transactions per day [104], Spark to shuffle hundreds of terabytes of big data [35], and Groupon to personalize coupons for 48 million customers in a day [103]. NASA, Microsoft, Twitter, Verizon, CapitalOne, and Weight Watchers are among many other users of actor concurrency [67, 102]. This requires these applications to be free from actor bugs, that occur in the real world, and have tests that are effective in finding these bugs, which can cost large sums of money or even lives. Unlike multithreaded concurrency, in which lower-level threads communicate using shared memory and locks, in actor concurrency, higher-level actors communicate using asynchronous messages [47, 48]. This makes not only actor concurrency but also its bugs [58, 87, 108] fundamentally different from multithreaded concurrency [68, 98]. Previous work [58, 60, 61, 87, 107, 108] discusses some of these differences.

Mutation testing [16, 76, 91] is a well-established technique that transforms the source or the binary, code of an application to induce its likely bugs and evaluate the effectiveness of its tests in finding these bugs. Mutation testing is available for a broad spectrum of applications and their bugs, ranging from web [124] to mobile [106] to machine learning [88], and is used at scale in companies like Google [122, 123] and Facebook [64]. However,
there still is no mutation testing for actor concurrency that uses real-world actor bugs. The only relevant work is the work by Jagannath et al. [89] that proposes 12 mutation operators in ActorFoundry [57]. However, these operators are based on the individual experiences of the authors, and not a curated set of real bugs, are not implemented, are not evaluated, and are often specific to ActorFoundry’s syntax and semantics and not applicable to today’s industrial-strength implementations of actor concurrency. ActorFoundry is a now-defunct research prototype actor language with no industrial use. We discuss this work further in detail throughout the paper.

In this paper, we propose \(\mu\text{Akka}\), a framework for mutation testing of Akka actor concurrency using real actor bugs. Akka [105], from LightBend, is an industrial-strength implementation of actor concurrency, among others such as Orleans [65], from Microsoft, and Erlang [56], from Ericsson. We focus on Akka because it is growing faster than others. For example, in the past five years, there are 3,727 Akka questions and answers [55] in Stack Overflow, which is 1.8 and 34.2x more than Erlang and Orleans, respectively. Similarly, there are 10,034 Akka applications [54] in GitHub, which is 1.5 and 7.3x more than Erlang and Orleans.

To design, implement, and evaluate \(\mu\text{Akka}\), we take the following major steps:

1. manually analyze a recent set of 186 real Akka bugs [58] from Stack Overflow and GitHub to understand their causes.
2. design a set of 32 mutation operators, with 138 source code changes in Akka API, to emulate these causes and induce bugs.
3. implement these operators in an Eclipse plugin for Java Akka.
4. use the plugin to generate 11,736 mutants of 10 real GitHub applications, with 446,444 lines of code and 7,871 tests.
5. run these tests on these mutants to measure the quality of mutants and the effectiveness of tests in detecting and killing these mutants, using ease-of-killing, duplicity, subsumption, and mutation score metrics that previous work uses often.
6. use PIT [72] to generate 26,177 mutants to compare \(\mu\text{Akka}\) and PIT mutant quality and test effectiveness.
7. manually analyze the bug coverage and overlap of \(\mu\text{Akka}\), PIT and Jagannath et al.’s [89] actor operators.
8. discuss the implications of our findings.

PIT is a popular [99] mutation testing tool with traditional logic operators such as Math that replaces mathematical operands *, -, +, /, %. Among others, we find that:

- **Mutant quality**: \(\mu\text{Akka}\) mutants are high quality: 2x harder to kill, 3x less duplicate, and 1.3x less subsumed than PIT.
- **Test effectiveness**: Tests are ineffective for \(\mu\text{Akka}\) mutants: 1.3x less effective in covering and 2.3x less effective in killing \(\mu\text{Akka}\) mutants than PIT.
- **Bug coverage**: \(\mu\text{Akka}\) bug coverage is high: 3.3x more bug coverage than Jagannath et al.’s. \(\mu\text{Akka}\) bug coverage benefits from the addition of PIT: \(\mu\text{Akka} + \text{PIT}\) cover 1.1x more bugs than \(\mu\text{Akka}\) alone and 9.5x more than PIT alone.

A few implications of our findings are in predictive, selective, and sampling mutation testing and actor concurrency testing.

## 2 BACKGROUND

In this section, we discuss the basics of actor concurrency, Akka, and mutation testing that we use throughout the paper.

### 2.1 Actor Concurrency

#### Basic actor concurrency

Unlike multithreaded concurrency, in which a program is a set of threads that communicate using shared memory and locks, in basic actor concurrency [47, 48], the program is a set of *actors* that communicate by *sending*, *receiving*, and *processing* of asynchronous messages. An actor has its own thread of execution and behavior and makes its state accessible only through messages, to avoid sharing. To send a message, a *sender* actor sends a fire-and-forget message without waiting and *blocking* for its response. To receive the message, a receiver actor enqueues the message in its *mailbox*. To process the message, the receiver dequeues the message from its mailbox and executes it sequentially and to the end before processing the next message in the mailbox. During the processing, an actor can change state and behavior, send a message, or create a new actor.

#### Akka actor concurrency

To allow for the development of real-world applications, Akka extends the basic actor concurrency with several necessary features, most of which are *programmatic* and *dynamic*. These features are *actor path* to locate a local or remote actor, *life cycle* to manage the actor life, *parental hierarchy and supervision* to manage the actor creation and failure, *configuration* to configure an application settings, *actor system* to provide the actor dispatch and scheduling, *message sendings*, *interaction patterns* to support more ways of actor interactions, *stashing* to buffer the actor messages for delayed processing, *deployment* to deploy remote actors, and *clustering* for distribution of actors over network. These features are often the causes for Akka bugs [58, 87] and are discussed further throughout the paper. ActorFoundry [57] lacks most of these features.

### 2.2 Mutation Testing

#### Basics

Mutation testing [16, 76, 91] transforms the source or binary, code of an application to induce its likely bugs and evaluate the effectiveness of its tests in finding these bugs. For each bug, a mutation operator slightly changes the application code to emulate the cause of the bug and induce the bug. The operator generates a *mutant* of the application which is the original application plus the induced bug. A test is *effective* if it can *detect* and *kill* a mutant by distinguishing its behavior from the original application behavior; otherwise the mutant stays *alive*. A test cannot kill a mutant that it does not cover. A test covers a mutant if its execution executes the mutated part of the mutant. Intuitively, the more mutants the tests cover and kill the more effective the tests are.

#### Types of mutants

There are different types of mutants, including *stillborn*, *easy-to-kill*, *trivial*, *redundant*, *duplicate*, *subsuming*, and *equivalent*. For a mutant \(m\) of an original application \(p\) and the set of tests \(T_{\text{cover},m}\) and \(T_{\text{kill},m}\) that, respectively, cover and kill \(m\), \(m\) is stillborn if it includes syntactic and semantic errors that a compiler can catch and thus prevent its compilation. A mutant \(m\) is easy-to-kill if a large number \(k\) of tests in \(T_{\text{cover},m}\) kill it. Following previous work [117], we set \(k\) to 97.5%. A mutant \(m\) is trivial if all the tests in \(T_{\text{cover},m}\) can kill \(m\). Triviality is a special case of ease-of-killing. A mutant \(m\) is redundant if tests that kill another mutant \(m'\) can also kill \(m\). A redundant mutant is either duplicate or subsumed. A mutant \(m\) is the duplicate of \(m'\) if their behaviors are the
functional equivalent of each other, but not equal to \( p \). Duplicate mutants are syntactically different from each other but semantically the same. A subsuming mutant \( m \) subsumes a subsumed mutant \( m' \) if \( \mathcal{F}_{\text{kill}}^m \) is a subset of \( \mathcal{F}_{\text{kill}}^{m'} \). \( m \) is equivalent if its behavior is the functional equivalent of \( p \). An equivalent mutant is syntactically different from the original application but semantically the same.

Automatic determination of all duplicate and equivalent mutants is proven to be undecidable [53, 69]. Our definition approximates duplicity using the execution of tests [118]. Approximation of equivalence requires complex heuristics and analyses, such as weakest precondition [62], and is out of the scope of this work.

Our definitions follow similar definitions from previous work for stillborn [16, 91], easy-to-kill [117], trivial [85, 94], redundant [16, 52, 94, 115], duplicate [16, 119], subsuming [16, 52, 94], subsumed [16, 52, 94], and equivalent [16, 52, 91, 94, 119].

**Quality of mutants** The quality of mutants is critical to mutation testing since non-quality mutants could bias the testing. For example, previous work [118] shows that using subsumed mutants, which are non-quality, could, on average, bias the conclusions of 62% of arbitrary sets of mutation testings. A mutant \( m \) is quality, useful [92, 94], or valuable to mutation testing, if it is killable and its killing requires the addition of a new test to set of tests \( \overrightarrow{T}_{\text{cov}} \) that cover it; otherwise it is non-quality. Trivial, easy-to-kill, redundant, and equivalent mutants are among the most well-known non-quality mutants [16, 93, 94, 117]. This is because, all and a large subset of tests in \( \overrightarrow{T}_{\text{cov}} \) can kill a trivial and easy-to-kill \( m \), respectively, with no need for a new test; a non-empty subset of \( \overrightarrow{T}_{\text{cov}} \) that kills \( m' \) can also kill its redundant \( m \) with no need for new tests; and there is no test that can kill an equivalent \( m \). Our definition follows similar definitions from previous work for quality mutants [16, 92, 94, 117~119].

**Effectiveness of tests** Mutation testing evaluates the effectiveness of tests in finding bugs that it induces. **Mutation score** [91] is the most well-known metric to measure the effectiveness of tests. For the set of all tests \( \overrightarrow{T}_{\text{cov}} \) that cover at least a mutant, the traditional mutation score is the ratio of the number of mutants that tests in \( \overrightarrow{T}_{\text{cov}} \) kill to the number of all mutants. Our definitions follow similar definitions from previous work for the mutation score [16, 91, 115].

**Mutant coverage** Neither all tests cover all the mutants nor all mutants are covered by all the tests. **Mutant coverage** is the metric that measures the coverage of mutants. Our definition follows the standard definition of coverage [86].

**Bug coverage** A mutation operator covers a bug if it can emulate the cause of the bug and induce the bug. Our definition follows similar definitions from previous work for bug coverage [106].

### 3 METHODOLOGY

In this section, we discuss our methodology to analyze Akka actor bugs, design and group \( \mu \text{Akka} \) mutation operators for these bugs, and measure the quality of mutants that \( \mu \text{Akka} \) and PIT operators generate, the effectiveness of tests to identify these mutants, and the coverage of bugs by \( \mu \text{Akka} \) and Jagannath et al.’s operators.

#### 3.1 Akka Actor Bugs

For Akka bugs, we use a set \( \mathcal{B} \) of 186 real bugs from a recent previous work [58]. \( \mathcal{B} \) includes 130 bugs from Stack Overflow questions and answers and 56 bugs from GitHub applications. These bugs cover a broad spectrum of causes, ranging from API confusion to model confusion to missteps in the application logic, and a broad spectrum of symptoms, ranging from incorrect messaging to incorrect termination to unexpected application behavior.

#### 3.2 Mutation Operators

**Design** For each bug in \( \mathcal{B} \), we take the following steps to design the mutation operator that induces the bug. First, we manually analyze the bug to understand its cause. For example, there is a bug [19] in \( \mathcal{B} \) in which an actor cannot be created with a cause that its name is not unique. Akka requires unique actor names. Second, we manually search Akka API and its documentation to identify methods that can emulate this cause by small changes in their syntax [46, 76, 91]. Several methods may emulate the same cause. For example, \texttt{actorOf(String name)} and \texttt{actorOf()}} are two Akka methods that, their invocations, create an actor with a given and random name, respectively. Changing these invocations to \texttt{actorOf(String name)} could emulate the cause in which the actor name is not unique, if \texttt{name} is an actor name that exists already. We ensure that the syntactic changes of our operators do not violate the compile-time syntax and semantic requirements of Akka Java API. Third, we select a name for the mutation operator that describes the changes it makes. For example, we give the name \texttt{CHANGE NAME} to the operator that changes \texttt{actorOf(String name)} to \texttt{actorOf(String name)}

We use the open card sort [79] to identify API methods, their source code changes, and the name of a mutation operator. In the open card sort, there are no predefined API methods, source code changes, and operator names; instead, they are developed during the sorting process. To sort, the first and second authors individually analyze the bugs and reiterate and refine until they agree. The second author is a Software Engineer and Programming Languages professor with extensive expertise in actor and multithreaded concurrent systems. The first author is a Ph.D. student with extensive coursework in concurrent and mobile systems. The third and fourth authors are Software Engineer professors with extensive expertise in concurrent and streaming systems, and testing and fault localization, respectively. First three authors have several years of extensive industrial work experience. In total, we design 32 mutation operators using source code changes in 138 Akka API. Table 1 shows these operators and their short descriptions. The formalization and API source code changes for these operators can be found in our replication package [27].

**Grouping** We use the same open card sort to group the mutation operators, based on the semantic relation between the bug causes they emulate. For example, \texttt{CHANGE NAME} and \texttt{CHANGE HIERARCHY} operators that change the related name and the hierarchy parts of the path of an actor. In Akka, the path of an actor denotes its physical location in a system and...
includes its name and hierarchy, among others. In total, we group our operators into 8 groups. Table 1 shows these groups.

**Compile-time and logic bugs** To avoid the generation of stillborn mutants, we do not design mutation operators for compile-time bugs. Generation and compilation attempts for stillborn mutants decrease the efficiency of mutation testing because they can neither be compiled nor executed. A bug is a compile-time bug if its cause is a syntactic or semantic issue that prevents its successful compilation. For example, there is a bug [18] in $\mathcal{R}$ in which the compilation fails because of using an undeclared variable. There are 21 (11.3%) compile-time bug in $\mathcal{R}$. Similarly, to avoid the redesigning of non-Akka and traditional operators, that already exist [16], we do not emulate logic bugs. A bug is a logic bug if its cause is a misstep in the application logic. For example, there is a bug [20] in $\mathcal{R}$ in which the lookup and message delivery for a remote actor fails because an skipped if conditional creates the wrong path for the actor. The traditional operator Negate Conditional in PIT [72] can emulate this bug by negating the condition of the conditional. There are 65 (35.0%) logic bugs in $\mathcal{R}$. In total, we design mutation operators to emulate the causes of 100 (53.8%) bugs in $\mathcal{R}$ that are neither compile-time nor logic bugs.

### 3.3 Evaluation

**Quality of mutants** We use three well-known metrics to measure the quality of mutants that $\mu$Akka and PIT operators generate. These metrics are ease-to-killing, duplicity, and subsumption, which Section 2 defines. Intuitively, the lower the number of easy-to-kill, duplicate, and subsumed mutants the higher the quality. Previous work uses the number of ease-of-killing [117], duplicity [16, 75, 119], and subsumption [16, 52, 94] often to measure mutant quality.

**Effectiveness of tests** We use the well-known metric mutation score, that Section 2 defines, to measure the effectiveness of our tests to identify $\mu$Akka and PIT mutants. Previous work uses mutation score [95, 120] often to measure the effectiveness of tests. We execute the tests to measure the mutant quality and test effectiveness. However, a test can be flaky and produce different outcomes for different executions. To prevent flaky tests from skewing measurements [121], we follow previous work [63] and use the average of three executions of our tests to calculate our metrics.

**Bug coverage** We use the same open card sort to identify $\mathcal{R}$ bugs that Jagannath et al.'s [89] operators can induce and cover and their overlap with $\mu$Akka.

### 4 MUTATION OPERATORS AND GROUPS

In this section, we discuss and illustrate the mutation operators using real bug examples. In addition, we compare our operators with Jagannath et al.'s [89]. Table 1 shows our 32 mutation operators, their eight groups 8 groups $\text{PATH}$, $\text{CONFIGURATION}$, $\text{COMMUNICATION TYPE}$, $\text{LIFE CYCLE}$, $\text{EXCEPTION}$, $\text{INTERACTION}$, $\text{RACE}$, and $\text{CLUSTER}$, and their short descriptions.

#### 4.1 PATH

In Akka, an actor has a path that denotes its physical location in a system. The path includes an address and a name. The address can be either local or remote. An actor is local if it is in the same Java Virtual Machine (JVM) and is remote otherwise. A local address specifies the actor system that the actor resides in and a hierarchy of its ancestors, following akka://actor system/hierarchy/name format. Actors form a hierarchy in which a parent creates its children. A remote address, also specifies a host, protocol, and a port for remote connections, following akka.protocol://actor system@host:port/hierarchy/name format. An Akka developer is responsible to understand actor paths and their different parts, and their programmatic and dynamic modifications, and manually provide correct values for these parts; otherwise incorrect actor paths can cause bugs, as the previous work [58] shows.

**Path group** includes six mutation operators that change the name, protocol, system, host, and port, and hierarchy parts of an actor path to induce the bugs that are caused by the use of incorrect values for these parts. For example, there are three bugs [1, 14, 19] in $\mathcal{R}$ in which an actor cannot be created because its name is not unique. Akka requires actor names to be unique in a system. Change Name incudes these bugs by changing the name of an actor to an actor name that already exists in the system. Similarly, there is a bug [3] in which an actor cannot be looked up and messages sent to it cannot be delivered because its hierarchy is missing an ancestor. Change Hierarchy induces this bug by changing the hierarchy of an actor to a hierarchy with one less random ancestor. Table 1 describes all the mutation operators in PATH and the bug causes they emulate.

Jagannath et al. [89] use their individual experiences to propose 12 unimplemented mutation operators in 3 groups for the research actor language ActorFoundry. These operators, inside parentheses, and their groups, outside parentheses are: Messaging (Remove Send/Receive, Modify Message Parameters, Reorder Message Parameter, Modify Message Name, Modify Message Recipient, and Change Synchronization Type), Constraint (Remove Constraint and Modify Constraint), and Creation/Deletion (Remove Creation/Deletion, Modify Creation Parameter, and Reorder Creation Parameters).

All six mutation operators in PATH are new and cannot be found in the previous work by Jagannath et al.’s.

### 4.2 Configuration

A configuration defines the properties of an application using a set of parameter and value pair settings. In Akka, a default configuration [50] defines 255 default settings. These settings define the properties of actors and actor systems of an application and their deployment, mailboxes, dispatchers, and routers, among others. For example, the mailbox can be configured to be either unbounded or bounded with a specific capacity for the number of its messages. An application configuration can override the default settings, define new settings, such as a new dispatcher [13] with a different max and min number of threads in its thread pool, or failback on another configuration for the settings that it does not define. An Akka developer is responsible to understand the default and application configurations, their parameter and value settings, overriding, and failback, and their programmatic modifications, and manually configure non-default settings; otherwise incorrect configurations can cause bugs [58].

**Configuration** includes six mutation operators that change Application, Failback, Mailbox, Dispatch, Rating, and Deployment settings to induce bugs that are caused by using incorrect
<table>
<thead>
<tr>
<th>Group</th>
<th>Mutation Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Path</td>
<td>name</td>
<td>change the name of an actor to an existing or random actor name</td>
</tr>
<tr>
<td></td>
<td>protocol</td>
<td>change the local protocol of an actor to a remote protocol and vice versa</td>
</tr>
<tr>
<td></td>
<td>actor system</td>
<td>change the actor system name of an actor to an existing or random actor name</td>
</tr>
<tr>
<td></td>
<td>host</td>
<td>change the host name or IP address of an actor to a random host name or IP address</td>
</tr>
<tr>
<td></td>
<td>port</td>
<td>change the port number of an actor’s host to a random port number</td>
</tr>
<tr>
<td></td>
<td>hierarchy</td>
<td>change the ancestral hierarchy of an actor to a hierarchy with one less random ancestor</td>
</tr>
<tr>
<td></td>
<td>application</td>
<td>change the application configuration to an empty configuration or drop it</td>
</tr>
<tr>
<td></td>
<td>fallback</td>
<td>change the fallback configuration to an empty configuration or drop it</td>
</tr>
<tr>
<td></td>
<td>mailbox</td>
<td>change the mailbox configuration of an actor to an empty or random configuration or drop it</td>
</tr>
<tr>
<td></td>
<td>dispatch</td>
<td>change the dispatch configuration of an actor to an empty or random configuration or drop it</td>
</tr>
<tr>
<td></td>
<td>routing</td>
<td>change the routing configuration of an actor to the default or no router configuration or drop it</td>
</tr>
<tr>
<td></td>
<td>deployment</td>
<td>change the deployment configuration of an actor to the default or drop the configuration</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Communication type</td>
<td>send type</td>
<td>change the set of message types that an actor sends to a set with one more random type</td>
</tr>
<tr>
<td></td>
<td>receive type</td>
<td>change the set of message types that an actor receives to a set with one less random type</td>
</tr>
<tr>
<td>Life cycle</td>
<td>lookup</td>
<td>change the lookup of an actor to occur in another existing or random actor system</td>
</tr>
<tr>
<td></td>
<td>starting</td>
<td>change the starting pre- and post-behaviors of an actor to empty or supertype behaviors</td>
</tr>
<tr>
<td></td>
<td>stopping</td>
<td>change the stopping pre- and post-behaviors of an actor to empty or supertype behaviors</td>
</tr>
<tr>
<td></td>
<td>restarting</td>
<td>change the restarting pre- and post-behaviors of an actor to empty or supertype behaviors</td>
</tr>
<tr>
<td></td>
<td>termination</td>
<td>change the termination of an actor or actor system by removing or delaying the termination</td>
</tr>
<tr>
<td></td>
<td>monitoring</td>
<td>change monitoring of an actor by removing the monitoring</td>
</tr>
<tr>
<td>Exception</td>
<td>failure</td>
<td>change the failure behavior of an actor to throw an exception</td>
</tr>
<tr>
<td></td>
<td>supervision</td>
<td>change the supervisory behavior of an actor to another random behavior</td>
</tr>
<tr>
<td>Interaction</td>
<td>synchrony</td>
<td>change the synchrony of a message that an actor sends from sync to async and vice versa</td>
</tr>
<tr>
<td></td>
<td>blocking</td>
<td>change the blocking behavior of an actor by increasing or decreasing its wait or sleep durations</td>
</tr>
<tr>
<td></td>
<td>timeout</td>
<td>change the timeout behavior of an actor by increasing or decreasing its timeout durations</td>
</tr>
<tr>
<td></td>
<td>forwarding</td>
<td>change the sender of a message that actor forwards to a random actor reference</td>
</tr>
<tr>
<td></td>
<td>sender</td>
<td>change the sender of a message that an actor sends to an existing actor reference</td>
</tr>
<tr>
<td></td>
<td>receiver</td>
<td>change the receiver of a message that an actor sends to an existing actor reference</td>
</tr>
<tr>
<td></td>
<td>scheduling</td>
<td>change the scheduling of a message that an actor sends by increasing or decreasing its delays</td>
</tr>
<tr>
<td>Race</td>
<td>sharing</td>
<td>change the sharing of data between an actor and its concurrent future or async tasks</td>
</tr>
<tr>
<td></td>
<td>ordering</td>
<td>change the ordering between two messages by delaying the first message</td>
</tr>
<tr>
<td>Cluster</td>
<td>subscription</td>
<td>change a cluster event that an actor subscribes for to another Akka cluster event</td>
</tr>
<tr>
<td></td>
<td>unsubscription</td>
<td>change a cluster event that an actor unsubscribes from to another Akka cluster event</td>
</tr>
</tbody>
</table>

Table 1: Mutation operators for actor concurrency in \( \mu \text{Akka} \), their groups, and descriptions.

settings for these configurations. For example, there is a bug [8] in \( \mathcal{B} \) in which an actor system cannot be configured and created because its application configuration does not load. Change Application induces this bug by replacing the application configuration with an empty configuration with no settings. Similarly, there is a bug [30] in which an actor cannot stash its messages because its mailbox is not configured properly. Change Mailbox induces this bug by removing the mailbox configuration of an actor.

All six mutation operators in Configuration are new and cannot be found in Jagannath et al.’s.

### 4.3 Communication Type

In Classic Akka, unlike Akka Typed [51], an actor neither knows about the types of messages that it may receive nor the type of messages it can send. An Akka developer is responsible to manually discover these message types and ensure that the actor can receive all the messages that others send to it and sends only messages that
others can receive; otherwise, sending or receiving messages with incorrect types can cause bugs [58].

**Communication Type** includes two mutation operators **Change Receive Type** and **Send Type** that change the type of messages that an actor can receive or send to induce the bugs that are caused by receiving or sending incorrect message types. For example, there are two bugs [22, 23] in B in which the application behavior is undesired [22] or terminates prematurely [22] because an actor receives a message of a type that it cannot process. **Change Receive Type** induces these bugs by changing the set of message types that a receiver actor can receive to another set with one less random message type. Similarly, **Change Send Type** can induce these bugs by changing the set of message types that a sender actor can send to another set with one more random message type.

Our **Change Receive Type** partially overlaps with Jagannath et al.’s Remove Send/Receive (RSR) operator, in their Messaging group, that ”mimics the omission of messages by removing sends/receives”. Similarly, our **Change Send Type** overlaps with their Messaging Modify Message Name (MMN) operator, in Messaging, that ”modifies the name [type] of a message being sent”.

### 4.4 Life Cycle

In Akka, an actor and its enclosing actor system go through different stages in their life cycle, such as creation, lookup, starting, stopping, restarting, termination, and monitoring. An Akka developer is responsible to understand these life cycles and their programmatic stages and manually manage these stages and their pre- and post-behaviors correctly; otherwise incorrect management of life cycles can cause bugs [58, 87].

**Life Cycle** includes six mutation operators that **Change Lookup, Starting, Stopping, Restarting, Termination, and Monitoring** of an actor and its actor system to induce the bugs that are caused by the incorrect managements of these stages. For example, there are two bugs [17, 25] in B in which the application does not shutdown [17] or consumes all its available memory [25] because its actor system and actors do not terminate. **Change Termination** induces these bugs by removing the termination of an actor and its enclosing actor system. Similarly, there is a bug [26] in which the termination of an actor goes unnoticed because the actor that is responsible to manage the termination is not monitoring the terminating actor. **Change Monitoring** induces this bug by removing the termination monitoring of an actor.

Our **Change Termination** partially overlaps with Jagannath et al.’s Remove Creation/Deletion (RCD) operator, from their Creation/Deletion group, that ”mimics the omission of creation/deletion of an actor by removing an actor creation/deletion”. All other five mutation operators in **Life Cycle** are new.

### 4.5 Exception

In Akka, a parent actor is not only responsible to create but also supervise its child actors and manage their failure when they throw exceptions. The **supervision** strategy of the parent specifies to either stop, resume, or restart the failed child or escalate the exception up the supervision hierarchy to the parent of the parent. An Akka developer is responsible to understand actor failures and exceptions, programmatic supervisory hierarchies, and supervision strategies and handle these exceptions correctly; otherwise, incorrect handling of exceptions can cause bugs [58].

**Exception** includes two mutation operators that **Change Failure** and **Supervision** behavior of an actor to induce the bugs that are caused by incorrect handlings of exceptions. For example, there are two bugs [6, 11] in B in which the application does not terminate [6] or swallows an exception [11] because an actor throws an exception that its ancestor does not handle properly. **Change Failure** induces these bugs by changing the behavior of an actor to throw an exception. Similarly, there are two bugs [2, 29], in which an actor does not restart properly because its parent is using the wrong supervision strategy. **Change Supervision** incudes these bugs by changing the supervision strategy of an actor.

Both mutation operators in **Exception** are new.

### 4.6 Interaction

In Akka, there are several ways in which an actor can interact with another. In addition to an asynchronous fire-and-forget message, a sender actor can send a synchronous request-response message to a receiver and wait and block for its response in a future variable for a timeout period. A future is a placeholder for an incomplete task with a result that is not ready yet. Similarly, a receiver can forward a message it receives to another actor, without changing the sender of the message, or schedule to send a message in specific intervals. An Akka developer is responsible to understand these programmatic interactions and their semantics and use them correctly; otherwise incorrect interactions can cause bugs [58, 108].

**Interaction** includes seven mutation operators that **Change Synchrony, Blocking, Timeout, Forwarding, Sender, Receiver, and Scheduling** of actor interactions to induce bugs that incorrect uses can cause. For example, there are two bugs [9, 12] in B in which a receiver actor cannot respond to its sender more than once because the temporary actor that a request-response message creates to receive the response, terminates after receiving the first response. **Change Synchrony** induces these bugs by changing an asynchronous fire-and-forget message to a synchronous request-reply message. Similarly, there is a bug [34] in which a receiver cannot process its messages because the messages are sent too fast. **Change Scheduling** induces this bug by decreasing the initial and interval delays between messages that a scheduler sends.

Our **Change Synchrony** overlaps with Jagannath et al.’s Change message Synchronization Type (CST), in Messaging, that ”changes a synchronous send to an asynchronous send and vice versa”. Similarly, our **Change Receiver** overlaps with their Modify Message Recipient (MMR), in Messaging, that ”modifies the recipient of a message”. All other five operators in **Interaction** are new.

### 4.7 Race

A race occurs if two concurrent computations access the same memory and one modifies the memory. In Akka, to avoid races, an actor processes its messages sequentially and one at a time. However, both lower-level data races and higher-level message races [60] are still possible. A data race occurs when a concurrent future or async task that runs outside the actor shares memory with the actor and either the actor or the task modifies the memory. Similarly, a message race occurs when two messages arrive at the same actor out
of their desired order and the processing of either of the messages modifies the actor memory. An Akka developer is responsible to understand data sharings and message orderings among all actors and non-actor concurrent computations of a system and make sure they are free from races; otherwise races could cause bugs [58, 87, 108].

RACE includes two mutation operators that Change Sharing of data and Ordering of messages to induce the bugs that incorrect sharing and message ordering can cause. For example, there are seven bugs [4, 5, 7, 10, 28, 32, 33] in which a response is not delivered to a sender actor [4, 5, 10, 28, 32] or the application does not behave as desired [7, 33] because an actor and its future task share the variable sender which the actor modifies. The variable sender is the sender of the current message that the actor is processing and changes when the actor starts to process another message. Change Sharing induces this bug by changing a random actor reference in a future or async task with getSender(). The method getSender accesses, reads and returns the value of the shared variable sender. Similarly, there is a bug [21] in which the application produces incorrect results because two messages that initiate the warmup of a simulation and production of the results arrive out of order. Change Sharing induces this bug by delaying the sending of a message to reorder the arrival of messages.

Our Change Sharing overlaps with Jagannath et al.’s Change (message) Reference Type (CRT), in Messaging, that “changes a versa” to change the sharing between actors. Change Ordering is new.

4.8 Cluster

In Akka, a cluster is a group of actor systems that provide distribution, load balancing, and failover for their actors. An actor system is a logical node of the cluster. An actor in the cluster can subscribe for the membership, domain, and reachability events of the cluster, receive messages when these events occur and process these messages accordingly. Similarly, the actor can unsubscribe from these events. An Akka developer is responsible to understand these events, their semantics, and their subscriptions and unsubscriptions and manage them correctly; otherwise incorrect subscriptions or unsubscriptions can cause bugs [58].

Cluster includes two mutation operators that Change Subscription and Unsubscription of actors in a cluster to induce bugs that are caused by incorrect subscriptions. For example, there is a bug [31] in which an actor misses a cluster leader change event because it subscribes for an incorrect member event instead of the correct domain event. The leader change event is a domain event. Change Subscription induces this bug by changing the cluster event that an actor subscribes to another random cluster event. Similarly, Change Unsubscription can induce this bug by changing the cluster event that an actor unsubscribes from.

Both mutation operators in Cluster are new.

5 Evaluation

In this section, we discuss the implementation of our mutation operators in μAkka, evaluate the mutant quality and test effectiveness of real application in μAkka and PIT, and study the bug coverage and overlap of μAkka, Jagannath et al.’s [89], and PIT operators.

5.1 Implementation

For real-world applicability, we implement our mutation operators as an Eclipse plugin, in a framework that we call μAkka. μAkka uses 138 source code changes of Java Akka API [90] to implement our 32 operators. For efficiency, in addition to not generating stillborn mutants, we integrate the following popular techniques, from previous work, into μAkka. First, we use conditional mutation [93] that uses conditional statements to integrate all the mutants and the original application into a single application. This allows a single compilation to compile all mutants all at once and efficiently instead of one by one and inefficiently. Second, we generate and use coverage information to execute mutants only if they are covered by tests [74]. This allows to not execute the tests that do not cover mutants and mutants that are not covered by tests.

5.2 Akka Applications

We use GitHub to randomly select a set of 10 mature and real-world Java Akka applications with a total of 446,444 lines of code. Table 2 shows these applications, which cover a broad spectrum of sizes, ranging from 261,951 to 15,527 to 3,136 lines of code, of domains, from Internet of Things (IoT) to web performance testing to money transfer, and of starts, from 19,600 to 390 to 5. An application is considered to be an Akka application if it uses Java Akka APIs in its source code. Due to compilation issues in Eclipse, we include the original application into a single application. This allows a single compilation to compile all mutants all at once and efficiently. First, we use conditional mutation that uses conditional statements to integrate all the mutants and the original application into a single application. This allows to not execute the tests that do not cover mutants and mutants that are not covered by tests.

Table 2: Real-world Akka applications from GitHub.

<table>
<thead>
<tr>
<th>Id</th>
<th>Project</th>
<th>LOC</th>
<th>Domain</th>
<th>Stars</th>
</tr>
</thead>
<tbody>
<tr>
<td>ditto</td>
<td>ditto [36]</td>
<td>261,951</td>
<td>internet of things (IoT)</td>
<td>390</td>
</tr>
<tr>
<td>lms</td>
<td>sunbird-lms [17]</td>
<td>78,633</td>
<td>learning management</td>
<td>29</td>
</tr>
<tr>
<td>wot</td>
<td>wot-vestrient [38]</td>
<td>52,019</td>
<td>web of things</td>
<td>23</td>
</tr>
<tr>
<td>flower</td>
<td>flower [39]</td>
<td>21,648</td>
<td>reactive microservices</td>
<td>518</td>
</tr>
<tr>
<td>comb</td>
<td>servicecomb [40]</td>
<td>15,527</td>
<td>transactional data</td>
<td>481</td>
</tr>
<tr>
<td>rhino</td>
<td>rhino [41]</td>
<td>13,882</td>
<td>web performance testing</td>
<td>16</td>
</tr>
<tr>
<td>parc</td>
<td>parcell [42]</td>
<td>13,209</td>
<td>asynchronous web</td>
<td>800</td>
</tr>
<tr>
<td>flink</td>
<td>flink-rpc [43]</td>
<td>8,326</td>
<td>web performance testing</td>
<td>19,600</td>
</tr>
<tr>
<td>fuse</td>
<td>fuse [44]</td>
<td>3,483</td>
<td>REST server</td>
<td>15</td>
</tr>
<tr>
<td>mony</td>
<td>money-transfer [45]</td>
<td>3,136</td>
<td>money transfer API</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 3 shows the number of mutants that different mutation operator groups generate. Gen, and the number of these mutants that the tests can cover, Cov. The table shows these numbers in aggregate, for all applications, and separately, for individual applications. Gen is the ratio of the number of mutants that an operator generates to the number of mutants that all operators generate. Cov is the ratio of the covered mutants of an operator to its generated mutants. Dark gray, light gray, and boxed denote higher, lower, and average values, respectively.

Table 3: Mutant generation and coverage.

5.3 μAkka’s Mutant Generation and Coverage

Table 3 shows the number of mutants that different mutation operator groups generate. Gen, and the number of these mutants that the tests can cover, Cov. The table shows these numbers in aggregate, for all applications, and separately, for individual applications. Gen is the ratio of the number of mutants that an operator generates to the number of mutants that all operators generate. Cov is the ratio of the covered mutants of an operator to its generated mutants. Dark gray, light gray, and boxed denote higher, lower, and average values, respectively. Generation According to Table 3, row Gen, for all applications in aggregate, the number of mutants that different mutation operator groups generate are substantially different, with interaction
5.4 μAkka’s Quality of Mutants

Table 4 shows the numbers of easy-to-kill, Eas, duplicate, Dup, and subsumed mutants, Sub, for μAkka mutation operator groups. Eas, Dup, and Sub, respectively, are ratios of easy-to-kill, duplicate, and subsumed mutants of an operator to its covered mutants.

Ease-of-killing According to Table 4, row Eas, less than a fifth (17.4%) of mutants are easy-to-kill and non-quality, whereas more than four fifth (82.6%) are hard-to-kill and quality, with substantially different ease-of-killing for mutants of different operators. Configuration mutants are the most (33.4%) easy-to-kill and PATH the least (8.7%), excluding Cluster.

Finding 3: \( \frac{3}{4} \) of μAkka mutants are hard-to-kill and quality.

Duplicity According to row Dup, more than a seventh (14.6%) of mutants are duplicate and non-quality, whereas about six seventh (85.4%) are unique and quality, with substantially different duplicity for mutants of different operators. PATH mutants are the most (24.6%) duplicate and EXCEPTION the least (6.4%), excluding Cluster.

Finding 4: \( \frac{3}{4} \) of μAkka mutants are unique and quality.

Subsumption According to row Sub, less than a quarter (24.2%) of mutants are subsumed and non-quality, whereas more than three quarters (75.8%) are subsuming and quality, with substantially different subsumption for mutants of different operators. PATH mutants are the most (29.9%) subsumed and RACE the least (8.0%), excluding Cluster.

Finding 5: \( \frac{3}{4} \) of μAkka mutants are subsuming and quality.

Altogether The quality of μAkka mutants differ substantially for its different operators. RACE mutants are the highest quality as the second least easy-to-kill, third least duplicate, and the second least subsumed, whereas CONFIGURATION mutants are the lowest quality as the seventh easy-to-kill, most duplicate, and subsumed.

Finding 6: Race, Interaction, Cluster, Exception, Life, Communication, PATH, and Config are highest to lowest μAkka mutants.

Table 3: Mutant generation, Gen, and coverage, Cov, for different mutation operator groups.

Table 5 shows the generation, coverage, and the quality of PIT mutants of our applications. Due to compilation and execution issues in PIT, we include six applications ditto, lms, wot, flower, parc, and fuse that we used previously. Generation, coverage, and

5.5 μAkka’s Test Effectiveness

Table 4 shows the traditional mutation scores for tests of different applications. According to Table 4, tests are effective in killing only a sixth (16.6%) of μAkka mutants, and leave about five sixth (83.3%) alive. In addition, Finding 2 says that the tests are ineffective in covering more than half of μAkka mutants.

Finding 7: Tests are ineffective not only in covering \( \frac{1}{2} \) of μAkka mutants but also in killing \( \frac{5}{6} \).

5.6 PIT’s Mutant Quality and Test Effectiveness
quality of $\mu$Akka mutants for these six applications can be easily calculated using Table 4.

<table>
<thead>
<tr>
<th>Id</th>
<th>PIT Mutations</th>
<th>Mutation Score</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cov</td>
<td>Eas</td>
</tr>
<tr>
<td>ditto</td>
<td>17,652</td>
<td>11,108</td>
</tr>
<tr>
<td>lms</td>
<td>2,436</td>
<td>915</td>
</tr>
<tr>
<td>wot</td>
<td>1,969</td>
<td>1,087</td>
</tr>
<tr>
<td>flower</td>
<td>2,565</td>
<td>470</td>
</tr>
<tr>
<td>parc</td>
<td>1,262</td>
<td>1,232</td>
</tr>
<tr>
<td>fuse</td>
<td>293</td>
<td>106</td>
</tr>
<tr>
<td>Total</td>
<td>26,177</td>
<td>14,909</td>
</tr>
</tbody>
</table>

Table 4: Mutant ease-of-killing, Eas, duplicity, Dup, and subsumption, Sub, for different mutation operator groups.

**Effectiveness of tests** Table 5 shows the traditional mutation scores for tests of our six applications. According to Table 5, tests are effective in killing more than a third (37.8%) of PIT mutants, which is 2.3x the 16.8% test effectiveness for $\mu$Akka mutants. In addition, Table 8 states that tests are 1.3x less effective in covering $\mu$Akka mutants than PIT.

**Finding 10:** Tests are 1.3x less effective in covering and 2.3x less effective in killing $\mu$Akka mutants than PIT.

5.7 Bug Coverage by PIT and Jagannath et al.’s

Figure 1 shows the coverages of non-logic and logic bugs, in our bug set $B$, by mutation operators in $\mu$Akka, Jagannath et al.’s [89], and PIT and the overlap of these coverages. The bug coverage of an operator is the ratio of the number of bugs that it can induce and cover to the number of all bugs in $B$. Similarly, the coverage overlap of an operator with another is the ratio of the number of bugs that the former covers to the latter. For example, $\mu$Akka INTERACTION covers about one seventh (13.5%) of bugs in $B$ and Jagannath et al.’s Messaging overlaps with INTERACTION in covering about one third (32.0%) of bugs that INTERACTION covers.

Figure 1: Bug coverage by $\mu$Akka, Jagannath et al.’s, and PIT.
In Figure 1, $\mu_{\text{Akka}}$ covers $\mathcal{B}$’s non-logic bugs and there is an overlap between bugs that $\mu_{\text{Akka}}$’s Communication type, Interaction, and Life Cycle cover and Jagannath et al.’s Messaging and Creation/Deletion. In total, Jagannath et al.’s cover about a third (30.7% = 31/101) of non-logic bugs that $\mu_{\text{Akka}}$ covers.

**Finding 11:** $\mu_{\text{Akka}}$ covers 3.3x more bugs than Jagannath et al.

For non-logic bugs that $\mu_{\text{Akka}}$ does not cover, PIT covers less than a fifth (18.5% = 12/65) of these bugs. Together, $\mu_{\text{Akka}}$ and PIT cover more than two third (68.1% = 113/166) of non-logic and logic bugs which is 1.1x the 60.8% coverage of $\mu_{\text{Akka}}$ alone and 9.5x the 7.2% coverage of PIT alone.

**Finding 12:** $\mu_{\text{Akka}}$ + PIT cover 1.1x more bugs than $\mu_{\text{Akka}}$.

6 IMPLICATIONS

**Predictive mutation testing for actor concurrency**

To improve efficiency, previous work [112, 126] uses characteristics of mutants as features to build and train models that can predict mutant killing without executing the mutants. For example, Mao et al. [112] uses characteristics such as the coverage of a mutant (numTestCover) and the operator group that generates the mutant (MutatorClass) for their 0.85 accurate predictions of mutant killing in 654 real-world Java applications with more than 4 million lines of code. While still non-existent, future prediction mutation testing tools for actor concurrency can use $\mu_{\text{Akka}}$ mutants, and their characteristics, such as mutation operator group, coverage, ease-of-killing, duplicity, and subsumption, to build and train models that predict the killing of actor concurrency mutants.

**N-selective and N%-sampling mutation testing for actor concurrency**

Similarly, to improve efficiency, previous work [111, 114] approximates mutation testing using a select set of mutation operators, that excludes N operators that generate the most/more mutants, while maintaining the mutation score. For example, Offutt and Rothermel [114] use 2- and 4-selective mutation testing, with 22 operators, in 10 Fortran applications and achieve mutant reductions of 24.0% and 41.4%. While still non-existent, future actor concurrency mutation testing can use $\mu_{\text{Akka}}$ operators, mutants, and statistics about their generation for N-selective testing. According to Table 3, a 4-selective testing that excludes the four operators in Communication type and Race groups, could potentially reduce our mutant numbers by 30.1%, if their exclusion maintains the mutation score. Similarly, previous work [83, 127] approximates mutation testing by sampling N% of mutants per criteria such mutation operator, method, or class. Future actor concurrency mutation testing can use $\mu_{\text{Akka}}$ operators, operator groups, and mutant quality as new guiding criteria for sampling.

**Actor concurrency testing**

To improve tests, previous work [73, 80] uses mutation testing to guide where to test and what to improve. For example, Fraser and Zeller [80] $\mu$TEST uses alive mutants to generate oracles and tests that kill 75% of all mutants in 10 Java libraries with 1,416 classes. Future actor concurrency testing can use $\mu_{\text{Akka}}$ to guide similar oracle and test generations.

7 THREATS TO VALIDITY

Some of our decisions during this work could be a threat to its validity. The bug set $\mathcal{B}$ that we use to understand Akka bugs and their causes may not be representative of all Akka bugs and could be a threat. However, the large number of bugs in $\mathcal{B}$ from both popular Stack Overflow and GitHub and the large number of Stack Overflow posts and GitHub commits that the previous work [58] uses to construct $\mathcal{B}$ could help mitigate this threat. The manual analysis that we use to understand the bugs, design and group mutation operators, and understand bug coverages can be a threat. To minimize this threat, we use well-known sorting techniques [79], with multiple sorters, that previous work proposes and often uses [49, 58, 59, 70, 125]. The metrics that we use to measure the mutant quality and test effectiveness can be another threat. To minimize this threat, we use well-known metrics that previous work proposes and often uses for mutant quality [16, 52, 75, 91, 94, 117, 119] and test effectiveness [52, 91, 95, 118, 120]. The applications that we use in our evaluations may not be a representative of all Akka applications and could be a threat. To minimize this threat, we select a random set of applications that are different in sizes, domains, number of stars, and developers.

8 RELATED WORK

**Concurrency**

The work by Jagannath et al. [89] is the closest to our work. We discuss this work and its semantic and bug coverage overlap with out work in detail in Sections 1, 4, and 5. Jagannath et al. [89] propose 12 mutation operators, in 3 groups, for ActorFoundry [57], that are based on individual experiences of authors and not a curated set of real bugs, are not implemented, are not evaluated, and are specific to the syntax and semantics of ActorFoundry and not applicable to today’s industrial-strength implementations of actor concurrency. Six out of our 32 mutation operators (18.8%) and the remaining twenty-six (81.2%) in Path, Configuration, Life Cycle, Interaction, Race, and Cluster groups are new. There are six operators that are unique to Jagannath et al.’s. Two Remove Constraint (RC) and Modify Constraint (MC) are inapplicable to Akka because Akka does not allow conditional constraints to disable the receiving of messages. For the remaining four, Modify Message Parameter (MMP), Reorder Message Parameter (RMP), Modify Creation Parameter (MCP), and Reorder Creation Parameter (RCP), there were no bugs in $\mathcal{B}$ that required a similar operator.

Previous work proposes ConMan [68] and CCMutator [98] to mutate multithreaded concurrency and its constructs, such as threads, locks, conditional variables, and atomic blocks, in languages like Java and C/C++. However, the fundamental differences between actor and multithreaded concurrency, the syntax and semantics of their constructs, and their bugs make multithreaded mutation testing inapplicable to actor concurrency.

**Languages and paradigms**

Previous work proposes mutation testing for different programming paradigms, such as functional [100], object- [109], aspect- [116], and declarative- oriented [24] programming, and different programming models, such web [124], mobile [106], and machine learning [88]. Previous work also proposes mutation testing for specification [66], modeling [78], and a broad range of programming languages, such as Java [110], JavaScript [113], C [71], C++ [15], C# [77], and Ruby [101]. However, none of these works design, implement, and evaluate mutation testing for actor concurrency using real bugs.
**Mutant quality** Previous work propose several techniques, such as subsuming [52, 96], dominator [97], disjoint [95], minimal [52], and surface [84] mutants to reduce non-quality trivial, redundant, and equivalent mutants. Others, quantify the mutant usefulness using the triviality, equivalence and dominance of the mutant and relate the usefulness to the program context of the mutant [92]. However, none evaluate the quality of actor mutants.

**Efficiency** Previous works propose Comutation [82] to select a small subset of multithreaded mutation operators with less number of mutants but the same test effectiveness and MutMut [81] for efficient execution of multithreaded mutants. However, selective and efficient mutation testing are outside the scope of this paper.

9 CONCLUSIONS AND FUTURE WORK

In this work, we propose \(\mu\text{Akka}\) for mutation testing of Akka actor concurrency. To design, implement, and evaluate \(\mu\text{Akka}\), we manually analyze a set of 186 real Akka bugs, design 32 mutation operators to induce these bugs, implement these operators in an Eclipse plugin, generate 11.7k mutants of 10 real applications, measure the quality of mutants and effectiveness of tests, generate 26.2k PIT mutants and compare with \(\mu\text{Akka}\), analyze the bug coverage and overlap between \(\mu\text{Akka}\), PIT, and actor operators in Jagannath et al.’s, and discuss a few implications of our findings. One avenue of future work is to use program context to predict and avoid generation of lower quality mutants [92]. Another avenue is predictive mutation testing for actor concurrency using machine learning.

DATA AVAILABILITY

All the data and tools that we use in this work are publicly available in our replication package [27]. These include \(\mu\text{Akka}\)’s mutation operators, their formal definitions, Akka API source code changes, and the source code of our Eclipse plugin.

ACKNOWLEDGEMENTS

We would like to thank anonymous reviewers for their valuable comments. Moradi Moghadam and Bagherzadeh were supported, in part, by MSGC and NSF grants 80NSSC20M0124 and CNS-22-13763. Khatchadourian was supported, in part, by NSF grant CCF-22-00343. Bagheri was supported, in part, by NSF grants 2124116 and 2139845.

REFERENCES

yosiz656b.


Mehdi Bagherzadeh and Raffi Khatchadourian. Going big: A large-scale study on what big data developers ask. In ESEC/FSE 2019


Sebastien Bardin, Michael Delahaye, Robin David, Nikolai Kosmatov, Mike Papadakis, Yves Le Traon, and Jean-Yves Marion. Sound and quasi-complete detectability of unsafe test requirements. In ICST ’15.


Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satis Chandra, and Erik Meijer. What it would take to use mutation testing in an industry - a study at a Citrix CFQ product. In ICSE-SEIP ’21.


Paul E. Black, Vadim Okun, and Yaacov Yesha. Mutation operators for specifica-

Duc Le, Mohammad Amin Alipour, Rahul Gopinath, and Alex Groce. Mutchek: An extensible tool for mutation testing of haskell programs. In ICST ’14.


Anna Derezinika and Karol Kowalski. Object-oriented mutation applied in common intermediate language programs originated from C. In ICST ’11.

Sally Fincher and Joshua Tenenberg. Making sense of card sorting data. In Search Perspectives ’18.

Brandon Hedden and Xinzhui Zhao. A comprehensive study on bugs in actor systems. In ICPP ’18.

Vilas Jagannath, Milos Gligoric, Steven Lauterburg, Darko Marinov, and Guha Garga. Mutation operators for actor systems. In ICS’TW ’10.

Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. TSE ’11, 37(5).


Brandon Hedden and Xinzhui Zhao. A comprehensive study on bugs in actor systems. In ICPP ’18.


[120] Mike Papadakis, Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults. In ICSE ’16.


Received 2023-02-02; accepted 2023-07-27