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

Mohsen Moradi Moghadam
Oakland University
USA
moradimoghadam@oakland.edu

Mehdi Bagherzadeh
Oakland University
USA
mbagherzadeh@oakland.edu

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0327-0/23/12…$15.00
https://doi.org/10.1145/3611643.3616362

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:
Mohsen Moradi Moghadam, Mehdi Bagherzadeh, Raffi Khatchadourian, and Hamid Bagheri. 2023. μAkka: Mutation Testing for Actor Concurrency in Akka using Real-World Bugs . In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616362

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 real-time [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 ***μAkka***, 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 ***μAkka***, 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 ***μAkka*** and PIT mutant quality and test effectiveness.
(7) manually analyze the bug coverage and overlap of ***μAkka***, 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*** : ***μAkka*** mutants are high quality: 2x harder to kill, 3x less duplicate, and 1.3x less subsumed than PIT.
- ***Test effectiveness*** : Tests are ineffective for ***μAkka*** mutants: 1.3x less effective in covering and 2.3x less effective in killing ***μAkka*** mutants than PIT.
- ***Bug coverage*** : ***μAkka*** bug coverage is high: 3.3x more bug coverage than Jagannath et al.'s. And ***μAkka*** bug coverage benefits from the addition of PIT: ***μAkka*** + PIT cover 1.1x more bugs than ***μAkka*** 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, *dispatch* to assign threads to the actor message processings, *scheduling* to schedule the actor 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 a*live*. 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*, *subsumed*, and *equivalent*. For a mutant m of an original application p and the set of tests $\mathscr{T}^m_{cover}$ and $\mathscr{T}^m_{kill}$ 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 $\mathscr{T}^m_{cover}$ kill it. Following previous work [117], we set k to 97.5%. A mutant m is trivial if all the tests in $\mathscr{T}^m_{cover}$ 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 $\mathscr{T}^{m}_{kill}$ is a subset of $\mathscr{T}^{m'}_{kill}$. 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 $\mathscr{T}^{m}_{cov}$ 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 $\mathscr{T}^{m}_{cov}$ can kill a trivial and easy-to-kill m, respectively, with no need for a new test; a non-empty subset of $\mathscr{T}^{m}_{cov}$ 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 $\mathscr{T}_{cov}$ that cover at least a mutant, the traditional mutation score is the ratio of the number of mutants that tests in $\mathscr{T}_{cov}$ 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 *μ**Akka*** mutation operators for these bugs, and measure the quality of mutants that *μ**Akka*** and PIT operators generate, the effectiveness of tests to identify these mutants, and the coverage of bugs by *μ**Akka*** and Jagannath et al.'s operators.

3.1 Akka Actor Bugs

For Akka bugs, we use a set $\mathscr{B}$ of 186 real bugs from a recent previous work [58]. $\mathscr{B}$ 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 $\mathscr{B}$, 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 $\mathscr{B}$ 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, *actorOf(String name)* and *actorOf()* are two Akka methods that, their invocations, create an actor with a given and random name, respectively. Changing these invocations to *actorOf(String name')* could emulate the cause in which the actor name is not unique, if *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 Change Name to the operator that changes *actorOf(String name)* and *actorOf()* to *actorOf(String name')*. For a Stack Overflow bug, we analyze its questions, answers, and comments to understand its cause and design its operator. Similarly, for a GitHub bug, we analyze its commit, messages, original and modified code snippets, pull requests, and issues.

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, Change Path groups together Change Name and 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 $\mathscr{B}$ in which the compilation fails because of using an undeclared variable. There are 21 (11.3%) compile-time bug in $\mathscr{B}$. 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 $\mathscr{B}$ 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 $\mathscr{B}$. In total, we design mutation operators to emulate the causes of 100 (53.8%) bugs in $\mathscr{B}$ 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 μ***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 μ***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 $\mathscr{B}$ bugs that Jagannath et al.'s [89] operators can induce and cover and their overlap with μ***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 Path, Configuration, Communication type, Life cycle, Exception, Interaction, Race, and 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, Actor System, Host, 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 $\mathscr{B}$ 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 *fallback* 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 fallback, 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, Fallback, Mailbox, Dispatch, Routing, and Deployment settings to induce bugs that are caused by using incorrect

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

Table 1: Mutation operators for actor concurrency in *μAkka*, their groups, and descriptions.

settings for these configurations. For example, there is a bug [8] in $\mathscr{B}$ 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 $\mathscr{B}$ 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 $\mathscr{B}$ 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 $\mathscr{B}$ 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 $\mathscr{B}$ 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 receive 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 $\mathscr{B}$ 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 message sent by reference to a message sent by value and vice 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 unsubscriptons 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 $\mathscr{B}$ 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

Id	***Project***	***LOC***	***Domain***	***Stars***
ditto	ditto [36]	261,951	internet of things (IoT)	390
lms	sunbird-lms [37]	78,633	learning management	29
wot	wot-servient [38]	52,019	web of things	23
flower	flower [39]	21,648	reactive microservices	518
comb	servicecomb [40]	15,527	transactional data	481
rhino	rhino [41]	13,882	web performance testing	16
parc	parallec [42]	13,209	asyncronous web	800
flink	flink-rpc [43]	8,326	web performance testing	19,600
fuse	fuse [44]	3,483	REST server	15
mony	money-transfer [45]	3,136	money transfer API	5
Total		446,444		

Table 2: Real-world Akka applications from GitHub.

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 *connectivity*, *internal*, *things*, *gateway*, and *base* subsystems of *ditto* and the *rpc* subsystem of *flink*, and not all their subsystems.

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. $\boldsymbol{Gen}_{\%}$ is the ratio of the number of mutants that an operator generates to the number of mutants that all operators generate. $\boldsymbol{Cov}_{\%}$ 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 $\boldsymbol{Gen}_{\%}$, for all applications in aggregate, the number of mutants that different mutation operator groups generate are substantially different, with Interaction

Id	***Mutation group***																***Total***	
	Path		***Config***		***Comm***		***Life***		***Excep***		***Inter***		***Race***		***Cluster***			
	Gen	***Cov***	***Gen***	***Cov***	***Gen***	***Cov***	***Gen***	***Cov***	***Gen***	***Cov***	***Gen***	***Cov***	***Gen***	***Cov***	***Gen***	***Cov***	***Gen***	***Cov***
ditto	117	64	27	15	993	410	1,626	580	71	48	2,161	802	724	240	1	1	5,720	2,160
lms	0	0	3	1	125	82	706	425	0	0	759	499	468	320	0	0	2,061	1327
wot	23	19	39	4	84	32	235	90	5	3	207	91	124	52	2	0	719	291
flower	13	7	7	7	30	9	72	30	5	4	73	26	28	4	0	0	228	87
comb	12	12	1	1	29	24	43	43	2	2	49	49	4	4	0	0	140	135
rhino	7	5	1	1	24	11	29	15	3	2	38	23	8	4	0	0	110	61
parc	17	10	3	1	296	82	410	236	8	8	450	228	220	112	0	0	1,404	677
flink	4	4	2	2	34	19	110	61	6	6	115	56	56	24	0	0	372	172
fuse	68	18	3	0	22	7	53	14	3	2	32	10	12	0	0	0	193	51
mony	15	12	1	1	98	38	272	24	4	4	304	23	140	4	0	0	834	106
Total	276	151	87	33	1,735	714	1,518	622	107	79	4,188	1,807	1,784	764	3	1	11,732	5,067
Gen$_{\%}$	2.4		0.8		14.8		30.3		1.0		35.7		15.3		0.1			
Cov$_{\%}$		54.8		38.0		41.2		42.7		73.9		43.2		42.9		33.4		43.2

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

alone generating more than a third (35.7%) of the mutants, which is the most, and Configuration generating the least (0.8%). This excludes the outlier Cluster that generates 0.1% of mutants with only 3 mutants. Four operator groups, Interaction, Life cycle, Race, and Communication type, that include 17 out of 32 (53.1%) μ***Akka*** operators, together generate the majority (96.1%) of mutants whereas the other four, Cluster, Configuration, Exception, and Path, generate only a small minority (3.9%).

Finding 1: $\sim\frac{1}{2}$ of μ***Akka*** operators generate $>\frac{9}{10}$ of mutants.

Coverage Similarly, according to row ***Cov***$_{\%}$, tests can cover less than half (43.2%) of μ***Akka*** mutants, and leave more than half (57.8%) uncovered, with substantially different coverage for mutants of different operators. Exception mutants are the most (73.9%) covered and Configuration the least (38.0%), excluding Cluster.

Finding 2: Tests are ineffective in covering $>\frac{1}{2}$ of μ***Akka*** mutants.

5.4 μ*Akka*'s Quality of Mutants

Table 4 shows the numbers of easy-to-kill, ***Eas***, duplicate, ***Dup***, and subsumed mutants, ***Sub***, for μ***Akka*** 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: $>\frac{4}{5}$ of μ***Akka*** 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: $\sim\frac{6}{7}$ of μ***Akka*** 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 μ***Akka*** mutants are subsuming and quality.

Altogether The quality of μ***Akka*** 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 μ***Akka*** mutants.

5.5 μ*Akka*'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 μ***Akka*** 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 μ***Akka*** mutants.

Finding 7: Tests are ineffective not only in covering $>\frac{1}{2}$ of μ***Akka*** mutants but also in killing $>\frac{5}{6}$.

5.6 PIT's Mutant Quality and Test Effectiveness

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

Id	*Mutation Group*																								*Total*			*Mutation Score*
	Path			*Config*			*Comm*			*Life*			*Excep*			*Inter*			*Race*			*Cluster*						
	Eas	*Dup*	*Sub*	*Eas*	*Dup*	*Sub*	*Eas*	*Dup*	*Sub*	*Eas*	*Dup*	*Sub*	*Eas*	*Dup*	*Sub*	*Eas*	*Dup*	*Sub*	*Eas*	*Dup*	*Sub*	*Eas*	*Dup*	*Sub*	*Eas*	*Dup*	*Sub*	
ditto	3	10	14	5	2	5	83	46	125	130	131	142	11	2	5	142	116	120	31	19	23	1	0	0	406	326	524	16.1
lms	0	0	0	1	0	0	61	0	0	124	74	70	0	0	0	81	29	32	40	25	25	0	0	0	307	128	287	17.8
wot	4	5	4	3	2	2	10	8	3	9	2	2	2	0	0	8	0	0	7	0	0	0	0	0	43	17	14	9.4
flower	0	2	1	2	3	2	2	3	5	1	12	11	1	1	1	2	7	7	0	0	1	0	0	0	8	28	34	24.6
comb	0	0	0	0	0	0	7	4	18	9	11	22	0	0	0	7	9	11	0	0	0	0	0	0	24	24	62	56.5
rhino	1	4	4	0	0	0	0	5	5	1	3	3	2	1	1	0	1	1	0	0	0	0	0		2	14	19	19.1
parc	0	5	9	0	0	0	19	20	32	8	51	70	1	0	3	5	53	50	0	22	12	0	0	0	34	151	218	20.4
flink	0	0	1	0	0	0	11	2	8	21	13	8	1	1	1	11	3	7	5	0	0	0	0		49	19	30	21.5
fuse	4	11	11	0	0	0	1	1	1	4	5	5	1	0	1	0	4	4	0	0	0	0	0	0	10	21	13	16.1
mony	1	0	1	0	0	0	2	5	9	1	4	4	1	0	2	4	2	5	0	0	0	0	0	0	9	11	23	5.2
Total	13	37	45	11	7	9	196	94	206	308	306	337	20	5	14	260	224	237	83	66	61	1	0	0	892	739	1,224	16.6
$\textit{Eas}_{\%}$	8.7			33.4			27.5			20.3			25.4			14.4			10.9			100.0			17.7			
$\textit{Dup}_{\%}$		24.6			21.3			13.2			20.2			6.4			12.4			8.7			0.0			14.6		
$\textit{Sub}_{\%}$			29.9			27.3			28.9			22.3			17.8			13.2			8.0			0.0			24.2	

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

quality of μ***Akka*** mutants for these six applications can be easily calculated using Table 4.

Id	*PIT Mutations*					*Mutation Score*
	Gen	*Cov*	*Eas*	*Dup*	*Sub*	
ditto	17,652	11,108	5,734	5,727	3,906	44.9
lms	2,436	915	250	265	164	16.8
wot	1,969	1,087	604	404	331	38.4
flower	2,565	470	101	182	155	8.6
parc	1,262	1,232	162	388	229	40.0
fuse	293	106	70	65	55	26.3
Total	26,177	14,909	6,921	7,031	4,840	37.8
$\textit{Cov}_{\%}$		57.0				
$\textit{Eas}_{\%}$			46.5			
$\textit{Dup}_{\%}$				47.2		
$\textit{Sub}_{\%}$					32.5	

Table 5: Mutant quality and test effectiveness in PIT.

Coverage According to Table 5, row $\textit{Cov}_{\%}$, tests cover more than half (57.0%) of PIT mutants, and leave the other half (43%) uncovered, which is a third more than (1.3x) the 44.5% coverage of μ***Akka*** mutants, for these six applications.

Finding 8: Tests are 1.3x less effective in covering μ***Akka*** than PIT mutants.

Ease-of-killing, duplicity, and subsumption According to row $\textit{Eas}_{\%}$, less than a half (46.5%) of PIT mutants are easy-to-kill, which is more than twice (2.64x) the 17.6% ease-of-killing for μ***Akka*** mutants. Similarly, according to row $\textit{Dup}_{\%}$, less than a half (47.2%) of PIT mutants are duplicates, which is thrice (3.2x) the 14.7% duplicity for μ***Akka*** mutants. Finally, according to row $\textit{Sub}_{\%}$, less than a half (32.5%) of PIT mutants are subsumed, which is a third more than (1.3x) the 23.8% subsumption for μ***Akka*** mutants.

Finding 9: μ***Akka*** mutants are higher quality than PIT: 2x harder to kill, 3x less duplicate, and 1.3x less subsumed.

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 μ***Akka*** mutants. In addition, Finding 8 says tests are 1.3x less effective in covering μ***Akka*** mutants.

Finding 10: Tests are 1.3x less effective in covering and 2.3x less effective in killing μ***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 $\mathscr{B}$, by mutation operators in μ***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 $\mathscr{B}$. 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, μ***Akka*** Interaction covers about one seventh (13.5%) of bugs in $\mathscr{B}$ and Jagannath et al.'s Messaging overlaps with Interaction in covering about one third (32.0%) of bugs that Interaction covers.

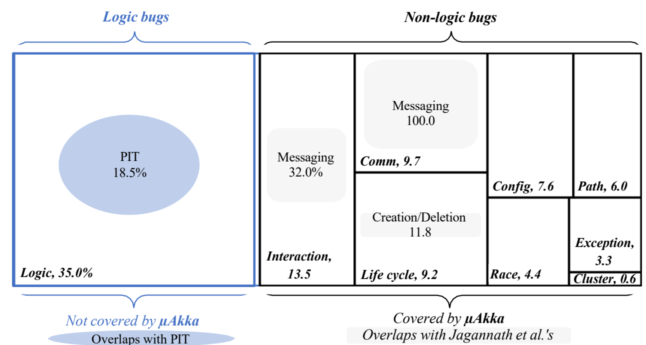

Figure 1: Bug coverage by μ*Akka*, Jagannath et al.'s, and PIT.

In Figure 1, μ***Akka*** covers $\mathscr{B}$'s non-logic bugs and there is an overlap between bugs that μ***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 μ***Akka*** covers.

Finding 11: μ***Akka*** covers 3.3x more bugs than Jagannath et al.

For non-logic bugs that μ***Akka*** does not cover, PIT covers less than a fifth (18.5%=12/65) of these bugs. Together, μ***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 μ***Akka*** alone and 9.5x the 7.2% coverage of PIT alone.

Finding 12: μ***Akka*** + PIT cover 1.1x more bugs than μ***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 μ***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 μ***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 μ***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] μTEST 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 μ***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 $\mathscr{B}$ 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 $\mathscr{B}$ 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 $\mathscr{B}$ 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 $\mathscr{B}$ 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 work proposes 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 ***μAkka*** for mutation testing of Akka actor concurrency. To design, implement, and evaluate ***μAkka***, 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 ***μAkka***, analyze the bug coverage and overlap between ***μAkka***, 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 ***μAkka***'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

[1] Stack Overflow. Actor name is not unique invalidactornameexception. https://www.stackoverflow.com/questions/11693562.

[2] Stack Overflow. Actor supervised by BackoffSupervisor loses stashed messages after restart. https://www.stackoverflow.com/questions/53933363.

[3] Stack Overflow. actorselection with relative path messages going to dead letters. https://www.stackoverflow.com/questions/23835157.

[4] Stack Overflow. Akka actor - sender points to dead letters. https://www.stackoverflow.com/questions/43396596.

[5] Stack Overflow. Akka actor cannot send back the message. https://www.stackoverflow.com/questions/28211255.

[6] Stack Overflow. Akka actor not terminating if an exception is thrown. https://stackoverflow.com/questions/6170227.

[7] Stack Overflow. Akka actors always times out waiting for future. https://www.stackoverflow.com/questions/36219778.

[8] GitHub. Akka .conf file configuration to .properties file. https://stackoverflow.com/questions/43517113.

[9] Stack Overflow. Akka context.parent unexpected value. https://www.stackoverflow.com/questions/19972524.

[10] Stack Overflow. Akka Dead Letters with Ask Pattern. https://www.stackoverflow.com/questions/25402349.

[11] Stack Overflow. Akka exception handling. https://stackoverflow.com/questions/14820489.

[12] Stack Overflow. Akka: waiting for multiple messages. https://www.stackoverflow.com/questions/20151944.

[13] Stack Overflow. akka.io dispatcher configuration exception. https://stackoverflow.com/questions/21686327/akka-io-dispatcher-configuration-exception.

[14] GitHub. Appmaster failed to recover. https://github.com/gearpump/gearpump/commit/02b234363e1fcb3f799e864205ccb05e8a53ee1e.

[15] Assessment of class mutation operators for C++ with the MuCPP mutation system. *Information and Software Technology '17*, 81.

[16] Chapter six - mutation testing advances: An analysis and survey. volume 112 of *Advances in Computers '19*.

[17] Stack Overflow. correctly terminate akka actors in scala. https://www.stackoverflow.com/questions/12324055.

[18] Stack Overflow. could not find implicit value for parameter system: akka.actor.ActorSystem. https://www.stackoverflow.com/questions/35202570.

[19] Stack Overflow. Dead-letter in akka scala actors. https://www.stackoverflow.com/questions/23327675.

[20] Stack Overflow. Dead Letters using Akka to create a message ring. https://www.stackoverflow.com/questions/43785208.

[21] GitHub. Delay datawriters initialization after warmup. https://github.com/gatling/gatling/commit/ba139ec991fb05a282b07ddbf20287d83ca32ce0.

[22] GitHub. error log when launching local. https://github.com/gearpump/gearpump/commit/8065694e5b7006953eeaa2cf21f4cbe8d02fd652.

[23] GitHub. fix occasional deathpactexception in httphostconnector. https://github.com/spray/spray/commit/e34da115fa43d4d46db0e7ae06eea7fbcbc4fdfd.

[24] Mutating database queries. *Information and Software Technology '7*, 49(4).

[25] Stack Overflow. Outofmemoryerror using akka actors. https://www.stackoverflow.com/questions/31995994.

[26] Stack Overflow. Parent actor doesn't receive termination message when child is stopped with poisonpill. https://stackoverflow.com/questions/49736277.

[27] Replication package. https://mbagherz.bitbucket.io/lab-correct-software/data/muAkka/.

[28] Stack Overflow. Scala Akka Actor - Dead Letters encountered. https://www.stackoverflow.com/questions/53200356.

[29] Stack Overflow. Send message to actor after restart from Supervisor. https://www.stackoverflow.com/questions/48446194.

[30] Stack Overflow. Setting bounded mailbox for an actor that uses stashing. https://stackoverflow.com/questions/20419990.

[31] GitHub. small fix for Leader role handover. https://github.com/rkrzewski/akka-cluster-etcd/commit/9ca03223eea1a989927fcb57fd024e4741515d33.

[32] Stack Overflow. Spray Dead Letter msg. https://www.stackoverflow.com/questions/30740132.

[33] GitHub. The restart application REST api returns false. https://github.com/gearpump/gearpump/commit/c6ab9f9f07b9a81f976e6e5d7e5389d18f68b0b7.

[34] GitHub. Too fast messaging. https://github.com/codingteam/horta-hell/commit/f7b104e57deb399044304ebf667b5a708d579d27.

[35] Akka actors in Spark. https://issues.apache.org/jira/browse/SPARK-5293, 2015.

[36] GitHub. https://github.com/eclipse/ditto, July 2022.

[37] GitHub. https://github.com/project-sunbird/sunbird-lms-service, July 2022.

[38] GitHub. https://github.com/sane-city/wot-servient, July 2022.

[39] GitHub. https://github.com/zhihuili/flower, July 2022.

[40] GitHub. https://github.com/apache/servicecomb-saga-actuator, July 2022.

[41] GitHub. https://github.com/ryos-io/Rhino, July 2022.

[42] GitHub. https://github.com/eBay/parallec, July 2022.

[43] GitHub. https://github.com/apache/flink/tree/master/flink-rpc, July 2022.

[44] GitHub. https://github.com/gibffe/fuse, July 2022.

[45] GitHub. https://github.com/kiamesdavies/money-transfer, July 2022.

[46] Allen T. Acree, Timothy A. Budd, Richard A. DeMillo, Richard J. Lipton, and Frederick G Sayward. Mutation analysis. Technical Report GIT-ICS-79/08, Georgia Institute of Technology, 1979.

[47] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. 1986.

[48] Gul Agha and Carl Hewitt. Concurrent programming using actors: Exploiting large-scale parallelism. In *Foundations of Software Technology and Theoretical Computer Science '85*.

[49] Syed Ahmed and Mehdi Bagherzadeh. What do concurrency developers ask about? a large-scale study using stack overflow. In *ESEM '18*.

[50] Akka Actor Reference Config File. https://github.com/akka/akka/blob/master/akka-actor/src/main/resources/reference.conf, June 2022.

[51] Akka Typed. https://doc.akka.io/docs/akka/current/typed/index.html, June 2022.

[52] Paul Ammann, Marcio Eduardo Delamaro, and Jeff Offutt. Establishing theoretical minimal sets of mutants. In *ICST '14*.

[53] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. 2008.

[54] Anonymous. Number of Akka, Erlang, and Orleans projects in GitHub, 2022. Accessed on 2022-08-04, Akka: https://tinyurl.com/yshr4ypn, Erlang: https://tinyurl.com/yz9zz25b, and Orleans: https://tinyurl.com/4jj9czry.

[55] Anonymous. Number of akka, erlang, and orleans questions and answers in Stack Overflow, 2022. Accessed on 2022-08-04. URL: https://tinyurl.com/ycxjjbb6.

[56] Joe Armstrong. *Programming Erlang: Software for a Concurrent World.* 2007.

[57] M. Astley. The actor foundry: A Java-based actor programming environment. Open Systems Laboratory. University of Illinois at Urbana-Champaign, 1999.

[58] Mehdi Bagherzadeh, Nicholas Fireman, Anas Shawesh, and Raffi Khatchadourian. Actor concurrency bugs: A comprehensive study on symptoms, root causes, api usages, and differences. *Proc. ACM Program. Lang.*, 4(OOPSLA).

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

[60] Mehdi Bagherzadeh and Hridesh Rajan. Order types: Static reasoning about message races in asynchronous message passing concurrency. In *AGERE '17*.

[61] Mehdi Bagherzadeh and Hridesh Rajan. Panini: A concurrent programming model for solving pervasive and oblivious interference. In *Proceedings of the 14th International Conference on Modularity*, MODULARITY 2015, pages 93–108, New York, NY, USA, 2015. ACM. **Best Papers**. URL: http://doi.acm.org/10.1145/2724525.2724568, doi:10.1145/2724525.2724568.

[62] Sebastien Bardin, Mickael Delahaye, Robin David, Nikolai Kosmatov, Mike Papadakis, Yves Le Traon, and Jean-Yves Marion. Sound and quasi-complete detection of infeasible test requirements. In *ICST '15*.

[63] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. Deflaker: Automatically detecting flaky tests. In *ICSE '18*.

[64] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. What it would take to use mutation testing in industry - a study at facebook. In *ICSE-SEIP '21*.

[65] Phil Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. Orleans: Distributed virtual actors for programmability and scalability. Technical Report MSR-TR-2014-41, 2014.

[66] Paul E. Black, Vadim Okun, and Yaacov Yesha. Mutation operators for specifications. In *ASE '00*.

[67] Robert L. Bocchino, Edward Gamble, Kim P. Gostelow, and Raphael R. Some. Spot: A programming language for verified flight software. *Ada Lett. 14*, 34(3).

[68] Jeremy S. Bradbury, James R. Cordy, and Juergen Dingel. Mutation operators for concurrent Java (J2SE 5.0). In *Mutation '06*.

[69] Timothy A. Budd and Dana Angluin. Two notions of correctness and their relation to testing. *Acta Inf. '82*, 18(1).

[70] Tatiana Castro Vélez, Raffi Khatchadourian, Mehdi Bagherzadeh, and Anita Raja. Challenges in migrating imperative Deep Learning programs to graph execution: An empirical study. In *International Conference on Mining Software Repositories*, MSR '22, pages 469–481. IEEE/ACM, ACM, May 2022. arXiv:2201.09953, doi:10.1145/3524842.3528455.

[71] Thierry Titcheu Chekam, Mike Papadakis, Yves Le Traon, and Mark Harman. An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In *ICSE '17*.

[72] Henry Coles. http://pitest.org/, 2023.

[73] Meenu Dave and Rashmi Agrawal. Mutation testing and test data generation approaches: A review. In *Smart Trends in Information Technology and Computer Communications '16*.

[74] M. E. Delamaro. *Proteum–A Test Environment Based on the Mutation Analysis.* PhD thesis, Masters thesis, University of Sao Paulo, Brazil, 1993.

[75] Pedro Delgado-Pérez, Sergio Segura, and Inmaculada Medina-Bulo. Assessment of C++ object-oriented mutation operators: A selective mutation approach. *Software Testing, Verification and Reliability '17*, 27.

[76] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer '78*, 11(4).

[77] Anna Derezinska and Karol Kowalski. Object-oriented mutation applied in common intermediate language programs originated from C. In *ICST '11*.

[78] Xavier Devroey, Gilles Perrouin, Mike Papadakis, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. Featured model-based mutation analysis. In *ICSE '16*.

[79] Sally Fincher and Josh Tenenberg. Making sense of card sorting data. *Expert Systems '05*, 22(3).

[80] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. In *ISSTA '10*.

[81] Milos Gligoric, Vilas Jagannath, and Darko Marinov. Mutmut: Efficient exploration for mutation testing of multithreaded code. In *ICST '10*.

[82] Milos Gligoric, Lingming Zhang, Cristiano Pereira, and Gilles Pokam. Selective mutation testing for concurrent code. In *ISSTA '13*.

[83] Rahul Gopinath, Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. How hard does mutation analysis have to be, anyway? In *ISSRE '15*.

[84] Rahul Gopinath, Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. Measuring effectiveness of mutant sets. In *ICSTW '16*.

[85] Rahul Gopinath, Philipp Görz, and Alex Groce. Mutation analysis: Answering the fuzzing challenge. *arXiv preprint arXiv:2201.11303*, 2022.

[86] Bernhard J. M. Grün, David Schuler, and Andreas Zeller. The impact of equivalent mutants. In *ICSTW '09*.

[87] Brandon Hedden and Xinghui Zhao. A comprehensive study on bugs in actor systems. In *ICPP '18*.

[88] Nargiz Humbatova, Gunel Jahangirova, and Paolo Tonella. Deepcrime: Mutation testing of deep learning systems based on real faults. In *ISSTA '21*.

[89] Vilas Jagannath, Milos Gligoric, Steven Lauterburg, Darko Marinov, and Gul Agha. Mutation operators for actor systems. In *ICSTW '10*.

[90] Java Akka API. https://doc.akka.io/japi/akka/current/, June 2022.

[91] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *TSE '11*, 37(5).

[92] René Just, Bob Kurtz, and Paul Ammann. Inferring mutant utility from program context. In *ISSTA '17*.

[93] Rene Just, Franz Schweiggert, and Gregory M. Kapfhammer. Major: An efficient and extensible tool for mutation analysis in a java compiler. In *ASE '11*.

[94] Samuel J. Kaufman, Ryan Featherman, Justin Alvin, Bob Kurtz, Paul Ammann, and René Just. Prioritizing mutants to guide mutation testing. In *ICSE '22*.

[95] Marinos Kintis, Mike Papadakis, and Nicos Malevris. Evaluating mutation testing alternatives: A collateral experiment. In *APSEC '10*.

[96] Bob Kurtz, Paul Ammann, Marcio E. Delamaro, Jeff Offutt, and Lin Deng. Mutant subsumption graphs. In *ICSTW '14*.

[97] Bob Kurtz, Paul Ammann, Jeff Offutt, and Mariet Kurtz. Are we there yet? how redundant and equivalent mutants affect determination of test completeness. In *ICSTW '16*.

[98] Markus Kusano and Chao Wang. Ccmutator: A mutation generator for concurrency constructs in multithreaded C/C++ applications. In *ASE '13*.

[99] Thomas Laurent, Mike Papadakis, Marinos Kintis, Christopher Henard, Yves Le Traon, and Anthony Ventresque. Assessing and improving the mutation testing practice of pit. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 430–435, 2017. doi:10.1109/ICST.2017.47.

[100] Duc Le, Mohammad Amin Alipour, Rahul Gopinath, and Alex Groce. Mucheck: An extensible tool for mutation testing of haskell programs. In *ISSTA '14*.

[101] Nan Li, Michael West, Anthony Escalona, and Vinicius H. S. Durelli. Mutation testing in practice using ruby. In *ICSTW '15*.

[102] Lightbend. Customer case studies. https://www.lightbend.com/case-studies#filter:akka, 2019.

[103] Lightbend. 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, 2020.

[104] Lightbend. 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, 2020.

[105] Lightbend. Akka. https://www.lightbend.com/akka-platform, 2022.

[106] Mario Linares-Vásquez, Gabriele Bavota, Michele Tufano, Kevin Moran, Massimiliano Di Penta, Christopher Vendome, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. Enabling mutation testing for android apps. In *ESEC/FSE '17*.

[107] Yuheng Long, Mehdi Bagherzadeh, Eric Lin, Ganesha Upadhyaya, and Hridesh Rajan. On ordering problems in message passing software. In *Proceedings of the 15th International Conference on Modularity*, MODULARITY 2016, pages 54–65, New York, NY, USA, 2016. ACM. URL: http://doi.acm.org/10.1145/2889443.2889444, doi:10.1145/2889443.2889444.

[108] Carmen Torres Lopez, Stefan Marr, Elisa Gonzalez Boix, and Hanspeter Mössenböck. A study of concurrency bugs and advanced development support for actor-based programs. In *Programming with Actors - State-of-the-Art and Research Perspectives '18*.

[109] Yu-Seung Ma, Yong-Rae Kwon, and Jeff Offutt. Inter-class mutation operators for java. In *ISSRE '02*.

[110] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. Mujava: An automated class mutation system: Research articles. *Softw. Test. Verif. Reliab.*, 15(2):97–133, jun 2005.

[111] José Carlos Maldonado, Ellen Francine Barbosa, Auri Marcelo Rizzo Vincenzi, and Márcio Eduardo Delamaro. *Evaluating N-Selective Mutation for C Programs: Unit and Integration Testing.*

[112] Dongyu Mao, Lingchao Chen, and Lingming Zhang. An extensive study on cross-project predictive mutation testing. In *ICST '19*.

[113] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. Efficient javascript mutation testing. In *ICST '13*.

[114] A. Jefferson Offutt, Gregg Rothermel, and Christian Zapf. An experimental evaluation of selective mutation. In *ICSE '93*.

[115] A. Jefferson Offutt and Ronald H. Untch. *Mutation 2000: Uniting the Orthogonal.*

[116] Elmahdi Omar and Sudipto Ghosh. An exploratory study of higher order mutation testing in aspect-oriented programming. In *ISSRE '12*.

[117] Mike Papadakis, Thierry Titcheu Chekam, and Yves Le Traon. Mutant quality indicators. In *ICSTW '18*.

[118] Mike Papadakis, Christopher Henard, Mark Harman, Yue Jia, and Yves Le Traon. Threats to the validity of mutation-based test assessment. In *ISSTA '16*.

[119] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *ICSE '15*.

[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 '18*.

[121] Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. A survey of flaky tests. *ACM Trans. Softw. Eng. Methodol. '21*, 31(1).

[122] Goran Petrović and Marko Ivanković. State of mutation testing at google. In *ICSE-SEIP '18*.

[123] Goran Petrovic, Marko Ivankovic, Gordon Fraser, and Rene Just. Practical mutation testing at scale: A view from google. *TSE '21*.

[124] Upsorn Praphamontripong, Jeff Offutt, Lin Deng, and JingJing Gu. An experimental evaluation of web mutation operators. In *ICSTW '16*.

[125] Yiming Tang, Raffi Khatchadourian, Mehdi Bagherzadeh, Rhia Singh, Ajani Stewart, and Anita Raja. An empirical study of refactorings and technical debt in Machine Learning systems. In *International Conference on Software Engineering*, ICSE '21, pages 238–250. IEEE/ACM, IEEE, May 2021. doi:10.1109/ICSE43902.2021.00033.

[126] Jie Zhang, Ziyi Wang, Lingming Zhang, Dan Hao, Lei Zang, Shiyang Cheng, and Lu Zhang. Predictive mutation testing. In *ISSTA '16*.

[127] Lingming Zhang, Milos Gligoric, Darko Marinov, and Sarfraz Khurshid. Operator-based and random mutant selection: Better together. In *ASE '13*.

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