A Delta-Debugging Approach to Assessing the Resilience of Actor Programs through Run-time Test Perturbations

Jonas De Bleser
jonas.de.bleser@vub.be
Vrije Universiteit Brussel
Brussels, Belgium

Dario Di Nucci
d.dinucci@uvt.nl
Tilburg University - JADS
’s-Hertogenbosch, The Netherlands

Coen De Roover
coen.de.roover@vub.be
Vrije Universiteit Brussel
Brussels, Belgium

ABSTRACT

Among distributed applications, the actor model is increasingly prevalent. This programming model organises applications into fully-isolated processes that communicate through asynchronous messaging. Supported by frameworks such as Akka and Orleans, it is believed to facilitate realising responsive, elastic and resilient distributed applications.

While these frameworks do provide abstractions for implementing resilience, it remains up to developers to use them correctly and to test that their implementation recovers from anticipated failures. As manually exploring the reaction to every possible failure scenario is infeasible, there is a need for automated means of testing the resilience of a distributed application.

We present the first automated approach to testing the resilience of actor programs. Our approach perturbs the execution of existing test cases and leverages delta debugging to explore all failure scenarios more efficiently. Moreover, we present a further optimisation that uses causality to prune away redundant perturbations and speed up the exploration. However, its effectiveness is sensitive to the program’s organisation and the actual location of the fault. Our experimental evaluation shows that our approach can speed up resilience testing by four times compared to random exploration.

CCS CONCEPTS

• **Computer systems organization** → **Reliability**; **Fault-tolerant network topologies**; • **Software and its engineering** → **Software testing and debugging**.

KEYWORDS

Resilience Testing, Delta Debugging, Fault Injection, Test Amplification

ACM Reference Format:

Jonas De Bleser, Dario Di Nucci, and Coen De Roover. 2020. A Delta-Debugging Approach to Assessing the Resilience of Actor Programs through Run-time Test Perturbations. In *Proceedings of ACM Conference (Conference’17).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Conference’17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The actor model [2, 26], which advocates the use of fully-isolated processes that communicate through asynchronous messaging, is increasingly popular among distributed systems. Originally embodied by programming languages such as Erlang and Elixir, it is now also supported by industrial-strength frameworks such as Akka[1] for the JVM or Orleans[2] for the .NET runtime.

Akka in particular has enjoyed adoption by large organisations such as Twitter and Amazon [37], as well as academic attention in the form of books on distributed systems [31, 41] and dedicated research [28, 45–47]. Besides elementary abstractions for defining actors and their communication, the Akka framework also facilitates the implementation of resilience against anticipated infrastructural failures (*e.g.,* network disconnections or node crashes). For instance, it provides support for guaranteed message delivery and for rebalancing actors across the nodes of a cluster.

Nevertheless, developers still need to (i) anticipate failure scenarios in their designs (*e.g.,* slow or lost messages), (ii) decide upon the corresponding resilience tactic (*e.g.,* at-least-once delivery mechanisms), and (iii) account correctly for all their implications (*e.g.,* process messages idempotently). An empirical study by Gao et al. [17] confirms that there are ample of opportunities for oversights and mistakes.

Despite the need for resilience testing, progress has been slow. The few techniques proposed in the literature for automated resilience testing all perturb a system’s execution by injecting faults at run time. All need to cope with the problem of exploring a large space of possible failure scenarios. The number of perturbations and perturbation targets to consider when generating failure scenarios is prohibitively large. Existing techniques explore failure scenarios either (i) randomly [29], (ii) by means of developer-provided specifications [25], (iii) heuristically [21], or (iv) by means of backward reasoning from a fault-sensitive outcome [4].

In this paper, we present an approach to resilience testing that combines test amplification [12] with delta debugging [49]. The former improves existing test cases by injecting faults during their execution, while the latter efficiently decides which faults to inject. In contrast to many approaches [6, 10, 29, 51] that follow the Chaos Engineering methodology, our approach also aims to be used during development as this poses no risk of service outages and data loss. Instead of relying on failure specifications [25], exploration heuristics [21], or prohibitively expensive reasoning [4], our approach uses the domain-specific information captured by developers in existing tests. In particular, our goal is to improve the

[1] https://akka.io
[2] https://dotnet.github.io/orleans

current testing strategy of a system by determining whether tests also keep succeeding under adverse conditions.

We have several reasons to believe that the combination of test amplification and delta debugging can expose resilience issues: (i) a significant amount of time is spent on software testing [40, 43] and tests are therefore likely to capture domain-specific information; (ii) developers tend to test the most important features (*i.e.,* "happy paths") [8, 30] first due to timing and budget constraints [7, 18]; (iii) previous work [48] found that the majority of catastrophic failures could have been prevented by performing simple testing on error handling code; and (iv) that many distributed system failures are caused by the untimely arrival of a single event [33].

To summarize, this paper makes the following contributions:

- The design of an automated resilience testing approach which combines test amplification with delta debugging to identify shortcomings in the implementation of resilience tactics in actor-based applications through perturbation of their test executions.
- The realization of this approach in a tool called CHAOKKA[3]. It automatically identifies mistakes in the implementation of resilience against actor restarts and message delivery failures in actor systems implemented with AKKA.
- An experimental evaluation of three exploration techniques: RT-R, RT-DD, and RT-DD-O. In particular, we show that the delta-debugging variants RT-DD and RT-DD-O consistently find resilience mistakes up to four times faster compared to the random exploration of RT-R.

The remainder of the paper is structured as follows. Section 2 introduces the actor model and the resilience tactics as supported by AKKA. In Section 3, we present our resilience testing approach and its realisation in CHAOKKA. The exploration strategies it can be configured with are discussed in Section 4. We evaluate the tool in Section 5, while we discuss the current limitations and challenges in Section 6. Finally, we discuss related work in Section 7.

2 BACKGROUND

The SCALA[4] ecosystem features AKKA, a modern implementation of the actor model [2, 26] where actors communicate through asynchronous message sending, rather than shared state. Listing 1 illustrates how developers can render an AKKA program resilient against delivery failures of message `CountCommand`, even across cluster migration restarts of the `GuaranteedDeliveryActor` actor. Listing 2 depicts the corresponding test case which uses SCALATEST[5] , the most popular SCALA testing framework [13].

2.1 Actors in AKKA

Actors in AKKA have local state, a message handler, and a mailbox in which messages are queued. Actors can (i) update their local state, (ii) change their message handler, (iii) send messages to other actors, and (iv) create new actors.

```
1  import akka.actor.{Actor, ActorRef}
2  import akka.persistence.{PersistentActor, AtLeastOnceDelivery}
3
4  trait Event
5  case class Plus(amount: Int)
6  case class PlusEvent(amount : Int) extends Event
7  case class CountCommand(id : Long, amount : Int)
8  case class ConfirmEvent(id : Long) extends Event
9  case class Confirm(id : Long)
10
11 class GuaranteedDeliveryActor(ref: ActorRef)
12   extends PersistentActor with AtLeastOnceDelivery {
13
14     override def receiveCommand: Receive = {
15         case Plus(amount) =>
16           persist(PlusEvent(amount))(updateState)
17         case Confirm(id) =>
18           persist(ConfirmEvent(id))(updateState)
19     }
20
21     override def receiveRecover: Receive = {
22         case e : Event => updateState(e)
23     }
24
25     def updateState(e: Event): Any = e match {
26         case PlusEvent(amount) =>
27           deliver(ref.path)(id => CountCommand(id, amount))
28         case ConfirmEvent(id) =>
29           confirmDelivery(id)
30     }
31
32     override def persistenceId: String = "actor-1"
33 }
34
35 class Accumulator extends Actor {
36     var count: Int = 0
37
38     override def receive: Receive = {
39         case CountCommand(id: Long, amount: Int) =>
40             count = count + amount
41             sender() ! Confirm(id)
42         case "result" =>
43             sender() ! count
44     }
45 }
```

Listing 1: Motivating example.

Class `Accumulator` on lines 35–45 implements an actor and defines its message handler as a partial function returned by the overriding method `receive`. Each `case` in the handler determines how a certain kind of message sent to the actor should be processed. Messages are removed one-by-one from the mailbox and processed by automatically applying the message handler. For instance, the `case` on lines 42-43 matches `"result"` messages where the handler will use the `!` operator to send the current value of `count` to the sender of the message.

Messages are sent to location-transparent addresses of type `ActorRef` (*e.g.,* as returned by `sender()` on line 43); whether the actor behind this address is local or remote is transparent to the sender. The actor's physical location can be changed through configuration without altering the code. This location transparency enables our tester to simulate a completely distributed deployment on a single JVM.

[3] https://github.com/jonas-db/chaokka
[4] https://www.scala-lang.org
[5] http://www.scalatest.org

2.2 Persistent Actors in Akka

Persistent actors persist their state according to the principle of Event Sourcing [15]. Persisted events are replayed whenever an actor is restarted after a failure or a cluster migration. A persistent actor is implemented by (i) inheriting from the trait `PersistentActor` (line 12), (ii) overriding `receiveCommand` to define the message handler (lines 14–18), (iii) overriding `receiveRecover` to define the handler that replays persisted events (lines 21–23), and (iv) defining a `persistenceId` to uniquely identify the entity in a journal where events are written to and read from (line 32). To persist an event, a developer must call `persist` (line 16) with the event to be persisted and a callback (*i.e.,* `updateState` on lines 25–30) to be executed whenever the given event has been persisted asynchronously.

2.3 Message Delivery Semantics in Akka

Akka uses at-most-once message delivery semantics as default. As a consequence, partial network failures can cause messages to be lost and therefore might never arrive at the receiver. To get stronger guarantees, Akka provides at-least-once message delivery semantics. Line 27 calls method `deliver` provided by the `AtLeastOnceDelivery` trait with the destination address of actor `ref` and a single-parameter callback. The callback is called with a unique identifier generated by the framework and returns the message `CountCommand` that has to be sent. The framework will periodically resend the message until an acknowledgement with that identifier has been registered. The actor that has to confirm a message sends a `Confirm` message back with the identifier `id` (line 41). Then, the handler of the receiving actor for `Confirm` messages calls method `confirmDelivery` to confirm the delivery (line 29).

2.4 Test Cases in ScalaTest

Listing 2 shows a test case written in ScalaTest. The test case starts by instantiating both actors on lines 2–4. Next, ten `Plus` messages are sent to `GuaranteedDeliveryActor` on line 6. After waiting for 2 seconds, the test sends a message `"result"` to `Accumulator` to retrieve the total sum. `expectMsg` will wait for the reply and then verify the result with an assertion.

```
1  "Accumulator" must "correctly accumulate numbers" in {
2      val a = system.actorOf(Props[Accumulator], name="A")
3      val gda = Props(new GuaranteedDeliveryActor(a))
4      val actor = system.actorOf(gda, name="GDA")
5
6      for (i <- 1 to 10) { actor ! Plus(i) }
7      Thread.sleep(2000)
8      a ! "result"
9
10     expectMsg((1 to 10).sum)
11 }
```

Listing 2: Test case for the motivating example.

2.5 Resilience Tactic Issues in Akka

As mentioned before, developers need to be aware of many different aspects to achieve a resilient system. In this paper, we focus on two issues that are related to duplicated messages and actor restarts.

Message Duplication. Implementing at-least-once message delivery can lead to two issues: (i) a message can arrive more than once (*i.e.,* due to a slow confirmation), and (ii) arrive out of order (*i.e.,* due to re-sending). We focus on the former issue as it is a known problem [24, 27] and the latter has been widely studied in the domain of concurrency issues (*e.g.,* [47]). The code shown in Listing 1 is a simplified version of a real-world example posted on StackOverflow[6]. In that post, a developer experienced an issue about message duplication: *"The problem is that I get different results each time I run this program. The correct answer is 49995000 but I don't always get that [when sending integers 1 to 9999 to GuaranteedDeliveryActor]".* At first sight, the implementation (*i.e.,* Listing 1) and test case (*i.e.,* Listing 2) look correct and seem to work in most cases. However, the developer forgot to take into account that `Accumulator` may receive a message more than once. For example, because the confirmation message sent by `GuaranteedDeliveryActor` was not received in time by `Accumulator`. The solution is either to remember and to not process duplicated messages by maintaining state or to make sure that the processing is idempotent, which appears to be nontrivial [27].

Actor Restart. Many distributed application failures are due to services that fail to recover their state after a restart [33, 42]. Ensuring that an actor's state is preserved across restarts is prone to the following problems: (i) developers may forget to persist all of the necessary state or, specific to event sourcing, (ii) developers may not replay the events from the journal correctly. While the `GuaranteedDeliveryActor` is resilient to restarts, its communication partner `Accumulator` is not. Any restart will reset its internal state such that `count` becomes 0. This, however, will not become clear from running the test and leads

Both selecting and implementing resilience tactics is far from trivial. Yet, there is no automated tool support for detecting resilience shortcomings in a distributed application.

3 OVERVIEW OF THE APPROACH

We introduce our approach for resilience testing by presenting the overall process in Section 3.1, defining the trace format in Section 3.2, and explaining the perturbations in Section 3.3.

3.1 Resilience Testing Process

Our resilience testing process is implemented in Chaokka. The tool expects a system implemented with the Akka framework and a test suite written with ScalaTest. It leverages ScalaTest to discover test cases and a Scala Build Tool (SBT) plugin in combination with AspectJ to instrument, monitor, and perturb the execution of each test case.

In summary, the process comprises 5 steps:

1. Test Discovery: information about each test case is extracted (*e.g.,* name, duration, and outcome) using ScalaTest and stored for later access.

2. Test Execution: an execution trace for each test case is collected by instrumenting the system under test with AspectJ and executing each test case through ScalaTest (Section 3.2).

3. Trace Analysis: the execution trace is analyzed to determine all perturbations and their targets (Section 3.3).

[6] https://stackoverflow.com/questions/27592304/akka-persistence-with-confirmed-delivery-gives-inconsistent-results

4. Perturbation Exploration: the exploration strategy repeatedly decides which perturbations are applied during each subsequent execution of the test case (Section 4).

5. Report: a resilience report which enumerates the found perturbations that cause a change in test outcome, as well as auxiliary information about the number of iterations, the duration, and general test information.

3.2 Test Case Execution Trace

The first step in our resilience testing process produces an *execution trace* for the program under test by instrumenting and executing one of its test cases. The trace is needed to capture all the actions that an actor can perform at runtime and enable the identification of potential perturbation targets. Formally, a trace t is a finite sequence of events $t = \langle e_1, e_2, \ldots, e_n \rangle$ where e_i is either a *Create*, *Send*, or *Turn* event. A *Create* event is logged whenever a new actor is created; a *Send* event whenever an asynchronous message h_{msg} is sent from l_{from} to k_{to}; and a *Turn* event whenever a message h_{msg} coming from l_{from} is processed by k_{to}. Note that a turn corresponds to the atomic application of the actor's message handler to a message from its mailbox. The location denotes an unique place in the system where the actor resides. Figure 1 depicts all captured information.

$$
\begin{aligned}
t &\in \mathit{Trace} = \langle e_1, e_2, \ldots, e_n \rangle \\
e &\in \mathit{Event} ::= \mathit{Create}(l_{\text{parent}}, k_{\text{child}}, b_{\text{persistent}}) \\
&\qquad \mid \ \mathit{Send}(l_{\text{from}}, k_{\text{to}}, h_{\text{msg}}, i_{\text{send}}, j_{\text{turn}}, b_{\text{alod}}) \\
&\qquad \mid \ \mathit{Turn}(l_{\text{from}}, k_{\text{to}}, h_{\text{msg}}, i_{\text{send}}, j_{\text{turn}}) \\
b &\in \mathit{Boolean} \text{ is a finite set of booleans} \\
h &\in \mathit{Hashcodes} \text{ is a finite set of hashcodes} \\
i, j &\in \mathit{Identifier} \text{ is a finite set of unique identifiers} \\
l, k &\in \mathit{Location} \text{ is a infinite set of actor locations}
\end{aligned}
$$

Figure 1: Execution trace events.

The identifiers i, j increase monotonically and uniquely identify each message that is sent and each turn in which a message is processed. For a *Send* event, i will be a new identifier, while j will be the identifier of the current turn. For a *Turn* event, j will be a new identifier, and i will be the identifier which was sent along with the message. In this way, every *Turn* event knows by which message it was caused, and every *Send* event knows from which turn it was sent. A higher identifier means that the event took place later in the trace. Based on these identifiers, we can detect the causality relation $\preceq \subseteq \mathit{Event} \times \mathit{Event}$ [14] between two trace events e and e' (*i.e.,* which turn sends a message and vice versa). The rules for this relation are shown in Figure 2. We leverage this relation in one of the exploration strategies discussed in Section 4.

Vector clocks [14] are usually employed to track this kind of relation for distributed systems. However, Akka's location-transparent actor references enable deployment reconfiguration in such a way that a single JVM suffices. Therefore, global identifiers suffice for our prototype implementation.

$$
\begin{aligned}
e &\preceq e' && \text{if they are the same event,} && (1)\\
e &\preceq e' && \text{if } e \text{ and } e' \text{ are turn events of the same actor} && (2)\\
& && \text{and } e \text{ happens before } e', \\
e &\preceq e' && \text{if } e \text{ is the send event of a message and} && (3)\\
& && e' \text{ is the turn event for the event } e, \text{ and} \\
e &\preceq e' && \text{if } e \preceq e'' \text{ and } e'' \preceq e' \ (\textit{i.e.,} \text{ transitivity}) && (4)
\end{aligned}
$$

Figure 2: Causality relation between trace events.

3.3 Perturbations

Our resilience testing process analyses a test execution trace to compute potential perturbation targets and repeatedly re-executes the test while perturbing the targets with their corresponding perturbation selected by the exploration strategy. To this end, every strategy generates a so-called *perturbation configuration* which is loaded by the tool on every test run. During test execution, every event is monitored, intercepted and perturbed when it conforms to a perturbation defined in the perturbation configuration. We introduce a perturbation for each resilience tactic issue (Section 2.5) to uncover defects in its implementation. We explain the rationale for each perturbation, summarized in Figure 3, as well as to which targets they are applied.

$$
\begin{aligned}
c &\in \mathit{Configuration} = \{p_1, p_2, \ldots, p_n\} \\
p &\in \mathit{Perturbation} ::= \mathit{Duplicate}(l_{\text{from}}, k_{\text{to}}, h_{\text{msg}}) \\
&\qquad \mid \ \mathit{Restart}(l_{\text{from}}, k_{\text{to}}, h_{\text{msg}}) \\
h &\in \mathit{Hashcodes} \text{ is a finite set of hashcodes} \\
l, k &\in \mathit{Location} \text{ is a infinite set of actor locations}
\end{aligned}
$$

Figure 3: Perturbation configurations.

Message Duplication. For messages sent using at-least-once delivery guarantees, the receiving actor might receive duplicated messages. As illustrated in Section 2, developers need to account for duplicates in the receiving actor by either remembering messages that have already been processed or by rendering its message processing idempotent. Our tester attempts to uncover defects in this implementation by generating a *Duplication* perturbation for every *Send* event of which the message was sent using at-least-once delivery semantics (*i.e.,* b_{alod} is true). The sender l_{from}, receiver k_{to}, and message hashcode h_{msg} are set correspondingly.

Actor Restart. For persistent actors that are restarted due to node failure or cluster migration, it might not recover to its last known state due to defects in the implementation of its state persistence or recovery. Our tester attempts to uncover such defects by generating a *Restart* perturbation for every *Send* event that targets a persistent actor (*i.e.,* $b_{\text{persistent}}$ is true). Restarts happen after any message, regardless of their message delivery semantics. The reason why we restart the actor after any message is because they internally transition to a new state, and at every transition, there might be a defect in the implementation. The sender l_{from}, receiver k_{to} (*i.e.,* the actor that is restarted), and message hashcode h_{msg} are set correspondingly.

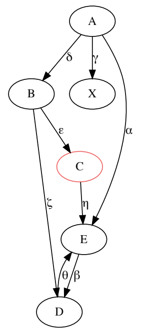

Figure 4: Illustrative actor system.

4 EXPLORATION STRATEGIES

Our resilience tester repeatedly re-executes a test, while applying a perturbation configuration. We present three exploration strategies that each determine the perturbation configurations in their way.

4.1 RT-R: a naive exploration

Given a set of perturbations P, the tester would ideally explore the power set $\mathcal{P}(P)$ within its given test budget as there are $|2^P|$ combinations of perturbation configurations. Exploration strategy RT-R therefore randomly applies perturbation configurations with *one* perturbation only, which is linear with respect to the cardinality of P. As such, every perturbation is applied individually and in random order. Indeed, this strategy will miss combinations of perturbations that lead to a defect but is out of the scope of this paper. We use RT-R as the baseline for our evaluation.

4.2 RT-DD: a delta debugging approach

To speed up resilience testing, we leverage the delta debugging algorithm [49] which has a logarithmic and quadratic time complexity in the best and worst case, respectively. We call the corresponding exploration strategy RT-DD.

The original delta debugging algorithm recursively tries to reduce a set of changes to satisfy 1-minimality (*i.e.,* removing any single change causes the failure to disappear). Our usage is slightly different as the tester does not know in advance of which perturbation configurations change the test outcome. Therefore, we consider the set of all possible perturbations as the initial perturbation configuration that *might* cause the change. In case the test outcome changes under this perturbation configuration, there *is* at least one perturbation responsible. Delta debugging will consequently reduce this configuration to the 1-minimal set of perturbations.

We briefly illustrate our algorithm using an example program whose communication topology is depicted in Figure 4. Every node represents an actor and every edge represents a sent message. All actors are persistent and every message is sent using at-least-once delivery guarantees, except for the processing acknowledgements which are sent using the default at-most-once guarantees (omitted from the figure). The order in which the messages are sent is indicated by the greek letter. For instance, messages ϵ and ζ are sent in the same turn after each other, but only after δ was received. For illustrative purposes, messages δ and γ are only sent after the confirmation of α, while all other messages are sent directly upon receiving a message. The example has a persistence defect in actor C which can be triggered by restarting the actor after processing ϵ from actor B.

```
1 ✗: Set(α,ᾱ,β,β̄,γ,γ̄,δ,δ̄,ϵ,ϵ̄,ζ,ζ̄,η,η̄,θ,θ̄)
2 ✗: Set(α,ᾱ,β,β̄,δ,δ̄,ϵ,θ)
3 ✓: Set(α,ᾱ,β,δ)
4 ✗: Set(β̄,δ̄,ϵ,θ)
5 ✓: Set(θ,β̄)
6 ✗: Set(δ̄,ϵ)
7 ✓: Set(δ̄)
8 ✗: Set(ϵ)
```

Figure 5: Steps of RT-DD to find the perturbation ϵ.

Figure 5 depicts the corresponding algorithmic steps. The input is a perturbation configuration that contains all *Restart* perturbations of the system. Step 1 tests this configuration which consists of both messages sent with at-least-once delivery semantics (denoted with greek letters) and messages sent with at-most-once semantics (denoted with greek letters with a bar). The ✗ outcome indicates that there is at least one problematic perturbation. The algorithm proceeds by splitting the configuration into two smaller configurations. The configuration selected in step 2 also results in a failing test outcome ✗ and is therefore further split. Step 3 determines that the perturbations do not affect the test outcome ✓. Therefore, Step 4 examines the other part of the configuration, which needs to be split into two again. The remaining steps of the algorithm determine that the test fails when actor C is restarted, after having received the message ϵ from actor B.

Note that another run of this algorithm might result in different partitions due to the non-determinism of the way configurations are represented (*i.e.,* the implementation of sets). This also shows that choosing a partitioning strategy can further optimise the outcome (*e.g.,* test perturbations of earlier messages first), as also suggested by Zeller *et al.* [49].

It is also important to understand that we *test* the resilience of a system, and not *verify* it. Therefore, it is no guarantee that the system is free of resilience defects when none of the applied perturbations causes a change in test outcome (*e.g.,* due to weak assertions). We discuss some of our assumptions and limitations in Section 6.

4.3 RT-DD-O: optimising delta debugging

When an actor sends messages to several other concurrent actors, independent execution paths may arise. Therefore actors on these paths might not affect each other as the state is not shared, even though one of them processed a message before the other. Inspired by the idea of hierarchical delta debugging [39], our final exploration strategy leverages the causality relation between actors to further reduce the perturbation space that has to be explored.

Algorithm 1 determines the causality relation from an execution trace. Essentially, the algorithm links one turn (*i.e.,* t_b on line 3)

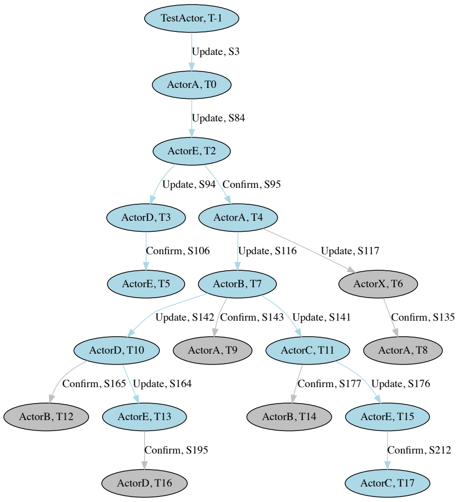

Figure 6: The causality relation of Figure 4.

to another turn (*i.e.,* t_a on line 1) using the message that was sent from within the former (*i.e.,* *se* on line 2) and that gave rise to the latter. Recall from Section 3.2, the send and turn identifiers i_{send}, j_{turn} are used to find out exactly which turns caused which sends, and which sends cause which turns. Finally, line 4 merges t_a to the current list of turns found at key t_b in the map using the merge operator (|+|).

The result of applying this algorithm to the trace of Figure 4 is shown in Figure 6. Nodes represent a unique actor turn, incoming edges represent the message that caused that turn, and outgoing edges represent an asynchronous message that was sent from within that turn. The numbers prefixed with *S* and *T* are the corresponding send and turn identifiers.

Algorithm 1: Determining the causality relation.

```
Input  : trace, an execution trace
Output : cr, the causality relation represented by mapping turns to a list of causally
         connected turns (i.e., Map[Turn,Turn*])
1 for t_a ← trace.turns do
2     se ← trace.sends.find(s => s.i_send == t_a.i_send) ;     // Find se that caused t_a
3     t_b ← trace.turns.find(t => se.j_turn == t.j_turn) ;      // Find t_b that caused se
4     cr ← cr |+| (t_b ↦ t_a) ;                                 // Thus, t_b caused t_a via se
5 return cr;
```

Whenever a test fails as a result of applying perturbations during its execution, there is an assertion about the state of an actor at location *l* that failed. Collecting all turns (and their causing messages) *of any actor* that happened before the last turn of the failing actor is one strategy to find the perturbations that might have caused the failure. For instance, if an assertion failed for actor `C` in Figure 4, all turns in Figure 6 would be held responsible as they all happen before that actor's last turn `T17`. However, it is clear that some turns (*e.g.,* `T6` and `T8`) cannot have affected `T17` as they reside on completely independent execution paths. While this strategy is trivial, it is suboptimal in performance.

A better strategy is not to use the execution trace, but to use the causality relation extracted from it. All turns that are causally connected to a specific actor are those found on all paths from the root to any turn of that actor in the causality relation. This is what Algorithm 2 determines.

Algorithm 2: Collect causally connected turns.

```
Input  : cr, the causality relation
         l, the location of an actor for which an assertion failed
         m, the maximum turn identifier
Output : collected, the set of causally connected turns
1  setOfPaths ← { cr(cr.root) } ;                          // All paths start from the root
2  collected ← ∅;
3  while setOfPaths != ∅ do                                // While there are unexplored paths
4      pathOfTurns ← setOfPaths.take(1)
5      lastTurnOnPath ← pathOfTurns.head                   // Last prepended turn on line 12
6      if lastTurnOnPath.k_to == l then                    // The turn is of our actor located at l
7          for turn ← pathOfTurns do
8              collected ← collected + turn                // Collect all turns on this path
9      connectedTurns ← cr.getOrElse(lastTurnOnPath, [])   // Turns caused by
        lastTurnOnPath
10     for turn ← connectedTurns do
11         if turn.j_turn <= m then                        // Turn identifier must be lower than m
12             setOfPaths ← setOfPaths + (turn :: pathOfTurns) // Prepend turn to path
13 return collected
```

In essence, it performs a breadth-first search to collect all paths to a given actor and returns all unique turns on these paths. The first parameter is the causality relation determined by Algorithm 1, the second parameter is the actor's location, and the last parameter specifies that only turns with an identifier lower than that identifier has to be collected. For all turns of actor `C`, this algorithm returns the set `{T0,T2,T4,T7,T11,T15,T17}`.

However, these turns are only a subset of the required one. The exploration strategy should also consider turns that might have affected one of the returned states. For instance, the turns on the path to `T5` should be included as well as these happened before the turn of `T15` which caused `T17` of actor `C`, and therefore `T5` might have affected the *run-time state* of `E`. This has to be repeated until we have every turn included, as shown in Algorithm 3.

Algorithm 3: Pruning perturbations.

```
Input  : cr, the causality relation
         l, the location of an actor for which an assertion failed
         c, a perturbation configuration
Output : c', a filtered perturbation configuration
1 turns ← collectCausallyConnectedTurns(cr, l, Integer.MAX) ;   // Step 1
2 affected ← ∅;
3 while turns != ∅ do                                   // Repeat Step 1 for each affected turn
4     turn ← turns.take(1)
5     affected ← visited + turn
6     extra ← collectCausallyConnectedTurns(cr, turn.k_to, turn.j_turn)
7     turns ← turns + (extra - visited)
8 return c.filter(p => affected.contains(s => s.l_from == p.l_from && s.k_to == p.k_to && s.h ==
    p.h))
```

The first step is shown on line 1, while the other step is determined through the while-loop on lines 3—8. For the example shown in Figure 4, the while-loop only makes sense for turns of actor `E` that happened before `T15`. The algorithm would determine that the turns on the paths `{T0,T2,T3,T5}` and `{T0,T2,T4,T7,T10,T13}` have to be included as well. Thus, Figure 6 depicts the final set by all blue messages and turns, while grey ones are pruned. Finally,

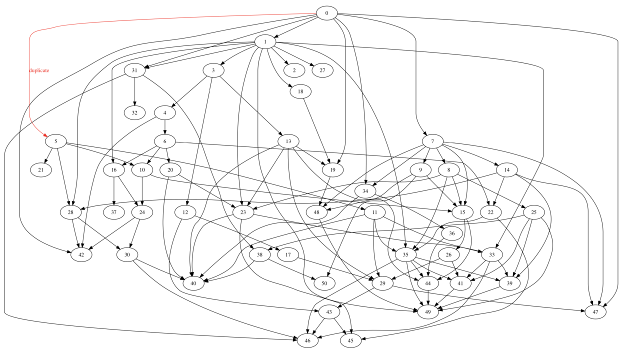

Figure 7: Generated actor system with a resilience defect.

line 8 filters the given perturbation configuration so that only those perturbations remain which were based on the determined turns and their causing messages. This means that our implementation relaxes the equality of messages as we use the combination of sender, receiver and message hash. Ideally, perturbations should be based on the send identifier instead of this combination. However, this requires an advanced replay mechanism of the whole actor system which is out of the scope of this paper.

It is also important to note that this optimisation only works when there are independent execution paths. In the worst case, RT-DD-O degrades to RT-DD. Compared to RT-R, this is still much more efficient. Nevertheless, we see several applications where independent execution paths occur such as publish/subscribe systems and microservice architectures as discussed in Section 6.1.

5 EVALUATION

We evaluate our approach by applying its prototype implementation Chaokka on many automatically generated actor systems, seeded with defects in the implementation of resilience against actor restarts and duplicated message. Through this experiment, we aim to answer the following research questions:

RQ$_1$: *How effective are the delta-debugging exploration strategies RT-DD and RT-DD-O compared to the random exploration strategy RT-R in detecting the seeded resilience defects?*

RQ$_2$: *What is the overhead of applying Chaokka’s perturbations on the execution of test cases?*

5.1 Design

As there is no open-source corpus of distributed actor systems that implement resilience tactics with known defects, we automatically generate actor systems for our experiments and randomly seed them with resilience defects. The communication topology of the generated actor systems is representative for those known from microservice architecture benchmarks [16, 54] and cloud services such as eBay[7]. That is, we assume that one actor corresponds to one microservice.

Figure 7 depicts an example of the actor systems generated for our experiments. In contrast to Akka’s default, all generated actors are resilient against restarts and all asynchronous messages are resilient against delivery failures. Each actor system consists of 50 numbered actors that persist a counter as their internal state. For each system, we generate a test case that sends a message to the entry point of the system (*i.e.,* actor 0) and asserts the system’s state after all communication has happened. During the execution of the system, messages are sent with at-least-once delivery semantics to one or more actors with a higher number. Each message changes the internal state by incrementing a counter value and persisting it subsequently. For each system, our generation process randomly selects n communication pairs from all pairs of actors (s, r) such that the receiver r has a higher number than the sender s (*i.e.,* the communication topology forms a directed acyclic graph). However, this process might result in a system where not every actor receives a message. Therefore, we extend the communication pairs such that every actor receives at least one message. These communication pairs are also used to generate assertions. In particular, we assert that the final counter value is equal to the number of paths from actor 0 to this actor. This number equals to the number of messages it will receive, and therefore equals the value of the counter. We simulate a defect in persistence by not persisting its counter value across restarts, and a defect in idempotence by not checking for duplicated messages. To answer **RQ$_1$** and **RQ$_2$**, we conduct the following experiments:

Experiment$_1$: We generate 10 actor systems, summarized in Table 1, and run our tool on mutants of these actor systems by seeding one defect in one of the actors with number 5, 25, or 45. These actors were selected as targets since they process their messages at different times in the execution. The resulting set of systems consists of 30 actor systems with varying size and defects, as shown in Table 1. The number of perturbations explored by each strategy is determined by the number of messages and the perturbation type. For each exploration strategy, we repeat the experiments for each system 10 times with a timeout of 30 minutes.

Experiment$_2$: We select the largest generated actor system from our previous experiment (*i.e.,* the one with 2008 messages) and systematically select and apply n perturbations, where n increases in steps of 100. We repeat this experiment 10 times and compare the execution time to assess the overhead of each perturbation.

All experiments are executed on an Ubuntu 18.04.3 instance with 252GB of RAM and 8 Intel(R) Xeon(R) CPU E5-2637 v3 @ 3.50GHz with Hyper-Threading enabled.

Table 1: The mean (A) and median (M) number of iterations needed to find a resilience defect, as well as the number of timeouts (T). Resilience defect is either restart actor (R) or duplicate message (D).

Messages		258	408	616	378	1026	626	706	854	1770	2008
Resilience Defect		D	D	D	R	D	R	R	R	D	D
Perturbations		129	204	308	378	513	626	706	854	885	1004
RT-R	A	52	63	32	49	34	62	49	46	42	41
	M	36	47	14	35	9	62	42	26	20	30
	T	0	4	13	1	13	6	2	6	12	14
RT-DD	A	12	12	13	13	15	14	14	14	14	15
	M	12	12	13	13	14	14	14	14	14	15
	T	0	0	0	0	0	0	0	0	0	0
RT-DD-O	A	8	10	11	8	11	8	8	10	13	10
	M	7	11	11	9	12	9	8	10	12	9
	T	0	0	0	0	0	0	0	0	0	0

[7]See https://youtu.be/U7X3qONf3sU?t=1182 for a description.

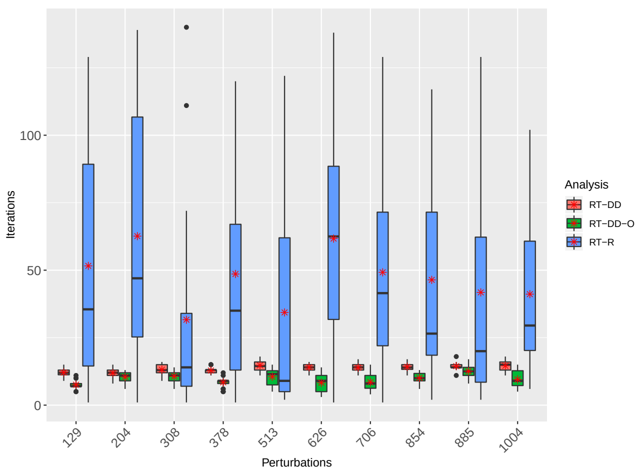

Figure 8: Performance of all three exploration strategies.

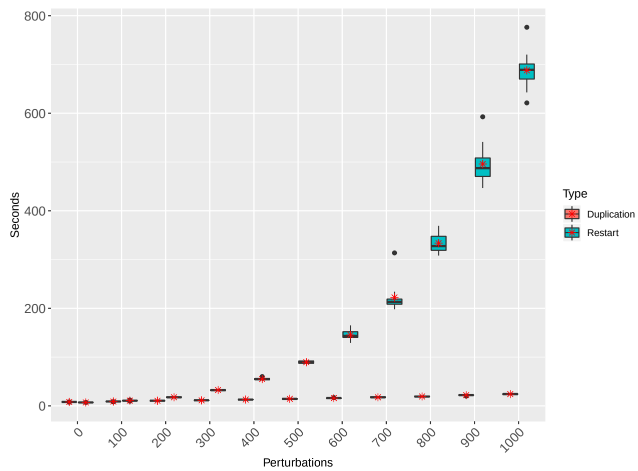

Figure 9: Overhead of perturbations.

5.2 Results

RQ$_1$. Table 1 on the previous page depicts the mean (rows A) and median (rows M) number of iterations that were required by each exploration strategy to find the seeded defect, as well as the number of runs that timed out (rows T) after 30 minutes. However, timeouts only occurred for RT-R as can be seen from that table. Figure 8 depicts the results of all runs, omitting runs that timed out.

It is clear that the number of iterations required by RT-R fluctuates widely, while RT-DD and RT-DD-O are much more stable and require fewer iterations to find the seeded defect. Note that, in our experiments, RT-R did not necessarily time out more often when an increasing number of perturbations needed to be explored. Again, this is due to its non-deterministic nature. As a testament to their efficiency, the delta-debugging strategies do not time out at all. For all experiments, it takes RT-R on average 33 and 37 iterations more to find the defect compared to RT-DD and RT-DD-O respectively. In relative terms, RT-R needs 370% of the iterations of RT-DD-O and 236% of those of RT-DD.

The performance of RT-DD-O is slightly better than that of RT-DD. In all experiments, it takes RT-DD on an average 4 iterations more compared to RT-DD-O. In relative terms, RT-DD needs 140% of the iterations of RT-DD-O. While not immediately apparent from Figure 8, the number of iterations required by RT-DD-O is sensitive to the location of the defect in the trace of the test case execution. For defects located early on in the execution, it is more likely that RT-DD-O can prune away a large part of the trace.

RQ$_1$ Summary

RT-DD and RT-DD-O outperform RT-R and need about four times fewer iterations for detecting a single failure. RT-DD-O demonstrates that causality can be leveraged to achieve a better performance than RT-DD. However, the improvements are highly dependent on program structure and fault location and can degrade to RT-DD in the worst case.

RQ$_2$. Figure 9 depicts box-and-whisker plots of the different execution times needed to run the test case with increasingly large perturbation configurations.

We observe that the execution overhead of configurations consisting of duplication perturbations grows linearly, while the overhead of configurations consisting of restart perturbations seems to grow exponentially. This is to be expected as asynchronous message sends, the bread and butter of the actor model, are fast and duplicating one message causes little overhead. The need to restart an actor, in contrast, should be rare and therefore does cause an overhead —which might be less outspoken for actors that persist and recover their state through other means than event sourcing. Nevertheless, the overhead of at most 13 minutes for the most expensive perturbation configurations is still within acceptable limits and indicates that it is feasible to incorporate the Chaokka prototype in a testing process. Moreover, there is ample room for improvements in its implementation.

RQ$_2$ Summary

The overhead of Chaokka is acceptable for large quantities of message duplication perturbations applied to a test case, but might become problematic for large quantities of actor restart perturbations. Mimicking restarts instead of native restarts by Akka could improve performance but requires additional engineering effort.

6 APPLICABILITY & LIMITATIONS

To the best of our knowledge, Chaokka is the first resilience testing approach for actor programs written in the Akka framework. We briefly discuss other potential application domains of our approach, as well as its assumptions and limitations.

6.1 Applicability

Actor frameworks. Our prototype targets Akka because it is the most popular implementation of the actor model for the JVM, with both a Java and a Scala implementation. However, our approach is equally applicable to actor frameworks for other languages such as Orleans, Pykka, Actix, *etc.* We have provided our tool as a reference implementation in the hope it may be adapted to these frameworks as well. It should suffice to intercept the run-time events related to the actor model and trace them in the format presented in Section 3.2.

Microservices. It is easy to draw similarities between the actor model and the microservice architecture [36]. One could argue that an actor is the smallest feasible granularity for such a service. Indeed, this is the point of view taken by LAGOM[8], a microservices framework built on top of AKKA. Therefore, our ideas should transpose easily to other frameworks such as SPRING CLOUD[9].

Message brokers. Our perturbations are equally applicable to systems that use message brokers such as KAFKA or RABBITMQ. Message brokers enable consumers to subscribe to messages published to a topic by independent producers. Some brokers support at-least-once delivery guarantees, but also at the cost of requiring idempotent processing.

6.2 Limitations

Deterministic Execution Traces. CHAOKKA extracts perturbation targets from the trace of a single test run only. Therefore, it might miss resilience defects when subsequent test runs create different actors and/or messages. CHAOKKA identifies messages by their sender, receiver and hashcode. Therefore, in case of their hashcode change, they will no longer be recognized as a perturbation target. Several approaches surveyed by Lopez *et al.* [38] can be used to detect and warn about non-determinism that is due to scheduling. Other sources of non-determinism (*e.g.,* random message payloads) could also be controlled by the tester which is common in dynamic symbolic execution [20].

Test outcome as recovery oracle. CHAOKKA needs a source of truth to determine whether a run-time perturbation is successfully recovered from. Related work has used developer-provided recovery specifications [25], contracts [32] and the outcome of test cases [1] just like CHAOKKA. Test cases have also been used as oracles in other applications [11, 23, 50]. However, they could be incorrect [5] and produce incorrect results.

Input programs. Our prototype deploys, monitors, and perturbs its input programs on a single node. Location-transparent actor references enable reconfiguring a distributed AKKA program so that a single JVM suffices. This transforms the actors from distributed processes into concurrent ones, but it might cause timing differences. To avoid this issue, several proven techniques have been proposed for tracing distributed applications [44]. We deem incorporating them a large engineering effort left for future work.

Threats to validity. We are aware that our experimental results are valid only for the defect-seeded actor systems that were randomly generated for our experiments. We have mitigated this threat by ensuring that their communication topologies are representative for those of known microservices, and this with varying message exchange densities and defect locations. Further evaluation of open and closed source applications is part of our future work.

7 RELATED WORK

We summarise the related work on resilience testing, delta debugging, and test amplification.

Resilience Testing. GREMLIN [25] tests the failure-handling capabilities of microservices in a language-agnostic manner by perturbing inter-service messages at the network layer. Testers need to specify failure scenarios and the corresponding recovery observations manually. Similarly, FATE and DESTINI [21] require specifications for their testing of the resilience of distributed middleware such as Cassandra against disk and network failures. They intercept and perturb system calls to this end. Our approach, in contrast, takes existing tests as specifications and focuses on generic mistakes that developers make in the implementation of resilience at the application level.

CHAOSMACHINE [51] validates or falsifies a resilience hypothesis about try-catch blocks. These hypotheses are either specified through annotations on the block or discovered through execution monitoring, and concern the difference in its behaviour in an execution with or without an exception (*e.g.,* the exception should be logged, or there should be no observable side-effects). Taking a Chaos Engineering approach, CHAOSMACHINE perturbs the system in production of which the monitored behaviour serves as an oracle. Its exploration strategy injects a single exception at a time, in contrast to the delta debugging approach taken in this paper.

CHAOS MONKEY [10] is a well-known NETFLIX Chaos Engineering tool that verifies in production whether the service is resilient to the termination of cloud resources. It has since been extended with other but equally coarse-grained production perturbations. NETFLIX has also experimented with Lineage-Driven Fault Injection [3], which reasons backwards from a run about possible failures that could affect the run's outcome. Proposed for data management systems, this so-called lineage comprises coarse-grained data partitioning and replication steps. An initial application of the technique to AKKA systems [19] with more fine-grained steps appeared to suffer from scalability issues.

Delta Debugging. Delta debugging [49] has been used in the context of web applications [23], web browsers [49], and microservices [53]. In particular, the work of Adamsen *et al.* [1] is closely related to ours as it subjects ANDROID test suites to adverse conditions (*e.g.,* device rotations) and leverage delta debugging to figure out the problematic ones. Our contribution is not only the transposition to the domain of actor-based systems but also the identification of the adverse conditions under which defects in the implementation of actor resilience tactics will occur. Zeller *et al.* remarked that the partitioning strategy affects the performance of delta debugging. The specific structure of inputs such as XML [39] and boolean formulas [9] has been used to speed up the process. Likewise, static and dynamic program slicing has been used to partition only the relevant parts of a program execution [22, 34]. The latter is similar to our approach as we use a dynamic causality slice to prune redundant perturbations.

Test Amplification. CHAOKKA modifies test execution which is one of the four ways to perform test amplification [12]. In this field, several works [11, 52] trigger unexpected exceptions during test runs to check the behaviour of a program in the presence of unanticipated scenarios. Leung *et al.* [35] use dynamic traces to find race conditions and non-determinism in the CUDA programming language.

[8] https://www.lagomframework.com
[9] https://spring.io/projects/spring-cloud

8 CONCLUSION

We have presented an automated approach for testing the resilience of actor-based programs against adverse conditions. The approach leverages existing tests by perturbing their execution and using their outcome as a resilience oracle. CHAOKKA implements this approach for the popular AKKA framework. As efficiently exploring the perturbation space is crucial to its success, we have proposed three exploration strategies and compared them on 30 representative and fault-seeded generated actor systems of increasing complexity. Our results show that the optimized delta debugging exploration strategy is up to four times faster than random exploration.

REFERENCES

[1] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. 2015. Systematic execution of android test suites in adverse conditions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*.
[2] Gul A Agha. 1985. *Actors: A model of concurrent computation in distributed systems.* Technical Report.
[3] Peter Alvaro, Kolton Andrus, Chris Sanden, Casey Rosenthal, Ali Basiri, and Lorin Hochstein. 2016. Automating Failure Testing Research at Internet Scale. In *Proceedings of the 7th ACM Symposium on Cloud Computing*.
[4] Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. 2015. Lineage-driven Fault Injection. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia).
[5] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2014. The oracle problem in software testing: A survey. *TSE* 41, 5 (2014).
[6] Ali Basiri, Niosha Behnam, Ruud De Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, and Casey Rosenthal. 2016. Chaos engineering. 33, 3 (2016).
[7] Moritz Beller, Georgios Gousios, Annibale Panichella, Sebastian Proksch, Sven Amann, and Andy Zaidman. 2017. Developer testing in the ide: Patterns, beliefs, and behavior. *bTransactions on Software Engineering* 45, 3 (2017).
[8] David Bowes, Tracy Hall, Jean Petric, Thomas Shippey, and Burak Turhan. 2017. How good are my tests?. In *2017 IEEE/ACM 8th WETSoM*.
[9] Robert Brummayer, Florian Lonsing, and Armin Biere. 2010. Automated testing and debugging of SAT and QBF solvers. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 44–57.
[10] Michael Alan Chang, Bredan Tschaen, Theophilus Benson, and Laurent Vanbever. 2015. Chaos monkey: Increasing sdn reliability through systematic network destruction. In *ACM SIGCOMM Computer Communication Review*, Vol. 45. ACM.
[11] Benoit Cornu, Lionel Seinturier, and Martin Monperrus. 2015. Exception handling analysis and transformation using fault injection: Study of resilience against unanticipated exceptions. *Information and Software Technology* 57 (2015), 66–76.
[12] Benjamin Danglot, Oscar Vera-Perez, Zhongxing Yu, Martin Monperrus, and Benoit Baudry. 2017. The Emerging Field of Test Amplification. *CoRR* (2017).
[13] Jonas De Bleser, Dario Di Nucci, and Coen De Roover. 2019. Assessing Diffusion and Perception of Test Smells in Scala Projects. In *Proceedings of the 16th International Conference on Mining Software Repositories*.
[14] Colin J Fidge. 1988. Partial orders for parallel debugging. In *ACM Sigplan Notices*.
[15] Martin Fowler. 2005. Event sourcing. (2005). https://martinfowler.com/eaaDev/EventSourcing.html
[16] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of ASPLOS*.
[17] Yu Gao, Wensheng Dou, Feng Qin, Chushu Gao, Dong Wang, Jun Wei, Ruirui Huang, Li Zhou, and Yongming Wu. 2018. An Empirical Study on Crash Recovery Bugs in Large-scale Distributed Systems. In *Proceedings of the 26th ESEC/FSE.*
[18] Vahid Garousi and Junji Zhi. 2013. A survey of software testing practices in Canada. *Journal of Systems and Software* 86, 5 (2013), 1354–1376.
[19] Yonas Ghidei. 2019. Lineage-driven Fault Injection for Actor-based Programs.
[20] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation.* 11.
[21] Haryadi S Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M Hellerstein, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. 2011. FATE and DESTINI: A framework for cloud recovery testing. In *Proceedings of Symposium on Networked Systems Design and Implementation.*
[22] Neelam Gupta, Haifeng He, Xiangyu Zhang, and Rajiv Gupta. 2005. Locating faulty code using failure-inducing chops. In *Proceedings of the 20th ACM/IEEE International Conference on Automated Software Engineering.*
[23] Mouna Hammoudi, Brian Burg, Gigon Bae, and Gregg Rothermel. 2015. On the use of delta debugging to reduce recordings and facilitate debugging of web applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 333–344.
[24] Pat Helland. 2012. Idempotence is not a medical condition. *Queue* 10, 4 (2012).
[25] Victor Heorhiadi, Shriram Rajagopalan, Hani Jamjoom, Michael K Reiter, and Vyas Sekar. 2016. Gremlin: Systematic resilience testing of microservices. In *IEEE 36th International Conference on Distributed Computing Systems.*
[26] Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence.*
[27] Waldemar Hummer, Florian Rosenberg, Fábio Oliveira, and Tamar Eilam. 2013. Testing idempotence for infrastructure as code. In *International Conference on Distributed Systems Platforms and Open Distributed Processing.*
[28] Shams M Imam and Vivek Sarkar. 2014. Savina-an actor benchmark suite: Enabling empirical evaluation of actor libraries. In *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control.*
[29] Yury Izrailevsky and Ariel Tseitlin. 2011. The netflix simian army. *Netflix* (2011).
[30] Pavneet Singh Kochhar, Tegawendé F Bissyandé, David Lo, and Lingxiao Jiang. 2013. Adoption of Software Testing in Open Source Projects–A Preliminary Study on 50,000 Projects. In *17th CSMR.*
[31] Roland Kuhn, Brian Hanafee, and Jamie Allen. 2017. *Reactive design patterns.*
[32] Yves Le Traon, Benoit Baudry, and J-M Jézéquel. 2006. Design by contract to improve software vigilance. *Transactions on Software Engineering* 32, 8 (2006).
[33] Tanakorn Leesatapornwongsa, Jeffrey F Lukman, Shan Lu, and Haryadi S Gunawi. 2016. TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. *ACM SIGPLAN Notices* 51, 4 (2016).
[34] Andreas Leitner, Manuel Oriol, Andreas Zeller, Ilinca Ciupa, and Bertrand Meyer. 2007. Efficient unit test case minimization. In *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering.*
[35] Alan Leung, Manish Gupta, Yuvraj Agarwal, Rajesh Gupta, Ranjit Jhala, and Sorin Lerner. 2012. Verifying GPU kernels by test amplification. In *ACM SIGPLAN Notices*, Vol. 47.
[36] James Lewis and Martin Fowler. 2014. Microservices. *martinfowler. com* (2014).
[37] Lightbend. 2020. *Lightbend Case Studies.* https://lightbend.com/case-studies
[38] Carmen Torres Lopez, Stefan Marr, Elisa Gonzalez Boix, and Hanspeter Mössenböck. 2018. A study of concurrency bugs and advanced development support for actor-based programs. In *Programming with Actors*. Springer, 155–185.
[39] Ghassan Misherghi and Zhendong Su. 2006. HDD: hierarchical delta debugging. In *Proceedings of the 28th international conference on Software engineering*. ACM.
[40] Glenford J Myers, Corey Sandler, and Tom Badgett. 2011. *The art of software testing.* John Wiley & Sons.
[41] Michael Nash and Wade Waldron. 2016. *Applied Akka Patterns: A Hands-On Guide to Designing Distributed Applications.* O'Reilly Media.
[42] Lennart Oldenburg, Xiangfeng Zhu, Kamala Ramasubramanian, and Peter Alvaro. 2019. Fixed It For You: Protocol Repair Using Lineage Graphs.. In *CIDR.*
[43] Roger S Pressman. 2005. *Software engineering: a practitioner's approach.*
[44] Raja R Sambasivan, Rodrigo Fonseca, Ilari Shafer, and Gregory R Ganger. 2014. So, you want to trace your distributed system? Key design insights from years of practical experience. *Parallel Data Lab* (2014).
[45] Gianluca Stivan, Andrea Peruffo, and Philipp Haller. 2015. Akka.js: towards a portable actor runtime environment. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control.*
[46] Samira Tasharofi, Peter Dinges, and Ralph E Johnson. 2013. Why do scala developers mix the actor model with other concurrency models?. In *European Conference on Object-Oriented Programming.*
[47] Samira Tasharofi, Michael Pradel, Yu Lin, and Ralph Johnson. 2013. Bita: Coverage-guided, automatic testing of actor programs. In *28th IEEE/ACM International Conference on Automated Software Engineering.*
[48] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U Jain, and Michael Stumm. 2014. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Symposium on Operating Systems Design and Implementation.*
[49] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (2002).
[50] Jie Zhang, Yiling Lou, Lingming Zhang, Dan Hao, Lu Zhang, and Hong Mei. 2016. Isomorphic Regression Testing: Executing Uncovered Branches Without Test Augmentation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering.*
[51] Long Zhang, Brice Morin, Philipp Haller, Benoit Baudry, and Martin Monperrus. 2018. A Chaos Engineering System for Live Analysis and Falsification of Exception-handling in the JVM. *arXiv preprint arXiv:1805.05246* (2018).
[52] Pingyu Zhang and Sebastian Elbaum. 2014. Amplifying Tests to Validate Exception Handling Code: An Extended Study in the Mobile Application Domain. *ACM Trans. Softw. Eng. Methodol.* 23, 4 (2014).
[53] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2018. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *TSE* (2018).
[54] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chenjie Xu, Chao Ji, and Wenyun Zhao. 2018. Benchmarking Microservice Systems for Software Engineering Research. In *Proceedings of the 40th International Conference on Software Engineering.*