An Automated Delta-Debugging Approach to Resilience Testing of Actor Systems through Fault Injection

Jonas De Bleser

Dissertation submitted in fulfillment of the requirement for
the degree of Doctor of Philosophy in Sciences

October 9, 2020

Promotor:
Prof. Dr. Coen De Roover, Vrije Universiteit Brussel, Belgium

Jury:
Prof. Dr. Viviane Jonckers, Vrije Universiteit Brussel, Belgium (chair)
Prof. Dr. Geraint Wiggins, Vrije Universiteit Brussel, Belgium (secretary)
Prof. Dr. Martin Monperrus, KTH Royal Institute of Technology, Sweden
Prof. Dr. Wolfgang De Meuter, Vrije Universiteit Brussel, Belgium
Prof. Dr. Kris Steenhaut, Vrije Universiteit Brussel, Belgium
Prof. Dr. Fabio Palomba, University of Salerno, Italy

Vrije Universiteit Brussel
Faculty of Sciences and Bio-engineering Sciences
Department of Computer Science
Software Languages Lab

Printed by
Crazy Copy Center Productions
VUB Pleinlaan 2, 1050 Brussel
Tel: +32 2 629 33 44
crazycopy@vub.ac.be
www.crazycopy.be

ISBN 9789493079755
NUR 980

Abstract

Until recently, software systems were typically designed with a monolithic architecture and deployed as a single executable. However, these systems struggle to keep up with the growing number of concurrent requests and increasingly shorter development cycles. Development teams therefore migrate to systems with a distributed architecture consisting of fine-grained services that can easily be scaled and developed independently. Although there are several ways to implement these services, the actor model is increasingly prevalent.

However, the distributed nature of these systems exposes them to abnormal conditions under which they may no longer remain operational. For example, the network might temporarily fail and could result in messages being lost. A resilient system can cope with such abnormal conditions by detecting them, limiting their negative impact, and restoring normal operation.

While existing actor model frameworks support the development of resilient systems, it remains the responsibility of developers to properly use this support and test the system's operation under abnormal conditions. Unfortunately, developers have been shown to have a tendency to test the system only under normal conditions. A system can therefore be less resilient than expected. Testament to this are the many outages and failures of systems that can lead to loss of turnover and customers. Therefore, automated approaches are needed to test the resilience of actor systems.

This dissertation presents an automated and dynamic analysis approach that injects faults during test execution to determine the resilience of the system. To efficiently explore all possible faults, we propose an exploration strategy based on delta debugging. This strategy can also be combined with optimizations to identify defects earlier. We explore one optimization where the causality of actor events is used to prune away certain faults, and a number of others where faults are given a priority to inject certain faults earlier. We implement our approach in the Chaokka tool. This tool is built for the actor model framework Akka and the testing framework ScalaTest which are widely used in practice. We also present an automated and static analysis approach to assess the quality of existing tests into which faults will be injected. We implement this approach in the Socrates tool and observe a low distribution of so-called test smells throughout the Scala ecosystem.

Finally, we demonstrate the applicability of our approach by testing the resilience of actor systems in which various defects are present. We conclude that our approach finds defects 5 times faster compared to a naive approach. This dissertation therefore provides a solid foundation to automatically and efficiently test system resilience.

Samenvatting

Tot voor kort werden softwaresystemen doorgaans ontworpen met een monolithische architectuur en uitgerold als één enkel uitvoerbaar bestand. Zulke systemen hebben echter moeite om een groeiend aantal gelijktijdige verzoeken en steeds kortere ontwikkelingscycli bij te houden. Ontwikkelingsteams migreren daarom naar systemen met een gedistribueerde architectuur die bestaat uit aparte services die gemakkelijk los van elkaar geschaald en ontwikkeld kunnen worden. Het actor model komt steeds vaker voor in de implementatie van deze services.

Het gedistribueerde karakter van deze systemen stelt ze echter bloot aan abnormale omstandigheden waaronder ze kunnen falen. Zo kan het netwerk bijvoorbeeld tijdelijk falen en kunnen berichten verloren gaan. Een veerkrachtig systeem kan met zulke abnormale omstandigheden overweg door ze te detecteren, hun negatieve impact te beperken, en de normale werking te herstellen.

Hoewel bestaande frameworks ondersteuning bieden voor het ontwikkelen van veerkrachtige systemen door middel van het actor model, blijft het de verantwoordelijkheid van ontwikkelaars om deze ondersteuning correct te gebruiken en de werking ervan onder abnormale omstandigheden te testen. Helaas is het aangetoond dat ontwikkelaars de neiging hebben om het systeem alleen onder normale omstandigheden te testen. Een systeem kan daarom minder veerkrachtig zijn dan gedacht. Een bewijs hiervan zijn de vele storingen van systemen die kunnen leiden tot het verlies van omzet en klanten. Er zijn daarom geautomatiseerde aanpakken nodig om de veerkracht van actor-gebaseerde systemen te testen.

Dit proefschrift presenteert een geautomatiseerde en dynamische aanpak die fouten injecteert tijdens de uitvoering van tests om de veerkracht van het systeem te bepalen. Om alle mogelijke fouten efficiënt te verkennen, presenteren wij een strategie op basis van delta debugging. Deze exploratie strategie kan ook gecombineerd worden met optimalisaties om defecten eerder te identificeren. We presenteren een optimalisatie waarbij de causaliteit van actoren wordt gebruikt om bepaalde fouten over te slaan, en een aantal andere waarbij fouten een prioriteit krijgen om bepaalde fouten eerder te injecteren. We implementeren onze aanpak in de tool Chaokka. Deze tool is gebouwd voor het actor model framework Akka en het testing framework ScalaTest die in de praktijk veel gebruikt worden. We presenteren ook een geautomatiseerde en statische aanpak om de kwaliteit van bestaande tests, waarin fouten geïnjecteerd zullen worden, te beoordelen. We implementeren deze aanpak in de tool Socrates en observeren een lage verspreiding van zogenaamde test smells doorheen het Scala ecosysteem.

Ten slotte demonstreren we de toepasbaarheid van onze aanpak door de veerkracht te testen van actor-gebaseerde systemen waarin verschillende defecten aanwezig zijn. We concluderen dat onze aanpak defecten 5 keer sneller vindt in vergelijking met een naïeve aanpak. Dit proefschrift biedt daarom een sterke basis aan om de veerkrachtigheid van systemen automatisch en efficiënt te testen.

Acknowledgements

I would like to take the opportunity to express my gratitude towards all the people who have supported me over the past four years and contributed to this dissertation, directly or indirectly.

First and foremost, I would like to express my gratitude and appreciation to Coen De Roover for being my promotor and whose expertise, understanding, and patience enabled me to present this dissertation. Thank you for your countless support and comments on my research during the past years. I am also extremely thankful towards my guiding post-doc Dario Di Nucci for teaching me how to write international conference papers that get accepted and providing me valuable feedback on my ideas. I would also like to extend my sincere thanks to Martin Monperrus, Fabio Palomba, Kris Steenhaut, Wolfgang De Meuter, Geraint Wiggins, and Viviane Jonckers for being the members of my jury and their insightful comments, discussions, and feedback. Furthermore, I would like to say thank you to Ahmed and Quentin for proofreading this dissertation.

Next, I would like to thank all members of the Software Languages Lab for spending their precious time with me, providing me with relevant insights and suggestions during research presentations, and accepting me for who I am. I would also like to thank the members of the Church of Iron (Florian, Christophe, Joeri) for their support during each workout. Looking at my current physique, it is clear that these workouts had exceptional success. Noah and Kevin, thanks for joining me on the daily commute and making the ride more enjoyable. Thank you Quentin for sharing the office with me and providing me with the necessary quietude to do pioneering research. Thank you Jens (yes, both) for interrupting my daily thoughts with conversations about static analysis, which I soundly over-approximated to be bottom. Thank you Maarten for the interesting discussions about dynamic analysis and helping me out with my ideas.

Finally, I'm grateful for the support that my family provided me throughout my entire life, without whose love and encouragement I would not have finished this dissertation. I would also like to thank my closest friends for the necessary relaxation and with whom I enjoyed numerous of conversations, parties, vacations, and events. I'm looking forward to many more of these once the COVID-19 pandemic is over. Absurdly enough, this pandemic has provided me with a sense of serenity to write this dissertation. Nevertheless, I sincerely hope that this pandemic will be over soon.

Again, thank you all, stay safe, and don't forget to read the remainder of this dissertation.
— Jonas

Contents

List of Algorithms

List of Figures

List of Listings

List of Tables

List of Acronyms

DD Delta Debugging

LDFI Lineage-driven Fault Injection

JVM Java Virtual Machine

SBT Scala Build Tool

AR Assertion Roulette

ET Eager Test

LT Lazy Test

GF General Fixture

MG Mystery Guest

SE Sensitive Equality

RO Resource Optimism

SHU Shuffle

RT Registration-Time

MT Message-Time

FI Fan-In

FO Fan-Out

FIFO Fan-In/Fan-Out

Chapter 1

Introduction

Imagine a world in which software systems aren't affected by hardware, software, nor human failures. Many developers would consider this a utopian world. Unfortunately, the real world is ruled by Murphy's law. Anything that can go wrong, will go wrong. And more likely than not, it will go wrong at the moment when least desired. Software testing has therefore become a standard activity of the development process.

Traditionally, developers would manually test the system to find defects and to ensure the correctness of their system. It is trivial to see that such a manual approach is far from efficient and does not scale to increasingly large and complex systems. In response, test automation frameworks were proposed and they have considerably improved the efficiency of software testing by means of automating the execution of tests. However, these frameworks provide limited support to implement the tests themselves. For example, developers do not get feedback on the quality or coverage of the tests. We therefore envision that these frameworks should become intelligent testing platforms which amplify the effort of developers. In order to demonstrate our vision, we present two approaches that integrate with current testing practices and amplify existing test suites implemented by developers.

Section 1.1 presents the context of this dissertation and explains the relationship between test automation, automated testing, and test amplification. Additionally, we highlight that more efficient means of testing are required as the industry is moving towards complex systems with a distributed architecture. Next, Section 1.2 defines the problem statement, while Section 1.3 explains the research goals and our approaches. Section 1.4 and Section 1.5 respectively lists the supporting publications and contributions of this dissertation. Finally, Section 1.6 presents the outline of this dissertation.

1.1 Context

Software testing [Bec03, Bei03] is a software engineering activity in which developers determine whether the system meets the specified requirements and works as expected. Decades ago, developers would test their monolithic systems right before the release through manual interaction. Developers would provide inputs, navigate through components, and compare outputs to determine the correctness of the system. However, such a manual testing approach does not longer fit today's needs. Not only should testing occur early in the development process, tests should also be executed continuously so that new and old defects are found immediately after a change. Furthermore, the complexity of contemporary systems is increasing as systems are migrating from a monolithic architecture towards a distributed architecture consisting of many services. This requires more advanced testing approaches as multiple services are now involved instead of one monolithic system. Additionally, the system should also be tested under conditions that are specific to a distributed environment. A manual testing approach can no longer extensively cover the entire behavioural spectrum of such systems. As a result, development teams have been seeking more efficient testing practices to keep up with the current demands of software testing.

Test Automation

One way to increase test efficiency is by reducing the human interaction and automating the execution of tests so they can be repeated reliably. For that reason, test automation frameworks (or testing frameworks) have been proposed to execute tests without human interaction. For example, JUnit was introduced around 1998 to test Java systems and is still actively.

To facilitate the automation of tests, testing frameworks provide support to execute tests with varying granularity. Unit tests have the smallest granularity as they are supposed to only test the functionality of a single component. This is a first line of defence against problems. When all unit tests pass, developers assume that individual components work as expected. Next, integration tests determine whether several components also work correctly when put together. Finally, end-to-end (E2E) testing simulates real user interaction with the system and determines whether the system under test works correctly from start to end. Given this level of automation, tests can now be executed repeatedly to ensure that the system still works after a change. The practice of repeating tests is called regression testing [YH12] and has become a default part of the software testing process. As a result, test efficiency has steadily improved over the past two decades.

Complex Distributed Systems

With the advent of complex systems that have to scale to millions of concurrent requests, increasingly more and complicated tests have to be implemented by developers. For example, we observe that development teams are increasingly migrating towards systems with a distributed architecture to keep up with the increasing demands of scalability and increasingly short development and deployment processes [MS20]. These systems need non-functional requirements such as availability, scalability, and resilience more than ever before. Yet, developers still have to come up with many ad-hoc tests by themselves as current testing frameworks provide limited support for such tests.

Moreover, the efficiency of humans in implementing these tests is not always optimal. The main reason is that test quality depends on the developer's effort and experience. For example, several studies [ECS15, BHP+17, BGP+17] indicated that developers only tend to cover the most likely execution paths of the system under normal conditions (*e.g.,* no exceptions or network failures). This is likely to negatively affect test efficiency in the long run as certain defects might only arise under abnormal conditions. It becomes clear that current testing frameworks do not longer suffice to test these increasingly more complex systems and therefore leave ample room for defects in the system.

Automated Testing

The problem of current testing frameworks becomes apparent when you look at the history: they were originally designed to support developers in automating manual tests, but not necessarily in the process of implementing them. Therefore, test efficiency is still largely dominated by humans because both their thoughts (*e.g.,* coverage of test scenarios) and their efforts (*e.g.,* quality of the implemented test cases) account for a significant part of the testing process. While education and training could improve the efficiency of humans, the possible gains remain limited. Automated testing approaches therefore try to replace the generation and implementation process of humans by computers.

Several approaches to automated testing have been proposed over the past decade based on techniques such as concolic testing [GKS05, QR11], search-based testing [HJ01, HJZ15], and fuzzing [SGA07]. However, they cannot always cover complex behaviour and typically require specifications or formal models of the expected behaviour which might not always be available. Moreover, these approaches do not always integrate with contemporary testing practices and frameworks which hinders adoption: humans remain the dominant producer of tests.

Towards Test Amplification

Despite the fact that humans might not implement each test in an optimal way, human-written tests have become widely available. It might therefore be better to embrace the effort of humans rather than trying to completely replace it with automated approaches. This has given rise to the idea of test amplification.

Danglot *et al.* [DVPY+19] introduced test amplification as an umbrella for the various activities that amplify (*i.e.,* augment, enhance, refactor, or optimize) test suites. For example, test cases could be analysed and refactored such that they are more maintainable, provide more coverage of the system, or contain better oracles to determine the correctness of the system.

While the field of test amplification is still nascent, several state-of-the-art approaches already show that test amplification can improve the software testing process in the contexts such as ANDROID applications (*e.g.,* [AMM15,ZE14]) and microservices architectures (*e.g.,* [RE19]). Moreover, we envision that amplifying existing test suites is a logical step forward and will enjoy increasing adoption in the future for several reasons. First, test amplification approaches have access to both system-specific executions and oracles. This is in contrast to many automated testing techniques that can only find system-agnostic errors such as crashes or unhandled exceptions. Second, testing suites are currently the key way to ensure that a system runs correctly. Unit, integration and end-to-end tests are unlikely to be replaced. Finally, a typical development team spends about 50% of their time and costs on software testing [MSB11]. It is therefore no surprise that test suites of software companies consist of a large number of tests that can be amplified automatically. In this way, tests can be further improved with minimal effort of developers. However, test amplification remains a largely unexplored field.

1.2 Problem Statement

We explained the need for optimizing test efficiency in order to keep up with the demands of testing contemporary systems. However, humans will likely remain a large producer of tests as Agile development practices [HH02] advocate to test early and often. As a result, extensive test suites with tests have become available, but their full potential remains largely unexploited at current times.

We therefore argue that the scope of testing frameworks should be broadened: they should not only provide support for automating the execution of tests, but also provide support for amplifying tests. Our vision is therefore that testing frameworks will evolve into intelligent testing platforms that provide a next step to improve the efficiency of the testing process. We formulate the problem statement and vision of this dissertation as follows:

> *"There is a need for intelligent testing platforms that amplify existing tests to further improve the test efficiency"*

1.3 Approach

Our research goal is to demonstrate the feasibility of our vision of intelligent testing platforms. To this end, we propose advancements in two of the four forms of test amplification [DVPY+19] as discussed below.

1. **Amplification by adding new tests as variants of existing ones.** The first form generates variants of existing test cases. For example, Baudry *et al.* [BFLT06] improve the mutation score of existing test suites by generating variants of them through a bacteriological algorithm, while Thummalapenta *et al.* [TMX+11] generalized unit tests into parameterized unit tests.

2. **Amplification by synthesizing new tests with respect to changes.** The second form synthesizes new tests as a reaction to a change or commit. While the first approach is applicable to all test cases, this one only concerns test cases that are changed in a commit. For example, Xu *et al.* [XR09] use concolic execution to only target branches that are not yet covered by the existing test suite after a change.

3. **Amplification by modifying existing test code.**
The third form modifies existing test cases to improve them in terms of quality, oracles, *etc.* Unlike the first way, no new test cases are generated. For example, Xie *et al.* [Xie06] amplifies object-oriented unit tests by adding assertions on the state of the receiver object and parameters in order to strengthen the test cases. *This form is of particular interest to us as Chapter 3 will present* Socrates *— an approach that analyses test code in order to improve its quality through refactorings.*

4. **Amplification by modifying test execution.**
The final form modifies the test execution in order to determine additional runtime information about the system under test. The work in this category typically instruments the system to intercept, modify and monitor certain locations in the system. For example, failures in the microservices architecture of eBay were found through test amplification of end-to-end tests [RE19]. *This form is of particular interest to us as Chapter 6 will present* Chaokka *— an approach that modifies test execution in order to assess a system's resilience.*

Our approaches incorporate automated program analyses that leverage information from existing tests. These analyses either extract information through a dynamic analysis (*i.e.,* by observing the execution of a test) or a static analysis (*i.e.,* by analysing the implementation of a test). Each analysis has its own strengths and weaknesses, but both kinds of information can be leveraged to achieve a particular engineering goal.

In particular, SOCRATES [DBDNDR19a, DBDNDR19b] is a general approach that improves the quality of tests by leveraging a static analysis, while CHAOKKA [DBDNDR20] is an approach that improves the resilience of systems by leveraging a dynamic analysis. Both are stepping stones towards our vision of intelligent testing platforms and are tailored to the specifics of the SCALA ecosystem which is particularly under-investigated in research.

Socrates	**Chaokka**
Analyses test code to improve the quality of tests	Analyses and modifies test executions to improve a system's resilience
Static analysis	Dynamic analysis

Table 1.1: SOCRATES and CHAOKKA are demonstrators of our vision.

1.3.1 Socrates: Test Smell Detection

First, we propose a static analysis approach to test amplification where existing test code is analysed to improve the quality of test code. Our motivation behind this work is the presence of *test smells* [vMBK01].

These smells indicate poor design choices of developers in the implementation of tests. For example, studies have shown the negative impact of test smells on maintainability [BQO+15] and flakiness of tests [PZ19]. Additionally, studies have shown relationships between test smells and defects in production and test code [TPB+16,SPZ+18]. Therefore, their absence contributes to an increased test efficiency and results in tests that become better subjects for test amplification at a later stage. However, existing studies about test smells and tool support to detect test smells are limited and tailored to the JAVA ecosystem. As a result, the current knowledge about test smells might not generalize to different ecosystems.

We therefore propose SOCRATES [DBDNDR19b, DBDNDR19a] as an automated approach to detecting test smells in the SCALA ecosystem. In particular, we implement SOCRATES for the most popular testing framework SCALATEST to show its integration with current testing practices and its potential applicability to the industry. Our approach requires a system and its test suite as input and will start by automatically discovering all tests in the test suite. Next, it executes a static analysis on the source code of the tests to detect 6 test smells. Our approach is unique in its detection because it leverages both syntactic and semantic information about the source code.

The output of SOCRATES is an overview that indicates which test cases are affected by which test smell. This tool therefore provides valuable and actionable feedback to increase the quality of tests. We will assess the test quality in the SCALA ecosystem by using SOCRATES on 164 SCALA projects.

1.3.2 Chaokka: Resilience Testing

Second, we propose a dynamic analysis approach to test amplification where test execution is modified to improve coverage of abnormal behaviour. Our motivation behind this work is the need for *resilience* [TKG09] in contemporary systems with a distributed architecture and fine-grained services.

A resilient system deals with abnormal conditions in a way that limits their negative impact on the system. That is, the system might temporarily degrade the quality of its services while recovering the system to its normal operation. However, studies [GZ13, ECS15] have shown that developers do not often test the system and its resilience mechanisms under abnormal conditions: exception handling code is usually of poor quality and developers tend to write tests to confirm behaviour, rather than to break it. It is therefore likely that a part of the system will be impacted under abnormal conditions, even when the system's normal operation was properly designed and tested.

Additionally, we observe that the actor model is increasingly used to implement services in a distributed architecture. In this model, actors are concurrent processes that exchange information through asynchronous messages as there is no shared memory in this model. However, despite the model its popularity, there are limited resilience testing approaches for actor systems.

We therefore propose Chaokka [DBDNDR20] as an automated approach to finding resilience defects. In particular, we implement Chaokka for the most popular actor model framework Akka to show its integration with current development practices and its potential applicability to the industry. Our approach requires an actor system and a test suite as input. It first observes the execution of the tests under normal conditions. Next, it repeatedly injects faults during the test executions to simulate the abnormal conditions for the system under test. A change in test outcome indicates a resilience defect that can be diagnosed through the injected faults.

The output of Chaokka is a report that provides valuable and actionable feedback to increase the resilience of systems. We assess the efficiency of different resilience analyses by using Chaokka on generated actor systems seeded with resilience defects. We consider Chaokka as the most novel of our contributions since it targets the least researched form of test amplification. Nevertheless, Socrates is equally important as it can ensure the quality of tests before being amplified by Chaokka.

1.4 Publications

This dissertation is supported by the following publications:

- **A Delta-Debugging Approach to Assessing the Resilience of Actor Programs through Run-time Test Perturbations.**
  2020 IEEE/ACM International Conference on Automation of Software Test (AST). De Bleser, J., Di Nucci, D., & De Roover, C. [DBDNDR20]
  This paper presents the core contribution of this dissertation. It introduces the resilience testing tool CHAOKKA for detecting issues in the implementation of resilience mechanisms. The tool analyses systems that are built with AKKA and tested with SCALATEST. The key contribution is the foundation for resilience testing of actor systems through fault injection and delta debugging with support for causality-based pruning.

- **Assessing Diffusion and Perception of Test Smells in Scala projects.**
  2019 IEEE/ACM International Conference on Mining Software Repositories (MSR). De Bleser, J., Di Nucci, D., & De Roover, C. [DBDNDR19a]
  This paper presents two studies about test smells. It introduces the test smell detection tool SOCRATES to detect test smells in test suites. The tool analyses tests in SCALATEST. The contribution is twofold: an empirical study of six kinds of test smells in 164 SCALA systems, as well as a survey on the perception and knowledge about test smells by 14 professional SCALA developers.

- **SoCRATES: Scala Radar for Test Smells.**
  2019 ACM SIGPLAN Symposium on Scala. De Bleser, J., Di Nucci, D., & De Roover, C. [DBDNDR19b]
  This paper presents the implementation and internal working of our test smell detection tool SOCRATES. The key contribution is the description of its architecture which enables developers to use and extend SOCRATES for further research in the domain of test smells.

Two additional publications document our academic track record, touching upon topics addressed in this dissertation such as mining software repositories in the SCALA ecosystem and bug detection in event-driven systems.

- **Mining Scala Framework Extensions for Recommendation Patterns.**
  2019 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). Pacheco, Y., De Bleser, J., Molderez, T., Di Nucci, D., De Meuter, W., & De Roover, C. [PDBM+19]
  This paper presents work about framework extension points and patterns. It introduces the tool SCALA-XP-MINER to mine for extension patterns in SCALA systems. The key contributions include the replication of a JAVA study in the context of the SCALA ecosystem, as well as the open-source tool SCALA-XP-MINER and a replication package to facilitate further research.

- **Static Taint Analysis of Event-driven Scheme Programs.**
  2017 European Lisp Symposium. De Bleser, J., Stiévenart, Q., Nicolay, J., & De Roover, C. [DBSNDR17]
  This paper presents work about static bug detection of event-driven systems. It introduces an Abstracting Abstract Machines (AAM) approach to detect generic bugs in an event-driven extension of Scheme.

1.5 Contributions

The contributions of this dissertation are twofold. We believe that both contributions demonstrate the feasibility of our vision of intelligent testing platforms. With respect to the state of the art on test smells, the contributions are published in two international conference papers (*i.e.,* [DBDNDR19a, DBDNDR19b]) and include:

- The design of an automated and static analysis approach to detect test smells including the transposition of six test smells introduced by Van Deursen *et al.* [vMBK01] for Java to Scala.
- The implementation of our approach for the testing framework ScalaTest in the open-source tool Socrates (**Sc**ala **Ra**dar for **Te**st **S**mells) to show its applicability. Its implementation provides a modular way to analyse tests for test smells and can be a foundation for further research.
- A survey with 14 professional Scala developers about test smells.
- A large-scale empirical study that analyses the diffusion of test smells in 164 open-source Scala systems hosted on Github which demonstrates the applicability of the approach.

With respect to the state of the art on resilience testing, the contributions are published in an international conference paper (*i.e.,* [DBDNDR20]) and include:

- The design of an automated and dynamic analysis approach to resilience testing which combines fault injection, delta debugging, and test amplification to find resilience defects in actor systems.
- The implementation of our approach for the actor framework Akka and the testing framework ScalaTest in the open-source tool Chaokka (**Chao**s in A**kka**) to show its applicability. Its implementation provides a modular way to define resilience analyses and can be a foundation for further research.
- Multiple optimizations including a pruning strategy based on the causality of actor events and actor-specific prioritization strategies.
- A study on actor systems with varying size and resilience defects to demonstrate the applicability of the approach.

1.6 Outline

The dissertation consists of 8 chapters and is organized as follows:

Introduction (Chapter 1) defines the context, problem, approach, and contributions of this dissertation.

The Scala Ecosystem (Chapter 2) presents the Scala ecosystem with its testing framework ScalaTest and its actor model framework Akka.

Socrates: Detecting Test Smells (Chapter 3) proposes Socrates as an automated and static analysis approach to detecting test smells. This chapter presents the implementation of our approach for the testing framework ScalaTest.

Empirical Study on Test Smells in the Scala Ecosystem (Chapter 4) presents our empirical study of 6 test smells in the Scala ecosystem and discusses the awareness of developers about test smells in ScalaTest.

State of The Art in Resilience Testing (Chapter 5) provides an overview of the related work on resilience testing and discusses our observations.

Chaokka: Resilience Testing (Chapter 6) proposes Chaokka as an automated and dynamic analysis approach to resilience testing. This chapter presents the implementation of our approach for the actor framework Akka and the testing framework ScalaTest.

Experimental Evaluation of Resilience Testing (Chapter 7) evaluates Chaokka and the efficacy of multiple resilience analyses that combine different exploration, pruning, and prioritization strategies.

Conclusion and Future Work (Chapter 8) summarizes this dissertation and discusses avenues for future work.

Chapter 2

The Scala Ecosystem

This chapter provides information about the Scala ecosystem and two of its frameworks. Our choice for situating this dissertation in the context of Scala is twofold. Not only does the ecosystem have its roots in the academic world [OAC+07], it also provides multiple frameworks for implementing and testing distributed systems that are increasingly adopted by the industry. In particular, we present the programming language Scala, the testing framework ScalaTest, and the actor model framework Akka.

Section 2.1 briefly presents the features of the programming language Scala. Next, Section 2.2 discusses the testing framework ScalaTest. We start by explaining how developers test their systems in theory to understand the concepts of software testing. Subsequently, we explain how developers write tests in practice using ScalaTest. Finally, Section 2.3 discusses the actor model framework Akka. We start by explaining the actor model in theory and describe how it is adopted by contemporary systems in practice. In particular, we discuss how actors and its corresponding mechanisms are implemented in Akka.

2.1 The Programming Language Scala

Scala[1] [OAC+07] is an industrial-strength programming language that has enjoyed a steady rise in popularity over the past years and has been adopted by many organizations such as Twitter, Paypal, and LinkedIn[2]. This statically-typed language combines object-oriented and functional programming in one concise, high-level language of which we discuss the different features that are important for understanding the remainder of this dissertation. Listing 2.1 provides the reader a first glimpse of Scala code.

```
class Stack[A] protected (array: Array[AnyRef], start: Int, end: Int)
  extends ArrayDeque[A](array, start, end)
    with IndexedSeqOps[A, Stack, Stack[A]]
    with StrictOptimizedSeqOps[A, Stack, Stack[A]]
    with IterableFactoryDefaults[A, Stack]
    with ArrayDequeOps[A, Stack, Stack[A]]
    with Cloneable[Stack[A]]
    with DefaultSerializable {

  def this(initialSize: Int = ArrayDeque.DefaultInitialSize) =
    this(ArrayDeque.alloc(initialSize), start = 0, end = 0)

  override def iterableFactory: SeqFactory[Stack] = Stack

  override protected[this] def stringPrefix = "Stack"

  def push(elem: A): this.type = prepend(elem)

  def push(elem1: A, elem2: A, elems: A*): this.type = {
    val k = elems.knownSize
    ensureSize(length + (if(k >= 0) k + 2 else 3))
    prepend(elem1).prepend(elem2).pushAll(elems)
  }

  def pushAll(elems: scala.collection.IterableOnce[A]): this.type =
    prependAll(elems match {
      case it: scala.collection.Seq[A] => it.view.reverse
      case it => IndexedSeq.from(it).view.reverse
    })

  def pop(): A = removeHead()

  def popAll(): scala.collection.Seq[A] = removeAllReverse()

  def popWhile(f: A => Boolean) = removeHeadWhile(f)

  final def top: A = head

  protected override def klone(): Stack[A] = {
    val bf = newSpecificBuilder
    bf ++= this
    bf.result()
  }

  override protected def ofArray(array: Array[AnyRef], end: Int) =
    new Stack(array, start = 0, end)
}
```

Listing 2.1: An excerpt of the `Stack` class provided by Scala.

[1] https://www.scala-lang.org

[2] https://www.lightbend.com/case-studies

The code above is the implementation of a stack data structure in Scala[3]. We will refer back to this code throughout the discussion of each feature below.

Interoperability. Scala runs on the Java Virtual Machine (jvm). This design choice enables developers to execute their Scala code on many different platforms and architectures such as Windows and Linux. This choice also facilitates the integration with popular programming languages such as Java. Additionally, Scala can also be used to build robust web applications by means of Scala.js which compiles Scala code to JavaScript. For several years, developers have been able to use Scala Native to build native applications that no longer run on the jvm.

Type Inference. One of the benefits of statically-typed languages is type checking at compile-time. However, this traditionally requires developers to declare the type of every variable which slows down development. Scala partially reduces this effort by providing a powerful type inference system. This system can often automatically determine types when the developer only provides a minimum amount of information. For example, type inference detects that `String` is the result type of the method `stringPrefix` (line 15).

Inheritance. Scala supports single inheritance but lacks support for multiple inheritance of classes. However, it features traits as a way to achieve multiple inheritance. This concept enables developers to inherit state and behaviour from multiple traits into a single class. Inherent to this concept is ambiguity: there may be situations where a particular feature is inherited from more than one parent class (*i.e.,* the diamond problem [BLS94]). This problem is resolved through linearization which puts classes and traits in a linear order to describe the chain of super calls at compile time. For example inheritance is used to extend from the class `ArrayDeque` (line 2) and to mix in behaviour from the traits (lines 3 to 8). The invariant type variable `A` is used throughout the code to support polymorphism.

Pattern Matching. Case classes are used to represent immutable structural data types in Scala. Specific to this kind of class is that they implement structural equality and have the ability to be deconstructed with pattern matching. For instance, Java does not have the ability to directly deconstruct constructor arguments of a type and match their value. Cumbersome code has to be written to do type casting (*i.e.,* applying the `instanceof` operator) and then comparing the values of the fields, while Scala enables you to do this directly through case classes and pattern matching. For example, the pattern matching functionality is used to support different behaviour of the method `pushAll` (line 26). It checks whether the parameter `elems` is of type `scala.collection.Seq[A]` (line 27) or any other type which requires conversion (line 28).

[3] https://github.com/scala/scala/blob/v2.13.2/src/library/scala/collection/mutable/Stack.scala

First-class functions. Functions are first-class values in Scala. Developers can leverage higher-order functions together with implicit conversions and case classes to create advanced domain-specific languages (DSL). Some of these characteristics have enabled the design of testing frameworks such as ScalaTest and ScalaCheck. Section 2.2 discusses the features of ScalaTest in more detail. For example, higher-order functions such as `popWhile` enable developers to pass around functions (line 35).

Concurrency models. Scala provides concurrency models such as shared memory, futures, and multithreading. An implementation of the actor model is also available as a framework under the name of Akka. This framework enables developers to build scalable, concurrent, and distributed systems which are some of the requirements for contemporary systems (*e.g.,* [KHA17, NW16]). Section 2.3 provides an in-depth discussion of this framework.

While the programming language Scala provides features that facilitate the development of software systems, developers remain uncertain whether their implementation works as expected. Software testing [Bec03, Bei03] has therefore become a standard practice in the development process.

2.2 The Testing Framework ScalaTest

The Scala ecosystem includes several testing frameworks of which ScalaTest[4] is the most popular testing framework [DBDNDR19a]. Compared to other testing frameworks, ScalaTest leverages some of Scala's advanced object-oriented and functional features to support various styles of defining tests and tests fixtures.

2.2.1 Writing Tests in Theory

We briefly describe the formal foundations of software testing to understand its concepts and make the connection between theory and practice.

$$
\begin{aligned}
p \in \mathit{Program} &= \text{a set of programs} \\
s \in \mathit{Specification} &= \text{a set of specifications} \\
t \in \mathit{Test} &= \text{a set of tests} \\
o \in \mathit{Oracle} &\subseteq \mathit{Test} \times \mathit{Program} \\
\mathit{corr} &\subseteq \mathit{Program} \times \mathit{Specification} \\
\mathit{corr}_t &\subseteq \mathit{Test} \times \mathit{Program} \times \mathit{Specification}
\end{aligned}
$$

Figure 2.1: A formal description of testing systems according to [SWH11].

[4] `http://www.scalatest.org`

Figure 2.1 shows a formal description of testing systems as defined by Staats *et al.* [SWH11]. The authors define a testing system as a tuple of the following 6 elements: In summary, a specification defines the intended behaviour that a given program is expected to adhere to. In an ideal world, the predicate *corr* would determine whether a specification holds for a given program. Unfortunately, the value of $corr(p, s)$ is generally not decidable because a complete specification might not be available or not all aspects of the program might be observable.

Therefore, two concepts are derived from the specification. Tests represent executions of the program with respect to the specification. Similarly, oracles approximate the specification and determine whether tests pass for a given program. The predicate $corr_t$ defines correctness with respect to a test. That is, $corr_t$ holds if and only if the specification holds for a program and a test. The outcome of this predicate is the outcome of test cases in testing frameworks. It is important to note that oracles and tests are likely to be approximations of a specification as summarized in Figure 2.2.

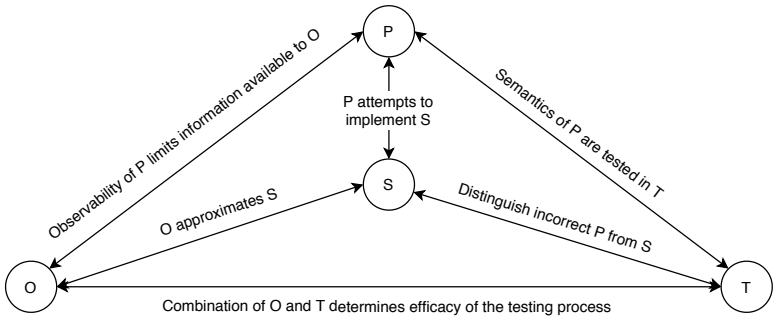

Figure 2.2: The relationship between programs (P), tests (T), oracles (O), and specifications (S). Adapted from [SWH11].

To understand why oracles approximate specifications, we start by the definition of a perfect oracle. Staats *et al.* [SWH11] define an oracle as perfect when it is both complete and sound. On the one hand, an oracle is complete when it is able to determine all violations for a given combination of test, program, and specification.

$$corr_t(t, p, s) \Rightarrow o(t, p)$$

However, oracles might be imperfect and may contain flaws in practice. For example, the method `push` does not violate the specification, but the oracle says otherwise because it misinterpreted its return value.

On the other hand, an oracle is sound when its judgements are always correct. Intuitively, a sound oracle fits the conventional wisdom about testing: we assume that the program is correct for the tests when the oracle says that they all pass.

$$o(t, p) \Rightarrow corr_t(t, p, s)$$

However, oracles might not be sound in practice [SWH11]. For example, the oracle might determine that the method `pop` returns an element from a non-empty stack such that the test passes. However, that element was not the first element on the stack which violates the specification. Ideally, one would have the perfect oracle that is both complete and sound.

$$\forall t, o : o(t, p) \Leftrightarrow corr_t(t, p, s)$$

Unfortunately, such an oracle (*i.e.,* the actual specification) is rarely available in practice. Therefore, developers have to manually compare the observed and expected behaviour by writing assertions that represent the oracle. That is, developers are approximating the specification as not everything might be observable. As a result, oracles may both fail to detect faults (*i.e.,* the oracle is incomplete) and may detect faults that do not exist (*i.e.,* the oracle is unsound). In the literature, the *test oracle problem* [BHM+14] represents the challenge of defining a correct oracle that distinguishes the correct behaviour from incorrect behaviour.

2.2.2 Writing Tests in Practice

To ease the burden of software testing, most programming language ecosystems include testing frameworks with different features. For instance, developers of Java and Python programs often use JUnit[5] and unittest[6] as their respective testing framework. These frameworks enable developers to write test suites that determine whether the observed behaviour is equal to the expected behaviour of a given system. In the next section, we discuss the features of the testing framework ScalaTest.

2.2.2.1 Testing Styles

Most testing frameworks provide developers with one specific *testing style* to write tests. For example, tests in JUnit are simply methods with an annotation, while tests in unittest are methods whose name starts with `test`. However, ScalaTest offers a spectrum of 8 different testing styles through an advanced domain-specific language. Developers are free to choose whatever style fits them and they can even use multiple styles in the same test suite. For example, Listing 2.2 on the next page shows the test class `StackSpec` which consists of two test cases (line 3–9 and line 11–16) and tests the class defined in Listing 2.1.

[5] https://junit.org

[6] https://docs.python.org/3/library/unittest.html

This class uses the `FlatSpec` style which dictates that tests are written in the form of *X should Y in*. The first test case uses assertions to compare the expected behaviour of `Stack` with the observed behaviour. For example, the first assertion (line 7) checks whether the result of popping the first element of the stack is the number 2. The second test case uses assertions to check whether an exception is thrown and whether the caught exception is indeed of type `NoSuchElementException`.

```
1 class StackSpec extends FlatSpec with Matchers {
2
3   "A Stack" should "pop values in last-in-first-out order" in {
4     val stack = new Stack[Int]
5     stack.push(1)
6     stack.push(2)
7     stack.pop() should be (2)
8     stack.pop() should be (1)
9   }
10
11  it should "throw NoSuchElementException when empty" in {
12    val emptyStack = new Stack[Int]
13    a [NoSuchElementException] should be thrownBy {
14      emptyStack.pop()
15    }
16  }
17 }
```

Listing 2.2: A test class for the `Stack` class consisting of two test cases.

2.2.2.2 Test Fixtures

Many tests often require the same set of data to execute the test. In terms of maintainability, it is therefore better to share the data across tests instead of duplicating them for each test. A *fixture* represents such data and can be as simple as a single variable, or as complex as an in-memory database. Test classes and test cases can then use the data defined in the fixtures. Typically, these fixtures are automatically initialized before the test and cleaned up after the test by the testing framework. In ScalaTest, fixtures can be defined in multiple ways such as first-class functions and traits. This results in a comprehensive set of features for defining, re-using, and composing test fixtures in a fine-grained manner. This is in contrast to JUnit and other testing frameworks where fixtures are applied to all test cases in a test, while ScalaTest also enables fixtures to be applied to single test cases. For example, we rewrote the code from Listing 2.2 such that it uses a fixture. The result is shown in Listing 2.3 where the fixture `StackFixture` is defined which provides a fresh instance of a stack. This particular fixture definition style is distinctive by its use of a trait (line 3) and anonymous class instance creation expressions (*e.g.,* `new StackFixture` (line 7)). The use of a trait provides the benefit that fixtures can be composed when necessary and improves maintainability in the long run. Each test case uses its own instance of the fixture such that the stack variable (line 4) is not shared between them. This avoids problems where one instance would be shared across multiple test cases and might result in flaky tests.

```
1  class StackSpec extends FlatSpec with Matchers {
2
3    trait StackFixture {
4      val stack = new Stack[Int]
5    }
6
7    "A Stack" should "pop values in LIFO order" in new StackFixture {
8      stack.push(1)
9      stack.push(2)
10     stack.pop() should be (2)
11     stack.pop() should be (1)
12   }
13
14   it should "throw NoSuchElementException" in new StackFixture {
15     a [NoSuchElementException] should be thrownBy {
16       stack.pop()
17     }
18   }
19 }
```

Listing 2.3: A test class for the `Stack` class consisting of two test cases that use the `StackFixture` fixture.

To conclude, we see that ScalaTest provides a wide variety of features which are not always present in other testing frameworks. This gives an opportunity to determine how these features of ScalaTest are used in practice and whether some of these affect the quality of existing tests. Analysing and refactoring existing tests is known to be one form of test amplification as discussed in the first chapter.

2.3 The Actor Model Framework Akka

Systems with a distributed architecture are built on top of services with different kinds of granularity. We observe that the actor model is increasingly used to implement these systems. The actor model is originally introduced by Hewitt *et al.* [HBS73] and later revised by Agha [Agh85] and differs from shared-memory models in several ways. This concurrency model presents actors as concurrent processes with a mailbox. An actor can do 3 things: process messages, spawn actors, and change their own behaviour and state. As there is no shared state in this model, actors need to exchange asynchronous messages to access or update each other's state. These messages are placed in the mailbox of the receiving actor and are processed in first-in-first-out (FIFO) order by default. These aspects are shown in Figure 2.3. The processing of one message by an actor defines a turn and each turn happens atomically (*i.e.,* there is no interleaving of other turns).

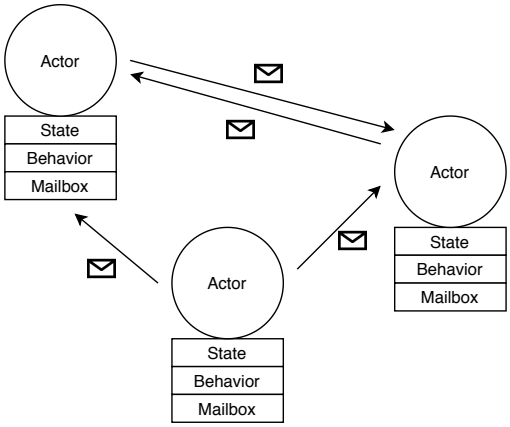

Figure 2.3: Each actor has a state, a behaviour, and a mailbox. Actors solely communicate with each other through asynchronous messages that are queued in their mailboxes. Every message is processed atomically in a turn and can modify the state and behaviour of an actor.

While there exist many frameworks and languages that adopted the actor model, AKKA[7] is among the most popular implementations of the actor model [HBS73]. It enables developers to build powerful reactive, concurrent, and distributed applications on the JVM. AKKA has enjoyed adoption[8] by large organizations such as PAYPAL and LINKEDIN which rely on AKKA to scale their services to the many concurrent requests of their users. Moreover, AKKA received attention in the form of books on distributed systems [KHA17, NW16] as well as academic papers [TPLJ13, TDJ13, SPH15, IS14].

Beyond implementing the original actor model, AKKA also provides additional features such as clustering, supervision mechanisms, actor persistence, and message delivery guarantees to facilitate the implementation of large-scale resilient systems.

2.3.1 Actors

AKKA provides a blueprint of an actor through the trait `akka.actor.Actor`. Listing 2.4 shows the class `Accumulator` that inherits from this trait and represents a service that accumulates numbers. The trait enforces the implementation of the method `receive` (line 6). This method returns a partial function that specifies the actor behaviour (*i.e.,* how and which messages are processed). The actor understands two types of messages: `CountCommand` (line 7) and `String` (line 10). While the former could match multiple instance of `CountCommand` (*e.g.,* different `amount`), the latter will only exactly match when the value is `"result"`.

[7] `https://akka.io`

[8] `https://www.lightbend.com/case-studies`

The actor mutates the state by changing the variable `count` to the value of `amount` and sends a `Confirm` message back in response to the former, while it will send the value of `count` to the sender in response to the latter. The `!` operator is used to send a message to an actor (*e.g.,* line 9 and 11), while `sender()` returns the actor which has sent the current message. Note that messages are sent to location-transparent addresses represented by the type `akka.actor.ActorRef` (*e.g.,* as returned by `sender()`). This transparency facilitates development as it does not matter whether the actor behind this address is local or remote. Akka will do the necessary actions to make sure the message arrives at the destination.

```
1 import akka.actor.Actor
2
3 class Accumulator extends Actor {
4   var count: Int = 0
5
6   override def receive: Receive = {
7     case CountCommand(id: Long, amount: Int) =>
8       count = count + amount
9       sender() ! Confirm(id)
10    case "result" =>
11      sender() ! count
12  }
13 }
```

Listing 2.4: The `Accumulator` actor in Akka.

2.3.2 Persistent Actors

One of the problems of actors is that they lose their state when restarted or migrated between clusters. To be resilient to restarts, Akka features persistent actors which follow the principles of Event Sourcing [Fow05]. A *command* is a message that requires the receiving actor to undertake an action. Whenever this command is valid, the actor will persist an *event* and execute the action once persisted. To reconstruct its original state, all persisted events are replayed whenever an actor is restarted (*e.g.,* upon a failure).

Akka provides the trait `akka.persistence.PersistentActor` as a blueprint of a persistent actor. Listing 2.5 shows the class `GuaranteedDeliveryActor` that implements this trait and represents a service that persists its state with event sourcing. This trait enforces the implementation of the methods `receiveCommand`, `receiveRecover`, and `persistenceId`. The method `receiveCommand` (line 13) is similar to the method `receive` as it specifies the behaviour of the actor. The method `receiveRecover` (line 18) defines how persisted events are replayed and the method `persistenceId` (line 27) defines how the entity is uniquely identified in the journal where events are persisted to and read from. To persist an event, a developer must call `persist` (line 14–15) with the event to be persisted and a callback (*i.e.,* `updateState`) to be executed when the given event has been persisted asynchronously.

```
1  import akka.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 ConfirmEvent(id : Long) extends Event
8  case class Confirm(id : Long)
9
10 class GuaranteedDeliveryActor(ref: ActorRef)
11   extends PersistentActor with AtLeastOnceDelivery {
12
13   override def receiveCommand: Receive = {
14     case Plus(a) => persist(PlusEvent(a))(updateState)
15     case Confirm(id) => persist(ConfirmEvent(id))(updateState)
16   }
17
18   override def receiveRecover: Receive = {
19     case e : Event => updateState(e)
20   }
21
22   def updateState(e: Event): Any = e match {
23     case PlusEvent(a) => deliver(ref.path)(id => CountCommand(id, a))
24     case ConfirmEvent(id) => confirmDelivery(id)
25   }
26
27   override def persistenceId: String = "actor-1"
28 }
```

Listing 2.5: A persistent actor in Akka.

2.3.3 Guaranteed Message Delivery

Recall that the expression `actor ! message` is used to send a message to an actor. By default, this message is sent with at-most-once message delivery semantics. This means that the message is sent only once and partial network failures or a crashes of the receiving side might result in a loss of the message. To be resilient to message loss, messages can be sent with at-least-once message delivery guarantees. Akka provides a blueprint of such a mechanism through the trait `AtLeastOnceDelivery` which provides the methods `deliver` and `confirmDelivery`. The method `deliver` (Listing 2.5, line 23) accepts two arguments which include the destination address of the actor and a callback. This callback is called with a unique identifier generated by Akka and returns the message `CountCommand` that has to be sent. The framework will periodically resend this message until an acknowledgement with that identifier has been registered through a call to the method `confirmDelivery` with the corresponding identifier. The actor that has to confirm a message sends a `Confirm` message back with that identifier `id` (Listing 2.4, line 9). The handler of the receiving actor for `Confirm` messages then calls method `confirmDelivery` to confirm the delivery (Listing 2.5, line 24).

2.3.4 The TestKit Library

Akka features the testing library TestKit[9] which seamlessly integrates with ScalaTest. It provides features such as timeout-aware assertions and means to collect specific messages to facilitate the testing of actor systems. Listing 2.6 shows a test class that uses features of TestKit and ScalaTest to check whether the actors from the previous sections are implemented correctly.

```
1  import akka.actor.{ActorSystem, Props}
2  import akka.testkit.{ImplicitSender, TestKit}
3  import org.scalatest.{BeforeAndAfterAll, FlatSpecLike, Matchers}
4  import scala.concurrent.duration._
5
6  class ExampleTest() extends TestKit(ActorSystem("SystemUnderTest"))
7    with FlatSpecLike with ImplicitSender
8    with Matchers with BeforeAndAfterAll {
9
10   "Accumulator" must "correctly accumulate numbers" in {
11     val a = system.actorOf(Props[Accumulator], name="A")
12     val props = Props(new GuaranteedDeliveryActor(a))
13     val actor = system.actorOf(props, name="GDA")
14
15     for (i <- 1 to 10) { actor ! Plus(i) }
16     Thread.sleep(2000)
17     a ! "result"
18
19     expectMsg((1 to 10).sum)
20   }
21
22   override def afterAll: Unit = {
23     TestKit.shutdownActorSystem(system, 5 * 60 seconds)
24   }
25 }
```

Listing 2.6: An integration test for both actors.

First, both actors are instantiated by calls to the method `system.actorOf` (lines 12–14) with an instance of the `Props` class. Note that this is different to how normal classes would be instantiated in Scala. Next, 10 `Plus` messages are sent to `GuaranteedDeliveryActor` (line 16). After waiting for 2 seconds, the test actor sends a message `"result"` to `Accumulator` to retrieve the total sum (line 18). The call to `expectMsg` (line 20) will wait by default for 3 seconds to receive a reply. When a reply is received, it will compare its payload to the expected result of `(1 to 10).sum`. Otherwise, an instance of `TimeoutException` is thrown.

2.4 Conclusion

This chapter presented the Scala ecosystem. First, we discussed the programming language Scala and briefly summarized its features. We consider this ecosystem a good fit for this dissertation as Scala has roots in the academic world and has been adopted by the industry over the past years. Second, we discussed the most popular testing framework ScalaTest. This testing framework provides different styles of testing, as well as different ways of defining fixtures.

[9] https://doc.akka.io/docs/akka/current/testing.html

These features are less available in existing testing frameworks which might affect the quality of tests. The next chapter therefore investigates the quality of tests in the Scala ecosystem. Finally, we discussed the actor model framework Akka. This framework provides the necessary means to build resilient and distributed complex systems. These systems will play a role in our resilience testing approach discussed in Chapter 6.

Chapter 3

Socrates: A Static Analysis Approach to Detecting Test Smells

As indicated in the first chapter, the majority of tests are still manually implemented. This results in tests with varying quality based on the effort and expertise of developers. However, studies have shown that low test quality can affect comprehensibility and maintainability in the long run. Testing frameworks should therefore help developers with the implementation of high quality tests, especially when they are used as input to test amplification approaches.

One way to assess test quality is by searching for so-called test smells. These test smells are poor design choices made by developers during the implementation of tests. Their absence should therefore indicate a high test quality. However, developers only have a limited number of tools available which can only detect smells in the Java ecosystem. Therefore, we explore the concept of test smells in the Scala ecosystem.

First, Section 3.1 presents the notion of a test smell and lists our motivations for this research by showing the negative impact of test smells. Section 3.2 then provides an overview of existing empirical studies and discusses the current observations about test smells. As concluded from the state of the art, the majority of the only studies detect test smells in the Java ecosystem. Next, Section 3.3 presents an overview of Socrates which is our static analysis approach to detecting test smells in ScalaTest. The next chapter will use Socrates to conduct an empirical study of test smells in the Scala ecosystem. This is followed by Section 3.4 which describes our unique test smell detection method. Section 3.5 then provides information about the test smells including their definitions, their transposed detection methods, and their manual refactoring methods to eliminate each smell. Finally, Section 3.6 briefly discusses the usage of Socrates and its possibilities to be extended with support for additional smells.

3.1 Motivation

As developers still write code manually, many poor design choices can be made during development. *Code smells* [Fow18] indicate bad characteristics of the system code, whereas *test smells* indicate potential problems with test code. The seminal paper of Van Deursen *et al.* [vMBK01] introduces the concept of test smells. These smells indicate poor design choices made by developers during the implementation of test cases, tests fixtures, or even complete test suites. The authors define a set of 11 test smells and their corresponding refactoring method.

Listing 3.1 shows a test case that exhibits a test smell related to asserting the `toString` method. This particular test case suffers from the smell Sensitive Equality as its assertion depends on the result of calling the method `toString`. We provide thorough explanations and examples later in this chapter.

```
1 "A recipe" should "have 100gr of Chocolate as only ingredient" in {
2   val il = List(Ingredient("Chocolate", 100))
3   val recipe = Recipe("Chocolate Cookies", il)
4   val result = "Recipe(Chocolate Cookies,List(Ingredient(Chocolate,100)))"
5
6   assert(recipe.toString() == result, "...")
7 }
```

Listing 3.1: An illustrative example of a test smell.

Our motivation to detect test smells in the Scala ecosystem is twofold and follows from the observations of our literature study discussed in Section 3.2.

3.1.1 Limited Context Diversity and Tool Support

Our literature study indicates that most empirical research on test smells is limited to the Java ecosystem where JUnit is the most popular testing framework. As a result, the current knowledge about test smells might be skewed. Furthermore, developers have limited access to open-source tool support to detect test smells which leaves ample room for low-quality tests.

However, having high-quality tests should be a priority as it facilitates maintenance and test amplification. For example, Abdi *et al.* [ARD19] hint that it is better to discard tests with low readability in the Smalltalk ecosystem before using them as input for test amplification. This is an important motivation as we will later amplify existing tests in Chapter 6.

3.1.2 Negative Impact on Software Aspects

Our literature study shows that test smells have a negative impact on different aspects of the system under test, as well as the tests themselves. In total, we found 5 publications and briefly discuss their observations.

1. **Impact on Defect Proneness.** In a study of 221 releases of 10 JAVA systems, Spadini *et al.* [SPZ+18] observed that tests with smells are more prone to changes and defects. Moreover, when tests exhibit test smells it also makes production code prone to defects.

2. **Impact on Flakiness.** In a study of 19532 JUNIT tests belonging to 18 JAVA systems, Palomba *et al.* [PZ17, PZ19][1] determined that 54% of the flaky tests can be attributed to the characteristics of the test smells. Refactoring test smells was also determined as an effective strategy to fix more than half of the flaky tests.

3. **Impact on Maintainability.** In a study of 987 JUNIT classes belonging to 25 open-source and 2 industrial JAVA systems, Bavota *et al.* [BQO+15] determine that test smells have a strong negative impact on program comprehension and maintenance.

4. **Impact on Production Code.** From a survey with 19 JAVA developers, Tufano *et al.* [TPB+16] concluded that test smells are not perceived as actual design problems. In a study of 152 open-source projects belonging to ecosystems APACHE and ECLIPSE, the authors also found a relationship between smells in test and production code.

Despite that these studies and surveys show the negative impact of test smells, the majority of developers remain unaware of test smells [TPB+16]. The next section presents the details of our literature study and was used to conclude these observations.

3.2 Literature Study

Before investigating test smells in the SCALA ecosystem, we explore the literature to identify the current knowledge about tests smells. Van Deursen *et al.* [vMBK01] provides the most prominent catalogue of test smells used in the literature. We therefore start our exploration with that seminal paper and do not consider others such as those of Meszaros [Mes07], Garousi *et al.* [GK18], or Peruma [Per18]. We explore all publications that cite the work of Van Deursen *et al.* and investigate what kind of smells they investigate and in which context this is done. Our exploration resulted in a list of 9 publications which includes the most relevant studies. We summarize each approach with a table that consists of the programming language, the testing framework, and the test smells that were considered in the study. A summary of our literature study is provided in the next section.

[1] Both papers were retracted because of an over-approximation of its results. However, the study still presents evidence for the impact of test smells on test flakiness.

Van Rompaey *et al.*

Van Rompaey *et al.* [VRDBDR07] define metrics-based detection methods for 2 test smells: General Fixture and Eager Test. They argue that the traditional means for test quality assurance (*i.e.,* human reviewing) is not a reliable means for test smell detection. Their study on Java systems with tests written in JUnit show that a metrics-based approach can be a more reliable detection mechanism. However, they did not consider metrics for other known test smells and indicate that the predictive power of their method is limited.

Java	JUnit	General Fixture, Eager Test

Table 3.1: Summary of Van Rompaey *et al.* [VRDBDR07]

Greiler *et al.*

Greiler *et al.* [GvS13, GZvS13] argue that there is a lack of tool support for developers to analyse and adjust test fixtures. To this end, the authors present TestHound as a tool that provides reports on test smells and recommends refactorings to eliminate them. The tool is able to analyse tests written in TestNG and JUnit. Their tool is only able to find the test smell called General Fixture (including variants) in large Java systems. Moreover, the result of a questionnaire with 13 developers shows that tool support is beneficial in understanding and adjusting test fixtures. However, the scope of their work is limited to General Fixture and it is unclear to what extent other test smells occurred in the analysed systems.

Java	JUnit, TestNG	General Fixture

Table 3.2: Summary of Greiler *et al.* [GvS13, GZvS13]

Bavota *et al.*

Bavota *et al.* [BQO+12, BQO+15] investigate the distribution of 11 test smells in an empirical study on Java open-source systems with tests written in JUnit. They present the first empirical evidence highlighting that test smells occur frequently. In particular, they demonstrate that 82% of the test suites were affected by at least one test smell. However, their tool sacrifices precision by using simple rules that overestimate the presence of test smells and the tool is not publicly available. Besides the empirical study, they conducted an survey with 20 students to assess the impact on software comprehension. Their results show that the majority of test smells have a strong negative impact and show the need for automated detection of test smells to improve software quality.

Java	JUnit	Eager Test, Lazy Test, Mystery Guest, Assertion Roulette, General Fixture, Sensitive Equality, Test Duplication, Test Run War, For Testers Only, Resource Optimism, Indirect Testing

Table 3.3: Summary of Bavota *et al.* [BQO+12, BQO+15]

Tufano *et al.*

Tufano *et al.* [TPB+16] investigate the perception of 5 test smells in a study with 19 developers. Their results indicate that developers lack knowledge about test smells and highlight the need for tool support.

Similar to Bavota *et al.*, they conduct an empirical study on 152 open-source Java systems to analyse the presence of test smells and their association with code smells. In particular, they use the tool of Bavota *et al.* [BQO+15] to detect test smells which might overestimate the presence of test smells. The results demonstrate that test smells are introduced upon the first commit and stay in the system for a long time (80% is not fixed after 1000 days). Additionally, they found several associations with code smells for the test smells Eager Test and Assertion Roulette. These observations only amplify the need for automated tool support, as already indicated by earlier works. One of the limitations of this work is that they did not consider the prevalence of test smells in an empirical study. This might have given another perspective as their data set is different from other studies.

Java	JUnit	Assertion Roulette, Eager Test, General Fixture, Mystery Guest, Sensitive Equality

Table 3.4: Summary of Tufano *et al.* [TPB+16]

Palomba *et al.*

Palomba *et al.* [PDNP+16] conducted an empirical study to understand the distribution of 8 test smells in 110 open-source Java systems. The main difference between this study and the previous ones is that they analyse JUnit tests that were automatically generated using EvoSuite [FA11]. Similar to Tufano *et al.*, they also investigate correlations of test smells with both other test smells and structural metrics (*e.g.,* number of classes or lines of code). Similar to previous works, the authors use the tool of Bavota *et al.* [BQO+15]. Their results confirm what is currently known in the state of the art: a high distribution where 83% of JUnit classes are affected by at least one test smell. Moreover, they found that the majority of the smells have strong positive correlations with structural

characteristics such as the number of classes and their lines of code. However, these results should be interpreted with caution as the distribution is based on generated tests —which might not be representative for human-written tests.

Java	JUnit	Eager Test, Resource Optimism, Mystery Guest, Assertion Roulette, General Fixture, Sensitive Equality, Test Duplication, Test Run War, For Testers Only, Indirect Testing

Table 3.5: Summary of Palomba *et al.* [PDNP+16]

Spadini *et al.*

Spadini *et al.* [SPZ+18] analysed 10 Java software systems to assess the association between the presence of 6 test smells and change- and defect-proneness of both test and production code. This work is also based on the test smell detection tool of Bavota *et al.* [BQO+15] but they did not include numbers about test smell distributions. However, the results confirm the negative impact of test smells on software quality attributes such as maintainability. Similar findings were reported by Palomba *et al.* [PZ17]. This work demonstrates that automated test smell detection tools do not only benefit the quality of test suites, but also the overall quality of the system as a whole.

Java	JUnit	Eager Test, Resource Optimism, Mystery Guest, Assertion Roulette, Indirect Testing, Sensitive Equality

Table 3.6: Summary of Spadini *et al.* [SPZ+18]

In a recent publication, Spadini *et al.* [SSO+20] analysed 1489 open-source projects to determine severity thresholds of test smells and conducted a study with 31 developers to evaluate the thresholds. This work uses the test smell detection tool TsDetect from Peruma [Per18] and the results therefore include additional smells. The results indicate that their severity thresholds are in line with how developers perceive test smells. Additionally, the participants agree that test smells have an impact on the maintainability of a test suite.

Java	JUnit	Assertion Roulette, Empty Test, Conditional Test Logic, General Fixture, Mystery Guest, Sleepy Test, Eager Test, Ignored Test, Resource Optimism, Magic Number Test, Verbose Test

Table 3.7: Summary of Spadini *et al.* [SSO+20]

3.2.1 Summary

Table 3.8 summarizes our exploration of the state of the art in test smells, including our approach at the bottom of the table. The results show that existing studies analyse a similar set of test smells, but also that they only targeted test smells in the Java ecosystem. On the one hand, that is because several studies use the same tool of Bavota *et al.* [BQO+15]. On the other hand, the majority of these test smells have clear definitions which makes them uniformly detectable.

	General Fixture	Mystery Guest	Resource Optimism	Eager Test	Lazy Test	Assertion Roulette	Sensitive Equality
Van Rompaey *et al.* [VRDBDR07]	✓			✓			
Greiler *et al.* [GvS13]	✓						
Greiler *et al.* [GZvS13]	✓						
Bavota *et al.* [BQO+12]	✓	✓	✓	✓	✓	✓	✓
Bavota *et al.* [BQO+15]	✓	✓	✓	✓	✓	✓	✓
Tufano *et al.* [TPB+16]	✓	✓		✓		✓	✓
Palomba *et al.* [PDNP+16]		✓	✓	✓		✓	✓
Spadini *et al.* [SPZ+18]		✓	✓	✓		✓	✓
Spadini *et al.* [SSO+20]	✓	✓	✓	✓		✓	
De Bleser *et al.* [DBDNDR19a]	✓	✓		✓	✓	✓	✓

Table 3.8: A summary of the test smells discussed in each study.

Note that we did not include the test smells Test Duplication (td), Test Run War (trw), For Testers Only (fto), and Indirect Testing (it) which were investigated in [BQO+12], [BQO+15], and [TPB+16] (except trw). We excluded these because we consider td to be defined overly general, trw to be too time-consuming to compute since there is an exponential number of possible orderings of tests, and both fto and it to be less important smells since we did not find them mentioned in other work. Additionally, some of these studies (*i.e.,* [SSO+20]) analyse test smells which were only proposed recently and are therefore excluded from the table. We leave their static detection for future work. The next section presents an overview of our approach to detecting test smells in the Scala ecosystem.

3.3 Overview of the Approach

As concluded from our literature study, there is a lack of automated tool support to detect test smells. Manually inspecting systems to find test smells is too time-consuming and error-prone. To this end, we present SOCRATES: **Sc**ala **Ra**dar for **Te**st **S**mells. SOCRATES statically detects all test smells enumerated in Table 3.8, except RESOURCE OPTIMISM which requires a dynamic analysis. We consider this set as a good choice since they are considered by the majority of existing studies and include the smells with the highest observed diffusion [TPB+16, PDNP+16]. Figure 3.1 shows the architecture of CHAOKKA. We discuss each step of the process in detail below.

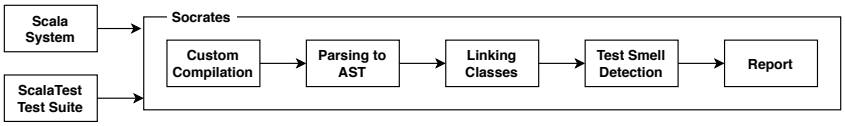

Figure 3.1: The architecture of SOCRATES.

In essence, SOCRATES works in several steps as discussed below:

1. **Input.** SOCRATES requires a SCALA system and its test suite written with the testing framework SCALATEST. Additionally, the project should use SBT as build tool.

2. **Compilation.** It automatically compiles the project using the SEMANTICDB[2] compiler plugin for SBT which exposes the semantic information maintained by the compiler (*i.e.,* symbol and type resolution),

3. **Parsing.** Next, it parses the test classes to Abstract Syntax Trees (ASTs), determines the used testing style for each test class, and collects all test cases within the test class.

4. **Linking.** The detection of test smells requires knowledge of the class under test. Therefore, we try to link the production class to each test class.

5. **Detection.** Finally, SOCRATES uses the gathered information of each test case (*i.e.,* syntactic and semantic information, as well as linked production class) to detect test smells.

6. **Report.** The output is shown through a user interface which indicates the absence or presence of each test smell for the collected test cases and test classes.

[2] https://scalameta.org/docs/semanticdb/guide.html

3.4 Syntactic and Semantic Information

While state-of-the-art approaches either rely on textual of syntactical information, SOCRATES also relies on semantic information such as types and symbols. We illustrate the difference between each type of information by means of the source code shown in Listing 3.2. This examples consists of a SCALA object that defines a `main` method which calls `println`.

```
1 object Test {
2   def main(args: Array[String]): Unit = {
3     println("hello world")
4   }
5 }
```

Listing 3.2: The object `Test` with its method `main`.

3.4.1 Information Extraction

First, SOCRATES starts by extracting syntactical information. Figure 3.2 shows the abstract syntax tree from Listing 3.2.

```
1  Defn.Object(
2    Nil,
3    Term.Name("Test"),
4    Template(
5      Nil,
6      Nil,
7      Self(Name(""), None),
8      List(
9        Defn.Def(
10         Nil,
11         Term.Name("main"),
12         Nil,
13         List(
14           List(
15             Term.Param(
16               Nil,
17               Term.Name("args"),
18               Some(
19                 Type.Apply(Type.Name("Array"), List(Type.Name("String")))
20               ),
21               None
22             )
23           )
24         ),
25         Some(Type.Name("Unit")),
26         Term.Block(
27           List(
28             Term.Apply(
29               Term.Name("println"),
30               List(Lit.String("hello world"))
31             )
32           )
33         )
34       )
35     )
36   )
37 )
```

Figure 3.2: The abstract syntax tree of the object `Test`.

This representation enables SOCRATES to find each part of the source code. For example, `Defn.Def` (line 9) represents the syntactic definition of the method `main`, while `Term.Apply` (line 28) represents the function application of `println` on the string `"hello world"` (line 30). Now, let's assume that we want to collect all invocations of the method `println`. This can be done by searching for all nodes that match the following pattern:

```
Term.Apply(Term.Name("println"), List(Lit.String(_)))
```

However, it might be that these patterns represent calls to a user-defined or imported `println` method, while we only want those patterns that call the predefined `println` method from the standard library of SCALA. This is where the semantic information comes into play. Figure 3.3 shows the generated SEMANTICDB for the illustrative example in Listing 3.2.

```
1  Summary:
2  Schema => SemanticDB v4
3  Uri => Test.scala
4  Text => empty
5  Language => Scala
6  Symbols => 3 entries
7  Occurrences => 7 entries
8
9  Symbols:
10 _empty_/Test. => final object Test extends AnyRef { +1 decls }
11 _empty_/Test.main(). => method main(args: Array[String]): Unit
12 _empty_/Test.main().(args) => param args: Array[String]
13
14 Occurrences:
15 [0:7..0:11) <= _empty_/Test.
16 [1:6..1:10) <= _empty_/Test.main().
17 [1:11..1:15) <= _empty_/Test.main().(args)
18 [1:17..1:22) => scala/Array#
19 [1:23..1:29) => scala/Predef.String#
20 [1:33..1:37) => scala/Unit#
21 [2:4..2:11) => scala/Predef.println(+1).
```

Figure 3.3: The SEMANTICDB payload of the object `Test`.

This document has two main sections: symbols and occurrences. We can find the symbol of a specific AST node by looking up its position in the occurrences, the returned value can then be looked up in the symbols section, which returns all necessary information about the symbol[3]. For example, `println` occurs at position `[2:4..2:11)` (line 21) which refers to symbol `scala/Predef.println(+1)`. As a result, patterns can be collected with 100% precision.

Among other things, SOCRATES also uses this semantic information to build the class hierarchy of the system. To do so, it collects the type, the fully qualified name and the parents of each class and links them together by traversing the inheritance chain. This hierarchy will be used in the next phase to determine which test classes inherit from a SCALATEST class.

[3] https://scalameta.org/docs/semanticdb/specification.html

3.4.2 Identification of Test Classes

SOCRATES only analyses the necessary files and detects these by assuming that the system follows the directory structure shown in Figure 3.4. The tool considers all classes in `src/main/scala` as production classes and `src/test/scala` as test classes.

```
src/
  main/
    resources/
       <files to include in main jar here>
    scala/
       <main Scala sources>
    java/
       <main Java sources>
  test/
    resources
       <files to include in test jar here>
    scala/
       <test Scala sources>
    java/
       <test Java sources>
```

Figure 3.4: The typical directory structure of system built with SBT.

Subsequently, we filter the test classes to only keep those that really represent a test class, and not simply an auxiliary class. We use the computed class hierarchy from the previous phase to check whether the test class inherits from a class defined by SCALATEST. The tool supports the majority of the testing styles from SCALATEST: `FlatSpec`, `FunSuite`, `WordSpec`, `FunSpec`, `FreeSpec`, `FeatureSpec`, `PropSpec`, and `RefSpec`. The tool also checks several variants of these styles: `{s}Like`, `Async{s}` and `Async{s}Like` where `{s}` is a style. For example, the tool will identify the following four styles of `FunSpec`: `FunSpec`, `FunSpecLike`, `AsyncFunSpec` and `AsyncFunSpecLike`.

3.4.3 Linking Test Classes to Production Classes

Some test smells such as LAZY TEST and EAGER TEST require information about the production class to correctly identify the smell. Our tool adopts a widely-used naming convention to determine the link between production and test class. The algorithm works as follows:

- let N be the fully qualified name of a test class
- determine whether N contains one of the following suffixes: `Test`, `Tests`, `TC`, `TestCase`, `Spec`, `Specification`, `Suite`, `Prop`
- if so, extract the suffix from N and return the class with the same name and package
- if not, then we are unable to determine the production class

For example, the suffix `Tests` is determined when the class `ShoppingCartTests` resides in the package `be.vub.soft`. This results in a production class that should exist in the package `be.vub.soft` and have the name `ShoppingCart`. We link a production class to a test class if and only if there exists a production class with the same name and which resides in the same package, other test classes are discarded. The test classes that adhere to these requirements are further analysed to determine the test cases and fixtures.

After collecting all necessary information, the final step of SOCRATES consists in detecting test smells in test classes and test cases. Some smells are specific to test classes (*e.g.,* GENERAL FIXTURE), while most of them are specific to test cases (*e.g.,* ASSERTION ROULETTE). Each detection method uses both AST and semantic information to detect test smells. We present the detection methods of each of the 6 test smells in the next section.

3.5 Static Detection Methods for Test Smells

For each test smell, we transpose the original definition from Van Deursen *et al.* to the context of SCALA. We provide an example instance, a static detection method, and a manual refactoring method to eliminate the smell and its negative impact. We will describe each test smell by means of the same `Ingredient` and `Recipe` classes shown in Listing 3.3.

```
1 case class Ingredient(name: String, weight: Int)
2 case class Recipe(name: String, ingredients: List[Ingredient]) {
3   def names: List[String] = ingredients.map(_.name)
4   def hasIngredients: Boolean = ingredients.nonEmpty
5 }
6 object Recipe {
7   def fromFile(file: BufferedSource): Recipe = ...
8 }
```

Listing 3.3: The illustrative SCALA classes under test.

3.5.1 Assertion Roulette (AR)

Definition. A test case that contains more than one assertion of which at least one does not provide a reason for assertion failure. In case the test fails, this test smell encumbers identifying which assertion failed and the reason why. Listing 3.4 depicts a SCALA example of this test smell and its resolution.

Detection Method. The detection method for this test smell amounts to finding all assertions in a test case and verifying that each assertion is provided with an additional argument.

Refactoring. SCALATEST complements the familiar `assert` with the more expressive `assertResult`, `assertThrows`, `cancel`, `assume`, and `fail`. Each takes the assertion's failure explanation as an optional argument. The framework also provides the `withClue` construct which uses its given parameter as the failure explanation for all of the assertions in its scope.

Assertion Roulette can therefore be resolved by either (i) providing a description or clue as an additional argument to `assert` and its variants, or by (ii) wrapping the assertions inside a `withClue` block.

```
"A recipe with one ingredient" should "have names=List('Chocolate')" in {
  val il = List(Ingredient("Chocolate", 100))
  val recipe = Recipe("Chocolate Cookies", il)
  assert(recipe.names.head == "Chocolate")
  assert(recipe.names.size == 1)
}
```

```
"A recipe with one ingredient" should "have names=List('Chocolate')" in {
  val il = List(Ingredient("Chocolate", 100))
  val recipe = Recipe("Chocolate Cookies", il)

  assert(recipe.names.head == "Chocolate",
    s"The name of the ingredient was ${recipe.names.head}")

  withClue(s"The size of the 'names' was ${recipe.names.size}") {
    assert(recipe.names.size == 1)
  }
}
```

Listing 3.4: An example of Assertion Roulette and its refactoring.

3.5.2 Eager Test (ET)

Definition. A test case that checks or uses more than one method of the class under test. Since its introduction [vMBK01], this smell has been somewhat broadly defined. It is left to interpretation which method calls count towards the maximum. Either all methods invoked on the class under test could count, or only the methods invoked on the same instance under test, or only the methods of which the return value is eventually used within an assertion. We have opted for the first interpretation in this study, but all others are valid too.

Detection Method. Our method to detect this smell consists of three steps: (i) identifying the class under test and collecting all of its methods; (ii) collecting the set of methods called from the test case; (iii) computing the size of the intersection of the outcomes (i) and (ii). If the intersection is larger than 1, more than one method is being tested by the test case.

Refactoring. Splitting the test into test cases that each test a single method of the class under test. For the example depicted in Listing 3.5, we opt to use a fixture to avoid duplicating the recipe object in each test case.

```
"The recipe" should "have two ingredients" in {
  val il = List(Ingredient("Cookie", 100), Ingredient("Milk", 200))
  val recipe = Recipe("Cookies and Milk", il)
  assert(recipe.hasIngredients, "...")
  assert(recipe.names.equals(List("Cookie", "Milk")), "...")
}
```

```
1 def fixture = new {
2   val il = List(Ingredient("Cookie", 100), Ingredient("Milk", 200))
3   val recipe = Recipe("Cookies and Milk", il)
4 }
5
6 "The recipe" should "have two ingredients" in {
7   val f = fixture
8   assert(f.recipe.names.equals(List("Cookie", "Milk")), "...")
9 }
10
11 "The recipe" should "have ingredients" in {
12   val f = fixture
13   assert(f.recipe.hasIngredients, "...")
14 }
```

Listing 3.5: An example of Eager Test and its refactoring.

3.5.3 General Fixture (GF)

Definition. A test fixture that is too general. Ideally, test cases should use all the fields provided by their fixture. This might be difficult to uphold when the fixture is shared by several test cases. ScalaTest features no less than 4 different means for defining and sharing fixtures. The detection methods and refactorings for this smell are four-fold too.

Type I - Global Fixture: Detection Method. ScalaTest has support to define fixtures through the traits `BeforeAndAfter` or `BeforeAndAfterEach`. These traits respectively enable providing code, as the value for a by-name parameter to methods `before` or `after`, that must run before or after the test or each test case of the test. This code typically initializes the fields used within the test or test case. The detection of this smell requires three steps: (i) identify a test class that mixes in one of these traits and calls their `before` or `after` methods, (ii) collect the set of fields assigned in the code provided as an argument to these methods, and (iii) determine whether a test case of the class does not reference one of the assigned fields.

Type I - Global Fixture: Refactoring. The test cases defined in Listing 3.6 share none of the fields defined in their common fixture. This instance of the smell can be eliminated by removing trait `BeforeAndAfter` from the test class, and by demoting the fields referenced in the argument to method `before` to local, immutable variables in the appropriate test case. In case groups of test cases each use a different group of fields, and the groups should remain together, ScalaTest supports defining a local fixture per individual test case rather than a global fixture for the entire test class —which can be reused as illustrated in the remainder of this section.

```
1 class RecipeTestSuite extends FlatSpec with BeforeAndAfter {
2   var emptyRecipe: Recipe = _
3   var recipe: Recipe = _
4
5   before {
6     emptyRecipe = Recipe("Empty", List.empty[Ingredient])
7     val il = List(Ingredient("Cookie", 100), Ingredient("Milk", 200))
8     recipe = Recipe("Cookies and Milk", il)
9   }
10
11  "The recipe" should "have two ingredients" in {
12    assert(recipe.names.equals(List("Cookie", "Milk")), "...")
13  }
14
15  "The empty recipe" should "have no ingredients" in {
16    assert(!emptyRecipe.hasIngredients, "...")
17  }
18 }
```

```
1 class RecipeTestSuite extends FlatSpec {
2   "The recipe" should "have two ingredients" in {
3     val il =  List(Ingredient("Cookie", 100), Ingredient("Milk", 200))
4     val recipe = Recipe("Cookies and Milk", il)
5     assert(recipe.names.equals(List("Cookie", "Milk")), "...")
6   }
7
8   "The empty recipe" should "have no ingredients" in {
9     val emptyRecipe = Recipe("Empty", List.empty[Ingredient])
10    assert(!emptyRecipe.hasIngredients, "...")
11  }
12 }
```

Listing 3.6: An example of General Fixture (Type I) and its refactoring.

Type II - Loan Fixture: Detection Method. So-called "loan fixture methods" are methods with a body that serves to set up and tear down fixture objects, respectively before and after the call from their body to the function provided to them as a parameter. Method `withRecipe` in Listing 3.7 is such a loan fixture method, calling its parameter `test` on line 8 with the fixture objects it has set up. The method itself is called from line 11 and line 16, for the purpose of loaning the objects to the test cases defined by its function argument on lines 12–13 and lines 17–18 respectively. Multiple loan fixture methods can be defined in a test class, and shared with the appropriate test cases. Despite the increase in expressiveness, this definition style is not less prone to the GF test smell. Detecting the GF smell in fixtures defined through loan fixture methods requires: (i) collecting the parameters of the function given as an argument for the call to the loan fixture method from the test case, and (ii) checking whether every parameter is referenced in the body of the function.

Type II - Loan Fixture: Refactoring. The fixture should be removed, in case the test case uses none of its objects, or split into several local fixtures. Note the changes in the parameter and argument lists as a result of the refactoring below, as well as the composition of two separate local fixtures for the last test case.

```
class RecipeTestSuiteLF extends FlatSpec {

  def withRecipe(test: (Recipe, Recipe) => Any) {
    val il1 = List(Ingredient("Cookie", 100), Ingredient("Milk", 200))
    val il2 = List(Ingredient("Eggs", 100), Ingredient("Bacon", 200))
    val cookiesAndMilk = Recipe("Cookies and Milk", il1)
    val baconAndEggs = Recipe("Eggs", il2)
    test(cookiesAndMilk, baconAndEggs)
  }

  "The recipe" should "have 2 ingredients (Eggs, Bacon)" in withRecipe {
    (cookiesAndMilk, baconAndEggs) =>
      assert(baconAndEggs.names.equals(List("Eggs", "Bacon")), "...")
  }

  "The recipe" should "have 2 ingredients" in withRecipe {
    (cookiesAndMilk, baconAndEggs) =>
      assert(cookiesAndMilk.ingredients.size == 2, "...")
  }

}
```

```
class RecipeTestSuite extends FlatSpec {

  def withCookiesAndMilk(test: (Recipe) => Any) {
    val il = List(Ingredient("Cookie", 100), Ingredient("Milk", 200))
    val cookiesAndMilk = Recipe("Cookies and Milk", il)
    test(cookiesAndMilk)
  }

  def withBaconAndEggs(test: (Recipe) => Any) {
    val il = List(Ingredient("Eggs", 100), Ingredient("Bacon", 200))
    val baconAndEggs = Recipe("Eggs", il)
    test(baconAndEggs)
  }

  "The recipe" should "have 2 ingredients (Eggs, Bacon)" in
    withBaconAndEggs {
      baconAndEggs =>
        assert(baconAndEggs.names.equals(List("Eggs", "Bacon")), "...")
  }

  "The recipe" should "have 2 ingredients" in withCookiesAndMilk {
    cookiesAndMilk =>
      assert(cookiesAndMilk.ingredients.size == 2, "...")
  }

  "Different recipes" should "not be equal" in withBaconAndEggs {
    baconAndEggs =>
      withCookiesAndMilk { cookiesAndMilk =>
        assert(cookiesAndMilk.equals(baconAndEggs), "...")
    }
  }

}
```

Listing 3.7: An example of General Fixture (Type II) and its refactoring.

Type III - Fixture Context: Detection Method. Fixture contexts are instances of an anonymous class that mixes in at least one trait such as `RecipeFixture` that provides *and* initializes fields for the fixture. Listing 3.8 uses these context at Lines 11–13 and lines 15–17. The body of the anonymous class itself corresponds to the test case, such as the `assert` expressions on lines 12 and 16. Note that multiple traits can be mixed into the "fixture context" object (*e.g.,* `new X with Y with Z`) as required by the fixture for a specific test case. Fixtures defined in this manner, as expressive it may be, are still prone to the GF test smell. Its detection requires: (i) collecting the fields mixed into and provided by the "fixture context" object, (ii) verifying whether every field is referenced in test case (*i.e.,* the body of the corresponding anonymous class creation expression).

Type III - Fixture Context: Refactoring. The refactoring consists of splitting the fixture into multiple smaller fixtures. A trait can be dedicated to each field, rendering them easier to compose as needed for individual test cases.

```
1  class RecipeTestSuiteFCO extends FlatSpec {
2
3    trait RecipeFixture {
4      val il1 = List(Ingredient("Cookie", 100), Ingredient("Milk", 200))
5      val il2 = List(Ingredient("Eggs", 100), Ingredient("Bacon", 200))
6      val cookiesAndMilk = Recipe("Cookies and Milk", il1)
7      val baconAndEggs = Recipe("Eggs", il2)
8    }
9
10   "The recipe" should "have two ingredients (Eggs, Bacon)"
11     in new RecipeFixture {
12       assert(baconAndEggs.names.equals(List("Eggs", "Bacon")), "...")
13   }
14
15   "The recipe" should "have two ingredients" in new RecipeFixture {
16     assert(cookiesAndMilk.ingredients.size == 2, "...")
17   }
18 }
```

```
1  class RecipeTestSuiteFCOR extends FlatSpec {
2
3    trait BaconAndEggsRecipe {
4      val il = List(Ingredient("Eggs", 100), Ingredient("Bacon", 200))
5      val baconAndEggs = Recipe("Eggs", il)
6    }
7
8    "The recipe" should "have two ingredients named Eggs and Bacon" in
9        new BaconAndEggsRecipe {
10           assert(baconAndEggs.names.equals(List("Eggs", "Bacon")), "...")
11     }
12 }
```

Listing 3.8: An example of General Fixture (Type III) and its refactoring.

Type IV - With Fixture: Detection Method. The traits defined in the package `org.scalatest.fixture` provide another way of using fixtures. Each of the test cases in such a class take the same fixture as parameter, such as `f` on line 13 of Listing 3.9. This fixture can be set up and torn down by overriding method `withFixture` in the test class, the body of which needs to apply the method's function parameter —which corresponds to the executed test case— to the fixture. This definition style eliminates some of the boilerplate involved in the "loan fixture method" style, but is only applicable when most test cases share the same fixture. The GF smell can manifest itself if the class defining the fixture (*e.g.,* `FixtureParam` on line 3) provides fields that are not referenced from a test case. It is convenient and common to use the `case class` feature of Scala to define the fixture class, which is the only variant we can support detecting without computationally expensive program analyses. The detection requires: (i) finding test classes that inherit from package `org.scalatest.fixture`, (ii) resolving the type of the argument to the function called from within `withFixture` to its type definition, and (iii) ensuring that all test cases within the class use the fields provided by this case class.

Type IV - With Fixture: Refactoring. There are multiple ways to eliminate this smell, but it is clear that the other definition styles such as Loan Fixture or Fixture Context can be of help. We refer the reader back to Listing 3.7 which shows a potential refactoring.

```
1  class RecipeTestSuiteWF extends fixture.FlatSpec {
2
3    case class FixtureParam(cookiesAndMilk: Recipe, baconAndEggs: Recipe)
4
5    def withFixture(test: OneArgTest): Outcome = {
6      val il1 = List(Ingredient("Cookie", 100), Ingredient("Milk", 200))
7      val il2 = List(Ingredient("Eggs", 100), Ingredient("Bacon", 200))
8      val cookiesAndMilk = Recipe("Cookies and Milk", il1)
9      val baconAndEggs = Recipe("Eggs", il2)
10
11     val theFixture = FixtureParam(cookiesAndMilk, baconAndEggs)
12     test(theFixture)
13   }
14
15   "The recipe" should "have two ingredients named Eggs and Bacon" in {
16     f => assert(f.baconAndEggs.names.equals(List("Eggs", "Bacon")), "...")
17   }
18
19   "The recipe" should "have two ingredients" in { f =>
20     assert(f.cookiesAndMilk.ingredients.size == 2, "...")
21   }
22 }
```

Listing 3.9: An example of General Fixture (Type IV).

3.5.4 Lazy Test (LT)

Definition. More than one test case that tests the same method while using identical fixtures. This smell affects test maintainability since assertions about one method should be in the same test case and not spread across different ones. Like Eager Test, the original definition [vMBK01] leaves some details to interpretation. We consider every call to the class under test as a potential cause of Lazy Test, irrespective of whether their results are used in an assertion.

Detection Method. A Lazy Test can be detected in three steps: (i) identify the class under test, (ii) for each test case, collect the set of methods that belong to that class under test, (iii) for each test case, compute the intersection of all other test cases. A non-empty intersection is indicative of a method that is referred to in multiple test cases.

Refactoring. Merge the individual test cases that execute the same method into a single one. The result is one test case per method of the class under test, which is said to improve the traceability between production and test code.

```
"The recipe" should "have zero ingredients" in {
  val recipe = Recipe("Cookies and Milk", List.empty)
  assert(!recipe.hasIngredients,
    s"The number of ingredients was ${recipe.ingredients.size}")
}

"The recipe" should "have two ingredients" in {
  val il = List(Ingredient("Cookie", 100), Ingredient("Milk", 200))
  val recipe = Recipe("Cookies and Milk", il)

  assert(recipe.hasIngredients,
    s"The number of ingredients was ${recipe.ingredients.size}")
}
```

```
"The recipe" should "zero ingredients" in {
  val emptyRecipe = Recipe("Cookies and Milk", List.empty)
  val il = List(Ingredient("Cookie", 100), Ingredient("Milk", 200))
  val recipe = Recipe("Cookies and Milk", il)

  assert(!emptyRecipe.hasIngredients,
    s"The number of ingredients was ${emptyRecipe.ingredients.size}")

  assert(recipe.hasIngredients,
    s"The number of ingredients was ${recipe.ingredients.size}")
}
```

Listing 3.10: An example of Lazy Test and its refactoring.

3.5.5 Mystery Guest (MG)

Definition. A test case that uses external resources that are not managed by a fixture. A drawback of this approach is that the interface to external resources might change over time necessitating an update of the test case, or that those resources might not be available when the test case is run, endangering the deterministic behaviour of the test. While Resource Optimism additionally requires the external resource to exists and be in a correct state, we consider the smell to be a special case of Mystery Guest. However, we do not this additional check as that path cannot always be statically determined. Nevertheless, the following refactoring resolves the common problem of both smells.

Detection Method. This smell can be detected by identifying test cases that contain resource instances such as `java.io.File`, `java.nio.file.Path`, `java.io.FileInputStream`, and `java.net.URI`, as well as factory methods for reading files such as `scala.io.Source#fromFile` and methods for connecting to databases such as `java.sql.DriverManager#getConnection`.

Refactoring. Manage resources explicitly in a fixture.

```
"A recipe" should "be able to be initialized from a file" in {
  val file = scala.io.Source.fromFile("ingredients_recipe.txt")
  val recipe = Recipe.fromFile(file)
    assert(recipe.ingredients.size == 20,  "...")
}
```

```
def withRecipeFile(test: BufferedSource => Any) {
  val path = "file.txt"
  val contents =
    """
    |BaconAndEggs
    |bacon,100
    |eggs, 200
    """.stripMargin

  Files.write(Paths.get(path), contents.getBytes(StandardCharsets.UTF_8))
  assume(new File(path).exists(), s"File $path did not exists")
  test(scala.io.Source.fromFile(path))
}

"A recipe" should "be able to be created from file" in withRecipeFile {
  file =>
    val recipe = Recipe.fromFile(file)
    assert(recipe.ingredients.size == 20,  "...")
}
```

Listing 3.11: An example of Mystery Guest and its refactoring.

3.5.6 Sensitive Equality (SE)

Definition. A test case with an assertion that compares the state of objects by means of their textual representation, *i.e.,* by means of the result of `toString()`.

Detection Method. This smell can be detected by (i) enumerating the assertions in a test case (as described in section 3.5.1) and by (ii) verifying whether they contain or rely on a call to the `toString()` method.

Refactoring. Compare the members of the object states structurally instead of relying on `toString()` of the wholes.

```
1 "A recipe" should "have as ingredient: Ingredient('Chocolate', 100)" in {
2   val il = List(Ingredient("Chocolate", 100))
3   val recipe = Recipe("Chocolate Cookies", il)
4   val result = "Recipe(Chocolate Cookies,List(Ingredient(Chocolate,100)))"
5   assert(recipe.toString() == result, "...")
6 }
```

```
1 "A recipe" should "have as ingredient: Ingredient('Chocolate', 100)" in {
2   val il = List(Ingredient("Chocolate", 100))
3   val recipe = Recipe("Chocolate Cookies", il)
4   assert(recipe.ingredients == List(Ingredient("Chocolate", 100)), "...")
5 }
```

Listing 3.12: An example of Sensitive Equality and its refactoring.

3.6 Implementation

We briefly discuss the usage and the possibilities to extend Socrates with support for additional smells as it could be used as a foundation for further research.

3.6.1 Usage

Socrates is available as an IntelliJ IDEA plugin that can be downloaded from Github[4]. The plugin must be installed before it can be used within a project. Figure 3.5 shows the options that can be used to configure the process.

The first option specifies the location of the `rt.jar`. This is needed to build the SemanticDB of the Scala standard library. The second option is the location of the `ivy2` cache. This folder contains all of external dependencies which are required to successfully compile the project. These jar files are required to build the SemanticDB. Multiple folders can be separated by colons (*e.g.,* `path1:path2`). The third option specifies the location of the SBT binary. This is required as Socrates will execute a SBT task that activates a compiler plugin in the background. The final option enables the developer to specify custom

[4] https://github.com/jonas-db/socrates

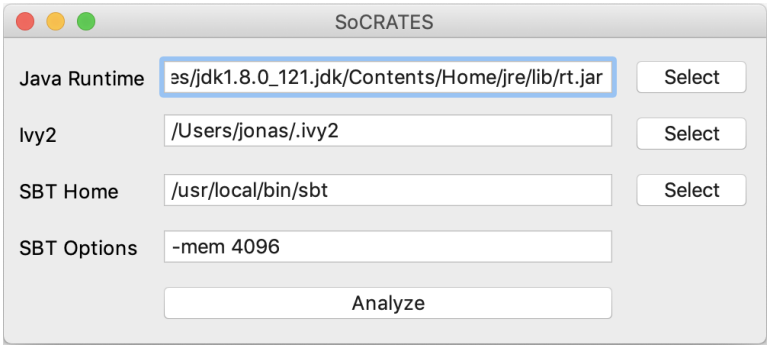

Figure 3.5: The required options to execute SOCRATES.

SBT options. By default, this includes an increased memory allocation because building the SEMANTICDB for a project can require a significant amount of memory. An example output of SOCRATES is shown in Figure 3.6.

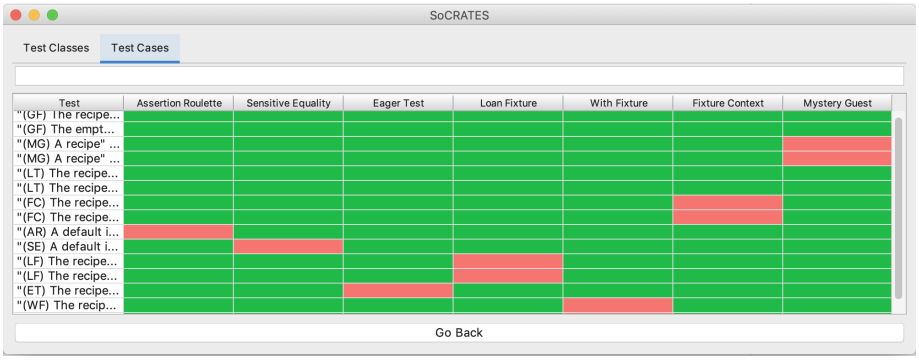

Figure 3.6: An overview of test smells reported by SOCRATES.

The active tab displayed in the figure reports the test smells related to test cases, while the other tab reports on the test smells GLOBAL FIXTURE and EAGER TEST related to test classes only. The results can be sorted by column and filtered to quickly investigate whether a given test class or test case exhibits a particular test smell.

3.6.2 Extension

Next to providing SOCRATES as an INTELLIJ IDEA plugin, we also provide it as an open-source project on GITHUB. This to facilitate further research as our literature study indicated a lack of open-source tool support. For that reason, we implemented SOCRATES from the bottom up to be extensible.

3.6.2.1 Compilation

Every project must be compiled with a SBT compiler plugin[5]. This plugin inserts dependencies and compilation options to generate the SemanticDB. All of this happens in the background by executing the command `sbt semanticdb`.

3.6.2.2 Identifying Test Cases

One should be able to implement their own test case identification strategies to detect a different testing style or framework, as well as their own test smell detection strategies to detect new smells. Listing 3.13 shows the class that a developer should implement to identify test cases, while Listing 3.14 shows an implementation of this class to detect test cases that use the testing style FunSuite.

```
1 abstract class Explorable {
2   def explore(ast: Tree, document: TextDocument): List[ScalaTestCase]
3 }
```

Listing 3.13: The class `Explorable`.

```
1 object FunSuite extends Explorable {
2
3   def explore(ast: Tree, d: TextDocument): List[ScalaTestCase] =
4     ast.collect({
5       case t@Term.Apply(
6           Term.Apply(Term.Name("test"),
7           Lit.String(_) :: _),
8           p) =>
9         val paramFixtures, loanFixtures, fixtureContexts = ...
10        ScalaTestCase(t, d, paramFixtures, loanFixtures, fixtureContexts)
11    })
12 }
```

Listing 3.14: An implementation of the class `Explorable`.

3.6.2.3 Identifying Test Smells

Listing 3.15 shows the classes that a developer can implement to detect new test smells. The class `TestClassTestSmell` and `TestCaseTestSmell` is used to detect smells in test classes and test cases respectively. Listing 3.16 shows the implementation of Sensitive Equality. We refer the reader to the implementation for more information and examples.

```
1 abstract class TestClassTestSmell extends TestSmell {
2   def verify(ts: ScalaTestClass, pdb: ProjectDatabase): Boolean
3 }
4
5 abstract class TestCaseTestSmell extends TestSmell {
6   def verify(
7     tc: ScalaTestCase,
8     ts: ScalaTestClass,
9     pdb: ProjectDatabase): Boolean
10 }
```

Listing 3.15: The classes `TestClassTestSmell` and `TestCaseTestSmell`.

[5] https://gist.github.com/olafurpg/a74404dfee6b3da03892af17357074d9

```
object SensitiveEquality extends TestCaseTestSmell {

  val assertions: PartialFunction[Tree, Tree] = {
    case a@Term.Apply(Term.Name(t), _) if t.equals("assert") => a
    case a@Term.Apply(Term.Name(t), _) if t.equals("assertResult")  => a
    case a@Term.Apply(Term.Name(t), _) if t.equals("assume") => a
    case a@Term.Apply(Term.Name(t), _) if t.equals("cancel") => a
    case a@Term.Apply(Term.Name(t), _) if t.equals("fail") => a
  }

  def hasToString(tree: Tree): Boolean = {
    tree.collect({
      case t@Term.Select(_, Term.Name("toString")) => t
    }).nonEmpty
  }

  override def verify(
    testCase: ScalaTestCase,
    testSuite: ScalaTestClass,
    pdb:ProjectDatabase): Boolean = {
      testCase.ast.collect(assertions).exists(a => hasToString(a))
  }

  override val description: String = "SensitiveEquality"
}
```

Listing 3.16: An implementation of the class `TestCaseTestSmell`.

3.7 Conclusion

This chapter presented our static analysis approach to detecting test smells. We started by defining the concept of test smells. Next, we presented our literature study and its three key observations which we used as our motivation for this research. First, the majority of the studies are based on systems in the Java ecosystem where the testing framework JUnit is the most prominent. Second, tests smells can have severe impact on multiple software aspects. Finally, developers have limited tool support which leaves ample room for low quality test suites. Based on these observations, we therefore proposed Chaokka to detect test smells in the Scala ecosystem. We explained our unique approach that uses both syntactic and semantic information. Additionally, we transposed test smell definitions to the Scala ecosystem and provided both a static detection and refactoring method. Finally, we explained several details on the implementation. In the next chapter, we will use Socrates to empirically investigate the prevalence of test smells in the Scala ecosystem.

Chapter 4

Empirical Study on Test Smells in the Scala Ecosystem

In the previous chapter, we observed that most empirical research on test smells focuses on the Java ecosystem where JUnit is the most prominently used testing framework. This might have skewed the current understanding of test smells. This chapter therefore presents two studies that aim to broaden the existing knowledge about test smells. For each study, we provide details about the design, the studies, and the results.

First, Section 4.1 discusses the knowledge of Scala developers about test smells. We conduct a survey to investigate whether developers can identified test smells. Similar to previous studies, we conclude that the majority of developers are not aware of test smells. Next, Section 4.2 presents an empirical study on the diffusion of test smells. We analyse the test suites of 164 Scala systems from Github. Our results indicate a lower diffusion of test smells in the Scala ecosystem. Finally, we conclude this research on test smells by discussing our observations.

4.1 Perception of Test Smells

This section reports on the first study where we assess whether developers are aware of test smells in the Scala ecosystem. Section 4.1.1 briefly highlights our motivations for this study. Section 4.1.2 presents the design of the study, while Section 4.1.3 discusses the results. Similar to studies in the context of the Java ecosystem, our results indicate that developers are not completely aware of test smells in the Scala ecosystem. On average, we find that only 5 out of 14 developers are able to identify existing instances of each test smell.

4.1.1 Motivations

Based on our exploration of the state of the art, we are only aware of four studies that surveyed developers about test smells. We briefly discuss them to explain our motivation for our survey.

Bavota *et al.* [BQO+15] complement the aforementioned diffusion results with a survey involving 61 participants ranging from students to professional developers. However, their main aim was not to assess the participants' perception of test smells, but to verify the impact of JUnit smells on software maintenance. The results of the user study show that test smells negatively impact program comprehension during maintenance activities.

Greiler *et al.* [GvS13] conducted a study on smells related to fixtures in JUnit and TestNG tests and surveyed 13 professionals. The results show that developers recognize that fixture-related smells are problematic and agree that they impact test maintenance negatively. However, it is not clear whether developers would be able to recognize the smells without the detection tool used in the study.

Tufano *et al.* [TPB+16] investigated developers' perception of JUnit test smells in a survey among 19 participants. Their study considers the same test smells, except for the omission of Lazy Test, and found that developers do not really perceive test smells as actual design problems (only in 17% of the cases) and are even less capable of identifying them precisely (only in 2% of the cases) without tool support.

Spadini *et al.* [SSO+20] analysed 1489 open-source projects to determine severity thresholds for test smells and conducted a study with 31 developers to evaluate their thresholds. The results shows that their severity thresholds are in line with the participants' perception of how test smells impact the maintainability of tests.

It is clear that these studies are conducted in the context of Java and JUnit with tests stemming from real-world systems. Therefore, we want to investigate the developers' perception of test smells in the context of Scala and ScalaTest with artificial tests. Our results should give us a broader view on the awareness of developers about test smells. Additionally, we are interested to see whether the results of our survey align with those observed in existing surveys.

4.1.2 Design

We conduct a survey with 14 Scala developers to understand the knowledge of developers about test smells. In particular, we assess whether developers perceive test smells (*i.e.,* assume or know something is wrong) and whether they can identify the corresponding cause (*i.e.,* explain the problem). This survey provides insights for researchers because our study broadens the knowledge of test smells in a different ecosystem and for developers of testing tools because we open-source our implementation of Socrates. Educators of software quality can use our results to see whether there is a need for education and training on test smells. In particular, this study aims to answer the following research question:

- **RQ$_1$:** *To what extent do developers perceive and identify tests smells in the Scala ecosystem?*

4.1.2.1 Survey

We invite members of Scala meet-up groups[1] to participate in our survey. This ensures that we have a representative sample of experienced and enthusiastic Scala developers. The purpose of the survey and the concept of a test smell are explained to the participant at the beginning of the survey. The questions of this survey are based on the test smells and their illustrative examples from Section 3.5. We provide a unit test that exhibits a particular smell for each question, without providing any additional information about the design issue. Each participant has to indicate whether he or she perceives the test smell and has to motivate their answer through the following questions:

- *Does this unit test exhibit a test smell according to your experience?*
- *If yes, indicate which piece(s) of code and/or which reasons might cause this test smell. If no, leave blank.*

The survey is hosted on Google Forms and is designed to be completed in approximately 20 minutes.

[1] https://www.meetup.com

4.1.2.2 Set of Data

In summary, we collected answers from 14 developers that were willing to participate in our survey and were collected over a time span of 2 weeks. The majority of the developers (10 out of 14) work on industrial systems and the remaining (4 out 14) work on open-source systems. These systems range from 10K to 100K lines of code. Furthermore, 13 participants have more than 5 years of experience developing in Scala, while 10 have more than 10 years of experience. Moreover, 11 out of 14 participants consider themselves experienced in the domain of software testing. Their experience is also reflected by the diversity of used testing frameworks: ScalaTest (79%), ScalaCheck (50%), Specs2 (43%), and JUnit (21%).

4.1.2.3 Studies

Our first study addresses $\mathbf{RQ}_1$ by conducting $\mathbf{Study}_1$. This study is based on the set of data discussed in Section 4.1.2.2.

> **$\mathbf{Study}_1$:** We analyse the answers of the survey in a quantitative and qualitative manner to determine the knowledge of Scala developers about test smells.

4.1.3 Results

Figure 4.1 shows the results of this survey in a quantitative manner. In particular, the plot shows the following:

- The number of cases in which test smells have been *perceived* by participants. A test smell is perceived whenever the participant answered yes to the question: *"Does this unit test exhibit a test smell according to your experience?"*.
- The number of cases in which test smells have been *identified* by the participants. A test smell is identified whenever the given explanation correctly pinpoints the cause of the test smell. The explanations are collected from the question: *"If yes, indicate which piece(s) of code and/or which reasons might cause this test smell. If no, leave blank."*.

A test smell can only be identified when it has been perceived by the participant. Arguably, the answers to the first question might have been biased which is why we consider the answers to the second question as truth. The number of developers that perceived the test smells is the combination of both numbers. For example, Assertion Roulette (ar) has been perceived 9 times in total, but only identified twice. This means that the 7 other participants incorrectly identified the smell and that 3 participants out of the 14 simply did not perceive this smell.

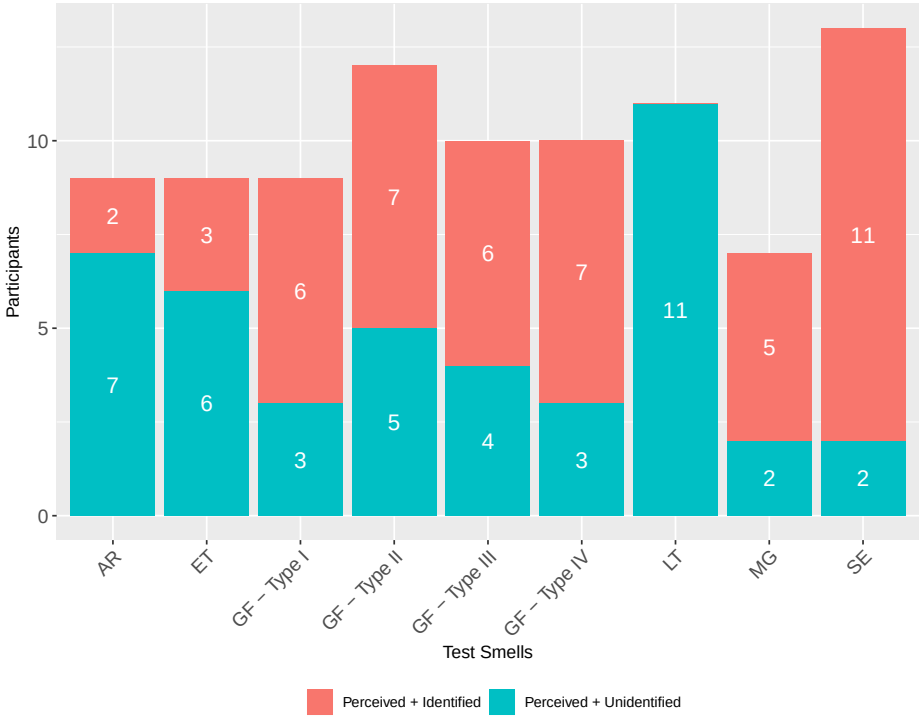

Figure 4.1: Absolute number of participants that perceived and identified the smells.

This plot shows that the test smells Sensitive Equality, General Fixture, and Mystery Guest are the most identified, while no developer was able to identify Lazy Test correctly. On average, only 5 out of 14 (36%) developers are able to explain the cause of these smells. This shows that developers are not able to correctly identify most of the smells, even though they perceive a design issue. We discuss the results of each test smell in a qualitative manner below.

Assertion Roulette. This smell was perceived by 9 out of 14 participants. However, only 2 of them identified the problem of the test having multiple assertions without a failure explanation (*e.g., [...] several assertions within same test case*). Indeed, contemporary integrated development environments (IDE) and test runners are able to pinpoint the failed assertion in the code and might be able to generate a simple error message based on the expressions in the assertion. This mitigates the potential negative impact of the smell on maintenance and comprehensibility.

Moreover, ScalaTest provides a more advanced domain-specific language to write assertions in a more readable and human-friendly way such as `string should startWith regex "Hel*o"`. The framework does not take an explicit failure explanation as argument but rather generates one itself based on the other operators in the sentence. These reasons might explain the low number of participants that are able to identify Assertion Roulette. It also raises questions about whether Assertion Roulette should still be considered a test smell.

General Fixture. On average, we observe that 10 out of 14 participants perceive each of the four variants and that 7 out of 14 participants are able to correctly identify the cause of these smells. It emerges that developers are well aware of the correct usage of fixtures, as confirmed by several detailed explanations of the participants such as: *"Two different test cases use the same fixture that is badly tailored for each. Each test case should have specific data made for it, reusing test data should remove code duplication when needed, in this case it's coupling the scenarios for no reason"* and *"unused fixtures in tests, plus in this case because each fixture is only used in one test you're paying a readability penalty for nothing"*.

Eager Test. The smell Eager Test is perceived by 9 out of 14 participants, but only 3 of them are able to correctly identify the cause of this smell. In these cases, the explanations are very short and to the point, such as *"Should only test one thing, and [...]"*. As mentioned in Section 3.5, this smell has some ambiguity. For some developers, it might not be a problem to assert multiple fields or methods if they belong to the same object.

Mystery Guest. Half of the 14 participants perceive that the unit test exhibited an instance of Mystery Guest. 5 of these 7 developers identify the problem with external resources. For example, one participant said *"[...] instead of a file would have simplified testing to avoid using external resources, there are also alternatives to generate tmp files for the test case, so that the file content and the test checks are easier to keep in sync"*). Despite fixtures with assumptions being the preferred approach to set up external resources, nobody mentioned this as a solution. This solution cancels tests whenever the fixture was not correctly initialized instead of being executed and ultimately failing.

Sensitive Equality. The majority of the participants (11 out of 14) perceive and correctly identify the smell Sensitive Equality. Their explanations are all very similar and correctly pinpoint the problem, namely the use of `toString` in an assertion: *"Testing 'toString' to check for behaviour is easily broken. Adding parameters to the class or renaming it would immediately break the test."*. These explanations reflect the high awareness of this smell.

Lazy Test. 11 out of 14 participants perceived this smell, but none was able to identify it correctly. It looks like many developers are simply not aware of this smell, or do not consider this to be a smell at all. We were not able to pinpoint the exact reason due to a lack of convincing answers. Surprisingly, multiple participants remarked an inconsistency between the description and the purpose of the test: *"Should only test one thing, and description should match assertion*, *"[...] Moreover, the description doesn't match the test: just verifying that [...]*, and *"The tests description does not match the assertions"* The inclusion of this inconsistency was unintentional, yet several developers perceive this as a smell which might indicate that developers heavily rely on the descriptions of a test to understand its purpose. This can also be related to ScalaTest itself and its different testing styles. They might encourage developers to describe their tests in a more natural way. This is in contrast to JUnit which only recently provides the `@DisplayName` annotation to describe their tests.

RQ_1 Summary

The most identified test smells are Sensitive Equality, General Fixture, and Mystery Guest, while no developer was able to identify Lazy Test correctly. Only 5 out of 14 (36%) developers are able to explain the cause of these smells. This aligns with the current knowledge that the majority of developers are not aware of test smells (*e.g.,* [TPB+16]).

4.1.4 Threats to Validity

We identify several threats that might influence the validity of our results and conclusions. In general, replications of the survey with a larger number of participants, a larger set of test smells, and more realistic test cases are desirable.

First, we identify the artificial nature of the test cases used in the survey as a threat. They might not reflect real and complex test cases which exhibit test smells. We opted for such examples so that developers can focus on the design issues of the tests without the need for long training sessions about the system under test.

Second, we are also aware that only showing examples of tests that exhibit a smell, along with its name, could have biased the participants. However, the impact is limited to the perception results only as participants had to give a correct explanation for the identification results. In the end, our results are similar to those observed in previous studies (*e.g.,* [TPB+16], [GvS13]).

Finally, the population of participants might have influenced the results. We were able to interview 14 Scala developers that work on open-source and industrial software systems, similar to the study by Tufano *et al.* [TPB+16]. While our aim was to get a representative population, we were limited to gather participants through mouth-to-mouth advertisement at Scala meetings and workshops.

4.2 Diffusion of Test Smells

This section reports on the study concerning the diffusion of test smells in the Scala ecosystem. Section 4.2.1 presents the design of the study, while Section 4.2.2 discusses the results. The results hint that test smells have a low diffusion across test classes in the Scala ecosystem — with Lazy Test, Eager Test, and Assertion Roulette as the most prevalent ones.

4.2.1 Design

We conduct an empirical study on a set of 164 open-source systems collected from Github to understand the diffusion of test smells in the Scala ecosystem. We apply our detection methods from Section 3.5 to detect six smells. In general, we aim to understand to what extent test smells are diffused in the context of Scala ecosystem and its testing framework ScalaTest. Therefore, this study benefits both researchers (*i.e.,* context diversity) and developers of testing tools (*i.e.,* are tools able to detect test smells and to what extent). In particular, this study aims to answer the following research questions:

- **RQ$_2$:** *To what extent are test smells spread in the Scala ecosystem?*
- **RQ$_3$:** *Which test smells occur more frequently in the Scala ecosystem?*

4.2.1.1 Data Set

To define our set of open-source Scala systems, we start by collecting all Scala repositories that were created on Github between January 2010 and July 2018 through the Github API. This results in an initial set of 72,619 systems. Next, we apply a selection process that discards projects that: (i) lack test classes, (ii) do not use SBT for build automation, or (iii) are outdated and no longer compile.

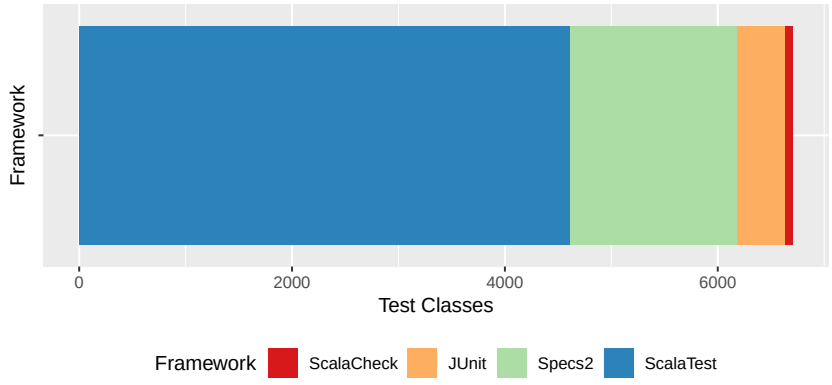

Figure 4.2: The usage of testing frameworks in the Scala ecosystem.

After this selection phase, 2,920 projects remain on which we conduct a preliminary analysis to assess which testing frameworks are the most prevalent. Figure 4.2 shows the resulting usage distribution of ScalaTest, Specs2, ScalaCheck and JUnit. It is apparent that ScalaTest is the most used testing framework, while the popular JUnit framework is barely used, despite the interoperability of Java and Scala. Given that ScalaTest is the most prominent testing framework in the Scala ecosystem, we further reduce our set to only those systems that use ScalaTest. Additionally, we filter out low-quality systems that have less than 1,000 lines of code in both production and test code. The resulting set consists of 164 systems. The collection and filtering was 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 4.1 summarizes several characteristics of these systems. Numbers are rounded to the nearest whole.

	1st Quartile	Mean	Median	3rd Quartile	Total
# of Production Files	26	75	48	86	12,266
# of Test Files	14	36	21	42	5,841
# of Production LOC	2,107	7,236	3,718	6,740	1,186,708
# of Test LOC	1,400	3,958	1,960	4,033	649,172
# of Test Classes	10	30	16	31	4,914
# of Test Cases	50	150	85	185	24,578

Table 4.1: The data set and its characteristics used in our empirical study.

As indicated previously, ScalaTest offers a wide range of testing styles. Table 4.2 summarizes the usage of each testing style in ScalaTest. Percentages are rounded to the nearest whole. We found `FlatSpec`, `FunSuite`, and `WordSpec` to be among the most popular styles. Their popularity might be explained by their similarity to xUnit testing frameworks. Additionally, these styles are presented first in the documentation of ScalaTest.

FlatSpec	FunSuite	WordSpec	FunSpec	FreeSpec	FeatureSpec	PropSpec	RefSpec
46%	27%	11%	11%	4%	1%	1%	0%

Table 4.2: The usage of testing styles in the Scala ecosystem.

4.2.1.2 Tool

We use our tool Socrates for the studies. With respect to existing tools, Socrates does not only use syntactic information from the abstract syntax trees, but also semantic information such as types and symbols. This should improve the precision with which test smells are detected, but might come at the expense of recall. Therefore, we manually investigate the precision and recall of Socrates by validating a statistically significant sample of the tests with a confidence level of 95% and a confidence interval of 5%. Precision measures whether the detected smells are indeed smells, while the recall measures how many test smells were not found by the tool.

The process involved two of the authors of [DBDNDR19a] which inspected 377 test cases to determine whether our tool correctly identified the smells in each test case or not. Disagreements during the validation were resolved by carefully checking both the code snippet and the test smell definitions. Table 4.3 shows the results of this manual validation. The precision and recall of SOCRATES for each test smell. The complete results are available in our appendix[2].

Test Smell	Precision	Recall
ASSERTION ROULETTE	100%	100%
EAGER TEST	96%	66%
GENERAL FIXTURE - TYPE I	97%	97%
GENERAL FIXTURE - TYPE II	-	-
GENERAL FIXTURE - TYPE III	100%	89%
GENERAL FIXTURE - TYPE IV	-	-
LAZY TEST	99%	75%
MYSTERY GUEST	100%	100%
SENSITIVE EQUALITY	100%	100%

Table 4.3: The precision and recall of SOCRATES for each test smell.

On average, SOCRATES achieves a high precision of 99% and modest recall of 90%. Our sample did not contain any test cases that exhibit type I and type II of GENERAL FIXTURE. These values are similar to state-of-the-art tools of previous empirical studies. For instance, the tool of Palomba *et al.* [PDNP+16] achieved a precision of 88% and a recall of 100%. Therefore, we deem SOCRATES sufficiently suitable for our studies.

4.2.1.3 Studies

We address $\mathbf{RQ}_2$ and $\mathbf{RQ}_3$ by conducting $\mathbf{Study}_2$ and $\mathbf{Study}_3$ respectively. These studies use the data set discussed in Section 4.2.1.1.

> **$\mathbf{Study}_2$:** We compute the number of systems and test classes that exhibit at least one of the six test smells.
>
> **$\mathbf{Study}_3$:** We compute the percentage of systems, test classes, and test cases that exhibit at least one instance of a particular smell.

4.2.2 Results

We discuss the systems and test classes that have at least one of the six test smells ($\mathbf{RQ}_2$) in Section 4.2.2.1, while Section 4.2.2.2 discusses foreach system, test class, and test case whether it contains a specific test smell ($\mathbf{RQ}_3$).

[2] https://figshare.com/articles/Assessing_Diffusion_and_Perception_of_Test_Smells_in_Scala_Projects/7836332

4.2.2.1 Study$_2$: At Least One Test Smell

We compute the number of systems and classes that exhibit at least one of the six test smells. We computed that 138 out of 164 (84%) systems and 1,381 out of 4,914 (28%) test classes are affected by at least one test smell. These numbers differ from those found by studies that assess test smell diffusion in the Java ecosystem. Additionally, we briefly compare our results to the work of Bavota *et al.* [BQO+15, PDNP+16] where the authors investigated the diffusion of test smells in 27 Java open-source systems. They found that 82% of the test classes is affected by at least one test smell. This is significantly higher than the 28% of test classes among the 164 Scala systems. While the studies differ in numbers, they tend to agree on which test smells are the most prevalent. Figure 4.3 shows the percentage of test classes affected by each test smell.

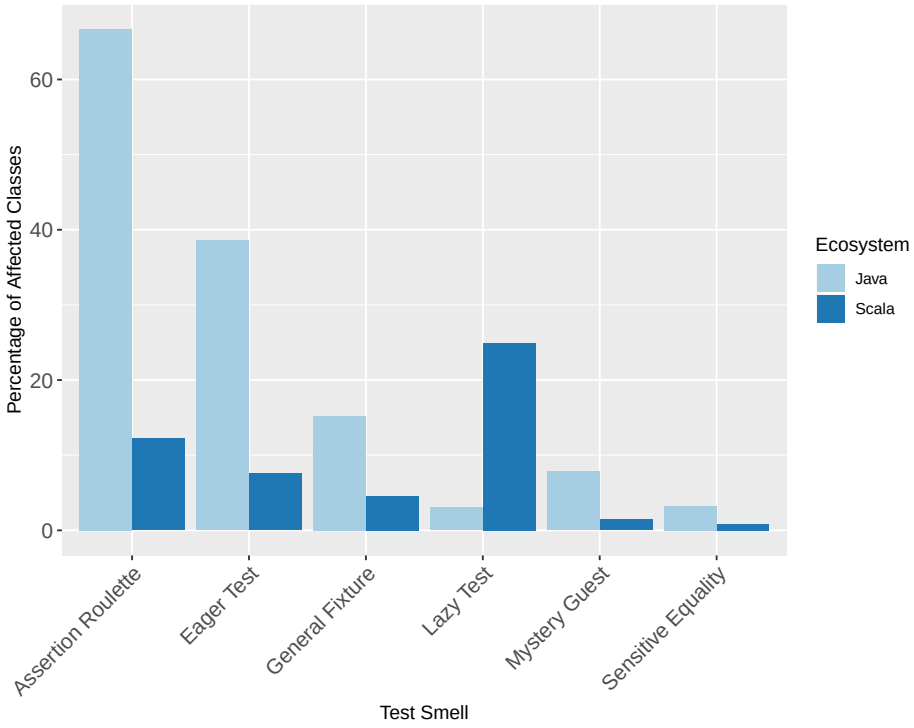

Figure 4.3: The mean percentage of test classes that are affected by a test smell.

It is clear from the plot that the ranking of each smell is similar across ecosystems, despite the difference in absolute numbers. Moreover, the studies do agree on Assertion Roulette, Eager Test, and General Fixture as the three most prevalent test smells when Test Code Duplication is discarded as the second most present test smell.

However, it is important to note that these findings are indicative and might not be compared in absolute numbers for several reasons: the set of systems is different; the languages differ; the number of tests smells differ; and different detection tools were used. Nevertheless, we list several plausible reasons for this difference. We believe this can provide insights for further research on test smells across different ecosystems.

First, the authors included Test Code Duplication and Indirect Testing as tests smells, affecting 35% and 11% of the test classes respectively. Given that code duplication is a general design smell and occurs in about 1 out of 3 classes, it is likely that this test smell has contributed to the high diffusion. The same applies for indirect testing, but to a lesser extent. Secondly, the authors state that their detection tool uses very simple detection rules that overestimate the presence of test smells in the code. Despite their modest precision, it might have influenced the projects and test classes that exhibit at least one test smell.

Second, an explanation for the lower diffusion of Eager Test might be that fields are by default public in Scala, and that its syntax enables protecting these fields at a later point in time by true accessor methods (*i.e.,* that do not directly return or set the value of the field) without having to substitute method calls for field accesses.[3] For the JUnit study, calls to accessor methods —including those returning or setting the field's value directly— still count towards those considered in the Eager Test (and Lazy Test) detection rule. We recommend researchers to exclude them instead.

Next, an explanation for the lower diffusion of Assertion Roulette among the ScalaTest classes might be that the framework features a popular DSL for specifying assertions as `should`-based sentences. For example, developers can specify `X should contain Y` or `a [Exception] should be thrownBy` for which the framework does not take an explicit failure explanation message as argument but rather generates one itself based on the other operators in the sentence. Our analysis therefore does not consider them as an assertion without explanation. Such a DSL might not be as popular or comprehensive for JUnit, but we recommend their use from the perspective of co-evolution of assertion and explanation.

Finally, JUnit does not provide limited means for defining fine-grained fixtures. This might have influenced developers to make more mistakes, resulting in a higher diffusion of General Fixture. Since fixtures play an important role in test suites, and therefore occur often, it might have contributed to the higher diffusion across test classes and systems. We therefore recommend that unit testing frameworks include features that support the fine-grained definition and sharing of case-specific fixtures, and that developers use them. These fixtures do not only make tests more maintainable, they also avoid code duplication across test cases.

[3] A pair of getter `field` and setter `field_=` methods can be defined so that existing `field` read and writes become calls to the appropriate method.

RQ_2 Summary

The Scala ecosystem has a high number (84%) of systems that are affected by at least one test smell, but only few (28%) of their test classes exhibit them. Compared to the Java ecosystem, we conclude that the diffusion of test smells is lower in the Scala ecosystem. Nevertheless, we observe a similar trend in the diffusion of each test smell across these ecosystems.

4.2.2.2 Study$_3$: At Least One of Each Test Smell

Figure 4.4 shows the diffusion of each test smell in the Scala ecosystem, while Table 4.4 shows the percentage of systems, test classes, and test cases that exhibit at least one instance of a particular smell. Percentages are rounded to the nearest whole. We discuss each test smell in detail.

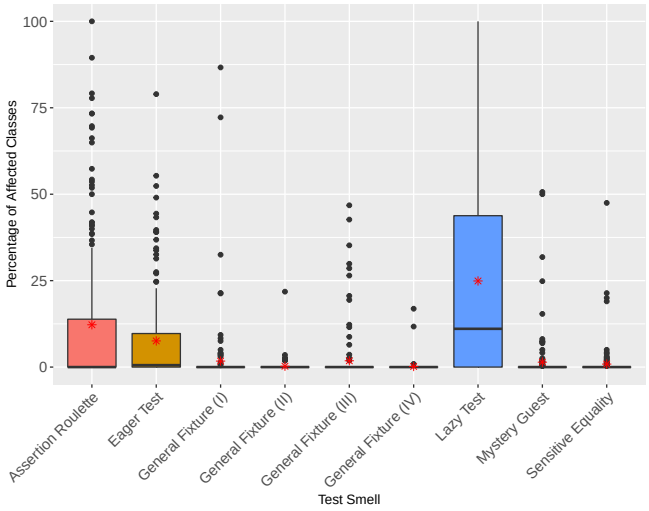

Figure 4.4: The diffusion of test smells in test classes across Scala systems.

Assertion Roulette. Assertion Roulette occurs in almost half of projects (45%) and in 16% of the test classes.

General Fixture. We go into more detail for fixtures as they are an important and advanced feature of ScalaTest (*cf.* Section 3.5.3). General Fixture occurs in about 1 out of 4 projects (27%) and in 3% of the test classes. Note that these results are the sum of the four types of General Fixture. In contrast to JUnit, where the fixture of a test class applies to all of its test cases by default, traits and first-class functions in Scala enable ScalaTest to support fine-grained fixture definitions.

Test Smell	% per System	% per Test Class	% per Test Case
Assertion Roulette	45%	16%	13%
Eager Test	52%	7%	6%
General Fixture - Type I	10%	1%	1%
General Fixture - Type II	5%	0%	0%
General Fixture - Type III	10%	2%	2%
General Fixture - Type IV	2%	0%	0%
Lazy Test	62%	11%	23%
Mystery Guest	16%	2%	1%
Sensitive Equality	13%	3%	1%

Table 4.4: The percentage of systems, test classes, and test cases exhibiting the different test smells.

Table 4.5 shows the distribution of each test fixture definition style. Percentages are rounded to the nearest whole. We observe that the majority (*i.e.,* 79%) of the tests use type I and type III, followed by type II (17%) and type IV (4%). About 49% (*i.e.,* type I and type IV) use a fixture that applies to the whole test class which indicates that fine-grained fixtures are indeed used.

Type I - Global	Type II - Loan Method	Type III - Context Object	Type IV - With
45%	17%	34%	4%

Table 4.5: The distribution of each test fixture definition style.

Eager Test. Eager Test occurs in about 1 out of 2 projects (52%), but only in 7% of the test classes. The results show that Eager Test does not frequently occur in multiple test classes, but rather in few test classes in half of the projects.

Mystery Guest. Mystery Guest is present in about 1 out of 7 (16%) projects, but in only 2% of all test classes. It is not surprising that only few test classes exhibit this smell as tests occasionally use external resources.

Sensitive Equality. Sensitive Equality is present in 13% and 3% of the projects and test classes respectively.

Lazy Test. Lazy Test occurs in almost 2 out of 3 projects (62%), or in 11% of the test classes. A plausible explanation for this high occurrence is the fact that developers are not aware of this smell as found in Section 4.1.3. Additionally, our tool does not take into account nested tests which is a feature of ScalaTest to reduce boilerplate. We made this design decision as it greatly simplified the detection of this smell. As a result, Socrates considers nested test cases to be lazy and this might have impacted the high diffusion of this smell. Indeed, from a manual inspection of several systems we found that our results over-approximate the diffusion of Lazy Test when nesting is incorporated.

RQ_3 Summary

In the Scala ecosystem, Lazy Test (62%), Eager Test (52%), and Assertion Roulette (45%) are the three most prevalent test smells, while General Fixture (27%), Mystery Guest (16%), and Sensitive Equality (13%) are the least prevalent.

4.2.3 Threats to Validity

We identify several threats that might impact the validity of the results of the our study. First, the set of 164 systems is only a subset of the many open-source Scala systems publicly available. We considered several criteria to select these systems such as the size of the system and test suite (*i.e.,* $+1,000$ lines of code), the build automation (*i.e.,* sbt v0.13+), and the relevance (*i.e.,* Scala v2.11+), and its testing framework (*i.e.,* ScalaTest). These systems might not be representative for real industrial software systems. Second, we detect test smells by means of Socrates which is implemented entirely from the bottom up by ourselves. Its precision and recall are thus major factors in the validity of our results. Therefore, we manually validated the output of Socrates on a sample of test cases in Section 4.2.1.2 and concluded that we achieve numbers similar to state-of-the-art tools [BQO+12, PDNP+16]. Third, some definitions of test smells leave details open to interpretation. We discuss our strict detection rules in Section 3.5 and used these throughout the whole study. However, these might differ from the rules used in the studies by Bavota *et al.* [BQO+12, BQO+15] to which we compare some of our results. Unfortunately, their tool is not open-source so we cannot discuss the similarities or differences in detail. Moreover, our tool incorporates semantic information (*i.e.,* type and symbol resolution, class hierarchy) next to syntactic information which is the only information considered by state-of-the-art tools. Fourth, the results might not hold for other test smells as we only covered 6 test smells that are among the most analysed smells in existing studies. We consider the analysis of the remaining test smells as future work. Finally, the corresponding production class for each test class is identified through naming conventions such as those used in existing studies [BQO+12, PDNP+16, SPZ+18]. We acknowledge that Socrates might have missed some links between production and test classes. Additionally, we assume a directory structure to detect the folder with unit tests (*i.e.,* `system/src/test`). However, this is only a convention which developers are not obliged to follow. This might have resulted in tests and test smells left to be undetected. In general, empirical studies with a large number of systems including industrial systems are desirable.

4.3 Conclusion

This chapter presented a survey and an empirical study about test smells in the Scala ecosystem. First, our survey assessed the perception of developers about test smells. The results indicate that developers are not completely aware of test smells. Second, our empirical study investigated the diffusion of test smells in Scala systems hosted on Github. The results indicate a lower diffusion of test smells in Scala systems compared to Java systems. The lower number of test smells could also indicate that test suites in the Scala ecosystem are suitable for test amplification, and hence used as input to our resilience testing approach.

Chapter 5

State of The Art in Resilience Testing

Resilience is a quality of contemporary systems that continues to gain traction over the past decade. However, its meaning is not always clear as the literature interchanges resilience with other software quality attributes such as availability, dependability and reliability. Moreover, the number of existing resilience testing approaches is limited and these approaches do not always provide the necessary means to efficiently detect resilience defects, let alone find defects in actor systems. This chapter therefore explores the state of the art in resilience testing.

Section 5.1 defines the meaning of resilience as well as its relation to other software quality attributes. Next, we discuss multiple techniques related to resilience testing. First, Section 5.2 presents fault injection along with its terminology and typical architecture. This technique is often used as foundation for resilience testing. Next, Section 5.3 presents Chaos Engineering which is an approach to test the resilience of a system in production environments through fault injection. Subsequently, we explain two techniques that can efficiently explore a fault space: Lineage-Driven Fault Injection (LDFI) and Delta Debugging (DD). Section 5.4 explains LDFI as a technique that leverages redundancy to efficiently explore the fault space, while Section 5.5 explains the necessary terminology and details of the delta debugging algorithms. We provide examples to illustrate the internal workings of each algorithm and motivate our choice for using delta debugging.

Section 5.6 then provides an overview of the state of the art. We categorize existing work into four categories based on their exploration strategy and assess each of these approaches on a set of properties that are desirable for resilience testing approaches. Finally, we summarize our observations in Section 5.7 and incorporate these into our approach to resilience testing presented in the next chapter.

5.1 Resilience and Its Meaning

Over the past decade, resilience continued to gain traction. However, its meaning is not always clear. The word resilience originates from the Latin verb *resilire*[1], which means "to jump back" or "to rebound". It is a combination of the prefix *re-* (*i.e.,* "back") and the verb *salire* (*i.e.,* "to jump") and thus literally means the act of jumping back. However, in order to determine what it means to be resilient in the year 2020 we start our discourse by determining contemporary definitions of resilience found in the literature:

> **The Reactive Manifesto[2]:** *"The system stays responsive in the face of failure. [...] Failures are contained within each component, isolating components from each other and thereby ensuring that parts of the system can fail and recover without compromising the system as a whole. [...]"*
>
> **Reactive Design Patterns [KHA17] and Merrian-Webster[3]:** *"The ability of a substance or object to spring back into shape or the capacity to recover quickly from difficulties"*
>
> **IBM[4]:** *"Software solution resiliency refers to the ability of a solution to absorb the impact of a problem in one or more parts of a system, while continuing to provide an acceptable service level to the business."*
>
> **The Dependability Community [Lap08]:** *"The persistence of service delivery that can justifiably be trusted, when facing changes."*

From these definitions, it is clear that resilience is a dynamic concept where action and reaction occurs when the system experiences conditions that are not normal or unlikely to happen (*e.g.,* network failures or disk failures). Given the different definitions, we informally define resilience in this dissertation as follows:

Definition 5.1. Resilience.
A system is resilient when it continues to deliver its services under abnormal conditions through detection and recovery. The system might temporarily degrade its services until recovered.

5.1.1 The Concepts of a Resilient System

Figure 5.1 shows an illustrative timeline of how resilient systems react under abnormal conditions. By default, a resilient system runs its normal operation until an abnormal condition occurs. For example, an abnormal condition could be a network disruption that breaks the communication between two services. However, this condition might not always affect the system.

[1] https://www.wordsense.eu/resilire

[2] https://www.reactivemanifesto.org

[3] https://www.merriam-webster.com/dictionary/resilience

[4] https://www.ibm.com/developerworks/websphere/techjournal/1407_col_nasser/1407_col_nasser.html

For example, it might be that those two services are not communicating at that point in time. The impact starts when abnormal conditions cause abnormal events and degrade the normal operation. For example, one service did not receive a reply because of a network disruption. Resilience mechanisms are able to detect these abnormal events and actively try to recover from the detected abnormalities. For example, the mechanism tries to send the message again as part of the recovery. In the worst case, one service might temporarily fail to communicate with another service. It might also be that the reply is received but with a high latency. However, this degradation should remain temporary because otherwise the system is not considered to be resilient. Of course, some degradations might not be recoverable because a system can only be resilient to what it can control. After a successful recovery of the resilience mechanism, the system continues its normal operation as nothing happened. This shows that detection and recovery are important parts of a resilient system.

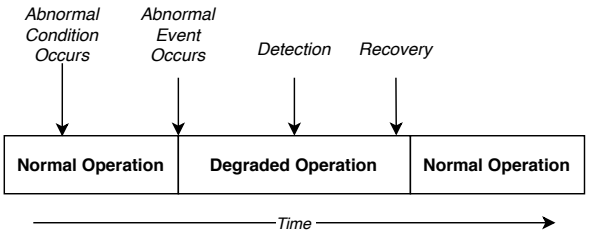

Figure 5.1: Resilient systems detect and recover from abnormal conditions.

It should also be clear that resilience is not a binary concept. Resilience is a matter of degree and differs between systems. For example, one system might be more resilient to disk failures, while the other might be more resilient to partial network failures. Additionally, resilience is closely related to other software quality attributes such as reliability, dependability, and availability.

For example, systems with high availability will have less impact of abnormal conditions and degraded operation during recovery since the system has other services available. Similarly, systems with high reliability have a low probability of failing. A system with low reliability could lead to more faults and thus require higher levels of resilience. A resilient system maintains a high reliability by providing its services under abnormal conditions. Performance is also related to resilience as resilience mechanisms need to detect and recover from abnormalities which could have an impact on throughput and latency. Dependability [ALR$^+$01, ALRL04] focusses on the broader concept of fault tolerance which can be achieved to means such as redundancy and resilience. Through these examples, it becomes clear that resilience is closely related to other quality attributes and indicates that a system should be more than resilient alone.

We summarize the key concepts of a resilient system in Figure 5.2. The goal of any system is to provide its services under normal operation. Resilient systems go one step further by also protecting them from failing under abnormal conditions and recovering them to their normal operation. This is done through incorporating resilience mechanisms such that they avoid the failure of services.

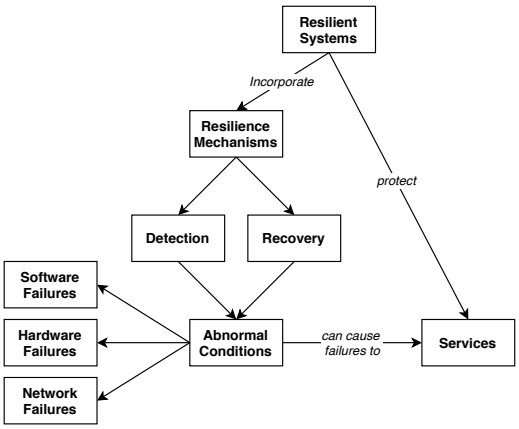

Figure 5.2: Resilient systems incorporate resilience mechanisms to protect their services from failing under abnormal conditions.

5.1.2 Incorporating Resilience Mechanisms is Difficult

Despite the need for resilience as reflected by frequent system outages [GDJ+11, HRJ+16], as well as new industrial practices such as Chaos Engineering (*cf.* Section 5.3) that advocate for testing resilience in production environments, implementing and testing system's resilience remains difficult.

5.1.2.1 The Implementation Process

The implementation of these resilience mechanisms remains non-trivial for multiple reasons. First, contemporary distributed systems are inherently complex due to their distributed communication and data requirements, as well as their dynamic scaling policies, cluster configurations, and infrastructure as code [Mor16]. Implementing resilience mechanisms to keep the system operational under abnormal conditions further increases the system's complexity. Second, studies such as [ECS15] have shown that resilience mechanisms (*e.g.,* try-catch blocks for exceptions) are commonly neglected by developers and tend to exhibit poor quality. Post-mortem reports of outages show that their cause is often due to missing or incorrect fault handling logic [JGS11]. Finally, resilience mechanisms can be spread across multiple components.

5.1.2.2 The Testing Process

Additionally, testing these resilience mechanisms under abnormal conditions is also hard for multiple reasons. As a result, developers remain unaware about whether their systems are indeed resilient.

First, developers often only test the most important features and focus on functionality under normal conditions [KBLJ13, BHP+17] (*i.e.,* "happy paths") due to timing and budget constraints [GZ13,BGP+17]. Heorhiadi *et al.* [HRJ+16] have a similar observation as they state that unit and integration tests are not able to catch exceptional behaviour since systems still crash even though all tests pass. Manually testing scenarios under abnormal conditions remains an additional, repeating, and time-consuming development task.

Second, the space of all possible combinations of faults is typically large. The majority of state-of-the-art approaches explore this space exhaustively or selectively. However, testing the resilience with an exhaustive approach might not always be feasible since it involves testing an exponential number of fault combinations. Selective approaches, in contrast, require developers to specify combinations of faults to be explored. However, this requires substantial effort from developers and might result in defects that are not found.

Third, guarantees about the resilience of individual components do not necessarily hold for systems composed from these components [ARH15]. Therefore, the interaction of multiple components should be tested under abnormal conditions to gain confidence in a system's resilience.

Finally, developers are accustomed to frameworks for unit and acceptance testing, yet these frameworks do not provide the necessary means to test a system under abnormal conditions. Currently, tests have to be cluttered with additional logic to simulate or trigger specific faults at certain moments. Developers might therefore skip testing the behaviour of their system under abnormal conditions. Testament to this is a study of crash reports [HRJ+16] which indicates that developers do not always test the resilience of the system under certain conditions.

5.2 Fault Injection

Failures can be prevented through extensive testing. *Fault Injection* [AALC96] is the process of deliberately introducing faults in a system in order to evaluate its ability to prevent or mitigate failures. Fault injection techniques can be either hardware-based, software-based, simulation-based, emulation-based, or based on a combination of multiple techniques. Each technique has its application domain, as well as it own advantages and disadvantages [ZAV+04]. In this dissertation, we focus on software-based fault injection.

5.2.1 Terminology

We use the terminology as defined by Avizienis *et al.* [ALR+01]. Figure 5.3 shows the relation between fault, error and failure.

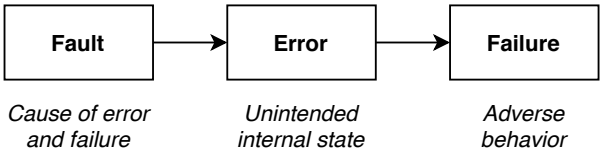

Figure 5.3: The relation between fault, error, and failure.

In a nutshell, a fault in a target might cause an error, and this error might result in a failure. Informally speaking, the granularity of fault targets can range from low-level instructions (*e.g.,* system calls) to language constructs (*e.g.,* exception handlers) to complete services (*e.g.,* payment service). A failure can occur due to the lack of mechanisms that prevent errors or the incorrect working of those mechanisms. For example, a wrong input (*i.e.,* fault) causes an exceptional state (*i.e.,* error), and this exception causes the system to crash (*i.e.,* failure). An exception handler could have prevented this failure.

Definition 5.2. Fault.
A fault is the adjudged or hypothesized cause of an error.

Definition 5.3. Error.
An error is that part of the system state that may cause a subsequent failure.

Definition 5.4. Failure.
A system failure is an event that occurs when the delivered service deviates from correct service.

Note that a fault might lead to different errors depending on the context. For example, a wrong input might cause a different exception depending on the state of the system. In turn, an error might be caused by several different faults or multiple faults at once. For example, two independent inputs or their combination might cause the same exception. Faults are said to be dormant when they do not result in an error, otherwise they are active. Similarly, errors are said to be latent when they are present but do not result in a failure. We also provide the following definitions related to our fault injection approach.

Definition 5.5. Fault Target.
A fault target is the component of a software system in which faults are injected.

Definition 5.6. Fault Type.
A fault type indicates what the injector should inject into a fault target.

Definition 5.7. Fault Model.
A fault model maps fault targets to fault types.

Definition 5.8. Fault Tuple.
A fault tuple consists of a fault target and a fault type defined by the fault model (*i.e.,* `(fault target, fault type)`) and is denoted as δ.

Definition 5.9. Fault space.
The fault space $\mathcal{F} = \{\delta_0, \delta_1, \ldots, \delta_n\}$ is the set of fault tuples determined by a given fault model and a given set of fault targets.

Definition 5.10. Fault Scenario. A fault scenario $f \subseteq \mathcal{F}$ consists of none, one or more faults. A fault scenario may or may not lead to a failure. We use $\emptyset$ to denote the empty fault scenario.

Given a system with components `C1` and `C2`, fault types `F1` and `F2`, and a fault model that maps `C1` $\mapsto$ `F1`, `C2` $\mapsto$ `F1`, and `C2` $\mapsto$ `F2`. A fault scenario could then be one of the following:

```
{}
{(C1, F1)}
{(C1, F1), (C2, F1)}
{(C2, F1), (C2, F2)}
```

That is, a fault scenario is an element of the power set of the fault space $\mathcal{F}$. Ordering does not matter here as the target should not only indicate the component, but also the timing if applicable.

Definition 5.11. Failure Scenario. A failure scenario is a fault scenario that causes a failure.

For example, the fault scenario `{(C2, F2)}` is a failure scenario when its injection results in a system crash. We denote this as follows:

`{(C2, F2)}` $\Rightarrow$ `crash`

5.2.2 Architecture

Figure 5.4 shows the architecture of fault injection approaches which consists of several components. The target is the system in which faults will be injected. Both the generator and injector stimulate the system, while the monitor only observes the system. The generator generates a workload or input which is used as to execute the system. The injector is responsible for injecting faults in certain fault targets. Finally, the monitor intercepts program execution and reports back information about the execution. Code instrumentation is used to add mechanisms that trigger a fault and intercept system calls to observe the system's execution during fault injection. The controller is the overarching component that manages the generator, injector and monitor. In this way, the monitor can communicate with the injector to indicate when a fault should be injected. A fault injection experiment typically consists of several runs where different fault scenarios are injected. We refer the reader to [ZAV+04, Nat11, NCM16] for more background information on fault injection.

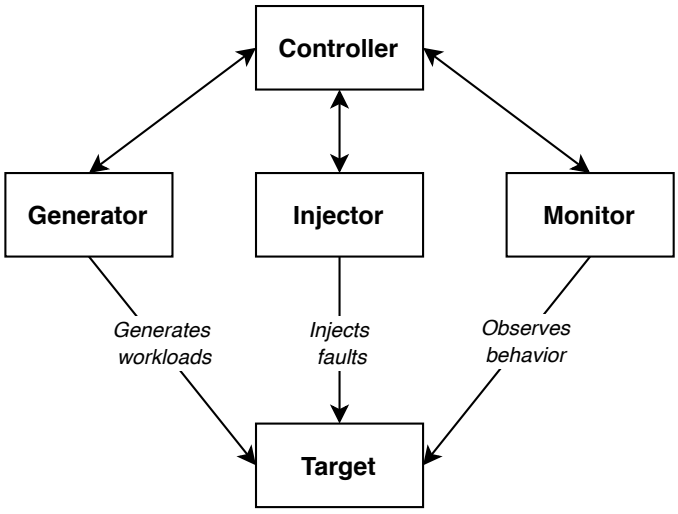

Figure 5.4: The typical architecture of a fault injection approach.

5.3 Chaos Engineering

Fault injection has been increasingly incorporated in resilience testing. In particular, NETFLIX began practising a form of resilience testing around 2008 [BBDR+16] by injecting faults in production systems.

At that moment, they made the shift from data centers to the cloud and figured out that a different approach was required to assess a system's resilience[5]:

> *"We have found that the best defence against major unexpected failures is to fail often. By frequently causing failures, we force our services to be built in a way that is more resilient."*

As a result, they devised a complete practice around this idea and named it *Chaos Engineering*[6] [BBDR+16]:

> *"Chaos Engineering is the discipline of experimenting on a system in order to build confidence in the system's capability to withstand turbulent conditions in production."*

Ideally, Chaos Engineering addresses the most significant weaknesses of systems before they affect services or users in production. There are five principles that describe an ideal application of Chaos Engineering:

1. **Build a hypothesis around steady-state behaviour.** Performance metrics such as throughput, error rates, and latency are usually used to define the steady state behaviour.
2. **Vary real-world events.** Typical events include hardware and software failures, but also non-failure events like a spike in requests.
3. **Run experiments in production.** Real traffic captures the most prominent paths through your system and will likely find real failures faster.
4. **Automate experiments to run continuously.** Manually experimenting with complex systems that change over time is not sustainable.
5. **Minimize blast radius.** Care must be taken when doing Chaos Engineering since experimenting in production can cause fatal consequences for services and users.

Chaos Engineering is considered to be a form of experimentation that generates new knowledge about the system. This differs from general testing as the outcome is known in advance. For example, chaos experiments could simulate hardware failures such as crashes, network disruptions, and disk failures where the granularity ranges from services to data centers, as well as simulate software failures such as throwing exceptions, low level I/O errors, and trigger different behaviour of methods.

[5] https://netflixtechblog.com

[6] http://principlesofchaos.org

5.3.1 Interest of Industry

The Chaos Engineering practice has been continuously improved over the last decade. As a result, companies have started adopting this practice and started developing other Chaos Engineering tools[7] to improve the resilience of their systems. In particular, Netflix has developed tools to facilitate and automate their experiments.

These tools have evolved in a suite of open-source tools called The Simian Army[8] to be used for systems hosted on Amazon Web Services (AWS). The Simian Army consists of services in the cloud for generating various kinds of failures, detecting abnormal conditions (*i.e.,* these services are called "monkeys" according to Netflix), and testing the ability to survive them. For instance, Chaos Monkey [CTBV15] randomly terminates virtual machine instances, Janitor Monkey searches for unused resources such as clusters or volumes for clean up, and Conformity Monkey detects instances that do not conform to predefined rules for security. Additionally, we present the most important tools and approaches that incorporate Chaos Engineering in Section 5.6.

Next to Netflix, there are many companies[9] interested in resilience testing their production systems including large organizations such as Google and Amazon [RKAL12], Microsoft [Nak15], and Facebook [Sve14]. Chaos Engineering has also been applied in software-intensive organizations such as financial institutions, healthcare provides, and online retailers [BBDR+16]. For example, some banks are known to verify the redundancy of their transactional systems by following the principles of Chaos Engineering. The online retailer Bol.com conducted a study[10] to tune resilience in their microservices architecture. This architecture is known to handle more than 10 million active users and over 22 million products at current times. This shows that companies see value in resilience testing and more companies are likely to incorporate methods like Chaos Engineering into their development process.

5.3.2 Observations

While Chaos Engineering is the dominant practice to assess resilience in production environments, we observe important similarities and distinctions with our approach to assess resilience in development environments.

> **Practice.** Chaos Engineering is a method for generating new information about the system when tested under abnormal conditions by investigating resilience hypotheses. For example, this practice aims to determine what happens with services or metrics when certain conditions occur. This gives developers insights and confidence that their systems will remain operational during these conditions.

[7] https://github.com/dastergon/awesome-chaos-engineering

[8] https://github.com/Netflix/SimianArmy

[9] https://coggle.it/diagram/WiKceGDAwgABrmyv

[10] https://dspace.library.uu.nl/handle/1874/366205

Our approach to resilience testing is a method to generate information as well. However, it will determine whether the functionality remains the same under certain conditions by leveraging the test oracle as source of truth (*i.e.,* assertions). This is in contrast to Chaos Engineering which typically doesn't have a desirable outcome in advance. It is a practice to determine these outcomes and does not focus on functional correctness.

Environment. The interest of Chaos Engineering lies in the behaviour of the system in production. The advantage of production environments is that user requests are representative behaviour of the system. This might lead to detecting problematic behaviours that might otherwise be difficult to foresee in development environments. However, we argue that the risk of experimenting with a production system might be too large when knowledge is limited. This dissertation therefore focuses on experimenting in development environments where we amplify existing test suites. Once developers are confident about their system's resilience in development environments, they can still apply practices such as Chaos Engineering in the production environment. As a result, we consider our approach as a first step to generate information about the system's functionality under abnormal conditions. Additionally, we argue that testing environments are becoming more representative of production environments due to containerization techniques such as Docker[11].

Oracle. The typical way to assess the effect of Chaos Engineering is by means of general metrics (*e.g.,* memory usage or throughput) or application-specific metrics (*e.g.,* number of video plays). Our approach leverages test oracles as a way to find resilience defects. The system might therefore be not resilient whenever the test oracle determines that the test fails.

5.4 Lineage-driven Fault Injection (LDFI)

Lineage-driven Fault Injection (LDFI) [ARH15] is a technique to efficiently explore fault scenarios to assess the fault tolerance of systems. In contrast to random or heuristic-based search strategies, LDFI is able to systematically cover the fault space and detect complex faults by leveraging data lineage [BKWC01, CWW00] and backwards reasoning. In this way, LDFI avoids exploring combinations of faults that are not possible in practice. For example, it can determine that certain fault scenarios can only be explored when a certain fault occurs first in the system's execution path.

5.4.0.1 Illustrative example

We explain the process of LDFI through an illustrative example, inspired by and adapted from [AAS+16].

[11] `https://docker.com`

The way that LDFI works is by capturing the lineage of a successful execution and then uses backwards reasoning. That is, it starts with the goal and reasons backward to find a sequence of steps that could prevent that successful execution. In this illustrative example, the successful outcome is data durability which is achieved through broadcasting the information to two independent databases. Such an outcome is typically defined by a correctness specification which can consist of invariants, assertions, pre and post conditions *etc.* In an informal way, backwards reasoning is similar to asking why an effect has happened (*i.e.,* which cause has which effect). Given our illustrative example, our first question therefore is:

"Why is this information durably stored?"

The answer to this question is that the information is stored in two separate databases. Consequently, we keep asking these kind of questions in order to determine the cause of this effect. The next question therefore is:

"Why is this information stored in two separate databases?"

The answer to this question is that a broadcast of that information is sent to both databases. Finally, the last question could be:

"Why is there a broadcast?"

The answer to this final question might be that the broadcast is the result of the client pressing a button. The result of this backwards reasoning about a successful outcome yields a lineage graph as shown in Figure 5.5. This is a directed graph where nodes represent steps and edges represent causality.

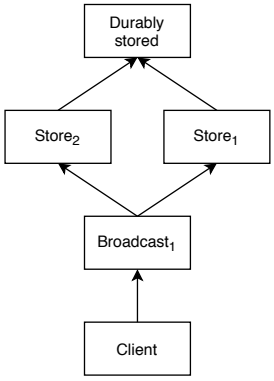

Figure 5.5: The lineage graph extracted from the system's execution.

That is, the lineage graph contains directed paths from causes to effects and hereby reveals redundant paths that yield the same outcome. Redundancy is an important part of LDFI as it considers the system to be fault tolerant when it finds an alternative (*i.e.,* redundant) execution path while injecting a fault scenario that achieves the same outcome.

Formally, the lineage graph consists of the nodes $\{Broadcast_1, Store_1, Store_2\}$. We omit the root and leaf node since these represent the cause and outcome. As a result, there are 2^3 combinations[12] that could prevent the desired outcome. For example, fault scenarios that target $Store_1$ or both $Store_1$ and $Broadcast_1$ might result in the data no longer being durable stored. However, this remains to be tested.

LDFI encodes these lineage graphs as a conjunction of disjunctions of propositional variables that can be either true or false. Every clause in the conjunction represents one path in the lineage graph. This is done for every path in the graph and results in a formula in conjunctive normal form (CNF) [Sch05]. For example, one path is encoded as a clause with two variables: $Store_1 \vee Broadcast_1$. The lineage graph from Figure 5.5 can be encoded as the combination of the two available paths:

$$\begin{aligned} & (Store_1 \vee Broadcast_1) \\ \wedge \quad & (Store_2 \vee Broadcast_1) \end{aligned}$$

A fault is simulated in a component (*e.g.,* message or database) when a variable is set to true in the formula. The goal of LDFI is, given such a formula, to find which variables need to be set to true to make the formula yield the value. This formula is solved as a SAT problem [DMB11] through a SAT solver (*e.g.,* Z3 [DMB08]). The system is correct with respect to a correctness specification when the formula is unsatisfiable (*i.e.,* the boolean formula results in false), otherwise a fault scenario will be presented (*i.e.,* the literals that are true). LDFI only generates the minimal solutions which results in the following 3 fault scenarios:

$$\begin{gathered} \{Broadcast_1\} \\ \{Store_1\} \\ \{Store_2\} \end{gathered}$$

For example, failing the broadcast (*i.e.,* setting $Broadcast_1$ to true) results in the formula to be true. Thus this fault has to be injected to determine whether the desired outcome indeed fails or whether redundant paths are found.

It is important to note that LDFI is an iterative process that extends the lineage graph over time with new knowledge (*i.e.,* redundant paths). For example, injecting a failure in $\{Broadcast_1\}$ (*i.e.,* the broadcast fails) results in a new lineage graph, as shown in Figure 5.6 on the next page.

[12] In fact, $2^3 - 1$ combinations as the empty combination cannot affect the outcome.

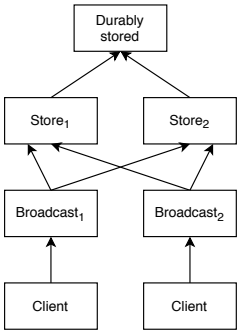

Figure 5.6: The extended lineage graph extracted from the system's execution.

In particular, this system seems to send multiple broadcasts whenever there was no acknowledgement from both databases. As a result, this system is fault tolerant with respect to the fault scenario $\{Broadcast_1\}$. The new lineage graph is encoded in the following CNF formula:

$$\begin{aligned} & (Store_1 \vee Broadcast_1) \\ \wedge \quad & (Store_2 \vee Broadcast_1) \\ \wedge \quad & (Store_1 \vee Broadcast_2) \\ \wedge \quad & (Store_2 \vee Broadcast_2) \end{aligned}$$

Solving this formula results in the following two minimal fault scenarios because setting these variables to true will resolve the formula to true (*i.e.,* the formula is satisfiable):

$$\{Store_1, Store_2\}$$
$$\{Broadcast_1, Broadcast_2\}$$

This is in contrast to a naive manner which would require to test all 2^4 combinations. This process is repeated up to some bound in time or the size of fault scenarios. For example, given that there are only two databases, it doesn't make sense to inject the first fault scenario as the outcome will always be incorrect. However, injecting the latter might again reveal new information about the system's redundancy as it might send the broadcast up to 5 times before a write is no longer durably stored. Again, it is important to understand that LDFI is able to explore the fault space much more efficient because execution paths and redundancy are explicit in the lineage graph.

We will not use LDFI in our approach because it focusses on redundancy and it poses several limitations. First, LDFI requires a fine-grained lineage in order to inject faults and be complete. In practice, collecting this lineage information seems not as trivial as it seems [AAS+16, CWC+19]. A trade-off between completeness and precision must be considered in order to get meaningful results. Second, LDFI returns solutions in no specific order and considers all faults as equal. This means that, while it covers the fault space and will find failures eventually, it might take longer to find them. Trivially, it is more beneficial when certain faults are prioritized and failures are found sooner. Finally, LDFI presents the decision problem as a SAT problem, which is known to be NP-complete[13]. Although a solution to an NP-complete problem can be verified fast (*i.e.,* verified in polynomial time), it remains slow (*i.e.,* exponential time) to find solutions.

5.5 Delta Debugging (DD)

"Yesterday, my program worked. Today, it does not. Why?" [Zel99] is a question that occurs ever so often during development. A good starting point is to examine the changes and narrow them down to a smaller set of changes that still causes the failure. However, manually examining large sets of changes quickly becomes infeasible. Delta Debugging [Zel99,ZH02] is a technique that automatically minimizes a given set of changes so that developers can immediately reproduce and resolve the failure. Zeller *et al.* proposed the automated delta debugging algorithm in [Zel99], but later revised it to the minimizing and general delta debugging algorithm [ZH02] which guarantees minimal solutions.

5.5.1 Terminology

The original papers present delta debugging in the context of minimizing program input or code changes. We adapt the definitions and algorithm presented in [Zel99, ZH02] to the context of our dissertation: minimizing fault scenarios.

Definition 5.12. The passing and failing fault scenario.
Delta debugging works by means of a passing fault scenario $f_{\checkmark}$ and a failing fault scenario $f_{\times}$. We consider $f_{\checkmark}$ to be initially empty (*i.e.,* $f_{\checkmark} = \emptyset$), while $f_{\times}$ consists of all possible faults (*i.e.,* $f_{\times} = \mathcal{F}$).

It will become clear that these two fault scenarios are the opposites of each other. For now, the reader can ignore the passing fault scenario ($f_{\checkmark}$) and only focus on the failing fault scenario ($f_{\times}$).

The delta debugging algorithm would typically start from a failing fault scenario that is much smaller than the set of all possible faults. The reason is that the algorithm was proposed as a debugging algorithm. Developers already had a failing fault scenario $f_{\times}$ and would use delta debugging to minimize it.

[13] https://en.wikipedia.org/wiki/NP-completeness

In our case, we do not have this information upfront which is why we start from the whole fault space. Note that $\mathcal{F}$ can be large for complex systems. For that reason, we choose to combine delta debugging in combination with test amplification (*cf.* **??**) to reduce this fault space. The set of faults is then only based on the execution of the test case which typically is only a part of the system's behaviour. Additionally, developers could manually reduce these failing scenarios $f_{✗}$ from the start by pruning or prioritizing certain faults (*cf.* Section 5.5.4). For example, developers might choose to not inject faults into certain components of the system. However, this requires domain knowledge which might not always be available.

Definition 5.13. The function *test*.
The function $test : \mathcal{P}(\mathcal{F}) \rightarrow \{✓, ✗, ?\}$ determines for a fault scenario $f \subseteq \mathcal{F}$ whether the test passes (✓) or not (✗). The function returns unresolved (*?*) whenever the result is indeterminate. Trivially, $test(f_{✓}) = test(\emptyset) = ✓$ and $test(f_{✗}) = test(\mathcal{F}) = ✗$.

The function *test* returns in all its simplicity the outcome of running the test case after injecting the given fault scenario. In our implementation, unresolved (*?*) is returned when the fault injector was not able to inject all faults. We start from the fact that $test(f_{✓}) = test(\emptyset) = ✓$ because *test* should pass when no faults are injected. Similarly, we start from $test(f_{✗}) = test(\mathcal{F}) = ✗$ because *test* should fail when all faults are injected. When $test(\emptyset) \neq ✓$ or $test(\mathcal{F}) \neq ✗$ there is no point in using delta debugging as the initial test already failed without faults or no failure was found in the fault space respectively.

Note that the goal of delta debugging is to minimize a given failing fault scenario to the smallest possible fault scenario. Ideally, that would mean finding the globally minimal fault scenario.

Definition 5.14. Globally minimal fault scenario.
A fault scenario $f \subseteq f_{✗}$ is the globally minimal fault scenario of $f_{✗}$ when it represents the set with the least faults that still causes the failure. That is, $\forall f' \subseteq f_{✗} : (|f'| < |f| \Rightarrow test(f') \neq ✗)$.

However, determining this global minimum is computationally expensive as it requires testing an exponential number of fault scenarios (*i.e.,* $2^{|f_{✗}|} - 2$). The two scenarios that are skipped are $test(f_{✓}) = ✓$ and $test(f_{✗}) = ✗$ as these are known. A typical alternative is then to find a local minimum.

Definition 5.15. Locally minimal fault scenario.
A fault scenario $f \subseteq f_{✗}$ is a locally minimal fault scenario of $f_{✗}$ when the following holds: $\forall f' \subset f : (test(f') \neq ✗)$.

Note that the set from which subsets are taken differs between a globally and locally minimal fault scenario (*i.e.,* $f_{✗}$ in contrast to f).

Although this might result in better performance when f is much smaller than $f_{✗}$, finding a local minimum still requires testing an exponential number of fault scenarios (*i.e.,* $2^{|f|} - 2$). Therefore, delta debugging searches for an approximation of the minimal fault scenarios. That is, a fault scenario that satisfies n-minimality: all fault scenarios that can be created by removing any combination of up to n faults from the n-minimal one, should no longer cause a failure (*i.e.,* ✓ or *?*).

Definition 5.16. ***n*-minimal fault scenario.**
The fault scenario $f \subseteq f_{✗}$ is n-minimal when all of testing subsets with up to n faults removed do not result in ✗: $\forall f' \subset f : |f| - |f'| \leq n \Rightarrow (test(f') \neq ✗)$

Naturally, we want to keep n small in order to have an acceptable time complexity and approximation. Therefore, the most interesting n-minimality is the 1-minimality: removing any single fault no longer makes it fail, but removing two or more faults might still make the fault scenario fail. This shows why it is an approximation in favour of performance: it might not be the globally minimal fault scenario.

Definition 5.17. 1-minimal fault scenario.
The 1-minimal fault scenario $f \subseteq f_{✗}$ is 1-minimal when the n-minimal difference holds with $n = 1$. That is, $\forall \delta_i \in f : test(f'_{✓} - \{\delta_i\}) \neq ✗$.

However, it is not efficient to test each scenario where a single element is removed. This brings us to the minimizing delta debugging algorithm [ZH02] which uses a strategy similar to binary search [Knu71]. We discuss its details on the next page.

5.5.2 The Minimizing Delta Debugging Algorithm

This minimizing delta debugging algorithm *ddm* [ZH02] takes as input a failing fault scenario $f_{\text{✗}}$ such that $test(f_{\text{✗}}) = \text{✗}$ and a function *test*. *ddm* calls the internal function ddm_2 which then recursively calls itself. We explain each of the four cases below the algorithm. The passing fault scenario $f'_{\text{✓}}$ remains $\emptyset$ through the whole algorithm. Its purpose will become clear in the remainder of this chapter where we discuss the general delta debugging algorithm (*cf.* Section 5.5.5). Delta debugging minimizes $f_{\text{✗}}$ by partitioning it into different subsets Δ_i. Each Δ_i has a cardinality equal (or almost equal due to uneven numbers) to $|\Delta_i| \approx |f'_{\text{✗}}|/n$, where the granularity n equals to 2 at the beginning to mimic a binary search. We define the function t as a typical implementation of the method $find$[14] on lists: it returns the first Δ_i that satisfies the condition $test(f \bullet \Delta_i) == \square$ where f is a fault scenario, $\bullet$ is a set operator (*i.e.,* $\cup$ or $-$), and $\square$ is one of the three possible outcomes of calling *test* (*i.e.,* ✓, ✗, or *?*). The algorithm proceeds to the next case when no Δ_i is found.

Definition 5.18. The Minimizing Delta Debugging Algorithm.

$$ddm(f_{\text{✗}}, test) = ddm_2(\emptyset, f_{\text{✗}}, 2)$$

$$ddm_2(f'_{\text{✓}}, f'_{\text{✗}}, n) = \begin{cases} ddm_2(f'_{\text{✓}}, \Delta', 2) & \text{if } \Delta' = t(f'_{\text{✓}}, \cup, \text{✗}) \quad (1) \\ ddm_2(f'_{\text{✓}}, f'_{\text{✗}} - \Delta', max(n-1, 2)) & \text{if } \Delta' = t(f'_{\text{✗}}, -, \text{✗}) \quad (2) \\ ddm_2(f'_{\text{✓}}, f'_{\text{✗}}, min(2n, |f'_{\text{✗}}|)) & \text{if } n < |f'_{\text{✗}}| \quad (3) \\ f'_{\text{✗}} & \text{otherwise} \quad (4) \end{cases}$$

where

$$f'_{\text{✗}} = \Delta_1 \cup \ldots \cup \Delta_n : (\forall i, j : \Delta_i \cap \Delta_j == \emptyset \wedge |\Delta_i| \approx |f'_{\text{✗}}|/n)$$

$$t(f, \bullet, \square) = \Delta_i : (\exists i \in \{1, \ldots, n\} : test(f \bullet \Delta_i) == \square)$$

precondition $ddm_2 : test(f'_{\text{✗}}) == \text{✗} \wedge n \leq |f'_{\text{✗}}|$

Case 1. Whenever the outcome of testing the fault scenario $f'_{\text{✓}} \cup \Delta_i$ is ✗, ddm_2 calls itself with $f'_{\text{✗}} = \Delta' = \Delta_i$ while keeping $n = 2$. Intuitively, we know that this fault scenario contains the faults we are looking for which is why we continue to minimize Δ' only.

Case 2. Whenever the outcome of testing the fault scenario $f'_{\text{✗}} - \Delta_i$ is ✗, we continue the search with $f'_{\text{✗}} = \Delta' = f'_{\text{✗}} - \Delta_i$ (*i.e.,* the complement), but change $n = max(n-1, 2)$. Intuitively, we know that the faults must be in the complement and not in the subset Δ_i as removing Δ_i still makes *test* fail. The reason why the recursive step chooses between the maximum $n-1$ and 2 is because the granularity stays the same in that way. Hence, some of the $n-1$ subsets from case 2 do not needed to be tested again in case 1. Otherwise, proceeding with $n = 2$ would require the algorithm to work down again until the previous granularity is achieved again.

[14] https://www.scala-lang.org/api/current/scala/collection/immutable/List.html#find(p:A=>Boolean):Option[A]

Case 3. All results of *test* gave either passing (✓) or indeterminate (*?*) results. To increase the chance of getting a ✗ as test outcome, the algorithm increases the granularity by dividing fault scenarios into $2n$ subsets instead of n. This is repeated until the granularity n reaches $|f'_{\text{✗}}|$. By then, the cardinality of each Δ_i equals to 1 and hence the failing fault scenario $f'_{\text{✗}}$ is 1-minimal already.

Case 4. The final step returns the 1-minimal failing fault scenario.

There are two important things to note. First, the function *test* must be deterministic, and thus the system must also have a deterministic execution. This is trivial to understand as *test* is the source of truth for the delta debugging algorithm. In order to support non-deterministic systems, developers have to ensure identical executions through record-replay mechanisms or frameworks. In this dissertation, we assume the execution to be deterministic. We refer the reader to Section 6.7.3 where we discuss this limitation in more detail.

Second, a fault scenario can cause multiple independent failures. The algorithm does not distinguish between those as *test* simply returns ✗ whenever there is a failure. Depending on the partitioning strategy, a different failure might be found instead of the original failure. This can be avoided by including more information about the failure instead of just returning the test outcome. For example, properties such as the location of the failure or the current call stack can be included. This means that the function *test* will only return ✗ when the failure was the same as the first found failure, while in all other cases it returns *?* whenever the found failure has different properties. This will return the fault scenario that caused the original failure. The remaining fault scenarios can still be found by repeating the delta debugging algorithm for each other failure. For simplicity, our implementation distinguishes between different failing test outcomes through their associated exception messages.

Figure 5.7 shows an illustrative example of the minimizing delta debugging algorithm where a fault scenario of 20 faults is simplified to a combination of 4 faults (1, 2, 3, 11) which required 72 executions of *test*. Step 3 is not explicitly shown but occurs when the value of n changes. Steps marked with a diamond (◇) are already tested in a previous test. For example, tests 3 and 4 perform the same test as in test 2 and 1 respectively. This indicates that a simple caching strategy can already reduce the number of performed tests. Indeed, caching enables the algorithm to skip 29 out of the 72 tests (40%) which results in 43 unique tests for the example below.

#	n	Δ_i	$\lvert\Delta_i\rvert$	Fault Scenario 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20	□	Case
1	2	Δ_0	10	1 2 3 4 5 6 7 8 9 10	✓	1
2	2	Δ_1	10	. . . . . 11 12 13 14 15 16 17 18 19 20	✓	1
3	2	Δ_0	10	. . . . . 11 12 13 14 15 16 17 18 19 20	✓	2°
4	2	Δ_1	10	1 2 3 4 5 6 7 8 9 10	✓	2°
5	4	Δ_0	5	1 2 3 4 5	✓	1
6	4	Δ_1	5	. . . . . 6 7 8 9 10	✓	1
7	4	Δ_2	5	. . . . . 11 12 13 14 15	✓	1
8	4	Δ_3	5	. . . . . 16 17 18 19 20	✓	1
9	4	Δ_0	15	. . . . . 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20	✓	2
10	4	Δ_1	15	1 2 3 4 5 . . . . . 11 12 13 14 15 16 17 18 19 20	✗	2
11	3	Δ_0	5	1 2 3 4 5	✓	1°
12	3	Δ_1	5	. . . . . 11 12 13 14 15	✓	1°
13	3	Δ_2	5	. . . . . 16 17 18 19 20	✓	1°
14	3	Δ_0	10	. . . . . 11 12 13 14 15 16 17 18 19 20	✓	2°
15	3	Δ_1	10	1 2 3 4 5 . . . . . 16 17 18 19 20	✓	2
16	3	Δ_2	10	1 2 3 4 5 . . . . . 11 12 13 14 15	✗	2
17	2	Δ_0	5	1 2 3 4 5	✓	1°
18	2	Δ_1	5	. . . . . 11 12 13 14 15	✓	1°
19	2	Δ_0	5	. . . . . 11 12 13 14 15	✓	2°
20	2	Δ_1	5	1 2 3 4 5	✓	2°
21	4	Δ_0	2	1 2	✓	1
22	4	Δ_1	2	. . 3 4	✓	1
23	4	Δ_2	2	. . . . 5 . . . . . 11	✓	1
24	4	Δ_3	2	. . . . . 12 13	✓	1
25	4	Δ_4	2	. . . . . 14 15	✓	1
26	4	Δ_0	8	. . 3 4 5 . . . . . 11 12 13 14 15	✓	2
27	4	Δ_1	8	1 2 . . 5 . . . . . 11 12 13 14 15	✓	2
28	4	Δ_2	8	1 2 3 4 . . . . . 12 13 14 15	✓	2
29	4	Δ_3	8	1 2 3 4 5 . . . . . 11 . . 14 15	✗	2
30	3	Δ_0	2	1 2	✓	1°
31	3	Δ_1	2	. . 3 4	✓	1°
32	3	Δ_2	2	. . . . 5 . . . . . 11	✓	1°
33	3	Δ_3	2	. . . . . 14 15	✓	1°
34	3	Δ_0	6	. . 3 4 5 . . . . . 11 . . 14 15	✓	2
35	3	Δ_1	6	1 2 . . 5 . . . . . 11 . . 14 15	✓	2
36	3	Δ_2	6	1 2 3 4 . . . . . 14 15	✓	2
37	3	Δ_3	6	1 2 3 4 5 . . . . . 11	✗	2
38	2	Δ_0	3	1 2 3	✓	1
39	2	Δ_1	3	. . . 4 5 . . . . . 11	✓	1
40	2	Δ_0	3	. . . 4 5 . . . . . 11	✓	2°
41	2	Δ_1	3	1 2 3	✓	2°
42	4	Δ_0	1	1	✓	1
43	4	Δ_1	1	. 2	✓	1
44	4	Δ_2	1	. . 3	✓	1
45	4	Δ_3	1	. . . 4	✓	1
46	4	Δ_4	1	. . . . 5	✓	1
47	4	Δ_5	1	. . . . . 11	✓	1
48	4	Δ_0	5	. 2 3 4 5 . . . . . 11	✓	2
49	4	Δ_1	5	1 . 3 4 5 . . . . . 11	✓	2
50	4	Δ_2	5	1 2 . 4 5 . . . . . 11	✓	2
51	4	Δ_3	5	1 2 3 . 5 . . . . . 11	✗	2
52	3	Δ_0	1	1	✓	1°
53	3	Δ_1	1	. 2	✓	1°
54	3	Δ_2	1	. . 3	✓	1°
55	3	Δ_3	1	. . . . 5	✓	1°
56	3	Δ_4	1	. . . . . 11	✓	1°
57	3	Δ_0	4	. 2 3 . 5 . . . . . 11	✓	2
58	3	Δ_1	4	1 . 3 . 5 . . . . . 11	✓	2
59	3	Δ_2	4	1 2 . . 5 . . . . . 11	✓	2
60	3	Δ_3	4	1 2 3 . . . . . 11	✗	2
61	2	Δ_0	2	1 2	✓	1°
62	2	Δ_1	2	. . 3 . . . . . 11	✓	1
63	2	Δ_0	2	. . 3 . . . . . 11	✓	2°
64	2	Δ_1	2	1 2	✓	2°
65	4	Δ_0	1	1	✓	1°
66	4	Δ_1	1	. 2	✓	1°
67	4	Δ_2	1	. . 3	✓	1°
68	4	Δ_3	1	. . . . . 11	✓	1°
69	4	Δ_0	3	. 2 3 . . . . . 11	✓	2
70	4	Δ_1	3	1 . 3 . . . . . 11	✓	2
71	4	Δ_2	3	1 2 . . . . . 11	✓	2
72	4	Δ_3	3	1 2 3	✓	2°

Figure 5.7: An example where a 1-minimal fault scenario is found in a non-monotone fault space.

5.5.3 Properties

Besides test execution being deterministic, the illustrative example above does not make any assumptions about the fault space. This brings us to two important properties.

Definition 5.19. Consistency.
A fault space is consistent if all its fault scenarios produce a determinate result. That is, $\forall f \subseteq \mathcal{F} : (test(f) \neq \textit{?})$.

While consistency will speed up the performance, this property might not always be present in practice. For example, injecting faults during test execution might stop the actor system from progressing. This could be considered an indeterminate result since the test didn't execute completely. Additionally, an indeterminate result also depends on the application domain. For example, it is much more likely to get indeterminate results when minimizing ASTs as not every AST node can be arbitrarily combined. As a result, compilation issues can arise which are considered to be indeterminate results as well.

Yet, delta debugging will still find a 1-minimal fault scenario even when the fault space is inconsistent. However, this will be at the expense of performance since the fault scenario will have to be partitioned in smaller ones, hence more fault scenarios have to be tested. This is guaranteed since the initial failing fault scenario was determined to fail. As a result, there must be a combination of faults that cause that failure.

Definition 5.20. Monotonicity.
A fault space is monotone if there are no compensating faults. Given a fault scenario f that fails the test, all fault scenarios consisting of at least all tuples of f will not pass. That is, $\forall f \subseteq \mathcal{F} : (test(f) = \text{✗} \Rightarrow \forall f' \supseteq f : (test(f') \neq \text{✓}))$. Similarly, given a fault scenario f that passes the test, all fault scenarios consisting of at most all tuples of f will not fail. That is, $\forall f \subseteq \mathcal{F} : (test(f) = \text{✓} \Rightarrow \forall f' \subseteq f : (test(f') \neq \text{✗}))$.

A monotone fault space implies that combinations of faults cannot cancel each other out. In this dissertation, we do not assume the fault space to be monotone. Monotonicity can speed up the performance as the function *test* can return ✓ whenever a superset of f has already passed the test and return ✗ whenever a subset of f has already failed the test. Naturally, checking this should be more efficient than doing a complete test execution. The delta debugging algorithm is able to find the 1-minimal fault scenario without any assumptions about monotonicity

Figure 5.8 shows the algorithm when a monotone fault space is assumed. Steps marked with a star (*) are skipped because the current fault scenario is either a subset of a cached passing fault scenario or a superset of a cached failing fault scenario. Indeed, besides the 29 tests that can be skipped due to caching, 26 additional tests can be skipped due to a monotone fault space. As a result, only 17 (23.6%) unique tests are required instead of the original 72.

#	n	Δ_i	$\lvert\Delta_i\rvert$	Fault Scenario 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20	□	Case
1	2	Δ_0	10	1 2 3 4 5 6 7 8 9 10	✓	1
2	2	Δ_1	10	. . . . . 11 12 13 14 15 16 17 18 19 20	✓	1
3	2	Δ_0	10	. . . . . 11 12 13 14 15 16 17 18 19 20	✓	2°
4	2	Δ_1	10	1 2 3 4 5 6 7 8 9 10	✓	2°
5	4	Δ_0	5	1 2 3 4 5	✓	1*
6	4	Δ_1	5	. . . . . 6 7 8 9 10	✓	1*
7	4	Δ_2	5	. . . . . 11 12 13 14 15	✓	1*
8	4	Δ_3	5	. . . . . 16 17 18 19 20	✓	1*
9	4	Δ_0	15	. . . . . 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20	✓	2
10	4	Δ_1	15	1 2 3 4 5 . . . . . 11 12 13 14 15 16 17 18 19 20	✗	2
11	3	Δ_0	5	1 2 3 4 5	✓	1°
12	3	Δ_1	5	. . . . . 11 12 13 14 15	✓	1°
13	3	Δ_2	5	. . . . . 16 17 18 19 20	✓	1°
14	3	Δ_0	10	. . . . . 11 12 13 14 15 16 17 18 19 20	✓	2°
15	3	Δ_1	10	1 2 3 4 5 . . . . . 16 17 18 19 20	✓	2
16	3	Δ_2	10	1 2 3 4 5 . . . . . 11 12 13 14 15	✗	2
17	2	Δ_0	5	1 2 3 4 5	✓	1°
18	2	Δ_1	5	. . . . . 11 12 13 14 15	✓	1°
19	2	Δ_0	5	. . . . . 11 12 13 14 15	✓	2°
20	2	Δ_1	5	1 2 3 4 5	✓	2°
21	4	Δ_0	2	1 2	✓	1*
22	4	Δ_1	2	. . 3 4	✓	1*
23	4	Δ_2	2	. . . . 5 . . . . . 11	✓	1
24	4	Δ_3	2	. . . . . 12 13	✓	1*
25	4	Δ_4	2	. . . . . 14 15	✓	1*
26	4	Δ_0	8	. . 3 4 5 . . . . . 11 12 13 14 15	✓	2
27	4	Δ_1	8	1 2 . . 5 . . . . . 11 12 13 14 15	✓	2
28	4	Δ_2	8	1 2 3 4 . . . . . 12 13 14 15	✓	2
29	4	Δ_3	8	1 2 3 4 5 . . . . . 11 . . 14 15	✗	2
30	3	Δ_0	2	1 2	✓	1°
31	3	Δ_1	2	. . 3 4	✓	1°
32	3	Δ_2	2	. . . . 5 . . . . . 11	✓	1°
33	3	Δ_3	2	. . . . . 14 15	✓	1°
34	3	Δ_0	6	. . 3 4 5 . . . . . 11 . . 14 15	✓	2*
35	3	Δ_1	6	1 2 . . 5 . . . . . 11 . . 14 15	✓	2*
36	3	Δ_2	6	1 2 3 4 . . . . . 14 15	✓	2*
37	3	Δ_3	6	1 2 3 4 5 . . . . . 11	✗	2
38	2	Δ_0	3	1 2 3	✓	1*
39	2	Δ_1	3	. . . 4 5 . . . . . 11	✓	1*
40	2	Δ_0	3	. . . 4 5 . . . . . 11	✓	2°
41	2	Δ_1	3	1 2 3	✓	2°
42	4	Δ_0	1	1	✓	1*
43	4	Δ_1	1	. 2	✓	1*
44	4	Δ_2	1	. . 3	✓	1*
45	4	Δ_3	1	. . . 4	✓	1*
46	4	Δ_4	1	. . . . 5	✓	1*
47	4	Δ_5	1	. . . . . 11	✓	1*
48	4	Δ_0	5	. 2 3 4 5 . . . . . 11	✓	2
49	4	Δ_1	5	1 . 3 4 5 . . . . . 11	✓	2
50	4	Δ_2	5	1 2 . 4 5 . . . . . 11	✓	2
51	4	Δ_3	5	1 2 3 . 5 . . . . . 11	✗	2
52	3	Δ_0	1	1	✓	1°
53	3	Δ_1	1	. 2	✓	1°
54	3	Δ_2	1	. . 3	✓	1°
55	3	Δ_3	1	. . . . 5	✓	1°
56	3	Δ_4	1	. . . . . 11	✓	1°
57	3	Δ_0	4	. 2 3 . 5 . . . . . 11	✓	2*
58	3	Δ_1	4	1 . 3 . 5 . . . . . 11	✓	2*
59	3	Δ_2	4	1 2 . . 5 . . . . . 11	✓	2*
60	3	Δ_3	4	1 2 3 . . . . . 11	✗	2
61	2	Δ_0	2	1 2	✓	1°
62	2	Δ_1	2	. . 3 . . . . . 11	✓	1*
63	2	Δ_0	2	. . 3 . . . . . 11	✓	2°
64	2	Δ_1	2	1 2	✓	2°
65	4	Δ_0	1	1	✓	1°
66	4	Δ_1	1	. 2	✓	1°
67	4	Δ_2	1	. . 3	✓	1°
68	4	Δ_3	1	. . . . . 11	✓	1°
69	4	Δ_0	3	. 2 3 . . . . . 11	✓	2*
70	4	Δ_1	3	1 . 3 . . . . . 11	✓	2*
71	4	Δ_2	3	1 2 . . . . . 11	✓	2*
72	4	Δ_3	3	1 2 3	✓	2°

Figure 5.8: An example where a 1-minimal fault scenario is found in a monotone fault space.

5.5.4 Partitioning Strategy

The performance of delta debugging is determined by the order in which each subset Δ_i is tested. While the algorithm seems to suggest that fault scenarios are partitioned and tested sequentially from left to right, it is not required. A fault scenario could sort its faults and then partition them such that each Δ_i is tested in a different order than partitioning an unordered fault scenario from left to right. Even better, these fault scenarios can be tested in parallel (*e.g.,* [HK16]).

We illustrate the possible gain and loss in performance by revisiting the illustrative example. By partitioning Δ_i in such a way that the faults 1, 2, 3, and 11 are together in a subset, we can actually achieve a better performance. Given that the fault space is monotone, we only need 12 unique tests compared to the 17 unique tests in Figure 5.8, an improvement of 29.4%. Similarly, in a non-monotone fault space the difference is even bigger. Here, we only need 16 tests instead of the 43 unique tests in Figure 5.7, an improvement of 62.8%.

#	n	Δ_i	$\lvert\Delta_i\rvert$	Fault Scenario 1 2 3 11 4 5 6 7 8 9 10 12 13 14 15 16 17 18 19 20	□	Case
1	2	Δ_0	10	1 2 3 11 4 5 6 7 8 9	✗	1
2	2	Δ_0	5	1 2 3 11 4	✗	1
3	2	Δ_0	2	1 2	✓	1
4	2	Δ_1	2	. . 3 11	✓	1
5	2	Δ_2	1	. . . . 4	✓	1
6	2	Δ_0	3	. . 3 11 4	✓	2
7	2	Δ_1	3	1 2 . . 4	✓	2
8	2	Δ_2	4	1 2 3 11	✗	2
9	2	Δ_0	2	1 2	✓	1°
10	2	Δ_1	2	. . 3 11	✓	1°
11	2	Δ_0	2	. . 3 11	✓	2°
12	2	Δ_1	2	1 2	✓	2°
13	4	Δ_0	1	1	✓	1*
14	4	Δ_1	1	. 2	✓	1*
15	4	Δ_2	1	. . 3	✓	1*
16	4	Δ_3	1	. . . 11	✓	1*
17	4	Δ_0	3	. 2 3 11	✓	2
18	4	Δ_1	3	1 . 3 11	✓	2
19	4	Δ_2	3	1 2 . 11	✓	2
20	4	Δ_3	3	1 2 3	✓	2

Figure 5.9: An optimal partitioning strategy can improve the performance of delta debugging.

However, a sub-optimal partitioning strategy can also deteriorate the performance. For example, Figure 5.10 shows the execution when the faults 1, 2, 3, and 11 no longer reside in the same Δ_i. In a monotone fault space, now 18 unique tests are required compared to the 17 unique tests in Figure 5.8. However, 67 unique tests are required when the fault space is non-monotone. This is in contrast to the 43 unique tests of Figure 5.7, a decline of 55.8%. Through these illustrative examples it becomes clear that partitioning strategies can affect the performance in both ways. We will later discuss several strategies to order faults based on actor-specific characteristics (*cf.* Section 6.6).

#	n	Δ_i	$\lvert\Delta_i\rvert$	Fault Scenario 1 4 5 6 7 2 8 9 10 11 12 13 14 15 16 17 18 19 20 3	□	Case
1	2	Δ_0	10	1 4 5 6 7 2 8 9 10 11	✓	1
2	2	Δ_1	10	. . . . . 12 13 14 15 16 17 18 19 20 3	✓	1
3	2	Δ_0	10	. . . . . 12 13 14 15 16 17 18 19 20 3	✓	2°
4	2	Δ_1	10	1 4 5 6 7 2 8 9 10 11	✓	2°
5	4	Δ_0	5	1 4 5 6 7	✓	1*
6	4	Δ_1	5	. . . . . 2 8 9 10 11	✓	1*
7	4	Δ_2	5	. . . . . 12 13 14 15 16	✓	1*
8	4	Δ_3	5	. . . . . 17 18 19 20 3	✓	1*
9	4	Δ_0	15	. . . . . 2 8 9 10 11 12 13 14 15 16 17 18 19 20 3	✓	2
10	4	Δ_1	15	1 4 5 6 7 . . . . 12 13 14 15 16 17 18 19 20 3	✓	2
11	4	Δ_2	15	1 4 5 6 7 2 8 9 10 11 . . . . . 17 18 19 20 3	✗	2
12	3	Δ_0	5	1 4 5 6 7	✓	1°
13	3	Δ_1	5	. . . . . 2 8 9 10 3	✓	1*
14	3	Δ_2	5	. . . . . 11 . . . . . 17 18 19 20 .	✓	1*
15	3	Δ_0	10	. . . . . 2 8 9 10 11 . . . . . 17 18 19 20 3	✓	2*
16	3	Δ_1	10	1 4 5 6 7 . . . . 11 . . . . . 17 18 19 20 .	✓	2
17	3	Δ_2	10	1 4 5 6 7 2 8 9 10 3	✓	2
18	6	Δ_0	2	1 4	✓	1*
19	6	Δ_1	2	. . 5 6	✓	1*
20	6	Δ_2	2	. . . . 7 3	✓	1*
21	6	Δ_3	2	. . . . . 2 8	✓	1*
22	6	Δ_4	2	. . . . . 9 10	✓	1*
23	6	Δ_5	2	. . . . . 11 . . . . . 17	✓	1*
24	6	Δ_6	2	. . . . . 18 19 . .	✓	1*
25	6	Δ_7	1	. . . . . 20 .	✓	1*
26	6	Δ_0	13	. . 5 6 7 2 8 9 10 11 . . . . . 17 18 19 20 3	✓	2
27	6	Δ_1	13	1 4 . . 7 2 8 9 10 11 . . . . . 17 18 19 20 3	✗	2
28	5	Δ_0	2	1 4	✓	1°
29	5	Δ_1	2	. . . . 7 2	✓	1*
30	5	Δ_2	2	. . . . . 8 9	✓	1*
31	5	Δ_3	2	. . . . . 10 11	✓	1*
32	5	Δ_4	2	. . . . . 17 18 . . .	✓	1*
33	5	Δ_5	2	. . . . . 19 20 .	✓	1*
34	5	Δ_6	1	 3	✓	1*
35	5	Δ_0	11	. . . . 7 2 8 9 10 11 . . . . 17 18 19 20 3	✓	2*
36	5	Δ_1	11	1 4 . . . . 8 9 10 11 . . . . 17 18 19 20 3	✓	2
37	5	Δ_2	11	1 4 . . 7 2 . . 10 11 . . . . 17 18 19 20 3	✗	2
38	4	Δ_0	2	1 4	✓	1°
39	4	Δ_1	2	. . . . 7 2	✓	1°
40	4	Δ_2	2	. . . . . 10 11	✓	1°
41	4	Δ_3	2	. . . . . 17 18 . . .	✓	1°
42	4	Δ_4	2	. . . . . 19 20 .	✓	1°
43	4	Δ_5	1	 3	✓	1°
44	4	Δ_0	9	. . . . 7 2 . . 10 11 . . . . 17 18 19 20 3	✓	2*
45	4	Δ_1	9	1 4 . . . . . 10 11 . . . . 17 18 19 20 3	✓	2*
46	4	Δ_2	9	1 4 . . 7 2 . . . . . 17 18 19 20 3	✓	2
47	4	Δ_3	9	1 4 . . 7 2 . . 10 11 . . . . . 19 20 3	✗	2
48	3	Δ_0	3	1 4 . . 7	✓	1*
49	3	Δ_1	3	. . . . . 2 . . 10 3	✓	1*
50	3	Δ_2	3	. . . . . 11 . . . . . 19 20 .	✓	1*
51	3	Δ_0	6	. . . . . 2 . . 10 11 . . . . . 19 20 3	✓	2*
52	3	Δ_1	6	1 4 . . 7 . . . . 11 . . . . . 19 20 .	✓	2*
53	3	Δ_2	6	1 4 . . 7 2 . . 10 3	✓	2*
54	6	Δ_0	1	1	✓	1*
55	6	Δ_1	1	. 4	✓	1*
56	6	Δ_2	1	. . . . 7	✓	1*
57	6	Δ_3	1	 3	✓	1°
58	6	Δ_4	1	. . . . . 2	✓	1*
59	6	Δ_5	1	. . . . . 10	✓	1*
60	6	Δ_6	1	. . . . . 11	✓	1*
61	6	Δ_7	1	. . . . . 19 . .	✓	1*
62	6	Δ_8	1	. . . . . 20 .	✓	1°
63	6	Δ_0	8	. 4 . . 7 2 . . 10 11 . . . . . 19 20 3	✓	2
64	6	Δ_1	8	1 . . . 7 2 . . 10 11 . . . . . 19 20 3	✗	2
65	5	Δ_0	1	1	✓	1°
66	5	Δ_1	1	. . . . 7	✓	1°
67	5	Δ_2	1	 3	✓	1°
68	5	Δ_3	1	. . . . . 2	✓	1°
69	5	Δ_4	1	. . . . . 10	✓	1°
70	5	Δ_5	1	. . . . . 11	✓	1°
71	5	Δ_6	1	. . . . . 19 . .	✓	1°
72	5	Δ_7	1	. . . . . 20 .	✓	1°
73	5	Δ_0	7	. . . . 7 2 . . 10 11 . . . . . 19 20 3	✓	2*
74	5	Δ_1	7	1 . . . . 2 . . 10 11 . . . . . 19 20 3	✗	2
75	4	Δ_0	1	1	✓	1°
76	4	Δ_1	1	. . . . 2	✓	1°
77	4	Δ_2	1	. . . . . 10	✓	1°
78	4	Δ_3	1	. . . . . 11	✓	1°
79	4	Δ_4	1	. . . . . 19 . .	✓	1°
80	4	Δ_5	1	. . . . . 20 .	✓	1°
81	4	Δ_6	1	 3	✓	1°
82	4	Δ_0	6	. . . . 2 . . 10 11 . . . . . 19 20 3	✓	2°
83	4	Δ_1	6	1 . . . . . 10 11 . . . . . 19 20 3	✓	2*
84	4	Δ_2	6	1 . . . . 2 . . . 11 . . . . . 19 20 3	✗	2
85	3	Δ_0	2	1 3	✓	1*
86	3	Δ_1	2	. . . . 2 . . . 11	✓	1*
87	3	Δ_2	2	. . . . . 19 20 .	✓	1°
88	3	Δ_0	4	. . . . 2 . . . 11 . . . . . 19 20 .	✓	2*
89	3	Δ_1	4	1 . . . . . 19 20 3	✓	2*
90	3	Δ_2	4	1 . . . . 2 . . . 11 3	✗	2
91	2	Δ_0	2	1 . . . . . 11	✓	1*
92	2	Δ_1	2	. . . . . 2 3	✓	1*
93	2	Δ_0	2	. . . . . 2 3	✓	2°
94	2	Δ_1	2	1 . . . . . 11	✓	2°
95	4	Δ_0	1	1	✓	1°
96	4	Δ_1	1	. . . . . 11	✓	1°
97	4	Δ_2	1	 3	✓	1°
98	4	Δ_3	1	. . . . . 2	✓	1°
99	4	Δ_0	3	. . . . . 2 . . . 11 3	✓	2*
100	4	Δ_1	3	1 . . . . 2 3	✓	2*
101	4	Δ_2	3	1 . . . . 2 . . . 11	✓	2*
102	4	Δ_3	3	1 . . . . . 11 3	✓	2*

Figure 5.10: A sub-optimal partitioning strategy can reduce the performance of delta debugging.

5.5.5 The General Delta Debugging Algorithm

As mentioned previously, the minimizing delta debugging algorithm produces a 1-minimal fault scenario as output. This means that we are only cutting away faults from the failing fault scenario in a way similar to binary search. However, minimization only considers the failing fault scenario ($f_{\text{✗}}$) and ignores the passing fault scenario ($f_{\text{✓}}$). Remember that the passing fault scenario remained equal to $\emptyset$ throughout the execution of the algorithm. However, the passing fault scenario can contain valuable information when the function *test* is expensive to call. In [ZH02], Zeller *et al.* indicate that it is generally more efficient to track this information to find the difference between a passing and failing fault scenario.

Therefore, the algorithm can be made more efficient by not only cutting away faults from the failing fault scenario, but also by adding faults to the passing fault scenario. By combining both strategies, we can narrow down the *difference* between a failing and a passing fault scenario. Intuitively, one is working from top to bottom (*i.e.,* minimizing the failing fault scenario), while another one is working from bottom to top (*i.e.,* maximizing the passing fault scenario), until they meet each other. As a result, the general delta debugging algorithm searches for n-minimal differences instead of n-minimal fault scenarios.

Definition 5.21. Minimal difference.
The difference $\Delta = f'_{\text{✗}} - f'_{\text{✓}}$ between two fault scenarios $f'_{\text{✓}}$ and $f'_{\text{✗}}$ with $\emptyset = f_{\text{✓}} \subseteq f'_{\text{✓}} \subset f'_{\text{✗}} \subseteq f_{\text{✗}}$ is minimal if for all subsets of Δ the following holds: adding a subset of faults to $f_{\text{✓}}$ no longer makes it pass, and removing that same subset from $f_{\text{✗}}$ no longer makes it fail. That is, $\forall \Delta_i \subset \Delta : test(f'_{\text{✓}} \cup \Delta_i) \neq \text{✓} \wedge test(f'_{\text{✗}} - \Delta_i) \neq \text{✗}$.

We adapt the definitions for n-minimality as follows:

Definition 5.22. n-minimal difference.
The difference $\Delta = f'_{\text{✗}} - f'_{\text{✓}}$ of two fault scenarios $f'_{\text{✓}}$ and $f'_{\text{✗}}$ with $\emptyset = f_{\text{✓}} \subseteq f'_{\text{✓}} \subset f'_{\text{✗}} \subseteq f_{\text{✗}}$ is n-minimal
when the minimal difference holds for all the subsets with a cardinality bounded by n. That is, $\forall \Delta_i \subset \Delta : (|\Delta_i| \leq n \Rightarrow (test(f'_{\text{✓}} \cup \Delta_i) \neq \text{✓} \wedge test(f'_{\text{✗}} - \Delta_i) \neq \text{✗}))$.

Definition 5.23. 1-minimal difference.
The 1-minimal difference $\Delta = f'_{\text{✗}} - f'_{\text{✓}}$ of two fault scenarios $f'_{\text{✓}}$ and $f'_{\text{✗}}$ when the n-minimal difference holds with $n = 1$. That is, $\forall \delta_i \in \Delta : |\Delta_i| \leq n \Rightarrow (test(f'_{\text{✓}} \cup \{\delta_i\}) \neq \text{✓} \wedge test(f'_{\text{✗}} - \{\delta_i\}) \neq \text{✗}$.

The concept of 1-minimality remains the same: a difference is 1-minimal when every individual fault contributes to the failure, but removing any combination of faults might still result in failure.

The general delta debugging algorithm takes as input a passing fault scenario $f_{\checkmark}$ such that $test(f_{\checkmark}) = test(\emptyset) = \checkmark$, a failing fault scenario $f_{\text{✗}}$ such that $test(f_{\text{✗}}) = \text{✗}$, and a function *test*. The partitioning strategy and the function t remain the same as with the minimizing delta debugging algorithm. We explain each of the six cases below the algorithm. The output is the tuple $(f'_{\checkmark}, f'_{\text{✗}})$ such that the difference $\Delta = f'_{\text{✗}} - f'_{\checkmark}$ is 1-minimal.

Definition 5.24. The General Delta Debugging Algorithm.

$$dd(f_{\text{✗}}, test) = dd_2(\emptyset, f_{\text{✗}}, 2)$$

$$dd_2(f'_{\checkmark}, f'_{\text{✗}}, n) = \begin{cases} dd_2(f'_{\checkmark}, f'_{\checkmark} \cup \Delta', 2) & \text{if } \Delta' = t(f'_{\checkmark}, \cup, \text{✗}) \quad (1) \\ dd_2(f'_{\text{✗}} - \Delta', f'_{\text{✗}}, 2) & \text{if } \Delta' = t(f'_{\text{✗}}, -, \checkmark) \quad (2) \\ dd_2(f'_{\checkmark} \cup \Delta', f'_{\text{✗}}, max(n-1, 2)) & \text{if } \Delta' = t(f'_{\checkmark}, \cup, \checkmark) \quad (3) \\ dd_2(f'_{\checkmark}, f'_{\text{✗}} - \Delta', max(n-1, 2)) & \text{if } \Delta' = t(f'_{\text{✗}}, -, \text{✗}) \quad (4) \\ dd_2(f'_{\checkmark}, f'_{\text{✗}}, min(2n, |\Delta|)) & \text{if } n < |\Delta| \quad (5) \\ (f'_{\checkmark}, f'_{\text{✗}}) & \text{otherwise} \quad (6) \end{cases}$$

where

$$\Delta = f'_{\text{✗}} - f'_{\checkmark} = \Delta_1 \cup \ldots \cup \Delta_n : (\forall i, j : \Delta_i \cap \Delta_j == \emptyset \wedge |\Delta_i| \approx |\Delta|/n)$$

$$t(f, \bullet, \square) = \Delta_i : (\exists i \in \{1, \ldots, n\} : test(f \bullet \Delta_i) == \square)$$

precondition $dd_2 : test(f'_{\checkmark}) == \checkmark \wedge test(f'_{\text{✗}}) == \text{✗} \wedge n \leq |\Delta|$

Case 1. Whenever a difference Δ_i is added to $f'_{\checkmark}$ and calling *test* with this difference fails (*i.e.,* $f'_{\checkmark} \cup \Delta_i == \text{✗}$), the recursive call continues with $f'_{\checkmark} = f'_{\checkmark}$, $f'_{\text{✗}} = f'_{\checkmark} \cup \Delta_i$, and $n = 2$. Intuitively, we know that *test* fails because either the difference Δ_i contains the faults or the faults are a combination of the passing fault scenario and the difference. Thus, we minimize the failing fault scenario.

Case 2. Whenever a difference Δ_i is removed from $f'_{\text{✗}}$ and calling *test* with this difference passes (*i.e.,* $f'_{\text{✗}} - \Delta_i == \checkmark$), the recursive call continues with $f'_{\checkmark} = f'_{\text{✗}} - \Delta_i$, $f'_{\text{✗}} = f'_{\text{✗}}$, and $n = 2$. Intuitively, we know that the difference contains the fault(s) as removing it from $f'_{\text{✗}}$ causes *test* to pass. Thus, we maximize the passing fault scenario.

Case 3. Whenever a difference Δ_i is added to $f'_{\checkmark}$ and calling *test* with this difference passes (*i.e.,* $f'_{\checkmark} \cup \Delta_i == \checkmark$), the recursive call continues with $f'_{\checkmark} = f'_{\checkmark} \cup \Delta_i$, $f'_{\text{✗}} = f'_{\text{✗}}$, and a possibly adapted n. Intuitively, we know that the difference does not contain any faults. Thus, we maximize the passing fault scenario.

Case 4. Whenever a difference Δ_i is removed from $f'_{\text{✗}}$ and calling *test* with this difference fails (*i.e.,* $f'_{\text{✗}} - \Delta_i == \text{✗}$), the recursive call continues with $f'_{\checkmark} = f'_{\checkmark}$, $f'_{\text{✗}} = f'_{\text{✗}} - \Delta_i$, and a possibly adapted n.

Case 5. All results of *test* gave indeterminate results. To increase the chance of getting a determinate (✓ or ✗) result, we can increase the granularity of each difference by dividing them into $2n$ parts instead of n. This is repeated until the granularity n is $|\Delta|$ as by then the difference consists of a single fault.

Case 6. The final step returns a passing and failing fault scenario of which the difference is 1-minimal.

The general delta debugging algorithm *dd* is a generalization of the minimizing delta debugging algorithm *ddm* presented earlier. This is easy to see as cases 2 and 3 are not executed when the function *test* only returns ✓ for $f'_{✓}$. However, it is important to note that the general delta debugging algorithms typically produces a different output then the minimizing delta debugging algorithm. The former searches for a 1-minimal difference between a passing and failing fault scenario, while the latter only searches for a 1-minimal failing fault scenario.

Figure 5.11 shows the application of the general delta debugging algorithm on the example from Figure 5.7. The algorithm yields the passing fault scenario $f'_{✓} = \{2, 3, 4, 5, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20\}$ and $\Delta = \{1\}$ as the 1-minimal difference.

#	n	Δ_i	$\lvert\Delta_i\rvert$	Fault Scenario 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20	□	Step
		$f_{✓}$		. . . . .		
		$f_{✗}$		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20		
		Δ		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20		
1	2	Δ_0	10	1 2 3 4 5 6 7 8 9 10	✓	1
2	2	Δ_1	10	. . . . . 11 12 13 14 15 16 17 18 19 20	✓	1
3	2	Δ_0	10	. . . . . 11 12 13 14 15 16 17 18 19 20	✓	$2^{\diamond}$
		$f_{✓}$		. . . . . 11 12 13 14 15 16 17 18 19 20		
		$f_{✗}$		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20		
		Δ		1 2 3 4 5 6 7 8 9 10		
4	2	Δ_0	15	1 2 3 4 5 . . . . 11 12 13 14 15 16 17 18 19 20	✗	1
		$f_{✓}$		. . . . . 11 12 13 14 15 16 17 18 19 20		
		$f_{✗}$		1 2 3 4 5 . . . . 11 12 13 14 15 16 17 18 19 20		
		Δ		1 2 3 4 5		
5	2	Δ_0	12	1 2 . . . . . 11 12 13 14 15 16 17 18 19 20	✓	1
6	2	Δ_1	12	. . 3 4 . . . . . 11 12 13 14 15 16 17 18 19 20	✓	1
7	2	Δ_2	11	. . . . 5 . . . . 11 12 13 14 15 16 17 18 19 20	✓	1
8	2	Δ_0	13	. . 3 4 5 . . . . 11 12 13 14 15 16 17 18 19 20	✓	2
		$f_{✓}$		. . 3 4 5 . . . . 11 12 13 14 15 16 17 18 19 20		
		$f_{✗}$		1 2 3 4 5 . . . . 11 12 13 14 15 16 17 18 19 20		
		Δ		1 2		
9	2	Δ_0	14	1 . 3 4 5 . . . . 11 12 13 14 15 16 17 18 19 20	✓	1
10	2	Δ_1	14	. 2 3 4 5 . . . . 11 12 13 14 15 16 17 18 19 20	✓	1
11	2	Δ_0	14	. 2 3 4 5 . . . . 11 12 13 14 15 16 17 18 19 20	✓	$2^{\diamond}$
		$f_{✓}$		. 2 3 4 5 . . . . 11 12 13 14 15 16 17 18 19 20		
		$f_{✗}$		1 2 3 4 5 . . . . 11 12 13 14 15 16 17 18 19 20		
		Δ		1		

Figure 5.11: The general delta debugging algorithm applied to our illustrative example.

Isolating this difference only requires 9 unique tests, compared to 43 unique tests for proving a 1-minimal fault scenario: an improvement of 79%. While the number of tests is indeed lower, developers have to take into account both the passing fault scenario and the 1-minimal difference which might pose difficulties for developers. In this case, developers still need to check 14 faults and are not aware that the failure is caused by the combination of faults 1, 2, 3, and 11. However, this might be clear for developers with domain knowledge. Nevertheless, the general delta debugging algorithm can give a significant improvement when the input are large fault scenarios (*e.g.,* 10^3 to 10^6 faults) as shown in [ZH02].

While both algorithms have their own purposes, we propose the following guidelines to developers. First, the partitioning strategy should be carefully selected as the examples show how it can affect performance. Secondly, we advise the use of the minimizing delta debugging algorithm in the ideal case as its output is a 1-minimal fault scenario at the expense of more tests. The general delta debugging algorithm could be applied when the function *test* takes a significant amount of time, the testing budget is limited, the fault space is large, or a combination of these. While fewer test executions are required, its output is a 1-minimal difference. However, the passing fault scenario can still be significantly large which might make it harder to find combinations of faults. However, for single faults the difference contains the single fault when the fault space is consistent. We conclude that there is no clear-cut choice between both algorithms as they have different purposes.

5.5.6 Complexity

To finalize the background on the delta debugging algorithm, we briefly discuss the best and worst cases in terms of time and difference cardinality. These complexities apply to both the minimizing and general delta debugging algorithm.

Definition 5.25. Time complexity.
The best-case time complexity of the delta debugging is $log_2(|f_{\text{✗}}|)$ when the fault space is consistent. The worst-case time complexity of delta debugging is quadratic: $|f_{\text{✗}}|^2 + 3|f_{\text{✗}}|$.

That is, the best case resembles a binary search algorithm. The example shown in Figure 5.12 presents an ideal case where the input is a fault scenario with 20 faults where only fault 1 causes a failure, all combinations with fault 1 fail, and all combinations without fault 1 pass. This example shows the logarithmic behaviour of delta debugging. That is, 4 tests are needed to find the fault since $log_2(|f_{\text{✗}}|) = log_2(20) \approx 4$. The outcome will be the same for both the minimizing and general delta debugging algorithm since only case 1 is executed for such cases.

#	n	Δ_i	$\lvert\Delta_i\rvert$	Fault Scenario 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20	□	Case
1	2	Δ_0	10	1 2 3 4 5 6 7 8 9 10	✗	1
2	2	Δ_0	5	1 2 3 4 5	✗	1
3	2	Δ_0	2	1 2	✗	1
4	2	Δ_0	1	1	✗	1

Figure 5.12: An illustrative example of the best-case performance.

The worst-case time complexity can be determined by splitting the algorithm into two phases: one phase where every test is indeterminate and thus requires doing recursive calls in case 3 which results in $4|f_{\text{✗}}|$ tests. The other phase is where the faults are found in the last test which results in $|f_{\text{✗}}|^2 - |f_{\text{✗}}|$ tests. This results in the worst-case time complexity of $4|f_{\text{✗}}| + |f_{\text{✗}}|^2 - |f_{\text{✗}}| = |f_{\text{✗}}|^2 + 3|f_{\text{✗}}|$. We refer the reader to [ZH02] for the complete proof. While a quadratic time complexity is still better than an exponential time complexity (*i.e.,* testing all subsets), Zeller *et al.* indicate that is unlikely that a fault scenario $f_{\text{✗}}$ would require $|f_{\text{✗}}|^2 + 3|f_{\text{✗}}|$ tests because case 1 quickly narrows down the fault scenario.

#	n	Δ_i	$\lvert\Delta_i\rvert$	Fault Scenario 1 2 3 4 5	□	Case
1	2	Δ_0	2	1 2 . . .	?	1
2	2	Δ_1	2	. . 3 4 .	?	1
3	2	Δ_2	1	 5	?	1
4	2	Δ_0	3	. . 3 4 5	?	2
5	2	Δ_1	3	1 2 . . 5	?	2
6	2	Δ_2	4	1 2 3 4 .	?	2
7	4	Δ_0	1	1	?	1
8	4	Δ_1	1	. 2 . . .	?	1
9	4	Δ_2	1	. . 3 . .	?	1
10	4	Δ_3	1	. . . 4 .	?	1
11	4	Δ_4	1	 5	?	1◇
12	4	Δ_0	4	. 2 3 4 5	?	2
13	4	Δ_1	4	1 . 3 4 5	?	2
14	4	Δ_2	4	1 2 . 4 5	?	2
15	4	Δ_3	4	1 2 3 . 5	?	2
16	4	Δ_4	4	1 2 3 4 .	?	2◇
17	5	Δ_0	1	1	?	1◇
18	5	Δ_1	1	. 2 . . .	?	1◇
19	5	Δ_2	1	. . 3 . .	?	1◇
20	5	Δ_3	1	. . . 4 .	?	1◇
21	5	Δ_4	1	 5	?	1◇
22	5	Δ_0	4	. 2 3 4 5	?	2◇
23	5	Δ_1	4	1 . 3 4 5	?	2◇
24	5	Δ_2	4	1 2 . 4 5	?	2◇
25	5	Δ_3	4	1 2 3 . 5	?	2◇
26	5	Δ_4	4	1 2 3 4 .	?	2◇

Figure 5.13: An illustrative example of the worst-case performance.

Definition 5.26. Difference cardinality.
The best-case cardinality of the 1-minimal difference Δ is $|dd(f_{\text{✗}})| = 1$ when the fault space is consistent.

That is, the general debugging algorithm always returns a single failure-inducing fault when there are only determinate results (✓ or ✗). This follows directly from the best-case time complexity. The cardinality of the difference might not be 1 when the fault space is not consistent, but a 1-minimal difference is always guaranteed.

5.6 Overview of Resilience Testing Approaches

We conduct a literature study to understand the current approaches to resilience testing. We review 15 approaches in the context of monolithic systems, distributed systems, microservice architectures, and mobile applications and summarize the results in Table 7.3. For each approach, we discuss the 8 properties discussed in Table 5.1. We group and discuss them according to the way they explore fault scenarios: developer-specified (Section 5.6.1), exhaustive (Section 5.6.2), LDFI (Section 5.6.3), and Delta Debugging (Section 5.6.4).

Properties	
Context	In which context are these approaches presented? What kind of systems are analysed?
Fault Model	Does the fault model support generic faults or faults specific a particular domain? What kind of fault targets does the fault model support? For example, approaches might only target a set of microservices and only inject failures related to the network.
Exploration	Is there a way to automatically generate and explore fault scenarios efficiently? Are fault scenarios automatically generated or do developers have to specify them by themselves? For example, automated approaches could reduce the effort of developers while manually-written fault scenarios might leave failures undetected.
Granularity	Does the approach explore combinations of faults? For example, approaches might only focus on combinations of two faults rather than a single one.
Pruning	Is it possible to narrow down the fault space with domain knowledge of developers? For example, developers might know that injecting faults in a specific component is useless because it has already been tested thoroughly or known to cause system failures. Pruning strategies could reduce the time to find failures.
Prioritization	Is it possible to prioritize certain fault scenarios according to strategies or developer knowledge? For example, historical data might indicate that new components should be tested first. Prioritizing strategies could reduce the time to find failures.
Environment	Is the approach used in a production or development environment? Is this choice motivated by risk or other reasons?
Oracle	What oracles are used to measure the resilience of a system after faults have been injected? For example, do developers need to manually check the outcome or is this automatically validated? Some approaches might require oracles or specifications in a custom language which might pose a barrier for usage.

Table 5.1: We discuss each approach based on the described 8 properties.

#	Name	Context	Fault Model	Exploration	Granularity	Pruning	Prioritization	Environment	Oracle
1	Gremlin [HRJ+16]	Microservice Architectures	Fail-stop/crash failures, performance/omission failures, and crash-recovery failures	Developer-specified	Multiple	No	No	Production	Gremlin Recipe
2	PreFail [JGS11]	Distributed Systems (Java)	Crash failures, disk failures, network failures, and node/rack-level network partitioning	Developer-specified	Multiple	Developer-specified policies	No	Development	Recovery Specification
3	Setsudo [GBJG15]	Distributed Systems (Java)	Node/link/disk failures, and network failures	Developer-specified	Multiple	No	No	Development	Metrics
4	Chaos Monkey [CTBV15]	Microservice Architectures (AWS)	Crash failures†	Exhaustive	Single	No	No	Production	Metrics
5	Fate and Destini [GDJ+11]	Distributed Systems (Java)	Crash failures, permanent disk failures, disk corruption failures, transient failures, and node/rack-level network partitioning	Exhaustive	Multiple	Dependency-based heuristics	No	Development	Destini Specification
6	ShortCircuit [CSM15]	Monolithic Systems (Java)	Try-catch failures	Exhaustive	Single	No	No	Development	Test Suites
7	ChaosMachine [ZMH+19]	Monolithic Systems (Java)	Try-catch failures	Exhaustive	Single	Developer-specified annotations	No	Production	Metrics
8	TripleAgent [ZM19]	Monolithic Systems (Java)	Try-catch failures	Exhaustive	Single	No	No	Development	Acceptability Oracle
9	ChaosOrca [SZM+19]	Microservice Architectures (Docker)	System call failures	Exhaustive	Single	Developer-specified calls	Call-based	Production	Metrics
10	Molly [ARH15]	Distributed Systems (Dedalus)	Crash-stop failures, message delivery failures, and (temporary) network partitions†	LDFI	Multiple	No	No	Development	Correctness Specification
11	ldfi-netflix [AAS+16]	Microservice Architectures	Crash failures, hardware failures, and software failures	LDFI	Multiple	No	No	Production	Metrics
12	ldfi-akka [Ghi19]	Actor Systems (Scala + Akka)	Crash failures and message delivery failures	LDFI	Multiple	No	No	Development	Correctness Specification
13	IntelliFT [CWC+19]	Microservice Architectures	Functional failures and performance failures	LDFI	Multiple	Feedback-based heuristics	Priority-based	Development	Test Suites
14	Madaari [RE19]	Microservice Architectures	Remote procedure call failures	LDFI	Multiple	No	Priority-based	Development	Test Suites
15	Thor [AMM15]	Mobile Applications (Android)	Android service failures	Delta Debugging	Multiple	Abstraction-based heuristics	No	Development	Test Suites
16	Chaokka [DBDNDR20]	Actor Systems (Scala + Akka)	Crash-recovery failures and message idempotency failures	Delta Debugging	Multiple	Causality-based heuristic	Priority-based	Development	Test Suites

Table 5.2: Overview of the start of the art work including our approach Chaokka at the bottom of the table. **Context.** Determines the kind of systems for which the approach is presented. **Fault Model.** Indicates the kind of failures that the fault model simulates. The symbol † indicates the lack of recovery failures (*e.g.*, crash-recovery failures). **Exploration.** Determines the way in which fault scenarios are explored. Manual requires developers to write specifications, while other techniques are automated. Exhaustive represents all (and possible redundant and infeasible) combinations of tuples. **Granularity.** Identifies whether the generated fault scenarios consist of single or multiple tuples. **Pruning.** Indicates whether additional optimizations are used to prune the fault space. **Prioritization.** Indicates whether fault scenarios are ordered by a specific property. **Environment.** Represents the environment in which the approach is presented and should be used. **Oracle.** Determines the way in which oracles are defined. Test suites and metrics are provided as-is and are part of the system, while specifications are not. These need to be written explicitly before the approach can be used and may require additional developer effort to learn the specification language.

5.6.1 Developer-specified Exploration of Fault Scenarios

This section discusses three approaches (*i.e.,* GREMLIN [HRJ+16], PREFAIL [JGS11], and SETSUDO [GBJG15]) that explore developer-specified fault scenarios.

5.6.1.1 Gremlin

Current post-mortem reports indicate that missing or faulty recovery logic are a significant cause of failures [HRJ+16]. However, this does not mean that developers do not implement resilience mechanisms. On the contrary, they implement them often but remain unaware of their effectiveness until the failure occurs [ECS15]. Heorhiadi *et al.* present GREMLIN [HRJ+16] as a resilience testing framework for microservice architectures which consist of loosely-coupled distributed processes that typically communicate via synchronous REST calls. Systems built with this architecture are becoming more prominent as the demand for horizontal scaling keeps increasing. However, moving towards such an architecture poses several challenges such as distributed communication and runtime heterogeneity that make resilience testing more difficult. GREMLIN has the following properties:

Fault Model. GREMLIN manipulates the network to detect failures of microservices. The reasons why GREMLIN does not directly inject faults into the microservices itself are threefold: failures can be emulated by manipulating the communication mechanisms between services (*e.g.,* drop messages to simulate a crashed service), failure recovery mechanisms can be observed from the network (*e.g.,* determine whether messages are retried), and runtime heterogeneity of microservices requires different instrumentation approaches. GREMLIN supports faults that cause the most common types of failures encountered in microservice architectures. These include failures due to a complete stop of a service (*i.e.,* crash-stop failures) and failures due to a service that recovers incorrectly (*i.e.,* crash-recovery failures), as well as performance failures.

Environment. GREMLIN is presented in the context of production environments. It has been integrated into IBM's Cloud[15] to test microservice architectures. GREMLIN does not generate the workload for the system and assumes that developers provide real or test requests.

Exploration. Fault scenarios are generated imperatively by developer-specified recipes as shown in Listing 5.1. The developers have the freedom to inject faults in any target (line 1), as well as use assertions as an oracle (line 2). Fault scenarios are thus one or more tuples. By means of these recipes, developers at IBM have successfully detected errors related to failure-handling logic and timeout handlers in a proprietary microservice application.

[15] https://www.ibm.com/cloud

However, there might exist a learning curve before developers become familiar with these recipes. A limitation of this approach is that it requires expertise and a mental model of the system which evolves over time. However, the authors indicate automatic recipe generation as a key area for future work.

```
1  Overload(ServiceB)
2  HasBoundedRetries(ServiceA, ServiceB, 5)
3
4  def Overload(Service1):
5    for s in dependents(Service1):
6      Abort(s, Service1, Error=503, Pattern='test-*',
7          On='request', Probability=.25)
8      Delay(s, Service1, Interval='100ms',
9          Pattern='test-*', On='request',Probability=.75)
```

Listing 5.1: A recipe in GREMLIN.

Pruning. Developers manually write recipes and hereby inherently prune the fault space. However, developers might also accidentally miss failures as recipes only represent certain fault scenarios. We therefore believe that pruning should only be used when the overall system has been thoroughly tested already (*e.g.,* skip faults in mature services).

Prioritization. There is no prioritization involved since fault scenarios are produced by the algorithmic recipes. It remains up to developers to decide in which order the recipes are executed. Nevertheless, developers might still outperform automated strategies with their domain knowledge.

Oracle. Each recipe contains assertions which are automatically checked. The focus of these assertions is mainly on the working of resilience mechanisms itself and not on the internal state of microservices. To this end, GREMLIN provides several base assertions to ease the work of the developer. For example, the assertion `HasBoundedRetries(Src, Dst, ...)` checks whether such a mechanism is implemented correctly between two given services. The developer can always query a centralized store with event logs in case custom assertions are needed.

5.6.1.2 PreFail

Joshi *et al.* [JGS11] empirically observed that recovery procedures of resilience mechanisms are often buggy in cloud systems such as HDFS, CASSANDRA, and ZOOKEEPER. The major reasons are that developers fail to anticipate failures or that they incorrectly implement the recovery mechanism. The authors indicate that previous work primarily addresses single failures during program execution despite cloud systems facing multiple failures. This is understandable as the challenge lies in the combinatorial explosion of multiple failures. In general, it is not feasible to explore all failure scenarios in practice. However, domain knowledge of developers can drastically reduce the fault space to be explored.

To this end, the authors present PreFail, a programmable tool for multiple-failure injection where developers can express heuristics to reduce the space of failure scenarios. The tool is split up into a failure-injection engine and failure-injection driver. The former is an fault injection tool that injects prescribed faults in components, while the latter generates fault scenarios based on developer-specified heuristics. PreFail has the following properties:

Fault Model. The targets of PreFail are defined by the failure-injection engine and includes Java library calls, network-level calls, disk-level calls, and system calls. PreFail then injects faults to determine the system's resilience to partitioning failures related to node, disk, and network.

Environment. PreFail is presented in the context of development environments. It requires a workload, but the format of these workloads remains unclear. One limitation is that PreFail does not control all kinds of non-determinism such as network message ordering.

Exploration. PreFail is generic and supports different use cases where fault scenarios can consists of one or more tuples. Combinations of two or three tuples are automatically generated but are later pruned by the policies provided by developers. This is essential to mitigate combinatorial explosion in the number of failure scenarios and makes testing for combinations feasible. An example of a heuristic is shown below and written in Python. This policy prunes away failure sequences where crashes are injected after a write I/O.

```
1 def flt(fs):
2   for f in failure_sequences:
3     fp = failure_injection_point(f)
4     isCrash = (fp['failure'] == 'crash')
5     isWrite = (fp['ioType'] == 'write')
6     isBefore = (fp['place'] == 'before')
7     if isCrash and (not (isWrite and isBefore)):
8       return False
9   return True
```

Listing 5.2: A policy in PreFail.

Pruning. There are many different ways in which developers can reduce the fault space. For example, policies can reduce the number of fault scenarios based on the kind of fault targets (*e.g.,* only two out of three replicas), the kind of fault types (*e.g.,* those that cause rack or disk failures), domain knowledge (*e.g.,* only target the first file write), probabilities (*e.g.,* likelihood of rack or disk failures), *etc.*

Prioritization. While the heuristics reduce the number of fault scenarios, they do not impose an order in which they are explored.

Oracle. The authors indicate that they wrote recovery specifications for every target workload to capture recovery bugs in the system. However, the language and kind of assertions of these specifications remains unclear.

5.6.1.3 Setsudo

Modern distributed systems are often designed to be fault-tolerant to partitioning. However, Ganai *et al.* [GBJG15] state that partition-tolerance is not tested rigorously in practice. To this end, the authors propose SETSUDO, a testing framework for distributed systems. This framework works with policies defined in the Perturbation Testing Policy Language (PTPL). This policy language enables developers to express fault scenarios in a declarative style with four constructs: targets, actions, pre hooks, and post hooks. Each of these can be combined without limitations to express fault scenarios. SETSUDO automatically translates these fault scenarios into actual fault injections during execution by intercepting targets through ASPECTJ and simulating varying failures. SETSUDO has the following properties:

Fault Model. SETSUDO supports different kinds of faults such as the ones that simulate network failures, network congestions, disk failures, data corruption, and node crashes. The fault targets vary between nodes, disks, links, *etc.*

Environment. SETSUDO is a testing framework and presented in the context of development environments. The authors indicate that client-side workloads are required but do not give information about their format.

Exploration. While fault scenarios are automatically generated from a test policy, it remains up to developers to provide the policy. Fault scenarios are one or more tuples. An example of such a policy S is shown below where each tuple x_i consists of a precondition, target, action, and postcondition. For example, `state-healthy` (line 5) is a predicate that checks the service is running, while `node-shardLeader-1` (line 8) refers to the service that is currently the leader.

```
1  S =(x0*(x1*((x2+(x3+x0))*x0)))
2
3  x0 = (state-solrSteady and state-zkSteady,
4  node-client, check-health, abort-error)
5  x1 = (state-healthy, node-client,
6  request-indexEmpty, state-indexEmpty)
7  x2 = (x21 * x22)
8  x21 = (true, node-shardLeader-1, down, wait-timed)
9  x22 = (true, node-shardLeader-1, up, wait-timed)
10 x3 = (x31 * x32)
11 x31 = (true, node-shardNonLeader-all, down, wait-timed)
12 x32 = (true, node-shardNonLeader-all, up, wait-timed)
```

Listing 5.3: A policy in SETSUDO.

Pruning. Developers manually write policies and hereby inherently prune fault scenarios. However, this can still generate many redundant fault scenarios which are not pruned. The authors plan to automatically remove such redundant fault scenarios in future work.

Prioritization. There is no prioritization of fault scenarios. While the authors indicate this is part of their future work, they give no information about how such a prioritization would work.

Oracle. The oracle is based on one of the following three anomalies: a slow response, no response, or an incorrect response. A monitor checks for any of these anomalies during each fault scenario execution. It automatically reports failures and stores information for later diagnosis.

5.6.2 Exhaustive Exploration of Fault Scenarios

This section discusses six approaches (*i.e.,* Chaos Monkey [CTBV15], Fate and Destini [GDJ+11], ChaosMachine [ZMH+19], ShortCircuit [CSM15], TripleAgent [ZM19], and ChaosOrca [SZM+19]) that automatically and exhaustively generate fault scenarios.

5.6.2.1 Chaos Monkey

Around 2010, Netflix presented Chaos Monkey [CTBV15] as one of the first Chaos Engineering tools. Netflix had started moving to the cloud a couple of years earlier because vertical scaling resulted in many points of failure and disruptions. It was expected that a shift to the cloud would reduce infrastructure-related issues since they no longer had to manage their own infrastructure and horizontal scaling would reduce single points of failure. Of course the cloud still incurs failures from time to time. Netflix needed a new approach to make their microservice architecture resilient to occasionally disappearing instances of their services on Amazon Web Services (AWS). Chaos Monkey has the following properties:

Fault Model. The fault targets are service instances deployed on AWS and the faults simulate terminations. Terminating an instance can simulate both network failures as well as instance failures. It does this at a much more frequent rate than typically seen in reality in order to gain confidence that an unlikely termination would not cause service disruptions. Chaos Monkey automatically detects the running instances. This is required as it is not feasible for developers to manually specify each instance in Netflix's microservice architecture. Not only does the architecture consists of thousands of instances running in the cloud, services also change over time, both in terms of location (*e.g.,* due to cluster migration) and number (*e.g.,* due to elasticity).

Environment. Given its goal, it is clear that Chaos Monkey targets production environments and uses real user requests as the workload. In particular, it targets systems running on AWS instances.

Exploration. Fault scenarios are automatically generated but contain only one tuple consisting of an AWS instance and a termination fault.

However, approaches that randomly generate fault scenarios cannot quantify progress. For example, it remains unclear whether all services were explored, neither whether all critical ones were considered. However, common reasons to use random exploration is its simplicity and its applicability in a setting where the system and its environment are considered a black box.

Pruning. The first version of Chaos Monkey did not allow developers to specify particular instances of services. Over the years, Chaos Monkey has become more sophisticated in the way it allows developers to specify termination frequencies, grouping, and exceptions[16].

Prioritization. Chaos Monkey randomly shuts down instances. As a result, there is not prioritization involved for services that might need more attention. For example, a recently-deployed service might cause more failures than mature ones.

Oracle. The approach of Chaos Monkey does not involve an automated oracle. It remains up to developers to manually assess the impact of these terminations through metrics and logs. This is also a reason why Chaos Monkey only terminates instances during business hours.

5.6.2.2 Fate and Destini

Gunawi *et al.* [GDJ+11] argue that failure recovery is challenging in cloud systems and indicate that it remains hard to systematically explore and test recovery mechanisms with current testing frameworks. The authors propose two advancements in cloud recovery testing: Fate and Destini. Fate systematically explores combinations of multiple failures and addresses the challenge of exponential explosion by means of pruning strategies. Destini is a specification language that enables developers to specify recovery behaviour. Fate has the following properties:

Fault Model. Fate targets so-called I/O points. These points are system or library calls that perform disk or network I/O. The fault model of Fate consists of faults that cause six failures: crash, permanent disk failure, disk corruption, node-level and rack-level partitioning, and transient failure.

Environment. The main purpose of Fate is to test recovery mechanisms during development. A workload is required to run the tool which can either be real or test requests to the cloud system. One limitation of Fate does not control all kinds of non-determinism such as network message ordering.

Exploration. Fault scenarios are one or more tuples. By default, Fate generates fault scenarios automatically and exhaustively. However, the authors indicate that exhaustive exploration remains a challenge.

[16] https://netflix.github.io/chaosmonkey/Configuring-behavior-via-Spinnaker

Pruning. Given the large fault scenario space, a simple filter can be used to reduce the fault space. For example, only faults in specific components should be explored. FATE also provides two built-in pruning strategies which can be enabled by developers to reduce the number of fault scenarios. These strategies are based on dependencies between faults. The first strategy effectively prunes away fault scenarios when two faults are not causally related (*i.e.,* the second occurs independently of the first). The second strategy prunes away fault scenarios where symmetric code is involved (*i.e.,* identical code that runs concurrently on different nodes).

Prioritization. While FATE supports so-called prioritization strategies, they only prune the search space but do not impose an ordering in which the fault scenarios are explored.

Oracle. While developers are accustomed to testing frameworks, the authors claim that these are limited in expressing recovery specifications. Therefore, DESTINI provides a DATALOG-based specification language that enables developers to write recovery specifications. An example of such a specification is shown below which informally throws an error whenever a `cnpEv` event occurs and the system is not in the expected state `stateY` for a given set of I/O points (P_i). FATE itself informs DESTINI about failure events such that it can decide whether the recovery specification is violated or not. However, this specification language might be an obstacle for developers as it imposes a learning curve. Moreover, replacing DESTINI does not seem possible as it is tightly coupled to FATE.

```
1  errX(P1,P2,P3) :- cnpEv (P1), NOT-IN stateY(P1,P2,_)
```

Listing 5.4: A DESTINI recovery specification.

5.6.2.3 ChaosMachine

Exceptions represent unexpected and abnormal conditions in a system. They occur so frequently that the majority of programming languages provide try-catch constructs to specify exceptional behaviour which is typically different from the normal one. It is essential that systems are resilient to these exceptions. Zhang *et al.* present CHAOSMACHINE [ZMH+19] as a tool that reveals the strengths and weaknesses of every try-catch block in the system. The approach inserts additional bytecode at the beginning of each block such that exceptions can be thrown during execution. The inserted code communicates with a controller that orchestrates the experiments to assess the resilience of the system. CHAOSMACHINE has the following properties:

Fault Model. CHAOSMACHINE tests the resilience of try-catch blocks by means of inserting code that throws exceptions at run-time. These exceptions vary across systems as they depend on the type of the exceptions that can be thrown.

The fault model is thus based on try-catch blocks as targets and exceptions as their faults. These blocks are automatically detected through the use of ASM[17]. Exceptions are only thrown at the beginning of a block. Note that a try-catch block can be split up to catch different kinds of exceptions. A fault tuple consist of a try-catch block, an exception, and a number that indicates how many times this exception has to be thrown (*e.g.,* only the first 10 times or always)

Environment. As the name suggests, ChaosMachine follows the principles of Chaos Engineering (*cf.* Section 5.3). That is, ChaosMachine allows developers to specify, discover and falsify hypotheses in order to assess the system's resilience to exceptions. By definition, this means that the tool is presented in the context of production environments.

Exploration. Fault scenarios are automatically generated and consist of a single tuple. Try-catch blocks are tested one at a time thus combinations of exceptions are not considered. By default, every try-catch block is exhaustively explored to detect failures.

Pruning. ChaosMachine considers hypotheses to test through developer-specified annotations as shown in Listing 5.5. Automated and exhaustive discovery is supported as well. Moreover, developers can limit hypothesis discovery to a specific package by configuring this explicitly. As a result, certain fault scenarios can be pruned by developers. This gives developers the freedom to leverage their domain knowledge in order to select critical try-catch blocks and validate their corresponding hypothesis.

```
1  state = SystemState.A;
2  try {
3    ... // an error is thrown
4    state = SystemState.B;
5  } catch (
6  @ChaosMachine(hypoth=Hypoth.RESILIENT)
7  Exception e
8  ) {
9    ... // handles the exception
10   state = SystemState.B;
11 }
```

Listing 5.5: A developer-specified resilience hypothesis.

Prioritization. ChaosMachine does not prioritize the try-catch blocks.

Oracle. The oracle is based on metrics gathered from the production environment through monitoring (*e.g.,* CPU usage) or developer-specified metrics. Developers have to specify thresholds on these metrics in order to determine the outcome of a hypothesis. According to these metrics, try-catch blocks are classified in four categories: resilient (*i.e.,* equivalent behaviour), observable (*i.e.,* user-visible effects), debuggable (*i.e.,* exception is logged), and silent (*i.e.,* neither observable nor debugging).

[17] https://asm.ow2.io

5.6.2.4 ShortCircuit

Cornu *et al.* [CSM15] focus on resilience against unanticipated exceptions. To this end, they present an approach called *short-circuit testing* (for brevity, we call this approach ShortCircuit) which aims to test resilience against exceptions by analysing their corresponding try-catch blocks. Each try-catch block is automatically detected through a static analysis and the collected type of the exception of each block is simply its statically declared type. These blocks are assessed according to two contracts: source-independence and pure resilience. The former indicates that a block's behaviour is identical under abnormal conditions (*i.e.,* when the try-catch block is executed), but still differs from behaviour under normal conditions. The latter indicates that the block's behaviour is always identical, regardless under which conditions. Moreover, source-independent blocks can be refactored to catch more exceptions. The authors propose *catch-stretching* to replace the exception type with one of its super types with as goal to make the try-catch block more resilient. ShortCircuit has the following properties:

Fault Model. The targets of this approach are try-catch blocks. The authors use a static analysis to determine all try-catch blocks and issue a standard run of all test cases to determine whether a test covers a try-catch block. In contrast to previous approaches, one could argue that this fault model has a dynamic behaviour (*i.e.,* the faults are based on the statically declared exception types which differ between programs).

Environment. ShortCircuit is meant to be used during development as it leverages existing test suites. This relates to the concept of Test Amplification (*cf.* **??**), similar to what we use in our approach.

Exploration. A fault scenario is automatically generated and consists of a single tuple. Only one try-catch block is tested at a time and an exception is only thrown at the beginning of a block. The approach exhaustively explores fault scenarios with try-catch blocks that are not explicitly triggered.

Pruning. ShortCircuit does not prune fault scenarios. Given a program and its test suite, it will detect all try-catch blocks in the program and assess them all by triggering an exception during successive test executions.

Prioritization. The authors do not indicate the possibility of prioritizing certain try-catch blocks. We believe ordering try-catch blocks (*e.g.,* by the number of calls to their enclosing method) can speed up detection.

Oracle. ShortCircuit uses test suites as an oracle to assess program correctness and to determine whether the try-catch blocks are source-independent or resilient. The latter can informally be considered as "the perfect plan B" (*e.g.,* read from database instead of cache) and is by construction source-independent. Systems with such try-catch blocks are therefore more desirable since behaviour remains the same under any condition.

5.6.2.5 TripleAgent

Zhang *et al.* [ZM19] propose failure-oblivious computing [RCD+04] to make systems resilient to uncaught and incorrectly handled exceptions. Their approach is implemented in the tool TripleAgent and serves two main purposes. First, it classifies locations (*i.e.,* statements in methods that can throw exceptions) into three categories according to the effect of the exception. A location is classified as either fragile (*i.e.,* a single exception causes unexpected behaviour), sensitive (*i.e.,* multiple exceptions cause unexpected behaviour), or immunized (*i.e.,* resilient to any number of exceptions). Second, it automatically indicates improvements to exception-handling problems. Every method on the stack after the source of an exception but before the default handling method can be a failure-oblivious method. A method is failure-oblivious when an instrumentation of it with a default try-catch block has no effect on the outcome when an exception is thrown in a method below. These failure-oblivious methods provide the same resilience guarantees as the original exception handling defined in a method higher up on the call stack. TripleAgent reports the methods between the source of exception and its handler that are found not to be failure-oblivious might cause resilience issues. TripleAgent has the following properties:

Fault Model. TripleAgent tests the resilience of methods against uncaught exceptions. Fault targets are statements in a method. Compared to other approaches where an exception is only injected at the beginning of a method, it injects checked exceptions before statements that throw an exception. Fault tuples therefore consist of a method, statement location and exception. In this way, the tool explores all possibilities and can differentiate the impact of exceptions at different locations. A static analysis is used to find out the statically-declared checked exceptions of each method. Only those exceptions are injected by TripleAgent. The exceptions can be injected in two ways: only when the target is reached for the first time, or every time the target is reached.

Environment. TripleAgent is presented in the context of development environments. It requires a Java system and a production-like workload as input. For example, the authors evaluate a file transfer client where the workload represents the user downloading a large file from the internet. The authors indicate that using TripleAgent in production is part of their long-term goal.

Exploration. Fault scenarios are automatically generated and consist of a single tuple. By default, the proposed algorithm selects all locations of all methods where checked exceptions can be thrown and thus exhaustively explores the fault space.

Pruning. Developers are not able to leverage their domain knowledge to skip certain methods, exceptions or locations.

Prioritization. TripleAgent does not support method prioritization.

Oracle. The authors indicate that no test suites are required, but instead requires a so-called acceptability oracle. This oracle defines acceptable behaviour in terms of generic and domain-specific oracles. For example, an acceptability oracle could be that the transfer client does not crash and exits normally and is able to download a complete file. However, it remains vague how these oracles are defined and in which language.

5.6.2.6 ChaosOrca

Simonsson *et al.* [SZM+19] apply the principles of Chaos Engineering to containerized applications. In particular, they present ChaosOrca, a tool that assesses the resilience of system calls in Docker[18] containers. Docker containers heavily rely on these system calls to communicate with the kernel of the operating system. As a result, the resilience of the application inside the container is correlated with the working of these calls. The tool automatically detects all system calls during the execution of the containerized application through system-level instrumentation (*i.e.,* bpftrace[19]). ChaosOrca has the following properties:

Fault Model. The fault targets of ChaosOrca are Docker containers and their processes within. To assess the resilience of these containers, it is enough to test the resilience to unexpected system calls results. The faults can cause delayed execution and error codes. While there are over 100 error codes, the evaluation only considers 6 of them which are related to resource and permission issues. A fault tuple therefore consists of a system call, and an error code or delay. The authors indicate that system call errors are representative of real-life failures. For example, the error code `EACCES` represents the lack of sufficient permissions for the attempted operation.

Environment. As the name suggests, ChaosOrca employs the principles of Chaos Engineering. By definition, this means that the tool targets systems in production. ChaosOrca is unique in the fact that it conducts experiments under production workload without instrumenting the application inside the Docker container.

Exploration. Fault scenarios consist of a single tuple and are automatically generated and exhaustively explored. Combinations of system calls are not considered.

Pruning. The approach can target single containers and single processes within them. Additionally, domain-knowledge of developers can be leveraged to indicate interesting system calls. By default, it exhaustively analyses all kinds of system calls.

[18] `https://www.docker.com`

[19] `https://github.com/iovisor/bpftrace`

Prioritization. By default, ChaosOrca orders system calls by the number of invocations and manually-specified system calls (*e.g.,* calls with a low number of invocations, yet interesting to test because it is a critical call). This increases the likelihood that the most important system calls are tested first. As a result, the confidence in resilience of the system increases faster.

Oracle. ChaosOrca records generic system metrics such as CPU usage, RAM usage, and network I/O. The authors indicate that the tool also supports additional metrics provided by developers. While the tool automatically computes metric differences, it remains unclear when the Chaos Engineering experiment fails and whether this always indicates an actual resilience issue (*e.g.,* whenever the metric changes or goes above a certain threshold).

5.6.3 LDFI-driven Exploration of Fault Scenarios

This section discusses five approaches (Molly [ARH15], ldfi-netflix [AAS+16], ldfi-akka [Ghi19], IntelliFT [CWC+19], and Madaari [RE19]) that automatically generate and explore fault scenarios using lineage-driven fault injection. We refer the reader back to Section 5.4 for details on LDFI.

5.6.3.1 Molly

The seminal paper of Alvaro *et al.* presents an implementation of LDFI called Molly [ARH15]. The authors use Molly to detect fault-tolerance bugs in large-scale complex distributed systems such as the reliable message queue Kafka and Paxos [Lam19]. The results indicate that Molly can find failures with an order of magnitude faster than a random exploration of the fault space. Molly was presented as a formal prototype where both the system and the correctness specification is specified in the Dedalus language [AMC+10] — a declarative rule-based language based on Datalog [EGM97]. Data lineage can therefore easily be extracted from program executions through program rewrites. The correctness specification can be expressed with pre and post conditions.

Fault Model. Molly focuses on crash-stop failures, message delivery failures, and network partition failures. The authors indicate crash-recovery failures as an avenue of future work. The presented fault model consists of three parameters: a logical time to bound the execution, a logical time at which message loss ceases, and a maximum number of crashes. The first parameter is required to guarantee completeness. The second one is to model intermittent message losses which eventually resolve. This also avoids infinite message losses which are uncommon [BK14] and would otherwise cause Molly to find counterexamples all the time (*i.e.,* by losing all messages). The last parameter indicates the maximum number of node crashes (*i.e.,* crash-stop) that can occur to avoid crashing all nodes.

Environment. MOLLY is not specific to development or production environments. However, we consider it to be mainly used in development environments. This is based on publications which indicate that deriving a lineage from systems in production environments is challenging. Additionally, the approach assumes that messages are successfully delivered and received in a deterministic order (*i.e.,* the execution is deterministic) which might not always be feasible to do in production environments.

Exploration. Fault scenarios consist of one or more tuples and are automatically generated and explored by LDFI.

Pruning. Compared to exhaustive approaches, LDFI prunes away many fault scenarios that are not possible for a given lineage. Developers can prune away some of these fault scenarios but makes LDFI incomplete.

Prioritization. By definition, LDFI does not incorporate any explicit ordering as it is represented as a decision problem. As a result, it will return fault scenarios based on the underlying implementation of the SAT-solver.

Oracle. The authors state that a correctness specification is required. For this reason, MOLLY provides built-in contracts with a pre and post condition to express distributed invariants.

5.6.3.2 LDFI-AKKA

Ghidei [Ghi19] explores a transposition of the ideas of MOLLY to the general-purpose programming language SCALA, in which data lineage cannot readily be extracted from a given program. The author presents LDFI-AKKA[20] which extends MOLLY to the general-purpose, object-oriented language SCALA where distributed programs are written using the actor framework AKKA.

The author utilizes program rewrites for inserting both logging and controlling constructs. SCALAFIX[21] is a well-known library to transform ASTs through developer-specified syntactic or semantic rules. For logging information about the actor system and its events, they make actors extend a specific trait and rewrite the receive block. LDFI-AKKA uses the collected information to build a lineage graph based on the exchanged actor messages during system execution. However, the evaluation hints that this approach might not scale on a fine-grained level of actor messages. In order to deterministically replay the execution, LDFI-AKKA wraps message sends within a check to see whether the message should be sent according to the previously recorded execution.

[20] https://github.com/KTH/ldfi-akka

[21] https://github.com/scalacenter/scalafix

Fault Model. This work extends and adapts MOLLY to the actor-based framework AKKA. The targets are therefore messages and actors, while the faults simulate message failures and node crashes.

Environment. The author does not indicate whether a development or production environment is targeted. However, given the limited scalability we deem this technique to be only usable in development environments. LDFI-AKKA is also only sound and complete under several assumptions. A synchronous execution model must be used (*i.e.,* all messages are eventually delivered and have a deterministic ordering, analogous to MOLLY [ARH15]), the program rewrites and SAT solver must be sound, and the programs must be deterministic in their nature.

Exploration. Fault scenarios consist of one or more tuples. LDFI-AKKA automatically generates and explores fault scenarios through LDFI. However, the evaluation hints this approach might not scale properly at a message-based granularity. This is a major limitation as AKKA systems are typically much larger than the programs used in the evaluation. These programs include (i) two examples presented in the original work of MOLLY [ARH15], (ii) illustrative examples[22] from the documentation of AKKA including the well-known Dining Philosophers [Dij78], and (iii) an implementation of the Observable Atomic Consistency Protocol [ZH18]. However, LDFI-AKKA is not able to execute the latter two. The former because the SAT solver was overwhelmed with too many literals, and the latter because the program deadlocks due to program rewrites that force a synchronous execution.

Pruning. LDFI-AKKA does not provide means to prune fault scenarios. However, such an optimization might reduce the burden on the SAT solver.

Prioritization. LDFI-AKKA does not provide a way to prioritize fault scenarios.

Oracle. The authors state that a correctness specification is required. An oracle is given in the form of a predicate implemented in SCALA.

5.6.3.3 LDFI-NETFLIX

Alvaro *et al.* [AAS+16] investigate how the prototype MOLLY [ARH15] can be adapted and used at the scale of NETFLIX. While this approach has no specific name, we call it LDFI-NETFLIX. The authors indicate three major problems and present their solutions with respect to NETFLIX's microservice architecture.

The first problem relates to the DEDALUS language which is used to implement the distributed system and makes it trivial to collect lineage information. However, there are too many systems to port at NETFLIX and not all source code is available which makes this infeasible.

[22] https://github.com/akka/akka-samples

Netflix therefore uses call graphs which are produced by their internal tracing system. These call graphs do not contain functions, but rather the services that participated in a user request. It is clear that the granularity of the lineage is much higher then the one achieved by the research prototype Molly [ARH15]. While this approach enables Netflix to do automated fault injection, it is fundamentally less precise.

The second problem lies in replayability. Given the complex and dynamic microservice infrastructure it is infeasible to replay user requests correctly as internal state changes, service versions change constantly, and not all services are idempotent for every request. To resolve this issue, they map individual requests to a set of request classes using an equivalence relation. Intuitively, two requests belong to the same request class when they interact with the same services. In this way, failure hypotheses found in one request can be tested in another request that maps to the same request class. In reality, these requests are not always deterministic and therefore this mapping only produces a class when the classifier has a high confidence that the request belongs to a class.

The final problem that needs to be resolved is to merge requests of the same request class into a single model. By default, a request does not contain redundancy because an alternative way is not explored in the original user requests. Therefore, Netflix incrementally builds a model of the redundant ways a request class can provide a desired outcome. This coarse-grained lineage model is extended over time (*i.e.,* grouping the call graphs of all requests of one request class into one model). LDFI uses this model to generate failure hypotheses. These hypotheses are then tested in subsequent requests that belong to the hypothesis its required request class.

Fault Model. Netflix uses a typical microservice architecture and therefore uses microservices as fault targets. The injected faults simulate failures caused by timeouts, internal server errors (*i.e.,* a 500 HTTP response code), and exception handlers.

Environment. This work is presented in the context of production environments. The authors indicate the difficulties and the trade-offs of using LDFI in practice.

Exploration. Fault scenarios consist of one or more tuples. The underlying implementation of LDFI automatically generates and explores fault scenarios. It is clear that the adaptation of LDFI from the research prototype Molly into Netflix's microservice architecture poses several challenges. However, the authors show that this is feasible with some trade-offs regarding data lineage and replayability. A more fine-grained lineage would reduce the fault scenarios that ultimately are false positives.

Pruning. LDFI already prunes a large part of the search space. There is no additional means to select fault scenarios.

Prioritization. By definition, LDFI does not incorporate any prioritization as it is represented as a decision problem. The authors indicate that detecting the most likely failure is part of their future work. However, this requires transforming the decision problem (*i.e.,* is there a set of faults) into an optimization problem (*i.e.,* which is the most likely set of faults). The following two papers present approaches to tackle this problem.

Oracle. The oracles are based on metrics captured during the experiment. However, multiple requests are tested to avoid false positives and the oracles only mark an experiment as failed when 75% of the requests result in a failure.

5.6.3.4 IntelliFT

Cui *et al.* [CWC+19] address two limitations of LDFI in a short paper with limited details. One limitation is that each user request is considered in isolation, without considering historical results of other requests. The other limitation is that many redundant fault scenarios are generated, which could be pruned by using previous results. To this end, the authors present IntelliFT, a feedback-based implementation of LDFI which guides the exploration more efficiently through historical results and heuristics, while leveraging existing integration test cases to simulate user requests.

The algorithm consists of two parts. The first part randomly explores some failure scenarios and measures their impact on test outcome. Next, the second part will select and explore the most likely fault scenarios guided by feedback of the subsequent fault scenarios. Similar to the previous paper, they use distributed logging to collect a lineage graph. The results show that their approach only tries 12% of the fault scenarios determined by LDFI to find failures with a naive implementation. However, it is unclear whether completeness remains guaranteed. IntelliFT has the following properties:

Fault Model. The fault targets are microservices and faults simulate the typical kind of failures in microservice architectures such as delay, disconnections, overload, and crashes.

Environment. IntelliFT uses integration tests to generate user requests and capture the internal service call graph. IntelliFT is therefore presented in the context of development environments.

Exploration. Fault scenarios consist of one or more tuples and are generated and explored by the underlying LDFI implementation.

Pruning. While LDFI explores every possible fault scenario, selection is used to filter the fault space by means of two heuristics. The first one is based on propagation of failures. In particular, when an upstream service observes the same error as its downstream service, it means that both cannot handle the same failure.

This means that the scenario were both services are injected with a fault can be pruned. The second one is based on the likelihood that more complex failure scenarios will not succeed when simple failure scenarios already fail.

Prioritization. The prioritization of fault scenarios is determined by three priorities: the priority of the request class, the priority of the fault type, and the priority of the service. These priorities are, however, *dynamically* computed based on subsequent executions of failure scenarios. Informally, a request class has a high priority when there is a high number of found failures and a low number of tested fault scenarios. A fault has a high priority when there is a high number of found failures and a low number of tested fault scenarios with this type of fault. A service has a high priority when the service is not capable of handling a high number of tested failure scenarios.

Oracle. The oracle is based on HTTP response codes for functional failures and on metrics for performance failures. For functional failures, a failure is detected when the HTTP response code is anything except 200. Otherwise, the absence of a failure is detected, except when the payload contains keywords such as error and exception. For performance failures, an upper bound is used on the metric (*e.g.,* latency).

5.6.3.5 Madaari

Raina *et al.* [RE19] presented MADAARI[23] during a conference talk. It is important to note that there are no scientific publications about this work and thus affects the level of detail at which we can describe this work. The motivation for the work is based on the fact that LDFI considers all faults as equal. However, this differs in reality because complex or new services are more likely to be prone to faults, compared to more robust or mature services. This intuition is backed up by historical post-mortem reports and logs [HRJ+16]. The proposed approach converts LDFI from a decision problem to an optimization problem. In particular, it introduces an order in which LDFI explores fault scenarios at EBAY. While LDFI originally was presented as a rather formal technique, it becomes clear that large companies are investing time and money to make this technique work for their large-scale architectures. MADAARI has the following properties:

Fault Model. The authors talk about failed remote procedure calls (RPC) in their real-world example. We therefore assume that remote procedure calls inside microservices are the fault targets and faults are response codes.

Environment. The authors show a real-world application of MADAARI to EBAY's payment platform. They use techniques similar to those from previous work to distil a lineage graph of their system.

[23] https://rsmemory.info/video/YoqS0F2ApamiY4g/jotb19-madaari.html

However, it remains unclear whether MADAARI is really used in the context of production environments as their use of tests as oracle seems to suggest otherwise. We therefore situate MADAARI in the context of development environments.

Exploration. Fault scenarios consists of one or more tuples and are automatically generated and explored by LDFI. As future work, the authors indicate that currently only time (*i.e.,* the order of the call graph) is considered and not state. The call graph already imposes an ordering in which faults are injected (*i.e.,* time). However, it might be interesting to only generate fault scenarios where the state of a fault target matches a particular predicate. For example, only inject a fault in a mail service when there are at least 5 mails queued for a given email address.

Pruning. The authors do not present any means to prune fault scenarios.

Prioritization. The authors use two kinds of priorities to order fault scenarios. The first priority is based on the depth of a node in the lineage graph, while the second one is based on the size of the sub graph below a given node. That is, the closer one node is to the root, the more likely that a failure of another node in the sub graph will manifest itself in that one node. Thus, when injecting a fault in one node and the outcome is still successful, the whole graph below that one node does not need to be explored any more.

Oracle. The authors propose end-to-end (E2E) tests in order to replay interactions and assertions to determine the outcome. Given the automated characteristics of E2E tests, we assume that no manual effort is required to determine the outcome.

5.6.4 DD-driven Exploration of Fault Scenarios

This final section discusses the approach of THOR [AMM15] which automatically explores fault scenarios using delta debugging (*cf.* Section 5.5).

5.6.4.1 Thor

Adamsen *et al.* [AMM15] observe that developers of mobile applications often ignore unusual behaviour in their test suites. That is, developers do not test for events that are unlikely to happen during the use of the application. In particular, the authors focus on events that should be neutral to the execution: whether they occur or not should not affect the outcome. To this end, Adamsen *et al.* [AMM15] present THOR. This tool automatically tests ANDROID applications under abnormal conditions by injecting neutral events in each test case and determining whether test outcome remains identical. THOR has the following properties:

Fault Model. The ANDROID framework consists of more than 60 different services that each have their own neutral events. That is, events that should not affect the outcome. For example, rotating a screen, temporary connection loss, or carrier changes are events that are considered to be neutral. THOR targets these services and injects different kinds of neutral events to mimic service actions that would not occur under normal conditions. These events are injected in the application immediately after instructions that trigger events (*e.g.,* click or swipe) in a test case. The fault tuples therefore consist of a trigger instruction of a test event and a set of neutral event sequences for a given service. In the evaluation, THOR only injects events related to the activity manager and the audio service.

Environment. THOR is presented in the context of development environments as it applies test amplification on existing test suites.

Exploration. Fault scenarios consist of one or more tuples and are explored by delta debugging. Important to note is that all neutral event sequences are concatenated together and injected aggressively after a trigger instruction. For example, the event sequences `pause-resume` targeting the activity manager and `wifi-4G-wifi` targeting the connection manager are concatenated a list `s=[pause-resume,wifi-4G-wifi]` such that a fault scenario would look similar to `{(clickX, s)}, {(clickY, s)}`. Merging all event sequences together is possible since neutral event sequences are closed under concatenation. THOR leverages delta debugging to find out the minimal neutral event sequence that causes the failure.

Pruning. Neutral events must be decided by developers as it can be a subjective matter. THOR does not enable developers to filter out certain trigger instructions. However, the authors present a way to reduce redundant injections by means of tracking abstract states. These states consist of an abstraction that includes the user interface state together with a trigger event and neutral events. Whenever such a state is already seen, the injection is skipped because any failure caused by that injection would likely have been found already. However, this mechanism may cause some failures to be missed.

Prioritization. Delta debugging its performance can be affected by changing the order in which partitions are tested (*cf.* Section 5.5.4). However, the authors do not discuss this matter and therefore assume that injection points or neutral events are not prioritized during testing.

Oracle. THOR amplifies existing test suites. Therefore, the oracle is based on the assertions that developers wrote and their effect on test outcome. No additional oracles or specifications are needed which facilitates the use for developers.

5.7 Observations

We now discuss the following observations made throughout our survey of the state of the art. Based on these observations, we distil the shape for our approach to resilience testing which we present in the next chapter.

Fault Model. The majority of the fault models can be divided into two categories. The first category groups models about exceptions, while the second one groups models about typical faults in distributed systems (*e.g.,* node, disk, and network failures). The approach of Adamsen *et al.* [AMM15] presents an atypical fault model: neutral events in ANDROID.

Our fault model consists of faults that try to uncover message idempotency failures and actor restart failures. These faults should be neutral in a resilient system: whether they occur or not should not affect functionality. The fault targets are fine-grained as they are the actual messages that are exchanged between actors.

Environment. About one third of the approaches is presented in the context of production environments due to their Chaos Engineering methodology, while the others are presented in the context of development environments. Both environments have their own merits. On the one hand, injecting faults in production environments can be risky as it might crash the system. On the other hand, development environments have to rely on artificial workloads which might not be representative of typical requests through the system.

Our approach is presented in the context of development environments because we consider our approach to be the first step towards a resilient system during the software development process.

Exploration. Several of the approaches require developers to write fault scenarios manually. However, these approaches require manual effort and extensive domain knowledge which might not be always available. Despite these drawbacks, it might outperform automated analyses as the fault scenarios capture only the fault space that is of interest to developers. The majority of the approaches use a default approach: exhaustively generate and explore all fault scenarios one by one. As a result, testing budget is wasted as many fault scenarios have to be explored. LDFI avoids exploring multiple fault scenarios by exploiting the system's redundancy. We also found one approach (*i.e.,* [AMM15]) that uses the delta debugging algorithm in combination with neutral events to efficiently explore the fault space. In particular, it iteratively explores one fault scenario consisting of all fault tuples and determines a minimal fault scenario when a failure is found.

Our approach automatically generates fault scenarios based on information of the system's execution and provides an exploration strategy based on delta debugging to find resilience defects efficiently.

Pruning. Several approaches provide means to prune the fault space. This is either by means of manually writing fault scenarios or by applying heuristics. It is vital to reduce the fault space when possible as the fault space is typically large for distributed systems.

Our approach provides a pruning strategy based on the causality relation between actors and their messages.

Prioritization. The majority of the approaches do not provide means to prioritize fault scenarios. This shows a major area of research to further optimize resilience testing.

Our approach provides means to prioritize fault scenarios based on actor-specific prioritization strategies. We evaluate the impact of the different strategies for each of our exploration strategies.

Oracle. The majority of approaches utilize test suites or metrics to conclude whether a failure occurs or not. Some use dedicated specification languages. It is known that developers put a vast amount of time and effort in test suites. As a result, they include domain knowledge that can be reused in the context of resilience testing.

Our approach leverages the test oracles from existing test suites and uses these to determine whether injecting faults in a test case leads to a failure.

5.8 Conclusion

This chapter provided an overview of the state of the art in resilience testing. First, we started by defining the meaning of resilience, its concepts, and its difficulties. Next, we discussed Fault Injection, Chaos Engineering, Lineage-Driven Fault Injection (LDFI), and Delta Debugging (DD) as they are closely related to resilience testing. We defined the necessary terminology such as faults, failures, and fault scenarios to understand the foundations of fault injection. Additionally, we illustrated the internal workings of LDFI and the minimizing and general delta debugging algorithm. Overall, this should provide enough terminology and background information to understand the remainder of this dissertation. We then presented the results of our literature study where we assessed existing resilience testing approaches on several properties. We concluded that existing work is limited for actor systems and also provides ample room for better exploration, pruning, and prioritization strategies. We grouped each of the approaches into four categories based their fault scenario exploration strategy: developer-specified, exhaustive, LDFI-based, or DD-based. We also identified three main categories of failures: failures related to exceptions, failures related to distributed systems, and failures related to neutral event sequences in mobile applications. Finally, we summarized our observations for each property and discussed how we will incorporate these observations into our approach to resilience testing. We present our approach and its implementation in the next chapter.

Chapter 6

Chaokka: A Dynamic Analysis Approach to Resilience Testing

As mentioned in the first chapter, contemporary systems are increasingly migrating to distributed architectures with fine-grained services. The actor model provides a foundation to build such distributed systems and has been widely adopted in the industry through frameworks such as Akka. However, the distributed nature of these systems exposes them to certain conditions to which they should be resilient. Despite that actor model frameworks provide multiple resilience mechanisms, it remains difficult to implement them and test whether they work as expected under such conditions. This chapter therefore describes our approach to resilience testing in the context of actor systems.

Section 6.1 presents the motivations behind this work. We describe the potential resilience defects that can occur during the implementation of resilient actor systems. Next, Section 6.2 provides an overview of Chaokka — our automated resilience testing approach that is built on top of the frameworks Akka and ScalaTest. The following sections then detail each step of Chaokka. Section 6.3 formally presents the concept of execution traces, causality, and fault scenarios in the context of the actor systems. Section 6.4 introduces three exploration strategies including our strategy based on delta debugging. In order to navigate more efficiently through the fault space, Section 6.5 discusses pruning strategies including our causality-based strategy, while Section 6.6 presents multiple prioritization strategies to find resilience defects sooner. Penultimately, Section 6.7 briefly discusses the usage of Chaokka and its possibilities to be extended with support for additional resilience defects and strategies. We will use Chaokka in the next chapter to determine the efficacy of resilience analyses with different combinations of strategies. Finally, Section 6.8 discusses several application domains of our resilience testing approach.

6.1 Motivation

Even though it is increasingly important in this age to have resilient distributed actor systems, it remains hard to implement and test resilience mechanisms (*cf.* Section 5.1.2). Not only because these mechanisms have to be integrated in systems that already are inherently complex, but also because automated resilience testing approaches are limited in number.

In particular for actor systems implemented with Akka, Section 6.1.1 exemplifies the potential defects in two resilience mechanisms, while Section 6.1.2 demonstrates how hard and cumbersome it is for developers to manually explore the behaviour of a system under abnormal conditions.

6.1.1 Difficulties of Implementing Resilience Mechanisms

To increase the resilience of systems, frameworks provide resilience mechanisms such as guaranteed message delivery, compartmentalization, actor supervision strategies, persistence of actor state, and circuit breakers to temporarily suspend actor interactions. We discussed several of these mechanisms in Section 2.3. However, implementing these resilience mechanisms further increases the system's complexity which might result in defects. We discuss one defect of the resilience mechanism that guarantees message delivery and one defect of the resilience mechanism that handles actor restarts.

6.1.1.1 Guaranteed Message Delivery

On the level of messages, developers need to account for message delivery failures. This can be mitigated by implementing at-least-once message delivery semantics. However, implementing this resilience mechanism can lead to two defects:

1. developers may forget that a message can arrive more than once, and
2. developers may not be aware that messages might arrive out of order.

Both defects arise when the receiving actor is implemented incorrectly. However, these defects are the result of the sender's resilience mechanism that repeatedly sends messages until it has received an acknowledgement. This implies that the mechanism is spread across two actors which might complicate the implementation. We only focus on the former defect as idempotency is a known problem [Hel12, HROE13] and the latter has been widely studied in general as a concurrency defect (*e.g.,* [LMBM18, TPLJ13]). There are two possibilities to avoid handling messages more than once: either by tracking the identifiers of messages to avoid processing the same message twice, or by making the message processing idempotent at the level of the business logic.

The example we have shown earlier in Listing 2.4 and Listing 2.5 on page 21 demonstrates the difficulty of implementing at-least-once delivery correctly. The code is actually a simplified version of a real-world question posted on StackOverflow[1]. In that question, a developer experienced a problem with messages arriving multiple times:

> *"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 the actor GuaranteedDeliveryActor]."*

At first sight, the implementation (*i.e.,* Listing 2.4 and Listing 2.5) and test case (*i.e.,* Listing 2.6) 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 to make sure that message processing is idempotent. However, systems might not always be designed for idempotence [Hel07]. Moreover, verifying idempotency appears to be non-trivial [HROE13] and challenging [RV13].

6.1.1.2 Event Sourcing

On the level of actors, developers need to account for actor restarts. An actor might be terminated and subsequently restarted by the framework when unhandled exceptions occur, certain supervision strategies are used, or actors are migrated from one cluster to another, or simply because actors had to be shutdown. This can be mitigated by implementing Event Sourcing [Fow05] which is the default mechanism of persistent actors. However, implementing this mechanism can lead to two defects:

1. developers may not persist all the necessary state, and
2. developers may not replay the actor state correctly.

We focus on both defects as many distributed system failures are due to services that fail to recover their state after a restart [OZRA19, LLLG16] with inconsistent states as a result [CDSL+19]. In fact, restarting actors can also result in sending messages again which might lead to the aforementioned problem of not handling messages in an idempotent way.

The actor `GuaranteedDeliveryActor` shown in Listing 2.5 is resilient to actor restarts, but its communication partner `Accumulator` shown in Listing 2.4 is not. Any restart will reset its internal state such that `count` becomes 0. This, however, will unlikely become clear from running the tests in normal conditions.

[1] `https://stackoverflow.com/questions/27592304`

6.1.1.3 Neutral Events

When we take a closer look at these events (*i.e.,* duplicate messages and actor restarts) that can cause these defects, we can informally say they should be *neutral* to the execution of the system. That is, it shouldn't make a difference when messages are received more than once or when actors are restarted at any moment in time. Deciding whether these events are truly neutral or not could depend on the system and must be decided by the developers. For example, an actor might perform additional side effects besides rehydrating its state upon restart. Nevertheless, this concept of neutrality is also considered in other work [AMM15] where events of ANDROID services are considered to be neutral to the execution of the application. For example, rotating the screen or changing the mobile network should not change the execution. We follow the same reasoning and consider these events to be neutral the execution of a resilient actor system.

6.1.2 Difficulties of Testing Resilience Mechanisms

One of the difficulties for developers remains to test these mechanisms under abnormal conditions and is therefore typically neglected. Studies have shown that tests often only cover the so-called happy paths[2] and therefore neglect exceptional behaviour [ECS15,JGS11]. As a result, developers stay in the dark about whether their resilience mechanisms indeed work as expected.

Recall from Section 2.3.4 that TESTKIT provides means to test actors. However, the abnormal conditions under which resilience mechanisms are meant to work correctly have to be simulated manually. Additionally, intercepting the target (*e.g.,* specific messages) requires overriding and partially re-implementing the internal behaviour of the actor. For example, simulating the conditions where messages can arrive multiple times requires developers to adapt the test case from Listing 2.6 to the one shown in Listing 6.1.

In particular, the method `receiveCommand` has to be overridden to find our target message (line 15–20) and `updateState` has to be changed to send the message twice (line 23–37). While the event `PlusEvent` is simple to handle in this case, it still requires duplicating the original implementation and modifying the call to `deliver` (line 27). This example shows that developers have to put significant effort to determine resilience defects as there is no support from AKKA, nor from TESTKIT itself to do such things. Testing approaches that can automatically determine such resilience defects are therefore needed.

[2] https://en.wikipedia.org/wiki/Happy_path

```
1  import akka.actor.{ActorSystem, Props}
2  import akka.testkit.{ImplicitSender, TestKit}
3  import org.scalatest.{BeforeAndAfterAll, FlatSpecLike, Matchers}
4  import scala.concurrent.duration._
5
6  class ExampleTest() extends TestKit(ActorSystem("SystemUnderTest"))
7    with FlatSpecLike with ImplicitSender
8    with Matchers with BeforeAndAfterAll {
9
10   "Accumulator" must "correctly accumulate numbers" in {
11
12     var target: Plus = null
13     val a = system.actorOf(Props[Accumulator], name="A")
14     val props = Props(new GuaranteedDeliveryActor(a) {
15       override def receiveCommand: Receive = {
16         case m@Plus(amount) =>
17           // Determine our target message
18           if(amount == 2) target = m
19           super.receiveCommand(m)
20         case e => super.receiveCommand(e)
21       }
22
23       override def updateState(e: Event) = e match {
24         case PlusEvent(amount) =>
25           var duplicate: CountCommand = null
26           // Deliver the original message
27           deliver(ref.path)(id => {
28             duplicate = CountCommand(id, amount)
29             duplicate
30           })
31           // Deliver the message one more time
32           if(target != null) {
33             context.actorSelection(ref.path) ! duplicate
34             target = null
35           }
36         case e => super.updateState(e)
37       }
38     })
39     val actor = system.actorOf(props, name="GDA")
40
41     for (i <- 1 to 10) { actor ! Plus(i) }
42     Thread.sleep(2000)
43     a ! "result"
44
45     expectMsg((1 to 10).sum)
46   }
47
48   override def afterAll: Unit = {
49     TestKit.shutdownActorSystem(system, 5 * 60 seconds)
50   }
51
52 }
```

Listing 6.1: A test case with additional code to find resilience defects.

6.2 Overview of the Approach

As observed in Chapter 5, there is a lack of automated tool support for testing the resilience of actor systems. Manually testing that resilience mechanisms work correctly under abnormal conditions might not only be time-consuming, but also error-prone due the inherent complexity and large fault space. To this end, we presented Chaokka: **Chao**s in A**kka**. Chaokka is an automated resilience testing tool for systems implemented for the Akka and ScalaTest frameworks. Figure 6.1 shows the architecture of Chaokka. We discuss each step of the process in detail below and refer the reader to Section 6.7 for details on its implementation.

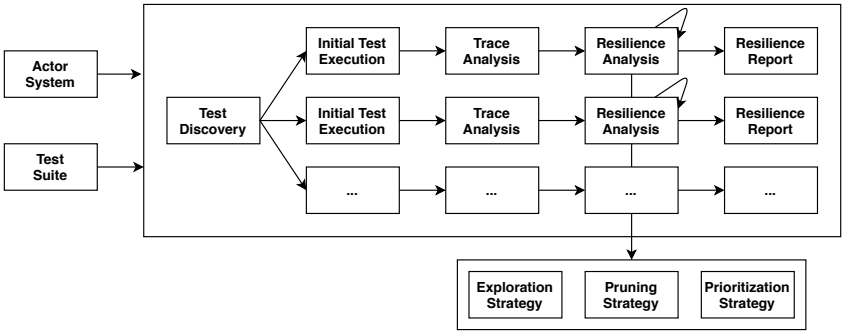

Figure 6.1: The architecture of Chaokka.

1. **Input.** Chaokka requires an actor system written with Akka and a test suite written with the ScalaTest framework. No modifications to the source or to the test code are required.

2. **Test Discovery.** Chaokka automatically discovers test cases written with ScalaTest and maintains test information in an index file which is built when Chaokka runs for the first time. This file maps test cases to their meta data such as name, duration, and test outcome. Subsequent runs will load this index file to initialise the tool. The index file only needs to be rebuild whenever the test suite is extended with new test cases.

3. **Resilience Testing.** Chaokka tests the resilience of the system through each test case individually. For each test case, it performs following steps:

   (a) **Initial Test Execution.** First, Chaokka hooks itself into the implementation of the Akka framework through AspectJ[3]. It executes an initial run of the test case and monitors the execution through these hooks to collect an execution trace (*cf.* Section 6.3.1). This trace captures all events about actors being started, messages being sent, and actors processing turns.

   (b) **Trace Analysis.** Next, Chaokka analyses the captured execution trace and determines the initial fault space. This is done by analysing the trace for fault targets and generating fault tuples by combining each target with a potential fault specified by the fault model. For example, only persistent actors can be injected with a fault that simulates a restart. Based on this fault space, different fault scenarios (*cf.* Section 6.3.3) are analysed by a resilience analysis.

   (c) **Resilience Analysis.** Finally, Chaokka executes a given resilience analysis which is composed of three strategies: an exploration strategy (*cf.* Section 6.4), a pruning strategy (*cf.* Section 6.5), and a prioritization strategy (*cf.* Section 6.6). The exploration strategy will determine in which systematic way the fault space is tested (*e.g.,* delta debugging), the pruning strategy will determine which faults remain in the fault space (*e.g.,* only faults related to specific actors), and the prioritization strategy will determine in which order the exploration strategy will explore faults (*e.g.,* faults related to complex actors first). This analysis repeats its exploration strategy until a failing fault scenario is found that satisfies certain requirements (*e.g.,* 1-minimality for delta debugging) and for which the test oracle determines that the test fails when the scenario is injected a run time.

4. **Output.** The output of Chaokka is a resilience report which includes the found fault scenario. This information is actionable as it contains which faults have to be injected in which fault target to make the system fail. While developers with domain knowledge could manually debug the system with this information, it might not be feasible for complex systems. We refer the reader to actor debugging tools such as Kompós [MLA+17] and Actoverse [SW17] that can be used by developers to set breakpoints on the targets of the failure-inducing faults. The combination of such tools and our resilience report should therefore help developers to detect and fix resilience defects.

[3] We refer the reader to the source code to get an overview of all hooks.

6.2.1 Fault Injection as Foundation

Our resilience testing approach adopts the typical fault injection architecture shown in Figure 6.2 and targets actor systems.

The generator component will use a given test suite and determine whether the actor system executes as expected during fault injection. The injector model will use our fault model specific to resilience defects in Akka. The monitor will observe the behaviour of the system, capture an execution trace, and determine the test outcome. Our approach uses the test oracle as its source of truth to determine resilience defects. The key reason for this choice is that test suites contain domain-specific execution scenarios and information about the expected behaviour in the form of assertions. This is in contrast to generic assertions that only assert whether the system produces a crash or whether an exception is thrown. Moreover, the choice for alternative oracles are limited. Manual observation of the system is difficult to scale, while oracles in the form of custom specifications might present a barrier to usage in practice due to their associated learning curve and expressiveness [GDJ+11]. Finally, the controller coordinates each component to test the resilience of the target system.

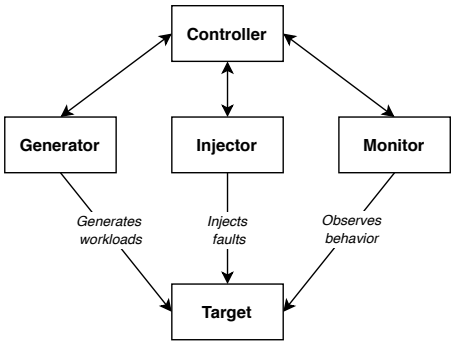

Figure 6.2: The fault injection architecture as the foundation of our approach.

6.3 Trace Analysis

The next sections provides information about the initial test execution and the captured execution trace. Section 6.3.1 describes the notion of an execution trace. Traces capture the necessary information about the execution of the actor system. Next, Section 6.3.2 presents the causality relation which partially orders actors events and is leveraged by one of our pruning strategies. Finally, Section 6.3.3 describes the semantics of fault scenarios in actor systems.

6.3.1 Execution Trace

Our resilience testing approach is based on the notion of an execution trace. This trace captures all events that occur in the actor system during test execution and will be analysed to get an initial fault scenario. Formally, a trace t is a finite set of events where each event is either a *Create*, *Send*, or *Turn* event. Our definition of a trace does not impose an ordering. Typically, events in a trace would be ordered in the way they are recorded. However, events of different actors can happen in parallel which renders the trace's total ordering as incorrect. We will determine the order of events through their send and turn identifiers as explained in the next section. Figure 6.3 shows the formal definition of a test execution trace and the information that it captures.

$$
\begin{aligned}
e \in \mathit{Event} ::= \; & \mathit{Create}(a_{\text{parent}}, b_{\text{child}}, f_{\text{persistent}}, m_{\mathit{registration}}) \\
\mid \; & \mathit{Send}(a_{\text{from}}, b_{\text{to}}, h_{\text{msg}}, i_{\text{send}}, j_{\text{turn}}, f_{\text{alod}}) \\
\mid \; & \mathit{Turn}(a_{\text{from}}, b_{\text{to}}, h_{\text{msg}}, i_{\text{send}}, j_{\text{turn}}) \\
t \in \mathit{Trace} = \; & \mathcal{P}(e_1, e_2, \ldots, e_n) \\
f \in \mathit{Flag} \; & \text{is a finite set of boolean flags} \\
h \in \mathit{Hashes} \; & \text{is a finite set of message hashes} \\
i, j \in \mathit{Identifier} \; & \text{is a finite set of unique identifiers} \\
a, b \in \mathit{Address} \; & \text{is a finite set of actor addresses} \\
m \in \mathit{Timestamp} \; & \text{is a finite set of timestamps}
\end{aligned}
$$

Figure 6.3: A formal description of execution traces.

The addresses a and b denote unique locations in the actor system. For instance, the address `akka://system@host.com:5678/user/actor-1` represents the actor with name `actor-1` that was created by the parent actor with name `user`. Both actors reside on the node `akka://system@host.com:5678` of the network. The identifiers i and j uniquely identify each message that is sent and each turn in which a message is processed. A global counter is used for each identifier.

1. **Create.** A *Create* event is captured whenever an actor at address a_{parent} creates a new actor at address b_{child}. The flag $f_{\text{persistent}}$ indicates whether this actor is persistent or not, while the timestamp $m_{\mathit{registration}}$ indicates the number of milliseconds elapsed since the UNIX epoch. These timestamps can be used to determine which actors are created before others.

2. **Send.** A *Send* event is captured whenever an asynchronous message with hash h_{msg} is sent from actor at address a_{from} to actor at address b_{to}. The flag f_{alod} indicates whether this message was sent with at-least-once message delivery semantics or not.

Note that synchronous messages are not supported out-of-the-box the actor model. Nevertheless, they can be simulated with asynchronous messages and blocking behaviour. The identifier i_{send} will be a new identifier, while j_{turn} will be the identifier of the current turn. In this way, we know from which turn this message was sent.

3. **Turn.** A *Turn* event is captured whenever a message with hash h_{msg} from the actor at address b_{from} is going to be processed by actor at address a_{to} (*i.e.,* the start of the turn). Recall that a turn corresponds to the atomic application of the actor's behaviour to a message taken from its mailbox. The turn identifier j_{turn} will be a new identifier, and i_{send} will be the send identifier which was sent along with the message. In this way, we know which message caused this turn.

6.3.2 The Causality Relation

The purpose of these send and turn identifiers is to order events of actors and determine which events caused which other events. In particular, each *Turn* event has knowledge about the message (*i.e.,* i_{send}) by which it was caused, and each *Send* event has knowledge about the turn (*i.e.,* j_{turn}) from which it was sent. Each identifier has its own global counter which is shared across all actors. When comparing two events of the same actor, an event with a higher identifier always happened after an event with a lower identifier.

Based on these identifiers, we can detect the causality relation $\preceq \subseteq \mathit{Event} \times \mathit{Event}$ [Fid88] between two trace events e and e' (*i.e.,* which turn sent a message and which message caused a turn) as shown in Figure 6.4.

$$
\begin{array}{lll}
e \preceq e' & \text{if they are the same event,} & (1) \\
e \preceq e' & \text{if } e \text{ and } e' \text{ are turn events of the same actor} & (2) \\
 & \text{with turn identifiers } j \text{ and } j' \text{ and } j < j', & \\
e \preceq e' & \text{if } e \text{ and } e' \text{ are send events of the same actor} & (3) \\
 & \text{with send identifiers } i \text{ and } i' \text{ and } i < i', & \\
e \preceq e' & \text{if } e \text{ is a turn event with turn identifier } j & (4) \\
 & \text{and } e' \text{ is a send event with turn identifier } j' & \\
 & \text{and } j == j', \text{ and} & \\
e \preceq e' & \text{if } e \text{ is a send event with send identifier } i & (5) \\
 & \text{and } e' \text{ is a turn event with send identifier } i' & \\
 & \text{and } i == i', \text{ and} & \\
e \preceq e' & \text{if } e \preceq e'' \text{ and } e'' \preceq e' \text{ (\textit{i.e.,} transitivity)} & (6)
\end{array}
$$

Figure 6.4: A formal description of the causality relation.

For example, when an actor has two *Turn* events e with $j_{\text{turn}} = 1$ and e' with $j_{\text{turn}} = 4$, it means that the event e happened before the event e' (*i.e.,* $e \preceq e'$). When an actor has two *Send* events s with $i_{\text{send}} = 5$ and s' with $i_{\text{send}} = 4$ it means that event s' happened before the event s (*i.e.,* $s' \preceq s$). Similarly, when an actor has the *Turn* event e with $j_{\text{turn}} = 4$ and two *Send* events e' with $j_{\text{turn}} = 4 \wedge i_{\text{send}} = 1$ and e'' with $j_{\text{turn}} = 4 \wedge i_{\text{send}} = 3$ it means that there were two messages sent in that turn (*i.e.,* $e \preceq e' \preceq e''$).

It is important to note that events are only partially ordered and that events of different actors do not have an order, unless they are causally related. For example, the *Turn* events e with $j_{\text{turn}} = 1$ of an actor, and e' with $j_{\text{turn}} = 4$ of another actor are not ordered by default. They can only be ordered when there exists a *Send* event e'' with $j_{\text{turn}} = 1 \wedge i_{\text{send}} = 1$ occurs at the turn with $j_{\text{turn}} = 1$ and when the turn $j_{\text{turn}} = 4$ process the message with $i_{\text{send}} = 1$. This is achieved by the transitivity of rules 4 (*i.e.,* a turn causes a send) and 5 (*i.e.,* a send causes a turn). This causality relation, also called the happens-before relation, provides an effective way to analyse the dynamic behaviour of systems. It has been widely used in concurrent program verification and testing [SRA03,SG03] as the extracted causal partial order can be investigated against a desired property. We will use this relation in order to collect all turns from an execution trace that are causally related to turns of a failing actor. This should reduce the fault space to be explored as not every turn in the system trace might be responsible for a given failure. Section 6.4 discusses this pruning strategy in further detail.

6.3.3 Actor-based Fault Scenarios

Our resilience testing approach analyses a trace to determine potential fault targets and their corresponding fault types. It then repeatedly executes the test while injecting different fault scenarios. To this end, every exploration strategy generates a fault scenario that consists of fault tuples (*cf.* Section 5.2). Each fault tuple consists of a fault target which is always a message. This message is identified by the sender and receiver address, as well as the hash of the message payload. Figure 6.5 formally defines fault scenarios.

$$
\begin{aligned}
t \in \mathit{FaultTarget} &= (a_{from}, b_{to}, h_{msg}) \\
p \in \mathit{FaultType} &= \mathit{Duplicate} \mid \mathit{Restart} \\
u \in \mathit{FaultTuple} &= (t, p) \\
s \in \mathit{FaultScenario} &= \mathcal{P}(u_1, u_2, \ldots, u_n) \\
h \in \mathit{Hash} &\text{ is a finite set of message hashes} \\
a, b \in \mathit{Address} &\text{ is a finite set of actor addresses}
\end{aligned}
$$

Figure 6.5: A formal description of fault scenarios.

We have two fault types: *Duplicate* and *Restart*. The former is injected in actors that receive messages with guaranteed message delivery, while the latter is injected in persistent actors. These faults simulate abnormal conditions that need to be handled by the persistence mechanism (*i.e., Restart*) or might be caused by the guaranteed message delivery mechanism (*i.e., Duplicate*). We refer the reader back to Section 6.1 for more details about defects in these resilience mechanisms.

1. **Duplicate.** The receiving actor might receive messages multiple times when messages are sent with guaranteed message delivery. This happens when the sender sends a message again because it has not received an acknowledgement in a timely manner.

   Our approach attempts to uncover defects in the implementation by generating a fault tuple with fault type *Duplicate* for every *Send* event of which the message was sent using at-least-once message delivery semantics (*i.e.,* f_{alod} is true). The sender a_{from}, the receiver b_{to}, and the message hash code h_{msg} are set correspondingly. Note that not every message will be duplicated as some are sent with at-most-once message delivery semantics. This fault helps developers to detect whether at-least-once delivery messages are correctly processed. For example, the fault tuple `((from, to, 12345), Duplicate)` is created when the send event is `Send(from, to, 12345, 1, 2, true)`. This will duplicate the message in the mailbox of the actor with address *to*.

2. **Restart.** Persistent actors that are restarted due to a node failure or migration to another node in the cluster might not recover to its last known state due to defects in the implementation of its state persistence or recovery mechanism. Crash-recovery failures belong to the most common types of modern-day failures in the cloud [HRJ+16]. In theory, this defect could also occur for actors without persistence. However, the recovery mechanism would simply be resetting its initial state. Therefore, we only focus on persistent actors as we want to assess whether the persistence mechanism was correctly implemented.

   Our approach attempts to uncover defects by generating a *Restart* fault for every *Send* event that targets a persistent actor (*i.e.,* $f_{\text{persistent}}$ is true). Restarts can happen after any message, regardless of their message delivery guarantees. The reason why we restart the actor after any message is because that is when they internally transition to a new state, and at every transition there might be a defect in the implementation. The sender a_{from}, the receiver b_{to} (*i.e.,* the actor that is restarted), and the message hash code h_{msg} are set correspondingly. This fault helps developers to detect whether events are correctly persisted and replayed during recovery. For example, the fault tuple `((from, to, 12345), Restart)` is created when `Send(from, to, 12345, 1, 2, _)` and `Create(p, to, true, 1)` are the send and create event respectively. This will restart the actor with address *to* after the turn of this message has finished.

Listing 6.2 depicts an illustrative example of a fault scenario in JSON format. This fault scenario consists of one fault tuple that duplicates a message sent from `TestActor` and received by actor `Accumulator`.

```
1 [{
2     "from": "akka://system/user/TestActor"
3     "to": "akka://system/user/Accumulator",
4     "message": "28259285928",
5     "duplication": true,
6 }]
```

Listing 6.2: An illustrative fault scenario where one message is duplicated.

The reader might ask himself why the faults targets are not identified by the send identifiers of each message since they already uniquely define each message. This is the ideal case in theory. However, our implementation does not handle non-determinism of the actor scheduler and does not replay test execution. As a result, send identifiers change across different test execution runs and are therefore not suitable to identify fault targets. We address this limitation in Section 6.7.3.

6.4 Exploration Strategies

Our resilience testing approach iteratively explores all possible fault scenarios. However, developers are bound by limited test budgets and therefore expect exploration strategies to be efficient. In this section, we present three exploration strategies that test fault scenarios in different ways.

6.4.1 Developer-specified Exploration

Similar to writing test cases, developers can write developer-specified fault scenarios such as shown in Listing 6.2. While this strategy can be useful when a specific fault scenario needs to be tested, it remains infeasible to manually explore a complete fault space. Not only because developers have to think about every possible fault scenario, but also because developers might not be able to reduce it to a minimal fault scenario. As a result, a fault scenario that causes a failure might be found but remains too large to identify the exact faults that cause the resilience defect. The next exploration strategy therefore automates the exploration to remove the manual effort.

6.4.2 Exhaustive Exploration

An automated but naive exploration strategy is to sequentially explore every possible fault scenario of a given fault space (*i.e.,* $f \in \mathcal{P}(\mathcal{F})$). By sorting and exploring fault scenarios in ascending order of cardinality, it will find the minimal fault scenario once a failure occurs.

While this strategy removes the manual effort, it might be infeasible to find a minimal fault scenario within the given test budget. The reason is straightforward: there is an exponential number (*i.e.,* $2^{|\mathcal{F}|}$) of fault scenarios to be tested. Therefore, we slightly adapt this exploration strategy so that it only explores fault scenarios consisting of a single fault. We will refer to this exploration strategy as NAIVE. This results in a linear performance with respect to the number of faults in the worst case. We will use this exploration strategy as our baseline when comparing the differences in performance of each exploration strategy. It is clear that this strategy will miss combinations of faults, but we do not consider combinations in this dissertation.

6.4.3 Delta Debugging Exploration

To the best of our knowledge, we are the first to propose the delta debugging algorithm [ZH02] in the context of resilience testing. We refer the reader back to Section 5.5 for the details of the algorithm.

6.4.3.1 Motivation

Our choice for incorporating the delta debugging algorithm into our resilience testing approach is motivated by multiple reasons:

Applicability. Several works indicate the applicability and the success of this technique. For example, Scott *et al.* [SBN+16] used it to minimize faulty executions of distributed systems, Brummayer *et al.* [BLB10] used it to debug boolean formula solvers, Zhou *et al.* used it to debug failures in microservice architectures [ZPX+19, ZPX+18], and Adamsen *et al.* [AMM15] used it to detect failures in ANDROID applications.

Combinations. Several outage reports [HRJ+16, GDJ+11] indicate that failures are caused by a combination of multiple faults. The delta debugging algorithm is able to find combinations of faults which is not trivial for manual or random approaches.

Partitioning. The algorithm is based on partitioning and testing fault scenarios. Therefore, a well-chosen partitioning strategy based on characteristics of the faults could speed up the algorithm. For example, Misherghi *et al.* [MS06] present hierarchical delta debugging to speed up the performance when the input is structured data (*e.g.,* AST or XML).

Ordering. Besides the partitioning strategy, the efficiency with which delta debugging finds fault scenarios also depends on the order in which the partitioned fault scenarios are tested. For example, developers can leverage both domain knowledge and prioritization strategies to test certain fault scenarios before others.

Monotonicity. It is not uncommon that combinations of faults undo each other. While the first version assumed monotonicity [Zel99], the general version does not make this assumption [ZH02] at the expense of more test executions (*cf.* Section 5.5.3).

Inconsistency. The algorithm finds a solution even when certain fault scenarios produce indeterminate results (*e.g.,* the system does not execute correctly and test outcome cannot be resolved). This is important as indeterminate results occur more often than not in practice [Zel99].

Scalability. While the definition of the delta debugging algorithm suggests that partitions of fault scenarios are tested sequentially, the reality differs. The fault scenarios can be tested in parallel as there are no dependencies between each other except the system under test. Hodovan *et al.* [HK16] investigate parallelism and achieve 4 to 5 times speedup compared to a sequential algorithm. Similarly, Zhou *et al.* [ZPX+19] present a technique for delta debugging microservice systems in parallel.

6.4.3.2 Illustrative Example

Recall that the delta debugging algorithm recursively tries to reduce the fault space to a 1-minimal fault scenario (*i.e.,* every single fault in the scenario is needed to trigger the failure). We will refer to this exploration strategy as DD. We illustrate this strategy by means of an illustrative actor system whose communication topology is shown in Figure 6.6.

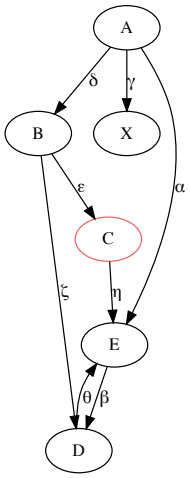

Figure 6.6: An illustrative actor system with a resilience defect.

Every node represents an actor and every directed edge represents a message that is being sent from one actor to another one. This actor system consists of 6 persistent actors that send 8 messages with guaranteed message delivery. We omitted the acknowledgement messages as response to each message from Figure 6.6. The order in which these messages are sent and received does not matter. The actor system has a defect in the resilience mechanism of actor C which incorrectly recovers the state. This problem can be found by restarting the actor after processing ϵ from actor B. This exploration strategy starts with the fault scenario that contains all *Restart* faults for each message in the system. For simplicity, we represent a fault scenario with the fault targets (*i.e.,* messages) only instead of the complete fault tuples.

$$\{\alpha, \bar{\alpha}, \beta, \bar{\beta}, \gamma, \bar{\gamma}, \delta, \bar{\delta}, \epsilon, \bar{\epsilon}, \zeta, \bar{\zeta}, \eta, \bar{\eta}, \theta, \bar{\theta}\}$$

This fault scenario consists of both messages sent with at-least-once message delivery guarantees (denoted with greek letters) and messages sent with at-most-once message delivery guarantees (denoted with greek letters with a bar). The former are the messages that alter the state, while the latter are the acknowledgement messages. We detail the corresponding steps of this exploration strategy below and use the marks ✓ and ✗ to indicate that the test succeeds and fails respectively. Trivially, the initial fault scenario results in a system failure when injected into the system during test execution because the tests fails whenever ϵ is included.

$$\{\alpha, \bar{\alpha}, \beta, \bar{\beta}, \gamma, \bar{\gamma}, \delta, \bar{\delta}, \epsilon, \bar{\epsilon}, \zeta, \bar{\zeta}, \eta, \bar{\eta}, \theta, \bar{\theta}\} \rightarrow \text{✗}$$

The algorithm proceeds by partitioning the scenario into two smaller scenarios. It continues with the first fault scenario shown below and determines that this scenario makes the test fail.

$$\{\alpha, \bar{\alpha}, \beta, \bar{\beta}, \delta, \bar{\delta}, \epsilon, \theta\} \rightarrow \text{✗}$$

Note that, while we initially ordered the faults for simplicity, the fault scenarios are partitioned in a non-deterministic way. The fault scenario would have been $\{\alpha, \bar{\alpha}, \beta, \bar{\beta}, \gamma, \bar{\gamma}, \delta, \bar{\delta}\}$ when the ordering would be based on the greek letters. In Section 6.6, we will discuss multiple prioritization strategies to make such ordering explicit. For now, the reader can assume that the ordering is unspecified. Next, the fault scenario is split up again and the algorithm continues with the first fault scenario.

$$\{\alpha, \bar{\alpha}, \beta, \delta\} \rightarrow \text{✓}$$

Since this fault scenario does not affect the test outcome, the algorithm proceeds to the next fault scenario and determines that the system fails.

$$\{\bar{\beta}, \bar{\delta}, \epsilon, \theta\} \rightarrow \text{✗}$$

As a result, this fault scenario needs to be partitioned again into smaller ones. 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.

$$\{\theta, \bar{\beta}\} \rightarrow \checkmark$$
$$\{\bar{\delta}, \epsilon\} \rightarrow \text{✗}$$
$$\{\bar{\delta}\} \rightarrow \checkmark$$
$$\{\epsilon\} \rightarrow \text{✗}$$

Clearly, this resembles the best case performance of the delta debugging algorithm. We refer the reader back to Section 5.5 for multiple other illustrative examples.

6.5 Pruning Strategies

From our exploration of the state of the art in resilience testing, we observed that several approaches use pruning strategies to reduce the fault space (*cf.* Section 5.7). Next to a developer-specified pruning strategy, we present one that can be used in combination with the exploration strategy based on delta debugging.

6.5.1 Developer-specified Pruning

Our approach includes a default pruning strategy where developers can manually specify which fault targets have to be pruned. This is similar to approaches that require developer-specified fault scenarios as discussed in Section 5.6.1. However, this strategy should only be used when developers have knowledge of the system as pruning too much can render the resilience testing approach ineffective. For example, pruning away all faults related to one specific actor might result in a smaller fault space, but can leave resilience defects uncovered.

6.5.2 Causality-based Pruning

Zeller *et al.* [ZH02] already hinted the potential benefits of combining systematic testing and program analysis. To the best of our knowledge, we are the first to propose a pruning strategy in the context of resilience that leverages ideas from program analysis [Wei84] and causality [Lam19].

In a nutshell, the strategy runs a causality analysis to determine which actor events are causally related. Based on this information, the strategy prunes the causality relation so that only *Send* and *Turn* events are retained that might have affected the actor for which an assertion has failed. Thus this strategy requires the address of the failing actor because only in that way can the strategy decide which events are causally related.

It then determines all fault targets that should not be pruned and removes every fault tuple from the current fault scenario that refers to these targets. Intuitively, this strategy prunes away messages that are not relevant to make the test fail and thus their removal reduces the fault space.

Algorithm 1 determines the causality relation from an execution trace. Essentially, the algorithm links one turn (*i.e.,* t_b on line 3) to another turn (*i.e.,* t_a on line 1) by means of the message that was sent from within the former (*i.e., se* on line 2) and that gave rise to the latter. Finally, line 4 merges ($|+|$) the turn t_a to the current list of turns found at turn t_b in the causality relation. We merge them since one turn can send multiple messages.

ALGORITHM 1: The algorithm to determine 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,List[Turn]]`)

1 **for** $t_a \leftarrow$ *trace.turns* **do**
  `// Find the` *Send se* `that caused` t_a
2 $se \leftarrow$ *trace.sends*.find($s \Rightarrow s.i_{send} == t_a.i_{send}$)
  `// Find the` *Turn* t_b `that caused` *se*
3 $t_b \leftarrow$ *trace.turns*.find($t \Rightarrow se.j_{turn} == t.j_{turn}$)
  `// Thus,` t_b `caused` t_a `via` *se*
4 $cr \leftarrow cr \ |+| \ (t_b \mapsto t_a)$
5 **return** *cr*

Recall from Section 6.3.1, that send and turn identifiers i_{send} and j_{turn} are used to find out exactly which turns caused which sends, and which sends caused which turns. The result of applying Algorithm 1 to the execution trace of Figure 6.6 yields the causality relation shown in Figure 6.7.

Here, nodes represent a unique actor turn, incoming edges represent the message that caused that turn, and outgoing edges represent a message that was sent from within that turn. The messages of type *Update* are sent with at-least-once delivery guarantees, while the messages of type *Confirm* are sent with at-most-once delivery guarantees and acknowledge the reception of *Update* messages. The numbers prefixed with S and T are the corresponding send and turn identifiers. The first node (*i.e.,* the turn of *TestActor*) and edge (*i.e.,* the message *Update*) simulates a request to the entry point of the system (*i.e.,* actor A). The different colours of the nodes can be ignored for now.

A naive way of finding all relevant faults is by collecting all turns of any actor which happened before the last turn of the actor for which an assertion failed. For example, all turns in Figure 6.7 would be collected when actor C fails because all turns happened before that actor's last turn (*i.e.,* `T17`). However, this is it is clear that some actor turns (*e.g.,* `T6` and `T8`) and their messages cannot have affected `T17` as there is no transitive causality between these actors. A better way is to use the causality relation extracted from the trace as shown in Figure 6.7.

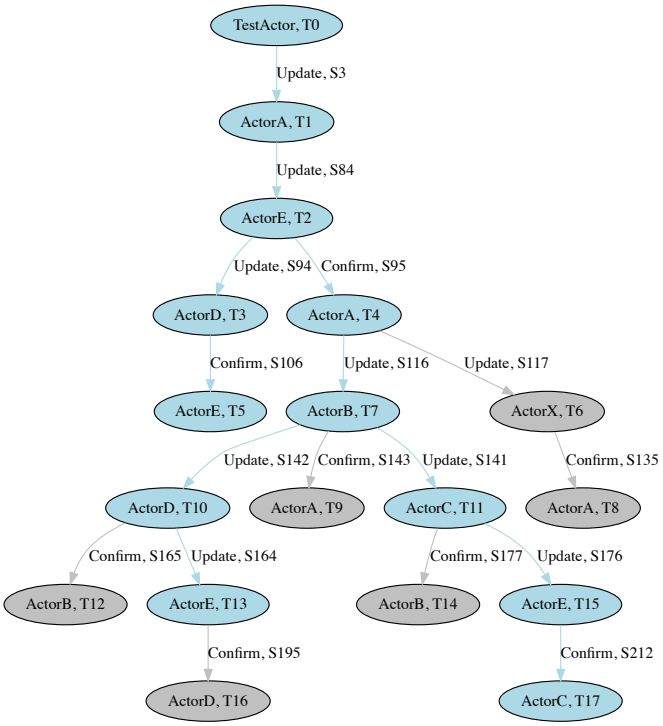

Figure 6.7: The causality relation extracted from the illustrative actor system.

We can use this relation to collect only those turns that are causally-connected to any turn of the failing actor. Those turns can be found by collecting all paths in the causality relation from the root to any turn of that actor. This is what Algorithm 2 determines which expects three arguments.

ALGORITHM 2: The algorithm to collect causally-connected turns.

```
   Input  : cr, the causality relation
            addr, the actor address for which an assertion failed
            m, the maximum turn identifier
   Output: collected, the set of causally connected turns
   // All paths start from the root
 1 setOfPaths ← { cr(cr.root) }
   // The set of causally connected turns
 2 collected ← ∅
   // While there are unexplored paths
 3 while setOfPaths ≠ ∅ do
       // Take the first path in the set
 4     pathOfTurns ← setOfPaths.take(1)
       // Take the last turn of that path (prepended at l12)
 5     lastTurnOnPath ← pathOfTurns.head
       // That turn is of an actor at address addr
 6     if lastTurnOnPath.b_to == addr then
           // Collect all turns on this path
 7         for turn ← pathOfTurns do
 8             collected ← collected + turn
       // Keep exploring, get turns caused by lastTurnOnPath
 9     connectedTurns ← cr.getOrElse(lastTurnOnPath, [])
       // Check if these turns are valid
10     for turn ← connectedTurns do
           // Turn identifier must be lower than m
11         if turn.j_turn ≤ m then
               // Prepend turn to path
12             setOfPaths ← setOfPaths + (turn :: pathOfTurns)
13 return collected
```

The first argument is the causality relation as determined by Algorithm 1, the second argument is the failing actor's address, and the last argument specifies the threshold below which turns need to be collected. As mentioned before, this pruning strategy requires the address of the actor for which an assertion failed. This information needs to be available in some format and accessible to the strategy. For example, our implementation extracts this information from the standard error message of the assertion. In essence, the algorithm performs a breadth-first search to collect all paths to a given actor and returns all unique turns on these paths.

For example, Algorithm 2 returns the following set of turns for actor C:

`{T1, T2, T4, T7, T11, T15, T17}`

However, this is only a subset of the required turns. For example, the turns on the path to turn `T5` of actor E should also be included as they might have influenced turn `T15` of actor E, which then caused the last turn `T17` of actor C. Therefore `T5` might have affected the run-time state of actor E and messages from that turn might have been affected by it. For example, an actor might change its state by incrementing a counter when a certain message arrives. This counter might later affect another actor because its value was included in a message that was sent to another actor which eventually failed due to that value. Therefore, turns that might have affected one of the returned turns should also be considered. Algorithm 3 repeats this process until every turn included.

ALGORITHM 3: The causality-based pruning strategy.

```
   Input  : cr, the causality relation
            addr, the actor address for which an assertion failed
            c, a fault scenario
   Output : c', the filtered fault scenario
   // Find all turns to the failing actor
 1 turns ← collectCausallyConnectedTurns(cr, addr, Integer.Max)
   // The set of causally-connected turns
 2 affected ← ∅
   // While there are unexplored turns
 3 while turns ≠ ∅ do
       // Take a turn
 4     turn ← turns.take(1)
       // This causally-connected turn might have affected the actor
 5     affected ← affected + turn
       // Find all causally connected turns for this turn
 6     extra ← collectCausallyConnectedTurns(cr, turn.k_to, turn.j_turn)
       // Extend only with turns we haven't visited yet
 7     turns ← turns + (extra - affected)
   // Pruning the fault scenarios.
 8 c' ← c.filter(p ⇒
 9         affected.contains(s ⇒
10           s.a_from == p.a_from ∧ s.b_to == p.b_to ∧ s.h_msg == p.h_msg))
11 return c'
```

Line 1 collects all turns of the failing actor, while the other turns are determined through the while-loop on lines 3–7. This loop computes the additional turns of actor E that happened before `T15`. Algorithm 3 would determine that the turns on the paths below have to be included as well:

`[T1, T2, T3, T5]` and `[T1, T2, T4, T7, T10, T13]`

This results in the final set of turns which are stored in the variable `affected` of Algorithm 3:

```
{T1, T2, T3, T4, T5, T7, T10, T11, T13, T15, T17}
```

Finally, the given fault scenario is filtered such that only fault tuples remain in the scenario if and only if their fault target (*i.e.,* message) causes one of the collected turns (line 8—10). Figure 6.7 depicts the final set of causally-connected turns in blue, while gray ones are pruned.

From the description of the causality-based pruning strategy, it should be clear that more fault tuples can be pruned when their fault targets are part of independent execution paths and do not have a transitive relation to the failing target actor. Such paths typically occur in microservices architectures and systems using publish and subscribe mechanisms (*cf.* Section 6.8). For example, an actor might broadcast a message to 10 actors which causes one actor to fail. As a result, 9 out of 10 faults can be pruned since the other 9 messages can have never affected the failing actor (*i.e.,* they are not causally connected). This strategy can also be further improved by tracking at which turn identifier an assertion fails. In that way, we only need to consider turns of the failing actor that have a smaller turn identifier than the tracked one. That is, line 1 of Algorithm 3 can use that turn identifier instead of the default `Integer.Max`. Note that this pruning strategy should only be combined with the exploration strategy where delta debugging is used. The reason is that the naive strategy already only tests fault scenarios consisting of one fault, hence there is nothing left to prune. In the next section, we discuss prioritization strategies which can be combined with any exploration strategy.

6.6 Prioritization Strategies

Recall Section 5.5.4 where we already discussed the impact of partitioning fault scenarios on the performance of delta debugging in particular. Prioritization strategies change the partitioning by first ordering faults according to a given priority and then partitioning them from left to right. These strategies address the desire to find failure-inducing faults as fast as possible and to optimally use the available test budget. For example, failures might be found sooner when faults are first injected in complex actors that receive the most messages. However, finding a good prioritization strategy remains challenging and can influence the performance of any exploration strategy in both ways. Additionally, the majority of existing resilience testing approaches do not provide the ability to order faults (*cf.* Section 5.7). In this dissertation, we therefore investigate prioritization strategies that are based on characteristics of the actor system. We are only aware of one publication (*i.e.,* [LKMA10]) that uses actor-specific prioritization strategies. However, that is in combination with dynamic partial-order reduction.

We therefore transpose two heuristics from that publication to the context of resilience testing and propose three new ones that are based on the number of interacting actors during the system's execution. Prioritization strategies only require an execution trace to compute the priority of each fault. The priority is always determined for the receiving actor of the fault target. We present each prioritization strategy in the following sections.

6.6.1 Shuffle (SHU)

This strategy computes the priority of fault tuple based on a random number. We use this strategy as the default prioritization strategy since it remains general. Algorithm 4 shows the trivial implementation of this strategy.

ALGORITHM 4: The prioritization strategy SHU.

```
   Input  : trace, the execution trace
   Output : mapping, a mapping of actor addresses to priorities
1  mapping ← []
   // Explore all registrations
2  for r ← trace.registrations do
       // Put the timestamp for that actor in the mapping
3      mapping ← mapping + (r.b_child ↦ Math.random())
4  return mapping
```

6.6.2 Registration Time (RT)

This strategy computes the priority for a fault tuple based on the registration timestamp (*i.e.,* $m_{registration}$) of the receiving actor (*i.e.,* b_{to}). This strategy is based on the *CreatedActor* heuristic from [LKMA10]. The reasoning behind this strategy is that actors that are registered first are more likely to be more complex than those registered later. For example, an actor that manages a set of worker actors to achieve a particular goal will be registered before the worker actors. Algorithm 5 shows the trivial implementation of this strategy.

ALGORITHM 5: The prioritization strategy RT.

```
   Input  : trace, the execution trace
   Output : mapping, a mapping of actor addresses to priorities
1  mapping ← []
   // Explore all registrations
2  for r ← trace.registrations do
       // Put the timestamp for that actor in the mapping
3      mapping ← mapping + (r.b_child ↦ r.m_registration)
4  return mapping
```

6.6.3 Message Time (MT)

This strategy computes the priority for a fault tuple based on the lowest turn identifier (*i.e.,* j_{turn}) of the receiving actor (*i.e.,* b_{to}). This identifier represents the first received message. This strategy is based on the *Queue* heuristic from [LKMA10]. The reasoning behind this strategy is that actors might be more complex when they receive messages early as they are more likely to be higher up in the communication topology. Algorithm 6 shows the implementation of this strategy.

ALGORITHM 6: The prioritization strategy MT.

```
   Input  : trace, the execution trace
   Output : mapping, a mapping of actor addresses to priorities
1  mapping ← []
   // Explore all turns
2  for turn ← trace.turns do
       // Get the current priority for this actor
3      priority ← mapping.getOrElse(turn.b_to, Integer.Max)
       // Turn identifier must be lower than current priority
4      if turn.j_turn < priority then
           // Replace the mapping
5          mapping ← mapping + (turn.b_to ↦ turn.j_turn)
6  return mapping
```

6.6.4 Actor Fan-In (FI)

This strategy computes the priority for a fault tuple based on the number of distinct actors *from* which the receiving actor (*i.e.,* b_{to}) receives messages. The reasoning behind this strategy is that actors are likely to be more complex when they receive messages from several different actors. For example, actors that implement the scatter-gather pattern[4] [Ver15] might benefit from this strategy. In that case, the aggregator actor accumulates responses from different actors and thus is more likely to be complex. Algorithm 7 shows the implementation of this strategy. The algorithm collects all messages and computes for each receiving actor (*i.e.,* b_{to}) a set of distinct sender actors (*i.e.,* a_{from}) and determines the priority on the cardinality of the set.

[4] https://www.enterpriseintegrationpatterns.com/patterns/messaging/BroadcastAggregate.html

ALGORITHM 7: The prioritization strategy FI.

```
   Input  : trace, the execution trace
   Output : mapping, a mapping of actor addresses to priorities
1  sets ← []
   // Explore all sends
2  for send ← trace.sends do
       // Get the current fan-in of receiving actor
3      set ← sets.getOrElse(send.b_to, ∅)
       // Update the mapping
4      sets ← sets + (send.b_to ↦ (set + send.a_from))
   // Use set cardinality as priority
5  mapping ← sets.map((address, set) ⇒ (address, set.size))
6  return mapping
```

6.6.5 Actor Fan-Out (FO)

Similar to the previous strategy, this strategy computes the priority for a fault tuple based on the number of distinct actors *to* which the receiving actor (*i.e.,* b_{to}) sends messages. While an actor might be more complex when it has a high fan-in, the same applies to actors that send messages to many different actors.

For example, the actor in the scatter-gather pattern will send messages to multiple different actors and therefore is likely to have a high fan-out value. Algorithm 8 shows the implementation of this strategy. The algorithm collects all messages and computes for each sender actor (*i.e.,* a_{from}) a set of distinct receiving actors (*i.e.,* b_{to}) and determines the priority on the set's cardinality.

ALGORITHM 8: The prioritization strategy FO.

```
   Input  : trace, the execution trace
   Output : mapping, a mapping of actor addresses to priorities
1  sets ← []
   // Explore all sends
2  for send ← trace.sends do
       // Get the current fan-out of sending actor
3      set ← sets.getOrElse(send.a_from, ∅)
       // Update the mapping
4      sets ← sets + (send.a_from ↦ (set + send.b_to))
   // Use size of sets as priority
5  mapping ← sets.map((address, set) ⇒ (address, set.size))
6  return mapping
```

6.6.6 Actor Fan-In/Fan-Out (FIFO)

The last prioritization strategy computes the priority for a fault tuple based on the ratio of the fan-in and fan-out priorities of the receiving actor. It returns `Integer.Max` as priority when there does not exist a fan-out value for the actor. The reasoning behind this strategy is that actors that communicate with different actors in both ways are more likely to be more complex. This should classify actors into three different groups: actors that receive from many distinct actors (*i.e.,* FIFO > 1), actors that send to many distinct actors (*i.e.,* FIFO < 1), and actors that send to and receive from many distinct actors (*i.e.,* FIFO ≈ 1). This ratio might be more representative then only the fan-in or fan-out heuristic. Algorithm 9 shows the implementation of this strategy.

ALGORITHM 9: The prioritization strategy FIFO.

Input : *fi*, a fan-in mapping
fo, a fan-out mapping
Output : *mapping*, a mapping of actor addresses to priorities

```
// Compute the ratio based on fan-in and fan-out
mapping ← fi.map((address, fip) ⇒
    // Get the fan-out priority
    fop ← fo.getOrElse(address, Integer.Max)
    // Replace the mapping
    (address ↦ fip/fop))
return mapping
```

6.6.7 Summary

We propose multiple prioritization strategies that use actor-specific characteristics to find failure-inducing faults sooner. Additionally, we provide an ascending and descending implementation for each strategy as it can be interesting to completely reverse the ordering. While determining effective strategies is a research domain on its own, we evaluate these different strategies in Section 7.2 to get a better understanding of how they can affect the performance of each exploration strategy. We leave more advanced strategies such as those using historical data or machine learning for a future avenue.

6.7 Implementation

We briefly discuss the implementation of Chaokka[5] as it can be a foundation for further research.

6.7.1 Usage

While we do not provide a user interface for this tool, running an analysis requires minimal effort of developers. Listing 6.3 demonstrates the minimal code required to execute a resilience analysis. This analysis uses our exploration strategy based on delta debugging with a developer-specified pruning strategy and ascending prioritization strategy.

```
1 val target = "path/to/target"
2 val suite = Some("MyTestSuite")
3 val test = Some("MyTestCase")
4
5 val runner = new ResilienceAnalysisRunner(target, suite, test)
6 val analysis = new DDResilienceAnalysis(PersistentActorRestart) with
   AscendingFanIn {
7   override def onInitialFilter(targets: List[Traceable]): List[Traceable]
     = targets.filter({
8       case s: Send => s.clazz.equals("Packet")
9       case _ => false
10  })
11 }
12
13 runner.initialize()
14 runner.analyse(p => analysis, 10)
```

Listing 6.3: A resilience analysis in Chaokka that tests for resilience defects with respect to actor restarts.

Developers must create an instance of the class `ResilienceAnalysisRunner` to run Chaokka and expects the target path of the system as a minimum (line 5). Optionally, one can provide the name of a test suite, or test suite and test case to only apply the analysis to a certain tests only. Next, one must choose which resilience analysis to run and create an instance of it (*i.e.,* `DDResilienceAnalysis`) (line 6). This analysis uses a developer-specified pruning strategy (*i.e.,* the method `onInitialFilter`) to only keep messages of type `Packet`. It also makes use of the ascending variant of the fan-in prioritization strategy (*i.e.,* `AscendingFanIn`).

As the fault type (*i.e.,* `PersistentActorRestart`), we choose to inject faults in persistent actors to check whether the system is resilient to actors being restarted. The developer then has to call the method `initialize` (line 13) which discovers tests in the target system or loads an existing index file, and the method `analyse` (line 14) with the number of times that the analysis has to be repeated. Once the runner completes successfully, the results will be available in the console and CSV files.

[5] https://github.com/jonas-db/chaokka

6.7.2 Extension

Chaokka is implemented from the ground up to be extensible. One should be able to implement other resilience analyses with exploration strategies and heuristics with minimal effort through the provided classes.

6.7.2.1 Perturbations

The abstract class `Perturbation` shown in Listing 6.4 defines the interface for implementing new types of faults. The method `pre` determines whether the fault can be injected in a given target (*i.e.,* `Traceable`). The method `inject` creates one or more fault tuples (*i.e.,* `ActorConfig`) which will consists of the target and the fault type.

```
abstract class Perturbation {
  def pre(perturbable: Traceable, report: TestReport): Boolean
  def inject[A <: Traceable](
    perturbable: A,
    report: TestReport,
    messageCandidates: Set[Traceable],
    actorCandidates: Set[Traceable]): List[ActorConfig]
}
```

Listing 6.4: The abstract class `Perturbation`.

For instance, Listing 6.5 shows the implementation of a perturbation which targets user messages sent with at-least-once delivery semantics. The method `pre` only returns true for these messages (*i.e.,* `Send`) when they are sent between two user actors, while the method `inject` creates the corresponding `ActorConfig` which consists of a fault target (*i.e.,* `ActorMessage`). The number 1 in the `ActorMessage` determines that the fault is a duplication of the current message.

```
  object AtLeastOnceDeliveryDuplication extends Perturbation {

    override def pre(p: Traceable, r: TestReport): Boolean = p match {
      case s: Send if
        s.spath.contains("/user/") && s.rpath.contains("/user/") &&
        r.atLeastOnceDeliveryMessages.contains(s.clazz)
          => true
      case _ => false
    }

  override def inject[Send](
    p: Send,
    report: TestReport,
    messageCandidates: Set[Traceable],
    actorCandidates: Set[Traceable]): List[ActorConfig] =
      p match {
        case Send(senderName, _, receiverName, _, hash, clazz, _, _) =>
          val message = ActorMessage(clazz.r, hash, 1, senderName.r)
          val config = ActorConfig(receiverName.r, List(message))
          List(config)
      }
}
```

Listing 6.5: The implementation of a fault that simulates a message that arrives multiple times due to the guaranteed message delivery.

6.7.2.2 Resilience Analysis

Secondly, the abstract class `ResilienceAnalysis` shown in Listing 6.6 defines the interface for implementing analyses with different exploration strategies. The method `run` should be implemented by the developer. The function `test` will execute the current test case with the passed set of faults (*i.e.,* `Set[ActorConfig]`). Several methods (line 11-15) are given a default implementation. The method `onInitialReport` (line 4–8) determines the fault targets for a given fault type (*i.e.,* value of `perturbation`), while the method `onSubsequentReport` is called for every subsequent test run. There will always be one initial run to get an initial trace, and then multiple test runs were a fault scenario is injected. The method `hasNext` determines whether there are fault scenarios left to be tested and the method `next` correspondingly provides this next scenario.

```
1 abstract class ResilienceAnalysis {
2   val perturbation: Perturbation
3   var perturbations: Set[ActorConfig] = Set.empty[ActorConfig]
4   def run(
5       test: Set[ActorConfig] => Status,
6       initialReport: TestReport,
7       start: Long,
8       timeout: Int): Unit
9   def name: String
10
11  def onInitialReport(r: Option[TestReport]): Unit = ...
12  def onSubsequentReport(report: Option[TestReport]): Unit = ...
13  def hasNext: Boolean = ...
14  def next(): ActorConfig =  ...
15  def onInitialFilter(targets: List[Traceable]): List[Traceable] = ...
16 }
```

Listing 6.6: The abstract class `ResilienceAnalysis`.

For example, Listing 6.7 shows the implementation of a resilience analysis that loads a fault scenario from a JSON file. The elements of the fault scenario are assigned to the variable `perturbations`. Since there is only one fault scenario to be explored, the method `hasNext` is overridden to return false.

```
1 class StaticResilienceAnalysis(
2     configFilePath: String,
3     override val perturbation: Perturbation)
4       extends ResilienceAnalysis {
5
6     perturbations = JSONParser.load(configFilePath, 0).toSet
7
8     override def run(
9       test: Set[ActorConfig] => Status,
10      initialReport: TestReport,
11      start: Long,
12      timeout: Int): Unit = {
13        test(perturbations)
14    }
15
16    override def hasNext: Boolean = false
17    override def name: String = "static"
18 }
```

Listing 6.7: The implementation of a resilience analysis that uses a developer-specified exploration strategy which loads a fault scenario from a JSON file.

6.7.2.3 Prioritization Strategies

Finally, the trait `PrioritizationStrategy` shown in Listing 6.8 defines the interface for implementing prioritization strategies. Such an implementation provides the method `compute` which returns the list of n partitions of $faults$ sorted according the priority of the fault tuples, the method `sorter` specifies the order of two given priorities (*i.e.,* ascending or descending order), and one variable `abbreviation` which is a string representation of the strategy for reporting purposes.

```
trait PrioritizationStrategy {
  def compute(faults: Set[ActorConfig], n: Int): List[Set[ActorConfig]]
  def sorter: (Int, Int) => Boolean
  val abbreviation: String
}
```

Listing 6.8: The trait `PrioritizationStrategy`.

For instance, Listing 6.9 shows the implementation of the `MessageTime` strategy. This strategy computes the priority of fault targets based on the identifier of the turn that processes the first message. We refer the reader back to Section 6.6 for the details of each prioritization strategy.

```
trait MessageTime extends PrioritizationStrategy {

    override def compute(faults: Set[ActorConfig], n: Int):
      List[Set[ActorConfig]] = this.initialReport match {
        case Some(r) =>
            var mapping: Map[String, Int] = Map.empty

            // Collect all turns and keep the first turn for every actor
            this.onInitialFilter(r.trace).foreach({
                case s: Turn =>
                    val current = mapping.getOrElse(s.rpath, s.turnID)
                    val nw = if(s.turnID < current) s.turnID else current
                    mapping = mapping + (s.rpath -> nw)
                case _ => ()
            })

            // Sort based on priority
            val heuristic: List[ActorConfig] = faults.toList.sortWith((a,
             b) => {
                val ai = mapping.getOrElse(a.actorName.toString, 0)
                val bi = mapping.getOrElse(b.actorName.toString, 0)

                sorter(ai, bi)
            })

            distribute(heuristic, faults, n).map(x => x.toSet)
        case None =>
            List.empty
    }

    override val abbreviation: String = "MessageTime"
}
```

Listing 6.9: The implementation of the MT prioritization strategy.

6.7.3 Limitations

CHAOKKA is the first resilience testing tool for actor systems and therefore has several limitations in terms of scalability, non-determinism, and test cases.

Scalability. CHAOKKA is currently limited to a single actor system hosted on a single physical machine. This limitation affects the kind and scale of systems that CHAOKKA can test. Nevertheless, AKKA is built around the notion of location transparent actor addresses which abstract away the difference between communicating with actors running on the same JVM or a different JVM (*e.g.,* a cluster of multiple machines). Configuration files or environment variables can be used to change the actor's location without altering the code. In this way, our approach can test a completely distributed deployment on a single JVM. Furthermore, one could wrongly assume that actor systems running on a single physical machine are resilient. Arguably, the probability for a message to be lost or an actor to be restarted is much lower on a single JVM then it is on a distributed system. Nevertheless, a shutdown of the whole system requires actors to recover in the same way as when actors would be distributed.

Non-determinism. CHAOKKA extracts fault targets from the trace of the first test execution only. Therefore, our approach might miss resilience defects when test executions are non-deterministic. For example, subsequent test executions might spawn different actors or exchange different messages, or both. We distinguish two kinds of non-determinism: due the scheduler (*e.g.,* one message arrives before another one) or due to data (*e.g.,* the result of the method `random`). CHAOKKA does not enforce prior execution paths during subsequent runs of a test and is therefore limited to deterministic systems. A consequence of not recording and replaying the actor schedule is that we have to identify messages by their sender, receiver and message hash, instead of the unique send identifier. CHAOKKA will therefore not be able to distinguish between multiple identical messages or detect changes in message payload across different runs of a test. Implementing the required record-replay mechanisms (*e.g.,* [AMB+18, LPV19, GKS05]) for AKKA is left for a future avenue.

Test Case Format. The causality-based pruning strategy expects that tests adhere to a specific format consisting of three steps: send message, wait for result, assert stable state. The first step consists of one message that is sent to the system to initiate a request, the second step blocks until a given condition is satisfied (*e.g.,* wait for the system to respond), and the final step consists of asserting the response, the system, or both. This approach greatly simplifies the implementation as the turn of the test actor and the sent message will be the root of our causality relation. Otherwise there might be multiple causality relations which makes the implementation of our approach more complex. Thus, this boils down to an implementation detail which can be solved through additional engineering efforts.

Soundness & Completeness. It is important to understand that we present a testing approach and not a verification approach. Chaokka therefore might not be complete and might not find all defects in the implementation of resilience mechanisms. However, incompleteness is typical for dynamic analyses as they are limited to what can be observed during one or more executions of the program under analysis. Moreover, Chaokka explores more behaviours of the system under test than defined in the original test suites. This can lead to failed test outcomes even when the system is resilient and vice versa. For example, it might be that the the oracle determines that the test failed because the assertions are too strong with respect to all possible executions. However, the quality of the existing test oracles is a general problem and beyond our control (*cf.* Section 2.2). We consider Chaokka to be sound when the test suites and their oracles are sound.

6.8 Application Domains

Our approach to resilience testing can be generalized to multiple other distributed architectures such as microservices and message brokers. We also provided Chaokka as an open-source tool so that developers can use its implementation as a foundation.

Actor Frameworks. Chaokka targets Akka since it is the most popular implementation of the actor model on the jvm, with both a Java and a Scala implementation. However, our approach should be equally applicable to actor frameworks and languages such as Erlang, Orleans, Pykka, Actix, *etc.* While some of these might offer fewer abstractions for implementing resilience mechanisms, ad-hoc implementations are also prone to the discussed implementation problems (*cf.* Section 6.1). To support applications built in these frameworks and languages, it should suffice to intercept actor events at run time and collect them in the proposed execution trace format (*cf.* Section 6.3.1).

Microservices. According to the Scala Developers Survey 2019[6], the Scala ecosystem is used by a majority of developers to implement microservices. It is easy to draw similarities between the actor model and microservices architectures [LF14] as 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[7], a microservices framework built on top of Akka. Our approach should therefore be transposable to microservice frameworks such as Lagom and Spring Cloud[8].

[6] https://scalacenter.github.io/scala-developer-survey-2019

[7] https://www.lagomframework.com

[8] https://spring.io/projects/spring-cloud

Message brokers. Message brokers such as KAFKA[9] or RABBITMQ[10] enable consumers to subscribe to messages which are published to a topic by independent producers. These are often used in combination with microservices, but also to connect multiple independent systems. Several of these brokers support mechanisms for at-least-once message delivery which exposes them to similar defects as presented in this dissertation. For example, consumers of message brokers should be aware that they can receive messages multiple times and need to take the corresponding actions.

6.9 Conclusion

This chapter presented our dynamic approach to resilience testing of actor systems. We started by motivating the need for automated resilience testing because implementing and testing resilient distributed actor systems remains difficult. Next, we provided an overview of CHAOKKA — our dynamic approach to resilience testing which aims to uncover resilience defects. It uses fault injection as its foundation to simulate the system under abnormal conditions. We then formally defined the concepts of execution traces, the causality relation, and fault scenarios in actor systems. Next, we proposed several exploration, pruning, and prioritization strategies that can be composed to form a resilience analysis. We presented our exploration strategy based on delta debugging and our pruning strategy based on causality of actor events. Moreover, we presented five different prioritization strategies that use actor-specific characteristics. Penultimately, we discussed the implementation of CHAOKKA including its usage, its extension possibilities, and its limitations. Finally, we discussed several application domains that are closely related and could benefit from our approach. The next chapter presents an evaluation of our approach by applying CHAOKKA on actor systems seeded with resilience defects.

[9] https://kafka.apache.org
[10] https://www.rabbitmq.com

Chapter 7

Experimental Evaluation of Resilience Testing

This chapter evaluates our approach to resilience testing. In particular, the goal of this evaluation is to examine the performance of resilience analyses that combine different exploration, pruning, and prioritization strategies. To this end, we run Chaokka on generated actor systems seeded with defects in their resilience mechanisms.

First, Section 7.1 presents a study on the performance of our approach to resilience testing. We compare the results of three resilience testing analyses configured with a default prioritization strategy. The analysis rt-n uses a naive exploration strategy, the analysis rt-dd uses our exploration strategy based on delta debugging, and the analysis rt-dd-o optimizes rt-dd with causality-based pruning. We also determine the overhead of our implementation for the actor model framework Akka by comparing test execution times with and without fault injection. Based on the mean number of test executions required to find a resilience defect, we observe that rt-dd and rt-dd-o are respectively about 3 and 5 times faster than rt-n. Next, Section 7.2 presents a study on the effect of prioritization strategies on the performance. We conclude that these strategies can deteriorate the performance of an analysis with a naive exploration strategy, while it has limited effect on the performance of analyses with our exploration strategy. Finally, we conclude this chapter by summarizing our observations.

7.1 Detection of Resilience Defects

This section reports on the performance of our approach and the execution overhead of our implementation. Section 7.1.1 presents the design of the study, while Section 7.1.2 presents the results. The results show that the exploration strategy based on delta debugging is able to achieve a speedup of about 3, while a speedup of about 5 is achieved when pruning is additionally enabled. The overhead of our implementation for the AKKA framework is limited when it simulates duplicate messages, but quickly increases when it simulates actor restarts.

7.1.1 Design

The goal of this evaluation is to understand the difference in performance between different resilience analyses when the default prioritization strategy (*i.e.,* SHU) is enabled. The analysis RT-N uses the exhaustive exploration strategy (*i.e.,* NAIVE), while the analysis RT-DD uses our exploration strategy (*i.e.,* DD). We also investigate the performance of the analysis RT-DD-O which combines DD with our causality-based pruning strategy. We refer the reader back to Chapter 6 for the details about each strategy. This study aims to answer the following research questions:

- **RQ$_1$:** *What is the performance of the resilience analyses* RT-DD *and* RT-DD-O *compared to* RT-N *in detecting the resilience defects?*
- **RQ$_2$:** *What is the overhead of* CHAOKKA *on the execution of test cases?*

7.1.1.1 Generation Process

We automatically generate actor systems for our studies and randomly seed them with resilience defects because we could not find representative open-source actor systems that have resilience mechanisms (*i.e.,* guaranteed message delivery and persistent actors) implemented in the majority of their actors. A true empirical study to evaluate our approach is therefore not possible.

Nevertheless, the generated actor systems feature communication topologies that are representative for known microservices architectures. For example, Figure 7.1 shows the architecture of EBAY's payment platform[1]. That is, we assume that one actor corresponds to one microservice. This evaluation approach is also used in the context of microservices binary trees of various depths are generated to represent systems (*e.g.,* GREMLIN [HRJ+16]).

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

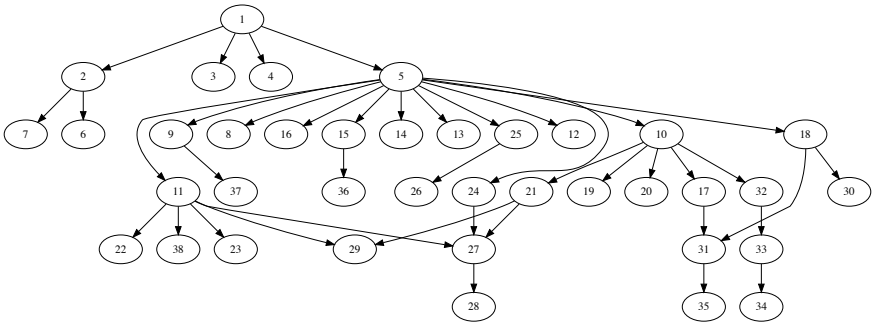

Figure 7.1: The communication topology of eBay's payment platform.

The generation algorithm is simple, yet effective enough to create representative topologies. This process consists of three steps. The input to this generation process is n and m, indicating the number of actors and messages respectively present in the system. First, we represent an actor system by a list of n identifiers where each identifier represents a single actor. We consider the first actor in the list to be the root actor (*i.e.,* the entry point of the system). Next, the method `generatePairs` shown in Listing 7.1 generates pairs of the form (x, y) from this list. Such a pair represents a connection where x sends a message to y. For simplicity, each actor understands only one type of message and therefore the pair adheres to the constraint that the identifier of x is always smaller than the identifier of y to avoid infinite loops.

```
1 def generatePairs(nodes: List[Int]): List[(Int, Int)] = {
2   var a: List[(Int, Int)] = List.empty
3
4   for(i <- 0 until nodes.size) {
5       for(j <- (i+1) until nodes.size) {
6           val pair = (nodes(i), nodes(j))
7           a = pair :: a
8       }
9   }
10
11  a
12 }
```

Listing 7.1: The method to generate unordered pairs from a list of nodes.

For example, calling the method `generatePairs` with `List(1, 2, 3, 4, 5)` will yield the following pairs:

```
(1,2), (1,3), (1,4), (1,5), (2,3), (2,4), (2,5), (3,4), (3,5), (4,5)
```

Next, the method `randomSubset` shown in Listing 7.2 selects an arbitrary subset of m pairs. The cardinality of this subset represents the number of messages that are exchanged in the actor system as each connection represents one message.

```
1 def randomSubset(pairs: List[(Int, Int)], m: Int) =
2   scala.util.Random.shuffle(pairs).take(m)
```

Listing 7.2: The method to select a random subset with cardinality m.

For example, selecting a subset of cardinality 5 will yield the following pairs:

```
(1,2), (2,4), (3,4), (3,5), (4,5)
```

However, each pair in this set does not connect with another pair as there is no connection between the actors 2 and 3. As a final step, we perform a final check to determine actors without a connection and mitigate this lack of connectivity. That is, we check whether every actor receives at least one message from another actor. This is repeated for each pair, excluding the pair that represents the root actor. A random connection is generated when an actor does not receive any message. For example, this step would detect that there doesn't exist such a pair `(_,3)` for the pairs `(3,4)` and `(3,5)`. That is, actor 3 does not receive any message as there does not exist a pair where the y is actor 3. As a result, it might generate the random pair such as `(1,3)` or `(2,3)` such that the pairs now define an actor system with n actors where there is at least one message between each actor:

```
(1,2), (1,3), (2,4), (3,4), (3,5), (4,5)
```

As a result, we get slightly more messages in the system then initially specified (*i.e.,* more than m messages). This will also be the reason why our generated systems do not have a rounded number of faults in our studies. Figure 7.2 shows a generated actor system consisting of 50 actors.

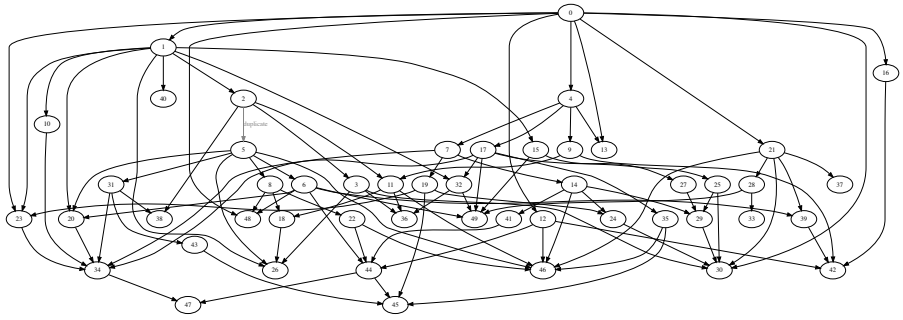

Figure 7.2: A generated actor system with a seeded resilience defect.

By default, each actor is resilient to actor restarts and message delivery failures. This is achieved by having each actor persisting a counter as its internal state and checking for duplicated messages by keeping track of which messages have already been received.

During the execution of the system, messages are sent with guaranteed message delivery to one or more actors and acknowledged with a message that is delivered at most once. Each message changes the internal state of the receiving actor. As indicated in Section 2.3.2, this is done through event sourcing. An event is persisted for each received message and the actor's counter value is incremented once the event is successfully persisted.

The actors of each generated system can simulate two kinds of resilience defects. The first defect is an incorrect implementation of the recovery mechanism by not rehydrating the counter value, while the second defect is an incorrect implementation of message idempotence by skipping the check for duplicated messages. For each system, we also generate a test case as shown in Listing 7.3.

```
def assertSystem(
  name: String,
  topology: List[(Int, List[Int])],
  messageDeliveryFaults: Set[Int] = Set(),
  persistenceFaults: Set[Int] = Set()): Unit = {

  // Stable after 5 seconds of no message activity
  Stabilizer.init(5 * 1000)

  // Instantiate the actors for the given topology
  val t = Topology(topology, messageDeliveryFaults, persistenceFaults)
  val rfs: Set[ActorRef] = Topology.generate(system, t, testActor)
  val m: Map[String, String] = rfs.map(x => (x.path.name -> x.path)).toMap

  // Expected state is the number of unique paths to that actor
  def findPaths = (r) => t.findPaths(r.path.name.substring(6).toInt)
  val expected = rfs.map(r => (r.path.toString -> findPaths(r))).toMap

  // Initiate the system with a request.
  val head = system.actorSelection(system.child(s"actor-0"))
  head ! Update(-1, 1)

  // Wait for a stable system state.
  Await.ready(Stabilizer.stabilize(), 60 * 60 seconds)

  // Ask the state of each actor.
  rfs.foreach(_ ! State)

  // Wait for replies.
  var resultAll: List[(Int, Int)] = List.empty[(Int, Int)]
  while(resultAll.size != expected.size) {
      val r = receiveOne(Duration.Inf)
      r match {
          case m: (Int, Int) => resultAll = m :: resultAll
          case m => m
      }
  }

  // Sort on actor name, then assert the state of each actor.
  val sorted = resultAll.sortBy(_._1).zipWithIndex
  sorted.foreach(r => {
      // Get the complete path for a given actor name
      val p = m("actor-"+r._2)
      // Assert that the actors state is the expected value
      assert(r._1._2 == expected(p), s"Something failed, error@[${p}]")
  })
}
```

Listing 7.3: A test case to test the behavior of our generated actor systems.

This test case sends a message to the entry point of the system (*i.e.,* the root actor) and asserts the system's state after all communication has happened. It does that by starting all actors for the given `topology` and uses the sets `messageDeliveryFaults` and `persistenceFaults` to determine which actors have to simulate a specific resilience defect (line 11—12). The next lines compute the expected state for each actor (line 16—17). This state is automatically computed based on the generated topology. In particular, we compute the final counter value of each actor by counting the number of paths from the root actor to each actor. This number of paths to each actor equals to the number of messages it will receive, and therefore equals the value of the counter. Next, it sends an initial message to the root actor (line 21). It will then wait until the system's execution is complete (line 24) and wait to receive a state update from each actor (line 27—37). Finally, it checks whether the received counter value is equal to the expected counter value for each actor (line 40—46). We will evaluate our approach based on these generated actor systems and test cases.

7.1.1.2 Data Set

We create our data set by repeating the generation process 10 times with $n = 50$, increasing m, and randomly choose the resilience defect type. This ensures that we generate a diverse set of actor systems with different communication topologies. Table 7.1 summarizes the data set which consists of 10 generated actor systems where each system has 50 actors, a varying number of messages, and a varying type of resilience defect.

#	1	2	3	4	5	6	7	8	9	10
Messages	258	408	616	378	1026	626	706	854	1770	2008
Resilience Defect	D	D	D	R	D	R	R	R	D	D
Fault Targets	129	204	308	378	513	626	706	854	885	1004

Table 7.1: The data set consists of 10 actor systems with a varying number of fault targets. The resilience defect is either related to restarting actors (R) or duplicating messages (D).

Remember that our approach uses messages as fault targets because the turns that process these messages represent a computational step of each actor. Therefore, we can find defects at a fine-grained level. Such a defect is either an implementation mistake related to the guaranteed message delivery mechanism which does not account for duplicate messages (D), or an implementation mistake related to the persistence mechanism which does not hydrate the state correctly after a restart (R). It is clear that the number of fault targets is half of the number of messages when the resilience defect is related to guaranteed message delivery. The reason for that is that the number of messages includes both messages sent with at-most-once and at-least-once message delivery semantics. However, duplication of messages can only occur for messages sent with at-least-once semantics.

7.1.1.3 Studies

Our evaluation consists of two studies. We address $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$ by conducting $\mathbf{Study}_1$ and $\mathbf{Study}_2$ respectively. Each study is based on the set of data discussed in Section 7.1.1.2.

$\mathbf{Study}_1$: We create 3 variants of each system of our data set by seeding the given resilience defect in one of the actors with number 5, 25, or 45. These actors reside at different locations in the communication topology. The resulting set of systems thus consists of 30 actor systems with varying size and defect location. We run each analysis (*i.e.,* RT-N, RT-DD, and RT-DD-O) on these actor systems with the default prioritization strategy enabled. We repeat the studies for each system 10 times with a timeout of 30 minutes. We compare the number of test executions to assess the performance of each analysis.

$\mathbf{Study}_2$: We select the largest generated actor system from our set of data (*i.e.,* the one with 2008 messages) and systematically select and apply n faults, where n increases in steps of 100. We repeat this study 10 times and compare the execution time to assess the overhead of each defect.

Caching of fault scenarios is enabled while monotonicity is disabled during each study. All studies 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.

7.1.2 Results

We discuss the performance of the analyses RT-N, RT-DD, and RT-DD-O ($\mathbf{RQ}_1$) in Section 7.1.2.1, while the overhead of CHAOKKA ($\mathbf{RQ}_2$) is discussed in Section 7.1.2.2. For each of the results, we provide box plots to visualize the results on a high level and provide several tables with detailed information to get a complete overview of the results. Note that all numbers are based on the 3 variants of each system and that these numbers are only based on analyses that did not time out.

7.1.2.1 $\mathbf{Study}_1$: Performance of The Resilience Analyses

Figure 7.3 shows the results of all analyses that did not time out after 30 minutes. It is clear that the number of test executions required by RT-N fluctuates widely, while RT-DD and RT-DD-O require fewer tests to find the seeded resilience defect and are much more stable in performance. We summarize the results in Table 7.2, while Table 7.3 summarizes the result for each system individually.

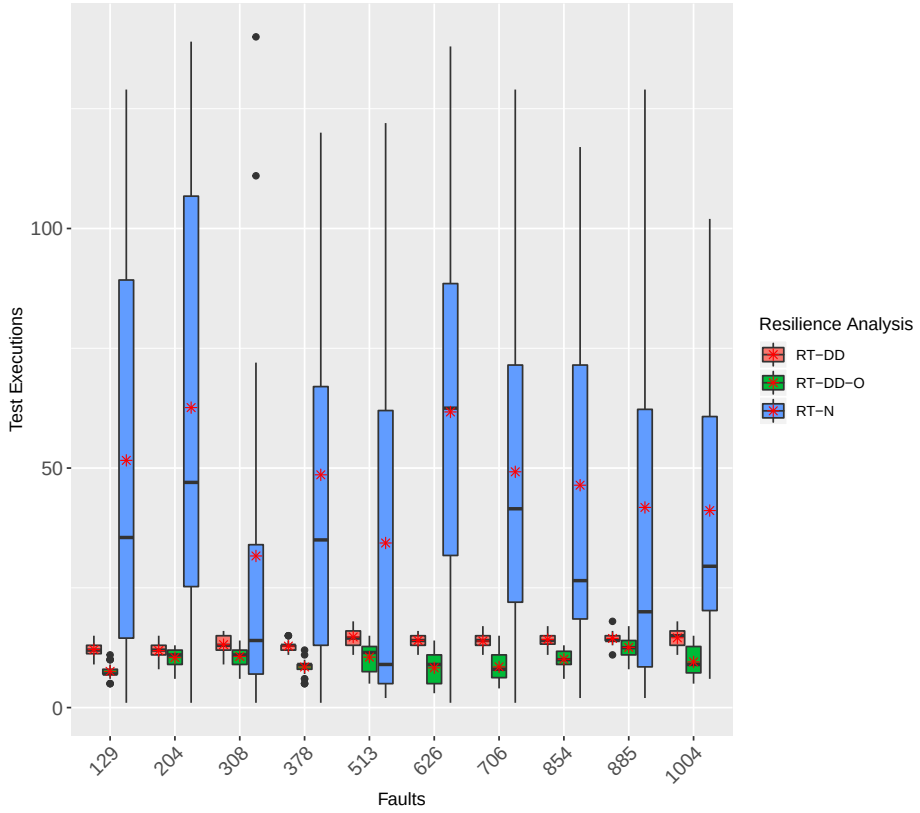

Figure 7.3: The number of test executions for all three resilience analyses.

	Mean	Median	Analyses	Timeouts
RT-N	47	33	229	71
RT-DD	14	15	300	0
RT-DD-O	10	11	300	0

Table 7.2: A summary of each resilience analysis executed on the set of 30 generated actor systems. It includes the mean and median number of test executions required to detect the seeded resilience defect, as well as the number of timeouts.

The following observations are based on the means shown in Table 7.2. It takes RT-N 33 and 37 tests more to find the defect compared to RT-DD and RT-DD-O respectively, while it takes RT-DD 4 tests more compared to RT-DD-O. RT-N therefore requires 3.36 and 4.70 times the test executions of RT-DD and RT-DD-O respectively, while RT-DD requires 1.4 times the tests of RT-DD-O. That is, RT-DD and RT-DD-O are respectively about 3 and 5 times faster than RT-N.

RT-DD-O is about one and a half times faster than RT-DD and indicates that our pruning strategy can further improve the performance.

We also discuss our observations based on the medians shown in Table 7.2. It takes RT-N 18 and 22 tests more to find the defect compared to RT-DD and RT-DD-O respectively, while it takes RT-DD 4 tests more compared to RT-DD-O. RT-N therefore requires 2.20 and 3.00 times the test executions of RT-DD-O and RT-DD respectively, while RT-DD requires 1.36 times the test executions of RT-DD-O. Based on the medians, RT-DD and RT-DD-O are about 2 and 3 times faster than RT-N. RT-DD-O remains about one and a half times faster than RT-DD.

#		1	2	3	4	5	6	7	8	9	10
Messages		258	408	616	378	1026	626	706	854	1770	2008
Resilience Defect		D	D	D	R	D	R	R	R	D	D
Fault Targets		129	204	308	378	513	626	706	854	885	1004
RT-N	Mean	52	63	32	49	34	62	49	46	42	41
	Median	36	47	14	35	9	62	42	26	20	30
	Timeouts	0	4	13	1	13	6	2	6	12	14
RT-DD	Mean	12	12	13	13	15	14	14	14	14	15
	Median	12	12	13	13	14	14	14	14	14	15
	Timeouts	0	0	0	0	0	0	0	0	0	0
RT-DD-O	Mean	8	10	11	8	11	8	8	10	13	10
	Median	7	11	11	9	12	9	8	10	12	9
	Timeouts	0	0	0	0	0	0	0	0	0	0

Table 7.3: The mean and median number of test executions needed to find a resilience defect, as well as the number of timeouts. These numbers are based on the 3 variants of each system. The resilience defect is either related to restarting actors (R) or duplicating messages (D).

Testament to the efficiency of both RT-DD and RT-DD-O is that they never timed out as shown in Table 7.3. This means that these analyses run for a total of 300 times without any timeouts (*i.e.,* 10 systems × 3 variants × 10 analyses). This is in contrast to RT-N which timed out in 71 of the cases and thus only has a total of 229 successful runs. Table 7.3 shows the number of timeouts for each system and analysis. This table also shows the mean and median number of required test executions to find the seeded resilience defect. Note that RT-N did not necessarily time out more often when an increasing number of faults needed to be explored because the default prioritization strategy randomly shuffles fault scenarios.

RQ_1 Summary

Both RT-DD and RT-DD-O outperform RT-N. The mean number of test executions of RT-DD and RT-DD-O is respectively about 3 and 5 times lower than RT-N to find a single resilience defect. RT-DD-O demonstrates that our causality-based pruning strategy can be leveraged to further improve performance.

7.1.2.2 Study$_2$: Overhead of Chaokka

Figure 7.4 shows the difference in test execution time when test cases are injected with increasingly larger fault scenarios.

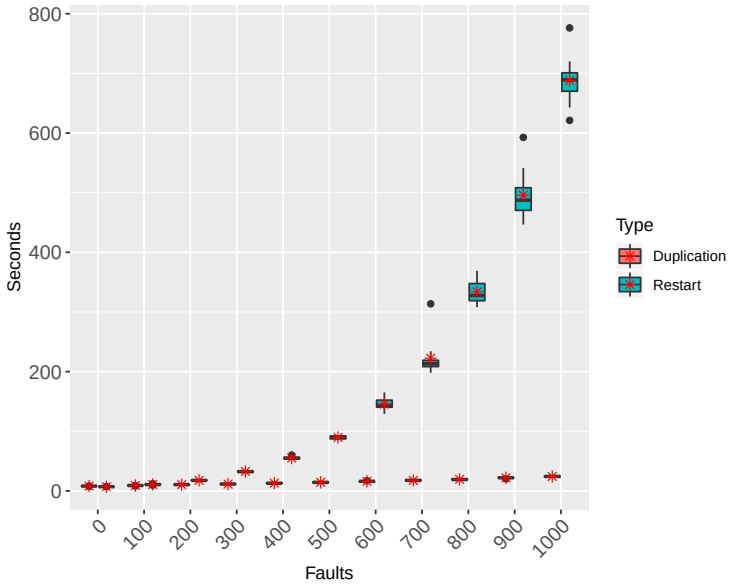

Figure 7.4: The overhead of Chaokka for each fault type.

We observe that the overhead of scenarios consisting of duplication faults grows linearly, yet at a very slow rate. On the contrary, the overhead of scenarios consisting of restart faults seems to grow exponentially at a fast rate. This was to be expected as sending and queueing asynchronous messages is computationally cheap and thus causes little overhead. However, restarting and hydrating an actor is computationally more expensive. In particular, the actor instance needs to be stopped, restarted, and hydrated again with all events from the journal which causes a larger overhead. However, this overhead might be less for actors that persist and recover their state through other means than event sourcing.

Nevertheless, the overhead of at most 13 minutes for a large fault scenario of 1000 faults is still within acceptable limits and indicates that Chaokka could be integrated into the testing process of particular systems. As shown in other studies (*e.g.,* [HRJ+16]), fault injection approaches inherently cause some overhead due to code instrumentation and monitoring. There is also ample room for improvements in our implementation which we discuss in Section 7.3.

RQ_2 Summary

CHAOKKA causes an acceptable execution overhead, but might become problematic when large fault scenarios consisting of faults related to persistent actors have to be explored. However, this overhead is specific to the implementation of our approach for the actor model framework AKKA.

7.2 Prioritization of Faults

This section reports on the performance of resilience analyses that combine prioritization strategies with the exploration strategies NAIVE and DD. We presented several prioritization strategies typical to actor systems in Section 6.6 and implemented them in CHAOKKA. Section 7.2.1 presents the design, while Section 7.2.2 presents the results of this study. The results show that the prioritization strategies have limited impact on the performance when combined with DD. The difference between ordering faults in ascending and descending order with respect to the prioritization strategy can significantly change the performance of analyses that use NAIVE, while the change remains limited for analyses that use DD.

7.2.1 Design

The goal of this study is to understand the difference in performance of analyses that combine each prioritization with the exploration strategy NAIVE or DD. This evaluation aims to answer the following research questions:

- **RQ_3:** *What is the impact on the performance when prioritization strategies are combined with the exploration strategy NAIVE or DD?*
- **RQ_4:** *What is the impact of ordering faults in ascending and in descending order?*

7.2.1.1 Studies

We address $\mathbf{RQ_3}$ and $\mathbf{RQ_4}$ by conducting a single study $\mathbf{Study_3}$.

$Study_3$: We use the largest actor system (*i.e.,* the system with 2008 messages) from our data set summarized in Table 7.3. We create 3 variants of this system by adding a single resilience defect related to guaranteed message delivery in the actors with number 5, 25, and 45. We test each of the 5 prioritization strategies separately in both ascending and descending order combined with the exploration strategy NAIVE or DD. This results in a set of 30 analyses because we have 3 varying systems and a total of 10 prioritization strategies. We repeat the analyses for each system 10 times with a timeout of 30 minutes. Caching of fault scenarios is enabled while monotonicity is disabled.

7.2.2 Results

We discuss the impact of prioritization strategies when combined with NAIVE and DD ($\mathbf{RQ}_3$) in Section 7.2.2.1, while we discuss the difference in ordering priorities ascending and descending order ($\mathbf{RQ}_4$) in Section 7.2.2.2. For each of the results, we provide box plots that visually summarize the results and provide several tables with detailed information to get a complete overview of the results. A dash (-) in the results indicates that the analysis was not able to find any resilience defect within the given time (*i.e.,* every run timed out).

7.2.2.1 Study$_3$: Impact of Prioritization Strategies

Figure 7.5 shows the number of test executions for each prioritization strategy. We prefix ascending and descending prioritization strategies with the letter *A* and *D* respectively. It only includes the results of analyses that did not time out after 30 minutes. Note that each box plot aggregates the results of the 3 variants of the system where the defect is either in actor 5, 25, or 45.

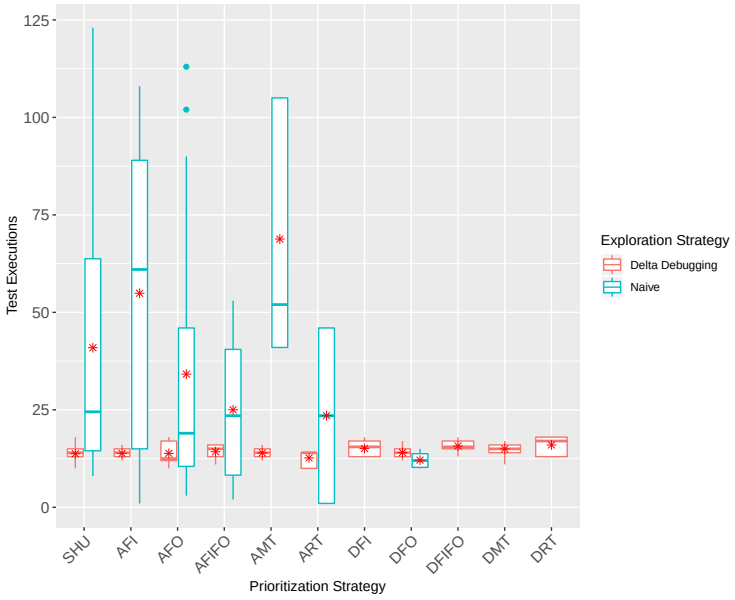

Figure 7.5: A summary of each prioritization strategy.

First, we discuss the results for the exploration strategy NAIVE. We observe that an analysis which combines NAIVE with the strategies DFI, DFIFO, DMT, or DRT is not able to find any resilience defect without timing out since there is no box plot in Figure 7.5.

This already indicates that strategies can deteriorate the performance and only cause timeouts. Analyses with the ascending variants of each prioritization strategy are able to find defects, but still time out in multiple cases. This is also clear from Table 7.4 which shows the mean and median number of test executions required to find the resilience defects, and the number of timeouts. Each column aggregates the results of the 3 variants of the system.

		SHU	AFI	AFO	AFIFO	AMT	ART	DFI	DFO	DFIFO	DMT	DRT
NAIVE	Mean	41	55	35	25	69	24	-	13	-	-	-
	Median	25	61	19	24	52	24	-	12	-	-	-
	Timeouts	14	10	11	20	25	20	30	20	30	30	30
DD	Mean	14	14	13	15	14	14	16	14	16	15	17
	Median	14	14	14	15	14	13	16	15	16	15	16
	Timeouts	0	0	0	0	0	0	0	0	0	0	0

Table 7.4: A summary of the mean and median number of required test executions of each prioritization strategy based on the analyses that did not time out.

Looking at the means and medians, we observe that the strategies AFI and AMT perform worse than the default strategy SHU. However, they have a different number of timeouts. Similarly, AFO, AFIFO, ART, and DFO have a better mean and median number of test executions than the strategy SHU, but time out more for certain defect locations. While the strategy DFO seems to achieve a small improvement on the number of test executions, it still times out in 20 of the 30 runs, and still has more timeouts than SHU. We also notice that the number of timeouts for AFI, AFIFO, ART, and DFO are a multiple of 10. Upon further inspection, the results show that the location of the resilience defect has played a role in these timeouts. This will become clear later in this chapter when we present the results for each defect location.

Next, we discuss the results for the exploration strategy DD. Table 7.4 clearly shows that none of the analyses timed out. However, looking at the results we only see a limited impact on performance. Looking at the means and medians, only AFO and ART are able to reduce the number of test executions by 1 compared to SHU (*i.e.,* an improvement of about 7%). We also observe that DRT is among the strategies that require the most number of test executions. In the worst case, this strategy increases the number of test executions with 3 (*i.e.,* a loss of about 21%). Based on Figure 7.5 and Table 7.4, we summarize the following conclusions for analyses that search for a single defect.

RQ_3 Summary

The results indicate that prioritization strategies have a limited impact when combined with the exploration strategy DD. However, the choice of a wrong prioritization strategy in combination with NAIVE can deteriorate the performance.

7.2.2.2 Study$_4$: Impact of Ascending and Descending Order

Figure 7.6 shows plots for each of the ascending prioritization strategies in isolation, while Table 7.5 shows the mean and median number of test executions of the analyses that did not time out. The results are split up for each defect location (*i.e.,* actor 5, 25, and 45) and the analyses are prefixed by the first letter of their exploration strategy (*i.e.,* N or D). From Table 7.5, observe that not every analysis with an ascending prioritization strategy finds the defects within 30 minutes when combined with NAIVE, while analyses with DD never time out. For example, while the strategy AFIFO was not able to find defects 25 and 45, the strategy AFO was able to find these but not able to find defect 5. As another example, we observe that the strategies AFI and ART are unable to find defect 45. Looking at the means of both strategies in Table 7.5, ART is able to find defects 5 and 25 respectively 19 and 2 times faster. However, the large speedup to find 5 is rather coincidental and is due to the internal working of AKKA.

		NAIVE			DD		
		N-5	N-25	N-45	D-5	D-25	D-45
SHU	Mean	123	**43**	33	16	14	13
	Median	123	18	25	15	14	13
	Timeouts	9	5	0	0	0	0
AFI	Mean	19	92	-	13	14	15
	Median	14	91	-	13	14	15
	Timeouts	0	0	10	0	0	0
AFO	Mean	-	59	**13**	18	**13**	**12**
	Median	-	59	13	18	13	12
	Timeouts	10	1	0	0	0	0
AFIFO	Mean	25	-	-	13	15	16
	Median	24	-	-	13	15	16
	Timeouts	0	10	10	0	0	0
AMT	Mean	52	-	73	15	15	14
	Median	52	-	73	15	15	14
	Timeouts	9	10	6	0	0	0
ART	Mean	**1**	46	-	**10**	14	14
	Median	1	46	-	10	14	14
	Timeouts	0	0	10	0	0	0

Table 7.5: The mean and median number of test executions required to find each resilience defect. These numbers are only based on analyses that use an ascending prioritization strategy and did not time out.

We also observe that combining ascending prioritization strategies with DD only slightly impacts performance. Additionally, the difference in performance for each defect location does not fluctuate as widely as it does when strategies are combined with NAIVE. On the one hand, combining NAIVE with the strategies ART (N-5), SHU (N-25), and AFO (N-45) require the lowest number of test executions to find each defect. On the other hand, combining DD with the strategies ART (D-5), AFO (D-25), and AFO (D-45) require the lowest number of test executions.

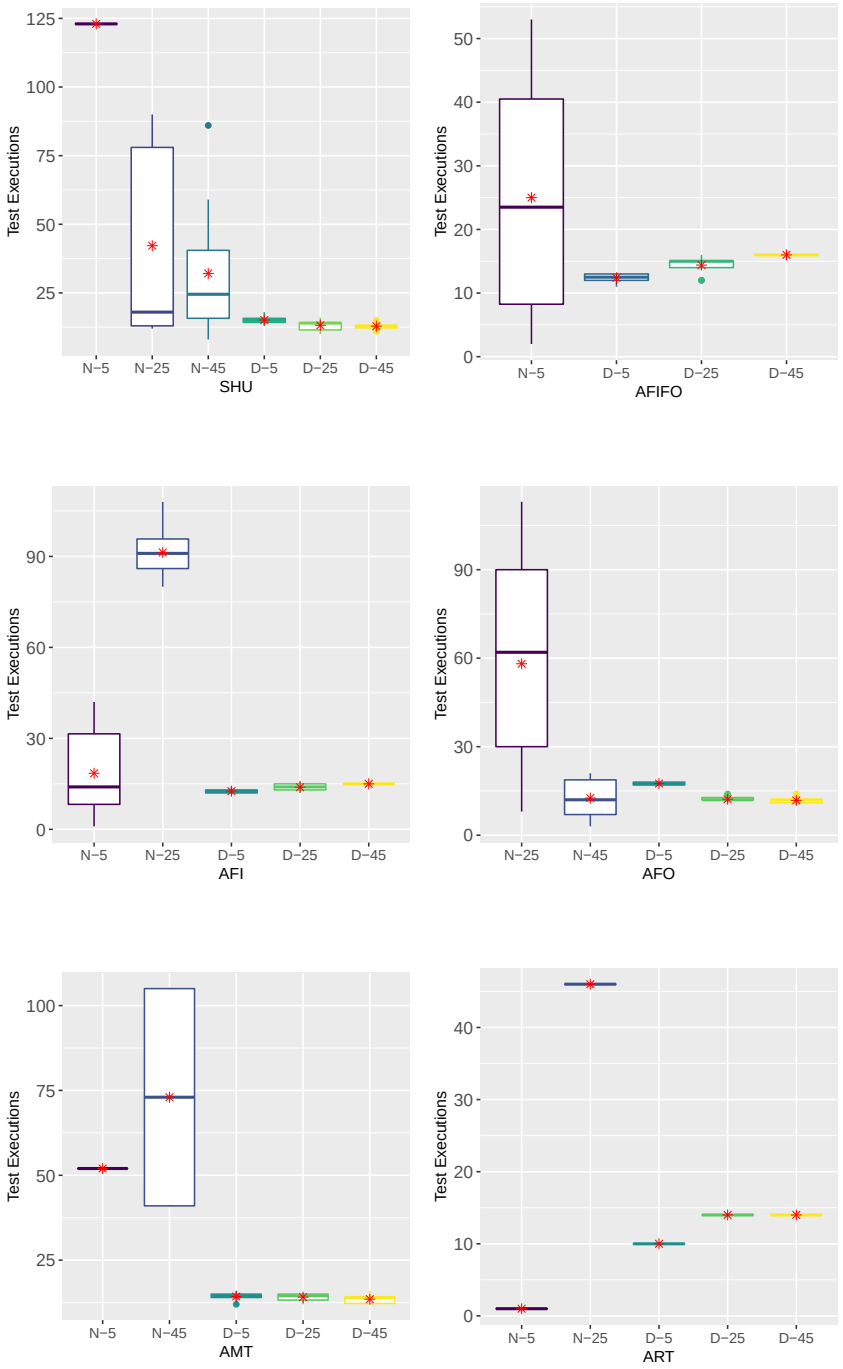

Figure 7.6: An overview of each analysis with ascending prioritization strategy.

Figure 7.7 shows plots for each of the descending prioritization strategies in isolation, while Table 7.6 shows the mean and median number of test executions of the analyses that did not time out. In contrast to the ascending strategies, we observe that many analyses time out when they use a descending variant in combination with NAIVE, while a combination with DD again never times out. That is, DFIFO, DFI, DMT, and DRT are not able to find any defect within the given time limit. Only the combination of DFO and NAIVE is able to find defect 5 without timing out and does this in a similar number of test executions as the combination of DFO and DD. Surprisingly, the strategy SHU seems to perform the best for these analyses.

		NAIVE			DD		
		N-5	N-25	N-45	D-5	D-25	D-45
SHU	Mean	123	**43**	**33**	16	**14**	**13**
	Median	123	18	25	15	14	13
	Timeouts	9	5	0	0	0	0
DFI	Mean	-	-	-	17	16	**13**
	Median	-	-	-	17	16	13
	Timeouts	10	10	10	0	0	0
DFO	Mean	**13**	-	-	**13**	16	15
	Median	12	-	-	13	16	14
	Timeouts	0	10	10	0	0	0
DFIFO	Mean	-	-	-	17	16	15
	Median	-	-	-	17	16	15
	Timeouts	10	10	10	0	0	0
DMT	Mean	-	-	-	15	15	16
	Median	-	-	-	14	15	16
	Timeouts	10	10	10	0	0	0
DRT	Mean	-	-	-	18	17	**13**
	Median	-	-	-	18	17	13
	Timeouts	10	10	10	0	0	0

Table 7.6: The mean and median number of test executions required to find each resilience defect. These numbers are only based on analyses that use a descending prioritization strategy and did not time out.

Based on the means in bold from Table 7.6, we make the following observations. On the one hand, combining NAIVE with the strategies DFO (N-5), SHU (N-25), and SHU (N-45) require the lowest number of test executions to find each defect. On the other hand, combining DD with the strategies DFO (D-5), SHU (D-25), and SHU (or DFI or DRT) (D-45) require the lowest number of test executions to find each defect.

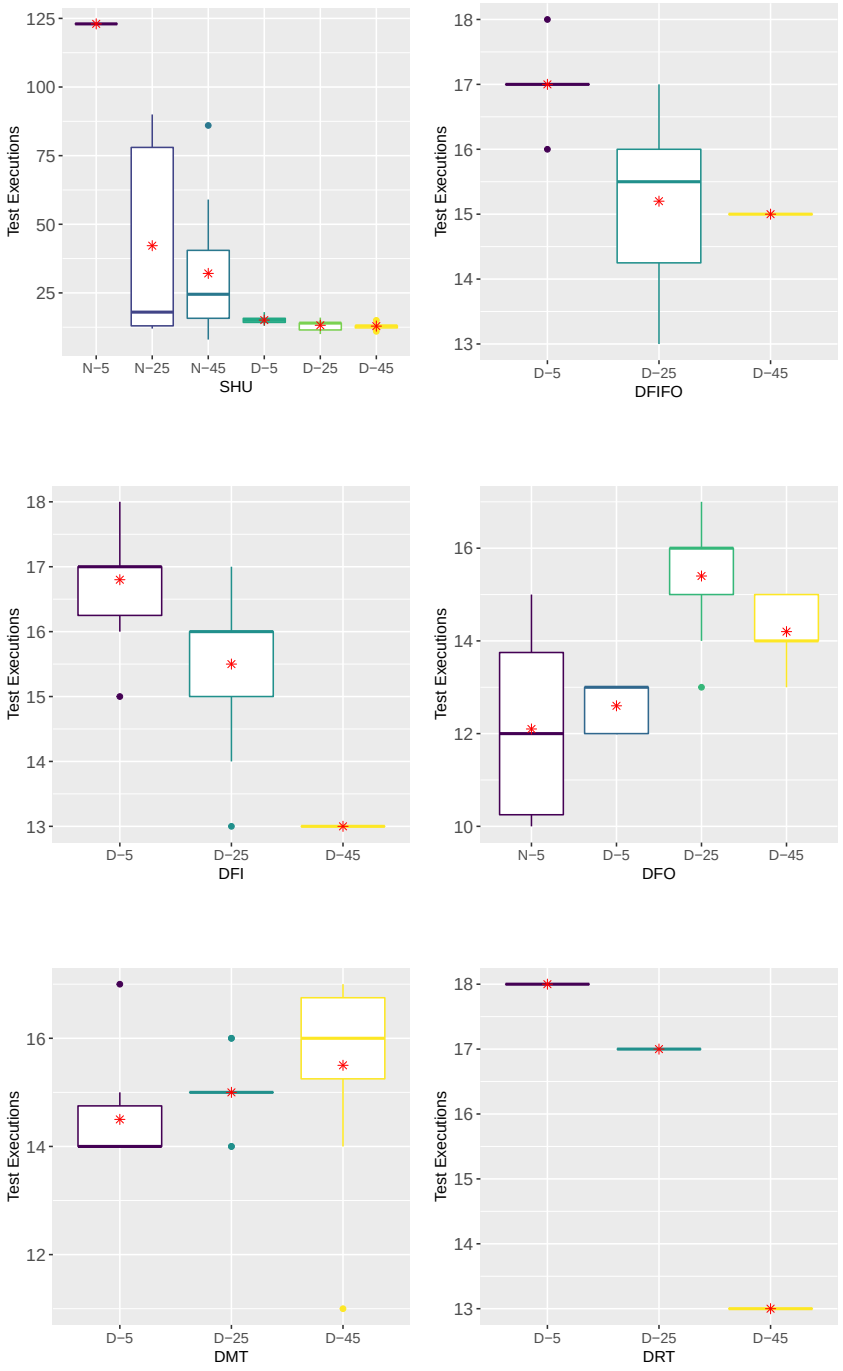

Figure 7.7: An overview of each analysis with descending prioritization strategy.

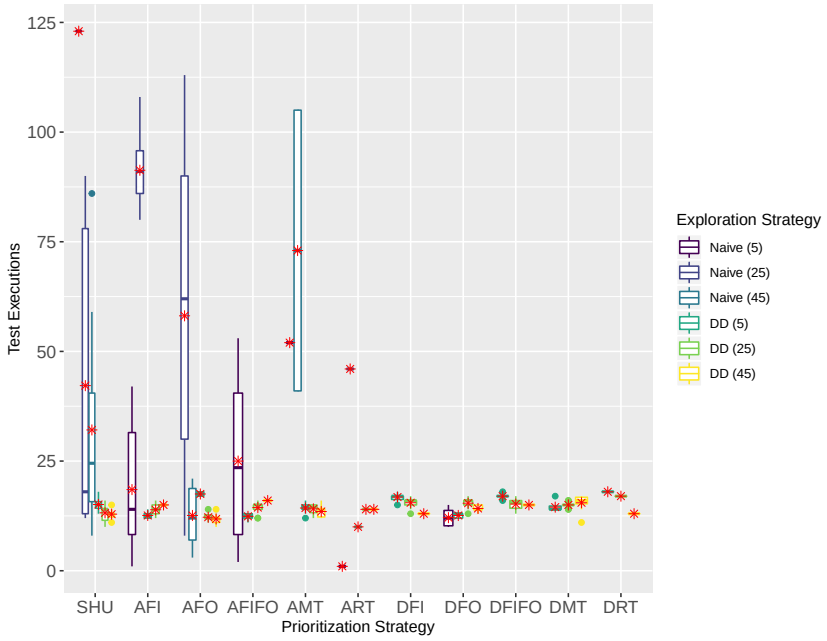

Figure 7.8: A complete overview of each prioritization strategy.

While Figure 7.5 aggregated the results of each variant, Figure 7.8 shows the results of each variant individually. It is important to note that some of these results might not be comparable because of the limited number of analyses that do not time out. For example, the combination of NAIVE and the strategy SHU had only 1 successful result (*i.e.,* 123). It could therefore be that this result is an outlier in a larger set of successful analyses. This is also the reason why we do not compute absolute performance differences between the different strategies. Nevertheless, the results show that an analysis with a naive exploration strategy does not perform well as reflected by the number of timeouts. We summarize our conclusions below.

RQ_4 Summary

There is a clear performance difference between ascending and descending variants of each prioritization strategy when combined with a naive exploration strategy. A sub-optimal prioritization strategy and the location of the resilience defect can be detrimental to the performance. In contrast, the performance difference is limited when analyses combine prioritization strategies with DD.

7.3 Discussion and Observations

We discuss several topics based on our results and observations.

1. **Combinations of Faults.** It is clear from the results that both RT-DD and RT-DD-O can achieve better performance than RT-N when searching for single faults. While our approach can already find combinations of faults, we leave an in-depth investigation of failures arising from combinations of faults for future work. Moreover, the general delta debugging algorithm is quadratic in the worst case, while a naive approach would be exponential since every possible combination has to tested. This shows that delta debugging can achieve sufficient performance even when combinations of faults have to be found.

2. **Fine-grained Priority.** Looking back at Table 7.5, we observe that several analyses did not always time out or didn't time out at all. For example, the combination of NAIVE with AMT timed out 9 times for defect 5, while it timed out 6 times for defect 45. This is due to the fact that fault tuples with the same priority are not ordered by other characteristics. For example, given that actor 1, 5, and 10 have the same priority, fault tuples δ_1, δ_5, and δ_{10} that have these actors as target are ordered in an unspecified way: the implementation might return $[\delta_1, \delta_5, \delta_{10}]$, or $[\delta_5, \delta_1, \delta_{10}]$, or any other permutation. This can result in a slightly different number of required test executions and can make the difference between timing out or not. We intentionally left the order of actors with the same priority unspecified as it remains general. That is, the prioritization strategies only use the given characteristic to compute the priority. A straightforward solution is to include other characteristics into the ordering such as actor names or message types.

3. **Determining Appropriate Strategies.** A resilience analysis that combines NAIVE with well-chosen prioritization strategies could outperform an analysis with our exploration strategy DD. For example, Table 7.5 indicates that an analysis with NAIVE and ART would only require 1 test execution to find defect 5, while it would take 10 test executions for an analysis based on DD. However, our results indicated that choosing a wrong prioritization strategy with a naive exploration strategy can deteriorate performance. Additionally, distributed systems are more likely to be complex and thus have a large fault space. Therefore, we consider our exploration strategy DD to be the best choice in general as our results show that it has a more stable performance compared to NAIVE. However, automatically detecting whether a certain combination of strategies is appropriate could further improve performance and preclude developers from deteriorating the performance with a wrong decision.

4. **Overhead of Test Execution.** Our implementation can be improved in several aspects. Currently, capturing execution traces, monitoring fault targets, and checking for faults in the fault scenario all occur on a single analysis thread. There might be more efficient ways to implement these mechanisms and thus improve the performance. Checkpointing [PBKL94] can be used as an improvement to the overhead of restarting the actor system for every test execution. There exist several utilities such as CRIU[2] that can be used to capture the state of a system on the JVM. Complex systems likely take several seconds or several minutes to start before any functionality is available which slows down the approach. Checkpointing could help as it can deploy a known state in matter of milliseconds[3].

   Another point of improvement is to automatically scale timeouts in tests as a test might fail because of the fault injection overhead. We made sure this was not the case for our analyses. However, this is a common problem in practice as timeouts can also occur when tests are run on a slower machine or network. The ScalaTest framework therefore provides a way to scale timeouts by a given factor. Thus, developers can easily adapt the timeouts of their tests and therefore mitigate aforementioned problems.

5. **Prioritizing Fault Types.** Given the larger overhead of restarting actors, we propose to test for issues related to guaranteed message delivery first and only then for issues related to actor restarts. The reasoning behind this is two-fold. First, the overhead of duplicating messages is small and thus can achieve results in a reasonable time. Second, actors can send messages during their recovery. As a result, this might cause duplicated messages on the receiver side. Additionally, the overhead of restarting actors quickly increases when more need to be started which might not fit the given testing budget for large systems. Nevertheless, several engineering efforts might be able to reduce this overhead.

7.4 Threats to Validity

While we put significant effort in attempting to collect Akka systems from Github, we were unable to find actor systems where a majority of actors implemented these resilience mechanisms. Therefore, we evaluated our approach on a set of generated actor systems and ensured that their communication topologies are representative for those of known microservice architectures (*e.g.,* Figure 7.1 from eBay) and that a varying number of messages and defect locations are incorporated to have a diverse data set. Nevertheless, we acknowledge that our results are only valid for the kind of actor systems that we generated and the seeded resilience defects that we implemented. Further evaluation requires representative systems and is left for a future avenue.

[2] https://github.com/xemul/criu

[3] https://www.jfokus.se/jfokus19-preso/Checkpointing-Java.pdf

7.5 Conclusion

This chapter presented an evaluation of our approach to resilience testing. The first study assessed the performance of different resilience analyses. The results indicate that our exploration strategy based on delta debugging with causality-based pruning is able to detect resilience defects about 5 times faster than a naive exploration strategy. The results of the second study indicate that the overhead of restarting actors is a significantly larger than the overhead of duplicating messages, but this overhead is caused by the AKKA framework itself. The third study investigated the impact of prioritization strategies on performance. The results show that the effect of prioritization strategies is limited when combined with DD, but can deteriorate the performance when combined with NAIVE. Additionally, the results show that the difference in performance between ascending and descending prioritization is larger when combined with NAIVE instead of DD. The results therefore indicate that our approach to resilience testing is an improvement to the state of the art as it lowers the amount of time it takes to detect resilience defects. Additionally, we show that our approach can be integrated with existing actor model frameworks (*i.e.,* AKKA) and testing frameworks (*i.e.,* SCALATEST) which should facilitate adoption and tool support. The next and final chapter concludes this dissertation by summarizing our discourse and discussing avenues of future work.

Chapter 8

Conclusion and Future Work

We discussed our vision of intelligent testing platforms that improve test efficiency throughout this dissertation. This vision originated from the fact that extensive test suites are available, but remain largely unexploited at current times. These tests have become available nowadays because testing frameworks have been widely integrated into the development process and software testing remains the go-to approach for developers to validate their implementation. However, contemporary testing frameworks do not provide support for amplifying these tests which leads to many of the developer's efforts to go to waste.

In this dissertation, we therefore propose two approaches that apply test amplification to exploit the full potential of tests. In particular, we presented Socrates as a static analysis approach to improving test quality and Chaokka as a dynamic analysis approach to improving resilience. Both approaches demonstrated the feasibility of our vision of intelligent testing platforms and are implemented for the Scala ecosystem to show its applicability to the industry.

This final chapter recapitulates the contents of this dissertation. Section 8.1 provides a summary of this dissertation by briefly highlighting the core of each chapter. Next, Section 8.2 restates our contributions related to our two approaches Socrates and Chaokka. Section 8.3 then discusses avenues for future work, while Section 8.4 concludes this dissertation.

8.1 Summary

Chapter 1 explained the concepts of test automation, testing frameworks, automated testing, and test amplification. Additionally, we discussed the shift towards systems with a distributed architecture and the fact that these systems are typically only tested under normal conditions. This is understandable as the increasingly complex systems and short development cycles make it infeasible for developers to always write optimal tests in terms of quality or coverage. We therefore indicated that existing test suites could be amplified to further improve the test efficiency automatically. For example, tests can be amplified to improve test quality (*e.g.,* achieving maintainable tests) or to cover additional behaviour (*e.g.,* testing the system under abnormal network conditions).

We therefore proposed our vision of intelligent testing platforms that provide support for test amplification. This dissertation therefore aimed to demonstrate that test amplification approaches can improve current testing practices. In particular, we proposed Socrates and Chaokka to demonstrate the feasibility of our vision and implemented them for the Scala ecosystem to show their applicability to the industry.

Chapter 2 presented the Scala ecosystem and two of its frameworks. We started by presenting the programming language Scala and its features. Next, we formally presented a testing system to understand the concepts of software testing and discussed these concepts in Scala's most popular testing framework ScalaTest. Finally, we introduced the actor model and its implementation in Scala's most popular actor model framework Akka, along with a discussion about the current testing practices for actor systems with the library TestKit.

Chapter 3 presented the concept of a test smell and its purpose as an indicator for test quality. Our research on this topic was motivated by the result of our extensive literature study. In particular, we observed that the majority of studies on test smells are conducted in the context of the Java ecosystem and that their corresponding tools are mostly not available. Moreover, several studies observed a negative impact of test smells on software aspects such as maintainability and defect proneness.

Therefore, we proposed Socrates — our automated and static analysis approach to detect test smells in the Scala ecosystem. Our approach is built on top of ScalaTest to show its applicability to the industry. We described that our approach is unique in its detection method because it uses both syntactic and semantic information. Additionally, we adapted 6 test smell definitions defined by Van Deursen *et al.* [VRDBDR07] and their corresponding refactoring methods to the specifics of ScalaTest. Finally, we discussed the implementation of Socrates including its usage and its extension possibilities.

Chapter 4 presented two empirical studies about test smells. First, we conducted a survey to investigate the perception of test smells by Scala developers and used Socrates to determine the diffusion of test smells in Scala systems found on Github. Our results show that developers have limited awareness of test smells, similar to what was found in existing surveys among Java developers. We also observed a lower diffusion of test smells in the Scala ecosystem than in the Java ecosystem. These results indicate that the current knowledge about test smells might not be generalizable across different ecosystems. Additionally, the results indicate that existing test suites in the Scala ecosystem have acceptable quality and could therefore be more suitable for test amplification.

Chapter 5 then presented the concepts of resilience and briefly highlighted the difficulties in implementing and testing resilience. In particular, resilience is increasingly important for systems with a distributed architecture as it exposes them to external conditions under which they may no longer remain operational. Systems that are resilient detect these conditions and recover from them through resilience mechanisms.

Next, we provided background information about techniques and approaches related to resilience testing. First, we explained Fault Injection [AALC96] as it is the foundation of most resilience testing approaches. It provides the necessary architecture to inject faults during the execution of the system in order to simulate certain conditions and events. Next, we discussed Chaos Engineering [BBDR$^+$16] as an approach to test the resilience of systems in production. Subsequently, we explained two techniques that can efficiently explore the fault space: Lineage-Driven Fault Injection [ARH15] and Delta Debugging [ZH02].

We then presented an extensive study of the state of the art to understand the current approaches to resilience testing. This study enabled us to identify their similarities and shortcomings of current approaches and categorize them accordingly. A key observation is that these approaches typically incorporate a spectrum of exploration, pruning, and prioritization strategies. We integrated these observations into our resilience testing approach.

Chapter 6 presented the difficulties of implementing and testing resilience mechanisms in the context of Akka as these resilience mechanisms are prone to several subtle implementation defects. First, developers could forget that messages could arrive multiple times because of guaranteed message delivery mechanisms. Second, developers could incorrectly persist or rehydrate the internal state of actors. Our motivation for this research results from the fact that developers currently have limited means to test the resilience of their actor systems.

Therefore, we proposed Chaokka — our automated and dynamic analysis approach to resilience testing of actor systems in the Scala ecosystem. Our approach is built on top of ScalaTest and Akka to show its applicability to the industry. Next, we provided an overview of our approach and explained our adoption of the fault injection architecture. In particular, our approach systematically injects faults during test execution to uncover resilience defects. Changes in test outcomes indicate resilience defects.

We then detailed the concepts of the trace analysis and resilience analysis. To recap, an execution trace of the test under normal conditions is captured for each test case. Next, the fault space is automatically generated by collecting the fault targets from the trace and combining them with their corresponding fault types. The resilience analysis then systematically explores this fault space by testing different fault scenarios in isolation. Each resilience analysis is composed of an exploration, a pruning, and a prioritization strategy.

To the best of our knowledge, we are the first to propose the use of delta debugging as an efficient exploration strategy in the context of actor systems and resilience testing. Additionally, we also proposed a pruning strategy based on the causality of actor events, as well as multiple prioritization strategies that use actor-specific characteristics. These strategies can further reduce the time to find resilience defects. Finally, we discussed the implementation of Chaokka including its usage and its extension possibilities.

Chapter 7 finally presented an experimental evaluation of our resilience testing approach. In particular, we execute resilience analyses with different combinations of strategies on a set of generated actor systems seeded with resilience defects. The studies indicate that resilience analyses with our exploration strategy based on delta debugging and our causality-based pruning strategy are about 5 times faster than analyses with a naive exploration strategy. Furthermore, the studies indicate that prioritization strategies have limited effect on the performance of analyses that use our exploration strategy based on delta debugging. However, it can have a detrimental effect on analyses with a naive exploration strategy resulting in uncovered resilience defects.

8.2 Contributions

This dissertation presents two key contributions: Socrates and Chaokka. We situated these approaches in the context of the Scala ecosystem (Chapter 2).

8.2.1 Socrates: Statically Detecting Test Smells

First, we identified the need for automated, efficient, and integrable approaches to detect test smells in order to improve test quality (Chapter 1). Our motivation behind this work was that current studies are limited to the context of the Java ecosystem and tools are not widely available. Our first contribution is therefore our static analysis approach to detecting test smells (Chapter 3 and Chapter 4):

- An automated and static analysis approach to find test smells.
- The implementation of our approach for the testing framework ScalaTest in the open-source tool Socrates.
- An empirical study of test smells in 164 open-source Scala systems.
- A survey on the awareness of 14 Scala developers about test smells.

8.2.2 Chaokka: Dynamically Testing Resilience

Second, we identified the need for automated, efficient, and integrable approaches to test the resilience of actor systems (Chapter 1). Our motivation behind this work is that state-of-the-art approaches to resilience testing are limited (Chapter 5) and do not exist for distributed actor systems. Our second contribution is therefore our dynamic analysis approach to resilience testing (Chapter 6 and Chapter 7).

- An automated and dynamic analysis approach to finding resilience defects in actor systems.
- The implementation of our approach for the actor framework Akka and the testing framework ScalaTest in the open-source tool Chaokka.
- An evaluation that demonstrates the impact and efficiency of our proposed exploration, pruning, and prioritization strategies.

We refer the reader back to the detailed conclusions about test smells and resilience testing in Chapter 4 and Chapter 7 respectively.

8.3 Future Work

We envision future research in both aspects of this dissertation and discuss several of these avenues in more detail below. With respect to test smells, we envision 5 avenues for future work:

1. **Different Ecosystems.** Our study indicated that the diffusion of test smells is lower compared to studies in the context of the Java ecosystem. However, it remains to be seen whether this is also true for other ecosystems. We therefore envision replications of these studies to get a broader view of how test smells are diffused across different ecosystems. While the implementation of Socrates is specific to the Scala ecosystem, it provides a solid foundation and architecture that can be adapted to detect test smells in other ecosystems. Additionally, we only transposed existing test smell definitions to the Scala ecosystem, but did not propose any new ones. We leave test smells specific to ScalaTest or Akka for future work.
2. **Dynamic Detection.** Some of the test smells were not detected because they require a dynamic analysis. For example, Resource Optimism would require the execution of the system to determine whether the resource exists. Similarly, Test Run War would require the tests to be executed in every possible order. While this smell could indeed be determined statically, it would require the computation of complex control and data flows. Therefore, a dynamic analysis is preferred. Ideally, Socrates would be extended with such support to enlarge the set of test smells it can detect.

3. **Test Smells and Metrics.** Several studies have observed a relationship between test smells and software aspects such as maintainability, defect proneness, and code smells. We believe that these are also present in different ecosystems. We therefore envision a replication of these studies for the SCALA ecosystem. These studies might also provide insights about which tests are the most suitable input for test amplification approaches.

4. **Severity of Test Smells.** Existing surveys show that developers are not always aware of test smells in their tests. However, this might also be because developers do not consider test smells as a severe issue. Unfortunately, the actual reason for the lack of awareness is typically neglected in surveys and thus might require more investigation in the future. For example, [SSO+20] proposed severity thresholds for test smells in order to urge developers to take action.

5. **Automated Refactoring.** SOCRATES only indicates test smells in the code, but does not provide support for automated refactorings that eliminate the smell from their test. We therefore envision such a feature so that developers no longer need to modify the test code themselves.

With respect to resilience testing, we envision 4 avenues for future work:

1. **Non-deterministic Actor Systems.** CHAOKKA only supports systems and tests that are deterministic in nature. Non-determinism can occur due to non-deterministic scheduling, non-deterministic data, or both. This might give different outcomes for subsequent runs and cause the delta debugging algorithm to produce incorrect fault scenarios as it expects the test function to be deterministic. Nevertheless, we identified several approaches [AMB+18,LPV19] in the work of Lopez *et al.* [LMBM18] that can be used to deterministically record and replay the scheduling so that actor systems have the same behaviour across multiple executions. CHAOKKA could additionally detect differences in subsequent execution traces to warn developers. Other sources of non-determinism (*e.g.,* random message payloads) could also be controlled by the tester which is commonly needed in dynamic symbolic execution [GKS05].

2. **Additional Resilience Defects.** While CHAOKKA supports the detection of two types of resilience defects, there might be other resilience mechanisms with their corresponding defects. For example, circuit breakers[1] provide a default response when they are in an open state. Such an open state only occurs under abnormal conditions and might not be understood by requesting services. Similarly, actors with supervision strategies[2] execute a certain recovery strategy upon abnormal conditions that occurred in an actor that they are supervising. Of course, developers might implement such an system-specific recovery strategy incorrectly.

[1] https://doc.akka.io/docs/akka/current/common/circuitbreaker.html

[2] https://doc.akka.io/docs/akka/current/typed/fault-tolerance.html

3. **Feedback-directed Resilience Testing.** CHAOKKA determines fault targets solely through the first test execution trace which is obtained by running the test under normal conditions. However, actor systems might change their execution when certain faults are injected. Therefore, we could incrementally build a model during the actual test execution that represents the system's execution so far (*e.g.,* which faults are injected into which actors). Additionally, we could track the effect of faults on certain fault targets by combining information from multiple tests. The resulting model could improve the efficiency of our approach to resilience testing. For example, certain fault tuples could be pruned from the fault space when they have already been thoroughly injected into other tests.

4. **Extended Evaluation.** While we invested significant effort in identifying open-source AKKA systems available on GITHUB, we were not able to find representative systems that extensively use persistent actors and at-least-once message delivery. Moreover, even if we did find such systems, they may not contain resilience defects and would still require manual effort to seed them with defects. Correspondingly, we evaluated our approach on a set of generated actor systems seeded with resilience defects. It therefore remains to be seen how our approach can detect resilience defects in open-source or proprietary AKKA systems.

8.4 Concluding Remarks

We believe that this dissertation and its proposed approaches demonstrate the feasibility of our vision of intelligent testing platforms. These platforms support current software testing practices and aim to improve test efficiency by leveraging existing test suites. To conclude, we summarize our two approaches.

First, we proposed SOCRATES as a static analysis approach to improving the quality of existing test suites. Our motivation behind this approach is the presence of test smells in the JAVA ecosystem, their negative impact of test smells on multiple software aspects, and the limited tool support for detecting smells in other ecosystems. Our automated approach therefore detects 6 test smells in an automated way for tests written with SCALATEST. Our empirical study of 164 SCALA systems indicated a low diffusion of test smells which shows that tests in SCALA have an acceptable quality. SOCRATES is one demonstrator of our vision and shows the possibilities of test amplification.

Second, we proposed CHAOKKA as a dynamic analysis approach to improve the resilience of actor systems. Our motivation behind this work is the increasing importance of resilience for systems with a distributed architecture and the limited tool support for finding resilience defects in an efficient manner. Our automated approach therefore leverages fault injection to find 2 types of resilience defects in actor systems written with AKKA. Moreover, our strategies based on delta debugging and causality are able to explore the fault space more efficiently.

Our experimental study of generated actor systems demonstrated that an analysis with our exploration and pruning strategy can find resilience defects up to 5 times faster compared to a naive analysis. Again, CHAOKKA supports the feasibility of our vision of testing platforms that amplify existing test suites and hereby increase test efficiency.

Bibliography

[AALC96] Dimiter Avresky, Jean Arlat, Jean-Claude Laprie, and Yves Crouzet. Fault injection for formal testing of fault tolerance. *Transactions on Reliability*, 1996.

[AAS+16] Peter Alvaro, Kolton Andrus, Chris Sanden, Casey Rosenthal, Ali Basiri, and Lorin Hochstein. Automating failure testing research at internet scale. In *Proceedings of the 7th ACM Symposium on Cloud Computing*, 2016.

[Agh85] Gul A Agha. Actors: A model of concurrent computation in distributed systems. Technical report, 1985.

[ALR+01] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, et al. *Fundamental concepts of dependability.* 2001.

[ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Transactions on Dependable and Secure Computing*, 2004.

[AMB+18] Dominik Aumayr, Stefan Marr, Clément Béra, Elisa Gonzalez Boix, and Hanspeter Mössenböck. Efficient and deterministic record & replay for actor languages. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, 2018.

[AMC+10] Peter Alvaro, William R Marczak, Neil Conway, Joseph M Hellerstein, David Maier, and Russell Sears. Dedalus: Datalog in time and space. In *International Datalog 2.0 Workshop*, 2010.

[AMM15] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. Systematic execution of android test suites in adverse conditions. In *Proceedings of the 24th International Symposium on Software Testing and Analysis*, 2015.

[ARD19] Mehrdad Abdi, Henrique Rocha, and Serge Demeyer. Test amplification in the pharo smalltalk ecosystem. In *Proceedings of the 14th International Workshop on Smalltalk Technologies*, 2019.

[ARH15] Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. Lineage-driven fault injection. In *Proceedings of the International Conference on Management of Data*, 2015.

[BBDR+16] Ali Basiri, Niosha Behnam, Ruud De Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, and Casey Rosenthal. Chaos engineering. *IEEE Software*, 2016.

[Bec03] Kent Beck. *Test-driven Development: By Example*. Addison-Wesley Professional, 2003.

[Bei03] Boris Beizer. *Software Testing Techniques*. 2003.

[BFLT06] Benoit Baudry, Franck Fleurey, and Yves Le Traon. Improving test suites for efficient fault localization. In *Proceedings of the 28th International Conference on Software Engineering*, 2006.

[BGP+17] Moritz Beller, Georgios Gousios, Annibale Panichella, Sebastian Proksch, Sven Amann, and Andy Zaidman. Developer testing in the ide: Patterns, beliefs, and behavior. *Transactions on Software Engineering*, 2017.

[BHM+14] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *Transactions on Software Engineering*, 2014.

[BHP+17] David Bowes, Tracy Hall, Jean Petric, Thomas Shippey, and Burak Turhan. How good are my tests? In *Proceedings of the 8th Workshop on Emerging Trends in Software Metrics*, 2017.

[BK14] Peter Bailis and Kyle Kingsbury. The network is reliable. *Queue*, 2014.

[BKWC01] Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. Why and where: A characterization of data provenance. In *International conference on database theory*. Springer, 2001.

[BLB10] Robert Brummayer, Florian Lonsing, and Armin Biere. Automated testing and debugging of SAT and QBF solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, 2010.

[BLS94] Niels Boyen, Carine Lucas, and Patrick Steyaert. Generalized mixin-based inheritance to support multiple inheritance. Technical report, vub-prog-tr-94-12, Vrije Universiteit Brussel, 1994.

[BQO+12] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and David Binkley. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *Proceedings of the 28th International Conference on Software Maintenance*, 2012.

[BQO+15] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. Are test smells really harmful? an empirical study. *Empirical Software Engineering*, 2015.

[CDSL+19] Domenico Cotroneo, Luigi De Simone, Pietro Liguori, Roberto Natella, and Nematollah Bidokhti. How bad can a bug get? an empirical analysis of software failures in the openstack cloud computing platform. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.

[CSM15] Benoit Cornu, Lionel Seinturier, and Martin Monperrus. Exception handling analysis and transformation using fault injection: Study of resilience against unanticipated exceptions. *Information and Software Technology*, 2015.

[CTBV15] Michael Alan Chang, Bredan Tschaen, Theophilus Benson, and Laurent Vanbever. Chaos monkey: Increasing SDN reliability through systematic network destruction. In *ACM Computer Communication Review*, 2015.

[CWC+19] Chengxu Cui, Guoquan Wu, Wei Chen, Jiaxing Zhu, and Jun Wei. Feedback-based, automated failure testing of microservice-based applications. *Preprint arXiv:1908.06466*, 2019.

[CWW00] Yingwei Cui, Jennifer Widom, and Janet L Wiener. Tracing the lineage of view data in a warehousing environment. *Transactions on Database Systems*, 2000.

[DBDNDR19a] Jonas De Bleser, Dario Di Nucci, and Coen De Roover. Assessing diffusion and perception of test smells in scala projects. In *Proceedings of the 16th International Conference on Mining Software Repositories*, 2019.

[DBDNDR19b] Jonas De Bleser, Dario Di Nucci, and Coen De Roover. Socrates: Scala radar for test smells. In *Proceedings of the 10th Symposium on Scala*, 2019.

[DBDNDR20] Jonas De Bleser, Dario Di Nucci, and Coen De Roover. A delta-debugging approach to assessing the resilience of actor programs through run-time test perturbations. In *Proceedings of the 1st International Conference on Automation of Software Test*, 2020.

[DBSNDR17] Jonas De Bleser, Quentin Stiévenart, Jens Nicolay, and Coen De Roover. Static taint analysis of event-driven scheme programs. In *10th European Lisp Symposium*, 2017.

[Dij78] Edsger W Dijkstra. Two starvation free solutions to a general exclusion problem. *Note EWD*, 1978.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[DMB11] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 2011.

[DVPY+19] Benjamin Danglot, Oscar Vera-Perez, Zhongxing Yu, Andy Zaidman, Martin Monperrus, and Benoit Baudry. A snowballing literature study on test amplification. *Journal of Systems and Software*, 2019.

[ECS15] Felipe Ebert, Fernando Castor, and Alexander Serebrenik. An exploratory study on exception handling bugs in java programs. *Journal of Systems and Software*, 2015.

[EGM97] Thomas Eiter, Georg Gottlob, and Heikki Mannila. Disjunctive datalog. *Transactions on Database Systems*, 1997.

[FA11] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 13th European Conference on Foundations of Software Engineering*, 2011.

[Fid88] Colin J Fidge. Partial orders for parallel debugging. In *ACM Sigplan Notices*, 1988.

[Fow05] Martin Fowler. Event sourcing. 2005.

[Fow18] Martin Fowler. *Refactoring: improving the design of existing code*. 2018.

[GBJG15] Malay Ganai, Gogul Balakrishnan, Pallavi Joshi, and Aarti Gupta. Setsudo: Pertubation-based testing framework for scalable distributed systems, 2015. US Patent App. 14/217,566.

[GDJ+11] 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. Fate and destini: A framework for cloud recovery testing. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation*, 2011.

[Ghi19] Yonas Ghidei. Lineage-driven fault injection for actor-based programs, 2019.

[GK18] Vahid Garousi and Barış Küçük. Smells in software test code: A survey of knowledge in industry and academia. *Journal of systems and software*, 138:52–81, 2018.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2005.

[GvS13] Michaela Greiler, Arie van Deursen, and Margaret-Anne Storey. Automated detection of test fixture strategies and smells. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, 2013.

[GZ13] Vahid Garousi and Junji Zhi. A survey of software testing practices in canada. *Journal of Systems and Software*, 2013.

[GZvS13] Michaela Greiler, Andy Zaidman, Arie van Deursen, and Margaret-Anne Storey. Strategies for avoiding text fixture smells during software evolution. In *Proceedings of the 10th Conference on Mining Software Repositories*, 2013.

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Conference on Artificial Intelligence*, 1973.

[Hel07] Pat Helland. Life beyond distributed transactions: an apostate's opinion. In *Proceedings of the 3rd Conference on Innovative DataSystems Research*, 2007.

[Hel12] Pat Helland. Idempotence is not a medical condition. *Queue*, 2012.

[HH02] James A Highsmith and Jim Highsmith. *Agile software development ecosystems*. 2002.

[HJ01] Mark Harman and Bryan F Jones. Search-based software engineering. *Information and software Technology*, 2001.

[HJZ15] Mark Harman, Yue Jia, and Yuanyuan Zhang. Achievements, open problems and challenges for search based software testing. In *Proceedings of the 8th International Conference on Software Testing, Verification and Validation*, 2015.

[HK16] Renáta Hodován and Ákos Kiss. Practical improvements to the minimizing delta debugging algorithm. In *Proceedings of the 11th International Conference on Software Engineering and Applications*, 2016.

[HRJ+16] Victor Heorhiadi, Shriram Rajagopalan, Hani Jamjoom, Michael K Reiter, and Vyas Sekar. Gremlin: Systematic resilience testing of microservices. In *Proceedings of the 36th International Conference on Distributed Computing Systems*, 2016.

[HROE13] Waldemar Hummer, Florian Rosenberg, Fábio Oliveira, and Tamar Eilam. Testing idempotence for infrastructure as code. In *International Conference on Distributed Systems Platforms and Open Distributed Processing*, 2013.

[IS14] Shams M Imam and Vivek Sarkar. 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*, 2014.

[JGS11] Pallavi Joshi, Haryadi S Gunawi, and Koushik Sen. Prefail: A programmable tool for multiple-failure injection. In *Proceedings of the International Conference on Object-oriented Programming Systems, Languages, and Applications*, 2011.

[KBLJ13] Pavneet Singh Kochhar, Tegawendé F Bissyandé, David Lo, and Lingxiao Jiang. Adoption of software testing in open source projects–a preliminary study on 50,000 projects. In *Proceedings of the 17th European Conference on Software Maintenance and Reengineering*, 2013.

[KHA17] Roland Kuhn, Brian Hanafee, and Jamie Allen. *Reactive design patterns*. 2017.

[Knu71] Donald E. Knuth. Optimum binary search trees. *Acta informatica*, 1971.

[Lam19] Leslie Lamport. The part-time parliament. In *Concurrency: the Works of Leslie Lamport*. 2019.

[Lap08] Jean-Claude Laprie. From dependability to resilience. In *Proceedings of the 38th International Conference On Dependable Systems and Networks*, 2008.

[LF14] James Lewis and Martin Fowler. Microservices. 2014.

[LKMA10] Steven Lauterburg, Rajesh K Karmani, Darko Marinov, and Gul Agha. Evaluating ordering heuristics for dynamic partial-order reduction techniques. In *Proceedings of the 12th International*

Conference on Fundamental Approaches to Software Engineering, 2010.

[LLLG16] Tanakorn Leesatapornwongsa, Jeffrey F Lukman, Shan Lu, and Haryadi S Gunawi. Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. *ACM SIGPLAN Notices*, 2016.

[LMBM18] 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*. 2018.

[LPV19] Ivan Lanese, Adrián Palacios, and Germán Vidal. Causal-consistent replay debugging for message passing programs. In *Proceedings of the 39th International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, 2019.

[Mes07] Gerard Meszaros. *xUnit test patterns: Refactoring test code*. 2007.

[MLA+17] Stefan Marr, Carmen Torres Lopez, Dominik Aumayr, Elisa Gonzalez Boix, and Hanspeter Mössenböck. Kόμπος: A platform for debugging complex concurrent applications. In *Proceedings of the 1st International Conference on the Art, Science and Engineering of Programming*, 2017.

[Mor16] Kief Morris. *Infrastructure as code: managing servers in the cloud*. 2016.

[MS06] Ghassan Misherghi and Zhendong Su. Hdd: hierarchical delta debugging. In *Proceedings of the 28th international conference on Software engineering*, 2006.

[MS20] Roger Magoulas and Steve Swoyer. Cloud adoption in 2020, 2020.

[MSB11] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. 2011.

[Nak15] Heather Nakama. Inside Azure Search: Chaos engineering. 2015.

[Nat11] Roberto Natella. *Achieving Representative Faultloads in Software Fault Injection*. PhD thesis, 2011.

[NCM16] Roberto Natella, Domenico Cotroneo, and Henrique S Madeira. Assessing dependability with software fault injection: A survey. *ACM Computing Surveys*, 2016.

[NW16] Michael Nash and Wade Waldron. *Applied Akka Patterns: A Hands-On Guide to Designing Distributed Applications*. 2016.

[OAC+07] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. The Scala language specification, 2007.

[OZRA19] Lennart Oldenburg, Xiangfeng Zhu, Kamala Ramasubramanian, and Peter Alvaro. Fixed it for you: Protocol repair using lineage graphs. In *The 9th Biennial Conference on Innovative Data Systems Research*, 2019.

[PBKL94] James S Plank, Micah Beck, Gerry Kingsley, and Kai Li. *Libckpt: Transparent checkpointing under unix*. 1994.

[PDBM+19] Yunior Pacheco, Jonas De Bleser, Tim Molderez, Dario Di Nucci, Wolfgang De Meuter, and Coen De Roover. Mining scala framework extensions for recommendation patterns. In *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering*, 2019.

[PDNP+16] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Rocco Oliveto, and Andrea De Lucia. On the diffusion of test smells in automatically generated test code: An empirical study. In *Proceedings of the 9th International Workshop on Search-Based Software Testing*, 2016.

[Per18] Anthony Shehan Ayam Peruma. What the smell? an empirical investigation on the distribution and severity of test smells in open source android applications. 2018.

[PZ17] Fabio Palomba and Andy Zaidman. Does refactoring of test smells induce fixing flaky tests? In *Proceedings of the 11th International Conference on Software Maintenance and Evolution*, 2017.

[PZ19] Fabio Palomba and Andy Zaidman. The smell of fear: On the relation between test smells and flaky tests. *Empirical Software Engineering*, 2019.

[QR11] Xiao Qu and Brian Robinson. A case study of concolic testing tools and their limitations. In *2011 International Symposium on Empirical Software Engineering and Measurement*, 2011.

[RCD+04] Martin C Rinard, Cristian Cadar, Daniel Dumitran, Daniel M Roy, Tudor Leu, and William S Beebee. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, 2004.

[RE19] Ashutosh Raina and Ramprasad Ellupuru. Madaari: Ordering for the monkeys. 2019.

[RKAL12] Jesse Robbins, Kripa Krishnan, John Allspaw, and Thomas A Limoncelli. Resilience engineering: learning to embrace failure. *Queue*, 2012.

[RV13] Ganesan Ramalingam and Kapil Vaswani. Fault tolerance via idempotence. In *Proceedings of the 40th Symposium on Principles of Programming Languages*, 2013.

[SBN+16] Colin Scott, Vjekoslav Brajkovic, George Necula, Arvind Krishnamurthy, and Scott Shenker. Minimizing faulty executions of distributed systems. In *Proceedings of the 13th Symposium on Networked Systems Design and Implementation*, 2016.

[Sch05] Rainer Schuler. An algorithm for the satisfiability problem of formulas in conjunctive normal form. *Journal of Algorithms*, 2005.

[SG03] Alper Sen and Vijay K Garg. Detecting temporal logic predicates in distributed programs using computation slicing. In *International Conference On Principles Of Distributed Systems*, 2003.

[SGA07] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. 2007.

[SPH15] Gianluca Stivan, Andrea Peruffo, and Philipp Haller. Akka.js: towards a portable actor runtime environment. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, 2015.

[SPZ+18] Davide Spadini, Fabio Palomba, Andy Zaidman, Magiel Bruntink, and Alberto Bacchelli. On the relation of test smells to software code quality. In *Proceedings of the International Conference on Software Maintenance and Evolution*, 2018.

[SRA03] Koushik Sen, Grigore Rosu, and Gul Agha. Runtime safety analysis of multithreaded programs. *ACM SIGSOFT Software Engineering Notes*, 2003.

[SSO+20] Davide Spadini, Martin Schvarcbacher, Ana-Maria Oprescu, Magiel Bruntink, and Alberto Bacchelli. Investigating severity thresholds for test smells. In *Proceedings of the 17th International Conference on Mining Software Repositories, MSR âĂŹ20*, 2020.

[Sve14] Yevgeniy Sverdlik. Facebook turned off entire data center to test resiliency. *Data Center Knowledge*, 2014.

[SW17] Kazuhiro Shibanai and Takuo Watanabe. Actoverse: a reversible debugger for actors. In *Proceedings of the 7th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, 2017.

[SWH11] Matt Staats, Michael W Whalen, and Mats PE Heimdahl. Programs, tests, and oracles: the foundations of testing revisited. In *Proceedings of the 33rd International Conference on Software Engineering*, 2011.

[SZM+19] Jesper Simonsson, Long Zhang, Brice Morin, Benoit Baudry, and Martin Monperrus. Observability and chaos engineering on system calls for containerized applications in docker. *Preprint arXiv:1907.13039*, 2019.

[TDJ13] Samira Tasharofi, Peter Dinges, and Ralph E Johnson. Why do scala developers mix the actor model with other concurrency models? In *Proceedings of the European Conference on Object-Oriented Programming*, 2013.

[TKG09] Kishor S Trivedi, Dong Seong Kim, and Rahul Ghosh. Resilience in computer systems and networks. In *2009 IEEE/ACM International Conference on Computer-Aided Design-Digest of Technical Papers*, pages 74–77. IEEE, 2009.

[TMX+11] Suresh Thummalapenta, Madhuri R Marri, Tao Xie, Nikolai Tillmann, and Jonathan De Halleux. Retrofitting unit tests for parameterized unit testing. In *International Conference on Fundamental Approaches to Software Engineering*, 2011.

[TPB+16] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. An empirical investigation into the nature of test smells. In *Proceedings of the 31st International Conference on Automated Software Engineering*, 2016.

[TPLJ13] Samira Tasharofi, Michael Pradel, Yu Lin, and Ralph Johnson. Bita: Coverage-guided, automatic testing of actor programs. In *Proceedings of the 28th International Conference on Automated Software Engineering*, 2013.

[Ver15] Vaughn Vernon. *Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka.* 2015.

[vMBK01] Arie van Deursen, Leon Moonen, Alex Bergh, and Gerard Kok. Refactoring test code. In *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering*, 2001.

[VRDBDR07] Bart Van Rompaey, Bart Du Bois, Serge Demeyer, and Matthias Rieger. On the detection of test smells: A metrics-based approach for general fixture and eager test. *Transactions on Software Engineering*, 2007.

[Wei84] Mark Weiser. Program slicing. *Transactions on Software Engineering*, 1984.

[Xie06] Tao Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *European Conference on Object-Oriented Programming*, 2006.

[XR09] Zhihong Xu and Gregg Rothermel. Directed test suite augmentation. In *Proceedings of the 16th Asia-Pacific Software Engineering Conference*, 2009.

[YH12] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: a survey. *Software testing, Verification and Reliability*, 2012.

[ZAV+04] Haissam Ziade, Rafic A Ayoubi, Raoul Velazco, et al. A survey on fault injection techniques. *The International Arab Journal of Information Technology*, 2004.

[ZE14] Pingyu Zhang and Sebastian Elbaum. Amplifying tests to validate exception handling code: An extended study in the mobile application domain. *Transactions on Software Engineering and Methodology*, 2014.

[Zel99] Andreas Zeller. Yesterday, my program worked. today, it does not. why? In *Software Engineering Notes*, 1999.

[ZH02] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *Transactions on Software Engineering*, 2002.

[ZH18] Xin Zhao and Philipp Haller. Observable atomic consistency for CvRDTs. *Journal of Computer Documentation*, 2018.

[ZM19] Long Zhang and Martin Monperrus. Tripleagent: Monitoring, perturbation and failure-obliviousness for automated resilience improvement in java applications. In *International Symposium on Software Reliability Engineering*, 2019.

[ZMH+19] Long Zhang, Brice Morin, Philipp Haller, Benoit Baudry, and Martin Monperrus. A chaos engineering system for live analysis and falsification of exception-handling in the JVM. *Transactions on Software Engineering*, 2019.

[ZPX+18] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Wenhai Li, Chao Ji, and Dan Ding. Delta debugging microservice systems. In *Proceedings of the 33rd International Conference on Automated Software Engineering*, 2018.

[ZPX+19] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. Delta debugging microservice systems with parallel optimization. *Transactions on Services Computing*, 2019.