Lineage-Driven Fault Injection for Actor-based Programs

YONAS GHIDEI
Lineage-driven Fault Injection for Actor-based Programs
Abstract

Lineage-driven fault injection (LDFI) is an approach for finding bugs or faults in distributed systems. Its current state-of-the-art implementation, Molly, takes programs written in the logic programming language Dedalus as its input. The problem is that Dedalus is a small and obscure language, used to serve as a proof of concept for Molly. Thus, the objective of this thesis is to extend and adapt Molly to a general-purpose, object-oriented language where distributed programs are written using the actor-based framework Akka. This thesis presents a novel concept for employing lineage-driven fault injection for actor-based programs in addition to implementing said concept to analyze existing Akka programs. The results show that the lineage-driven fault injector for Akka programs, ldfi-akka, is capable of successfully pinpointing the weaknesses of the programs that can be analyzed in a feasible amount of time. However, ldfi-akka struggles to analyze larger and complex programs as the underlying SAT-solver used is overwhelmed. The correctness of the analysis made by ldfi-akka is partially based on the subject programs’ ability to a) be rewritten in such a way that logging can be added and b) exhibit deterministic behavior across multiple runs. Conclusively, this study presents a novel approach to employ lineage-driven fault injection on actor-based programs and ldfi-akka, an implementation of LDFI on Akka programs.
Referat

Acknowledgements

I would like to thank my supervisors Martin Monperrus and Philipp Haller for their enthusiastic involvement in this thesis. The guidance I have received from them has been very helpful. I would especially like to thank Philipp Haller for all of the invaluable feedback and insight he has given me throughout the project. Lastly, I would like to thank Vizrt and Robert Olsson for their support.
## Contents

### Glossary

1 Introduction 3
   1.1 Background ................................................................. 3
   1.2 Problem ................................................................. 4
   1.3 Objective ................................................................. 4
   1.4 Research Questions ...................................................... 4
   1.5 Purpose ................................................................. 5
   1.6 Contributions .......................................................... 5
   1.7 Sustainability, Ethics and Societal Aspects ......................... 5
   1.8 Outline ................................................................. 6

2 Theoretical Background 7
   2.1 Logic Programming ...................................................... 7
   2.2 Dedalus ................................................................. 8
       2.2.1 Logical Time ...................................................... 8
       2.2.2 Asynchrony ...................................................... 8
   2.3 Lineage-driven Fault Injection ........................................ 9
       2.3.1 Data Lineage ...................................................... 9
       2.3.2 Failure Specification ............................................ 9
       2.3.3 Sweep .......................................................... 10
       2.3.4 Boolean Formulas ............................................... 11
   2.4 Molly ................................................................. 11
       2.4.1 Simple-Deliv in Dedalus ....................................... 12
       2.4.2 Injecting Failures ............................................... 13
   2.5 Akka ................................................................. 14
       2.5.1 Simple Concept ................................................... 14
       2.5.2 Checker Counter .................................................. 16
   2.6 Scala’s Design by Contract ............................................ 17

3 Related Work 19
   3.1 Model Checking ........................................................ 19
   3.2 Fault Injection ......................................................... 20
### CONTENTS

3.2.1 Simian Army ............................................. 20  
3.2.2 Failure Injection Testing ..................................... 20  
3.2.3 LDFI at Netflix ........................................... 20  

4 Research Method .................................................. 23  
4.1 Research strategy ............................................. 23  
4.2 Method ............................................................ 23  
4.2.1 Research type ............................................... 23  
4.2.2 Research approach ......................................... 24  
4.3 Research phases ............................................... 24  
4.3.1 Literature study ............................................. 24  
4.3.2 Practical study ............................................... 25  
4.4 Validity .......................................................... 25  

5 LDFI for Actor-based Programs ........................................ 27  
5.1 Boolean Encoding ............................................. 27  
5.1.1 Formulas as Paths ......................................... 27  
5.1.2 Minimal Solutions .......................................... 29  
5.2 Logical Time ................................................... 30  
5.3 Failure Specification ........................................... 32  
5.4 Evaluator ......................................................... 34  
5.4.1 Backward Step .............................................. 34  
5.4.2 Forward Step ................................................ 35  
5.4.3 Concrete Evaluator ......................................... 36  

6 LDFI for Akka .......................................................... 39  
6.1 Simple-Deliv in Akka .......................................... 39  
6.2 Data Lineage in Akka .......................................... 41  
6.2.1 Logging Configuration ...................................... 42  
6.2.2 Actor Logging .............................................. 42  
6.3 Parsing ............................................................ 43  
6.4 Controller ........................................................ 44  
6.5 Program Rewrite ............................................... 44  
6.5.1 Embedding Logging ......................................... 46  
6.5.2 Incorporation of Fault Injections ......................... 47  

7 Results ................................................................. 51  
7.1 Simple-Deliv .................................................... 51  
7.2 Retry-Deliv ...................................................... 53  
7.3 Hello World ..................................................... 53  
7.4 Persistence ....................................................... 55  
7.5 Dining Philosophers ............................................ 58  
7.6 Observable Atomic Consistency Protocol ...................... 60
CONTENTS

7.7 Evaluation ................................................................. 61

8 Correctness 63
  8.1 Ordering ................................................................. 63
  8.2 Large Formulas .......................................................... 64
  8.3 Rewrite ................................................................. 64

9 Discussion 65
  9.1 Logical Clock ............................................................ 65
  9.2 Rewrite ................................................................. 65
  9.3 Large Formulas .......................................................... 66
  9.4 Correctness .............................................................. 66

10 Conclusion 69
  10.1 Research Questions ................................................... 69
  10.2 Future Work ............................................................ 70
Glossary

Abstract syntax tree An abstract tree structure of some source code. Abbreviated as AST.

Byzantine failures Arbitrary deviations of processes’ behaviour.

Formal method Method to mathematically specify and verify computer software.

Invariant Condition that is expected to hold during the execution of a program.

JVM Java Virtual Machine. An abstract computing machine that runs Java programs.

Logical time Time that keeps tracks of the ordering of events or activities in a distributed system.

NP-complete decision problem Non deterministic polynomial time, a decision problem whose solution can be verified but not found (as of June 2018) in polynomial time.

SLF4J A logger that is used to abstract some of the commonly used java frameworks for logging, such as java.util.logging or log4j.

State space The set of states that are reachable in a computer program.

Static checker A tool to verify that a program source code follow the rules of a given programming language.
Chapter 1

Introduction

In response to the increased computing demands, modern enterprises have had their traditional monolithic system architectures pushed to the limit. Thus, the shift towards distributed systems such as microservices — a collection of loosely coupled services — has become increasingly palpable. In order to keep up with the trend, many enterprises are abandoning their current monolithic systems and entering the den of inherent vulnerability to non-determinism that distributed systems entail. Although this change has brought along benefits in terms of scalability and reliability, the instances in which distributed systems fail relative to monolithic ones are disparate.

1.1 Background

If a distributed system provides multiple ways of achieving the expected or successful outcome, even in the event of failure, it is said to be fault tolerant. Hence, various disciplines have emerged in order to determine whether a given distributed system has redundancy. One of these disciplines, Chaos Engineering, deliberately injects faults into a given distributed system in order to determine its resilience. The space of instances of failures in a given distributed system, however, increases exponentially with the number of individual components that it is comprised of. For large-scale distributed systems, it becomes infeasible to inject faults to every possible combination of interactions between the components. Lineage-driven fault injection (LDFI) is a method that specifically searches for injections that can prevent a successful or expected result from occurring. LDFI uses backward reasoning from an expected or successful outcome to determine which failures could arise in a given system. Thus, at each step, beginning from the last, it asks what could have prevented said outcome from occurring, and continues this process until it either finds a hypothesis (possible fault injection) or else concludes that the system is fault tolerant.
CHAPTER 1. INTRODUCTION

1.2 Problem

The problem is that the state-of-the-art implementation of LDFI, Molly, is restricted to taking distributed programs written in logic programming language Dedalus [3], as input. Consequently, it does not support standard imperative object-oriented languages, which might lead to implementation limitations seeing as Dedalus is a relatively obscure language within a not so widely used programming paradigm [21].

1.3 Objective

The objective of this thesis is to extend and adapt the state-of-the-art LDFI approach, which has so far only been realized for a logic programming language, to a general-purpose, object-oriented language where distributed programs are written using the actor model. The widely-used framework to write distributed programs using the actor model, Akka, has been implemented for the JVM and JavaScript runtimes and provides APIs for both Java and Scala. Thus, the goal of the thesis, more concretely, is to extract the lineage from the execution traces of Akka programs; design, model and implement a lineage-driven fault injector; and finally, determine whether such programs are fault tolerant or else provide the fault injections that proves otherwise.

1.4 Research Questions

In order to reach the objective of this thesis, a set of research questions are formulated. In essence, the research questions are formulated such that it is possible to determine the viability of performing lineage-driven fault injection on actor-based programs both in theory and in practice.

Furthermore, it has been shown that Molly, implementing lineage-driven fault injection for Dedalus programs, performs an analysis that is sound: every counterexample found corresponds to inputs that would cause the program to fail. Also, Molly provides completeness guarantees, i.e., it asserts that if no such counterexamples are found, then the program is fault tolerant. Moreover, Molly can extract the data lineage precisely because the distributed program is written in a logic programming language like Dedalus, where all computations are based on logical inferences and can thus be traced.

As a consequence, the following research questions are explored:

i) Is it possible to employ lineage-driven fault injection for actor-based programs?

ii) Can lineage-driven fault injection be used to analyze Akka programs?

iii) How can data lineage be extracted from execution traces of Akka programs?

iv) Is it possible to control the run-time execution of Akka programs?
v) How can we ensure that the analysis of the extracted data lineage is sound?

vi) Is a precise statement about completeness possible?

1.5 Purpose

The purpose of this thesis is to assist modern enterprises in their pursuit of building fault tolerant distributed systems. For large scale internet enterprises, reliable and available services are paramount to the business model. In companies like Netflix and Amazon, availability is defined in terms of the number of 9s after 99 percent \(^\text{\footnotesize 99\%}\). The discrepancy in downtime between two systems that have an availability rate of 99.99\% and 99.999\% respectively, is large. In a year, the downtime of the former system would be around 53 minutes whereas it would be around 5 minutes for the latter. Naturally, enterprises whose business model are reliant on availability would have an economic incentive to minimize their downtime by looking for methods that increase their systems fault tolerance. The expansion of promising software such as \textit{Molly} would therefore be in direct alignment with the purpose of the thesis.

1.6 Contributions

The main contributions of the study are summarized as follows:

- A lineage-driven fault injector for programs written in Akka, capable of finding failures violating the subjected program correctness specification or else conclude that the program is fault tolerant.

- An extension of LDFI to asynchronous actor-based programs, including novel approaches for (a) tracking lineage, as well as (b) failure specifications constraining fault injection.

- A mean of controlling the execution of Akka programs by employing an external controller that interacts with the program at run time.

- A set of rewriting rules that target paramount keystones in a given Akka program such as message passing and logging.

- Experimental results applying the approach to a set of existing Akka programs.

1.7 Sustainability, Ethics and Societal Aspects

Although the results of this study have no direct societal or ethical implications due to their theoretical nature, the advancement of technology has an impact, albeit a small one,
on society in all of its cornerstones nevertheless. More concretely, improving the reliability and availability of important services and by extension the technology of which many are dependant upon, results in an improvement for society as a whole.

With regards to sustainability, one could make the argument that the study contributes to less downtime for systems which in turn results in an elevated strain on the environment in terms of electricity usage. On the other hand, with the results of this and other similar studies, modern enterprises would be able to discard possibly redundant resources. As a consequence, we could see a decrease in the emissions that arise with regards to the energy consumption within the IT sector that would yield positive environmental effects.

1.8 Outline

The thesis is presented in the following structure:

Chapter 2. Theoretical Background This chapter includes detailed descriptions of the various technologies used to conduct this study.

Chapter 3. Related Work In this chapter, the previous and related work is reviewed.

Chapter 4. Research Method This chapter describes the thesis’ method in general and the quantitative method in particular.

Chapter 5. LDFI for Actor-based Programs In this chapter, a novel conceptual framework for how to apply lineage-driven fault injection for actor programs is given.

Chapter 6. LDFI for Akka In this chapter, a concrete implementation of the conceptual framework on actor programs written in Akka is detailed.

Chapter 7. Results In this chapter, the results of employing ldfi-akka on various Akka programs is presented.

Chapter 8. Correctness In this chapter, the correctness of the implementation is presented.

Chapter 9. Discussion In this chapter, a qualitative analysis of the evaluation is given. Moreover, alternative methods and theories are discussed.

Chapter 10. Conclusion This chapter includes the conclusion of the study while giving insight on possible future work.
Chapter 2

Theoretical Background

In this chapter, terms and concepts that are paramount to this study are introduced, explained and reasoned about. In section 2.1 a brief description of the axioms on which logic programming is built upon is given. These basics are expanded upon to give an introduction to the logic programming language Dedalus in section 2.2. Section 2.3 is a key section as it includes specifics about LDFI, the technique that this study attempts to expand. The state-of-the-art implementation of LDFI, Molly, is detailed in section 2.4 alongside some simple examples. Akka, a toolkit to build concurrent distributed systems for Java programs, is introduced in section 2.5. Lastly, a concise description of Scala’s informal design by contract statements is given in section 2.6.

The sections 2.2 - 2.4 including the concepts, reasoning, figures and terms, are in large part a review of the previous work by Alvaro, Rosen et al. [4] and Alvaro, Andrus et al. [5].

2.1 Logic Programming

Logic programming, a subset of the declarative programming paradigm, uses logical inference to conduct computations. To accomplish this, logic programming languages are comprised of two major components: rules and facts. The logic in a program is expressed as relations using a combination of rules and facts. Rules (and facts) are written in the form of clauses, following the general structure illustrated in Listing 2.1.

The rule can informally be stated as, if Body₁ is true, and Body₂ is true, and Bodyₙ is true, then Head is true. Facts, on the other hand, are rules without a body (implicitly inferred to be true) that always hold.

```
Head :- Body₁,Body₂...,Bodyₙ. /*Rule*/
Head :- true. ≡ Head. /*Fact*/
```

Listing 2.1: Rules and facts in logic programming.
CHAPTER 2. THEORETICAL BACKGROUND

parent(Alice, Bob).
parent(Bob, Charlie).
grandparent(X, Y) :- parent(X, Z), parent(Z, Y)

Listing 2.2: Sample logic program

Head₁ : − Body. /*Deductive*/
Head₂@next : − Body. /*Inductive*/
Body@1. /*Fact*/

Listing 2.3: Examples of rules and fact incorporating the notion of logical time.

Now consider the logic program in Listing 2.2. The first two lines are facts, which describe the parent relation between two atoms. The third line (a rule), states that if some X is a parent of some Z and Z is a parent of some Y; then X must be the grandparent of Y. Thus, if we query grandparent(A, Charlie), then A = Alice is inferred, as Bob is the parent of Charlie and Alice is the parent of Bob.

2.2 Dedalus

As a logic programming language, Datalog is based on the principles described in the previous section. In recent years, Datalog has been used as a foundation language in a variety of fields within computer science such as: compiler analysis, robotics and networking [3]. Dedalus, a subset of Datalog, provides a foundation for two major features of distributed systems: asynchronous processing and communication, and mutable state.

2.2.1 Logical Time

To express these features, Dedalus syntax incorporates the notion of logical time as an attribute in the head predicate. To accommodate for said notion, Dedalus are split into inductive rules, which incorporate logical time, and deductive rules which do not.

The deductive rule remains unchanged from the syntax described in the previous section. The inductive rule, however, states that if Body is true at time $t \in Z$, where $Z$ is the time domain, then Head is true at $t + 1$. The fact states that Body is true starting at logical time $t = 1$, and hence (by applying the inductive rule) $\forall k \in Z$, where $k \geq t$.

2.2.2 Asynchrony

Distributed systems are inherently vulnerable to non-determinism, as the behaviour of the system as a whole or its individual components are hard to predict. For instance, communication over a network might be delayed, interrupted or otherwise interfered with. As a result, results from logical inferences might come in the wrong order, and thus cause
2.3. LINEAGE-DRIVEN FAULT INJECTION

Listing 2.4: Asynchrony in Dedalus.

\[
\text{Head}(\zeta) : \text{Body}_1(\tau), \text{time}(\zeta), \text{choose}(\tau, \zeta); /* Asynch. Rule*/
\equiv
\text{Head@async : Body; /* Asynch. Rule*/}
\]

a program to behave differently and consequently lead to unexpected outcomes. In order to model non-deterministic behaviour, a non-deterministic construct, choice is introduced. Choice is used by the programmer to propose alternatives that the program later ”chooses” from. For instance, if the program were to fail at a certain time step, it backtracks and tries other alternatives. A rule is said to be asynchronous if the relation between logical time \( \zeta \in \mathbb{Z} \) and logical time \( \tau \in \mathbb{Z} \) is unknown. The asynchronous rule is hence constructed as depicted in Listing 2.4.

The rule can informally be read as, if Body is true at time \( \tau \) and current logical time is at \( \zeta \) and choose is true at both time \( \tau \) and \( \zeta \) then Head is true at time \( \zeta \).

2.3 Lineage-driven Fault Injection

Lineage-driven fault injection (LDFI) is a technique to determine whether a given distributed program is fault tolerant, or else provide counterexample(s) — possible inputs — that would cause it to fail [4]. The subsections to this section provide necessary details of the concepts used in the implementation of lineage-driven fault injection.

2.3.1 Data Lineage

Data lineage is defined as connecting system outcomes for a given execution to the data or messages that lead to that outcome [4]. As a consequence, using data lineage simplifies the process of tracing errors of a given execution to its root cause. Furthermore, data lineage enables step-by-step debugging of a given program execution, which can be visualized using lineage graphs.

2.3.2 Failure Specification

In order to simulate failures using LDFI, two main assumptions are made. First, LDFI assumes no Byzantine failures, i.e., the behaviour of each node is consistent for every observation, and second, it assumes that all messages are eventually delivered. Furthermore, it assumes that all delivered messages are received in a deterministic order, i.e., they are received in the order in which they were sent. Granted, said assumptions does not represent the behaviour of real-life in production systems. On the other hand, evaluations of an asynchronous distributed program in a synchronous simulation are made possible as a consequence. Given these limitations, a specification for admissible failures must be given, as it would not be worthwhile examining how a distributed system comprised of three
nodes behave if we crash all three nodes at the same time. Thus, the failure specification limits how many failures can arise, in addition to how they arise. The failure specification described by Alvaro, Rosen et al. consists of three parameters: end of time (EOT), end of finite failures (EFF) and Crashes. The EOT parameter bounds the number of permitted executions within a simulation. EFF represents the logical time up to which message loss is admissible, and lastly, Crashes specifies the maximum number of node crashes that is allowed. Consider the following failure specification:

\( fspec: \langle 3, 1, 1 \rangle \)

Informally, up to 3 executions should be explored, message loss at exactly logical time 0 or 1 are allowed, and up to 1 crash failures are permitted. Thus, the \( fspec \) imposes limitations on what, where and how we fail a given distributed system. As a consequence, each failure injection in LDFI must be admissible in regards to the failure specification. According to the above failure specification, crashing two nodes or cutting messages after logical time step 1 would not be admissible.

### 2.3.3 Sweep

In order to set the failure specification for a given distributed program, Molly performs a sweep, which can be reduced to an algorithm consisting of the steps illustrated in Figure 2.1. First, \( EFF \) is initialized to 0. \( EOT \) is thereafter incremented until an execution in which messages are sent is witnessed. A sweep is not performed in the case where a distributed program does not entail any communication, as there could not exist any failures that are of interest to the LDFI algorithm. Therefore, when an execution in which a message is sent is witnessed, \( EFF \) is increased until either an invariant violation is produced — in which case \( EOT \) is incremented — or \( EFF = EOT - 1 \), in which case both are incremented. This process is repeated until a user defined upper bound of a logical clock expires.
2.3.4 Boolean Formulas

Figure 2.2 depicts a single client, $C$ sending a broadcast $Bcast$, which is then intercepted by two replicas: $Rep_1$ and $Rep_2$. With only a single broadcast, and two replicas, the number of possible failures that can arise for set $S = \{Bcast, Rep_1, Rep_2\}$ is $2^3 = 8$, namely, the number of subsets in $S$. However, LDFI does not consider all of the possibilities in the failure space exhaustively, as it only looks for paths that lead to an successful outcome. The paths leading to successful outcomes — stable writes with regards to Figure 2.2 — are written as clauses in *Conjunctive Normal Form* (CNF). In a boolean formula, clauses contain *literals* — the smallest entities in a boolean formula — that can evaluate to either true or false. In this context, the literals correspond to nodes or messages within the distributed program. A boolean formula in CNF is written in clauses that are separated with logical AND operators, while the literals in each clause are separated with logical OR operators. The CNF-formula obtained from the lineage graph depicted in Figure 2.2 is thus:

$$(Bcast \lor Rep_1) \land (Bcast \lor Rep_2)$$

A stable write only occurs when $Bcast$ is intercepted by $Rep_1$ or $Rep_2$. As a result, the solutions (possible fault injections) to above boolean formula are:

$\{Bcast\}$, $\{Rep_1, Rep_2\}$

If $Bcast$ is set to true (i.e cut the message between the client and replicas) then no stable write occurs. Alternatively, setting both $Rep_1$ and $Rep_2$ to true (crashing them) also prevents a stable write. That is to say, the solutions corresponds to the crashes or message losses that needs to occur in order to prevent the system from reaching a successful outcome. At first glance, this is unintuitive because in this context, setting the literal to true correspond to failing it in the system. On elaborate scrutiny however, this makes sense, as the goal of LDFI is to find input that causes a given program to fail. In short, every solution to the obtained CNF-formula from the lineage graph represent possible fault injections that causes the system to fail.

2.4 Molly

*Molly* is the state-of-the-art implementation of the LDFI technique, depicted in Figure 2.3. It takes as input a distributed program written in Dedalus, together with its corresponding inputs and, pre- and post-conditions. Thereafter — in order to obtain the fspec — it performs a sweep. It proceeds by performing a *forward step*, extracting outcomes of a failure-free execution of the distributed program. It then continues to perform a *backward step*, where for every step of the execution, it extracts the lineage of the outcome and converts it to a boolean formula in CNF. The backward step then passes CNF-formula to the external SAT-solver which sends possible failure scenarios to the *evaluator*. The evaluator determines whether the iteration should stop, in which case it renders a *verdict*.
1 Otherwise, it concludes that the iteration should continue and transforms the possible failure scenarios into new inputs to be passed to the forward step for another iteration. In order to perform this analysis, the distributed program must consequently fulfill the following prerequisites:

i) There must be clearly stated pre- and post-conditions.

ii) The outcome of an execution must be clear, i.e., the outcome must either satisfy or violate the invariant.

iii) For every outcome, it must be possible to extract the data lineage.

2.4.1 Simple-Deliv in Dedalus

Now, consider the sample Dedalus program implementing a best-effort broadcast, simple-deliv; illustrated in Listing 2.5. The first line says that if some payload is broadcasted from

---

1The evaluator concludes whether the given distributed program is fault tolerant or else provide lineage together with program output that violates the invariants.
2.4. **MOLLY**

| log(Node, Pload) :- bcast(Node, Pload); |
| node(Node, Neighbor)@next :- node(Node, Neighbor); |
| log(Node, Pload)@next :- log(Node, Pload); |
| log(Node2, Pload)@async :- bcast(Node1, Pload), node(Node1, Node2); |

**Listing 2.5:** *simple-deliv*. A sample Dedalus program implementing a best-effort broadcast

| missing_log(A, P1) :- log(X, P1), node(X, A), notin log(A, P1); |
| pre(X, P1) :- log(X, P1), notin crash(_, X, _); |
| post(X, P1) :- log(X, P1), notin missing_log(_, P1); |

**Listing 2.6:** Correctness specification for *simple-deliv*

A certain node, then it must be logged by that node. The second line, an inductive rule, states that if two nodes are neighbors at a certain time step, then they are also neighbors for all the following time steps. Similarly, line 3 states that if a node logs some payload at a certain time step, then it is logged for the consecutive time steps. The last line represents a distributed rule that states: if some node broadcasts some payload while connected to some neighbor, then it must *eventually* (possibly at a different time) be logged by that neighbor.

Listing 2.6 implements the correctness specifications that are necessary to evaluate the program implemented in Listing 2.5. The first line says that a log is said to be missing if its payload exists for some node, while its neighbor has failed to log that same payload. The pre-condition for *simple-deliv* is that all nodes that have not been crashed must have logs of their payload. The post-condition says that all payloads logged for some node must also have been logged by all neighboring nodes. *Molly*’s goal is always to find admissible failure injections that satisfies the pre-condition while violating the post-condition.

### 2.4.2 Injecting Failures

Assume that we have a set of three neighboring nodes \{A, B, C\}. Moreover, the failure specification has been set to \(EOT=\text{inf}\) (we want to check the entire program), \(EFF=2\) and \(Crashes=1\). The notation for literals in the boolean formula is used by Alvaro, Rosen et al. to denote messages sent during an execution of a program is: \(O(Sender, Receiver, SenderTime)\). The first input is a single fact: \(bcast(A, data)@1\). The stage has now been set in order to run *Molly* on this sample program. After an successful outcome has been extracted from the forward step, *Molly* performs the backward step and extract the following CNF-formula: Informally, process \(A\) made a broadcast to its neighboring processes \(B\) and \(C\) at logical time 1. Finding possible failure injections for this naive best-effort broadcast

\[O(A, B, 1) \lor O(A, C, 1)\]
is simple, and is no match for Molly considering the above failure specification (\(EFF=1\) and \(Crashes=0\) would suffice to produce a violation). Starting backwards, it realizes that B only logged the message because it received it from A at time 1. Thus, this is be one (among other) hypothesis for the next iteration. The program is now run again, but this time, Molly prevents A from sending a message to B (message omission) at time 1. As a result, B does not receive the message from A and consequently does not log it, ending in a violation of the post-condition.

2.5 Akka

Akka is a toolkit for constructing message-driven distributed programs that run on the Java Virtual Machine (JVM). Akka implements the actor model, introduced in 1973 as a framework for a theoretical basis to conduct concurrent or parallel computations [11]. Whereas the actor model is a conceptual model, an actor system is a hierarchical structure comprised of actors. Actors in Akka are the minimal computational entities and are containers for state, behavior, a mailbox, child actors and a supervisor strategy [14]. Actors are represented and uniquely identified with actor references. A state in an actor refers to the data that it is currently holding, e.g., a counter or message. The behavior defines the actor’s actions in response to messages it receives. A mailbox is used to process an actors messages by implementing a first-in, first-out (FIFO) queue system. Each actor is capable of creating child-actors for which they can delegate different tasks. A parent actor has access to its children’s actor references and is responsible for their supervision. For instance, if an exception occurs within a child actor, then the supervisor (parent actor) is responsible for handling it. In response to receiving a message an actor can: send messages, mutate its state, change its behaviour or create another actor.

2.5.1 Simple Concept

Figure 2.4 illustrates a simple actor system comprised of four actors receiving and sending messages. The Checker actors sends messages to the mailbox of the Counter actor. The mailbox then sequentially handles the messages in the order of which they arrived (which might differ from the order in which they were sent). The Counter then receives the messages and decides on how to proceed based on the alternatives previously listed. In the figure, the Counter responds with messages designated to corresponding senders mailbox.

Listing 2.7 illustrates how the actor system in which the actors operate is instantiated using Akka. The Counter actor is created by specifying the system in which it should exist. Three Checker actors are created together with the reference to the Counter actor, allowing them to initiate requests to it. Props is a configuration object within Akka used to make sure that the created actor is properly registered in the actor system.
Figure 2.4: Illustration of a simple actor system using 4 actors.

```
object Start
object CheckerCounter extends App {
  val system = ActorSystem("system")
  val counter = system.actorOf(Props[Counter], "counter")
  for (i <- 1 to 3) {
    val name = "checker" + i
    system.actorOf(Checker.props(counter), name) ! Start
  }
}
```

Listing 2.7: An implementation of an actor system using Akka.
class Counter extends Actor {
    var count = 0
    def receive = {
        case Request =>
            if (count <= 100) {
                sender() ! count
                count = count + 1
            }
            else {
                self ! PoisonPill
                sender() ! PoisonPill
            }
        }
    }
}

Listing 2.8: Implementation of the Counter actor.

object Request
class Checker(counter: ActorRef) extends Actor {
    def receive = {
        case count:Int =>
            if (count % 4 == 0) {
                println("My integer is divisible by four: " + count)
                counter ! Request
            }
            else {
                counter ! Request
            }
        case Start => counter ! Request
    }
}

Listing 2.9: Implementation of the Checker actor.

2.5.2 Checker Counter

Now, consider the code shown in Listing 2.8 which implements the Counter actor. First, a class representing the Counter actor is implemented. Thereafter, the receive method is implemented which ensures that the actor can receive messages containing a Request object. In response to the message, the actor either sends a message with the current count to the sender, and then proceed to increment the counter variable; or kill itself and the sender if the actor is finished counting.

Listing 2.9 implements the behaviour of the Checker actor. The Checker actor receives an actor reference to the Counter actor when instantiated. In response to receiving the initial Start object from the outside, it sends a Request object to the Counter actor. Whenever it receives a count, it checks whether it is divisible by four, in which case the count is announced. Thereafter, the Checker actor always sends requests for another count (regardless of the outcome), and repeats the process until it is killed from the outside.
2.6. SCALA’S DESIGN BY CONTRACT

Scala’s informal design by contract facilities are comprised of the following statements: assert, assume, ensuring and require. A static checker expects to be able to prove all assert statements at compile time, while the assume statement is used to tell the static checker that it can trust it to be true, but not try to prove it. At run-time however, the statements behave the same, namely, they both throw the same exception if their conditions do not hold. The require statement is different from the assert statement as it assumes that if a condition violation occurs, then the user must have given input which is faulty rather than assuming that a logical error exists in the program’s source code. Ensuring is used as a form of assert statement that is applied to a given function’s return value. This study restricts itself to using require and ensuring to set the pre- and post-conditions for a given Scala program.

Listing 2.10 implements a method to set the age (instance variable age) of a class Person. The require statement makes sure that the input parameter is greater than 0, as there are no negative ages. The ensuring statement makes sure that the instance variable was actually set to the age specified in the input parameter when the method is finished executing.
Chapter 3

Related Work

In this chapter, two common disciplines that have been used to determine the resiliency or fault tolerance of software programs are introduced. Section 3.1 details specifics about what model checking is in general in addition to giving examples of how it has been used to assess distributed systems. In section 3.2 concise explanations of other fault injection techniques that have been used previously are given, in addition to highlighting their respective strengths and weaknesses.

3.1 Model Checking

Model checking is a formal method that is used to check whether a given system meets the requirements of a specified condition. For software programs, it requires enumeration of the state space while verifying that all possible states are valid in respect to the given specification. To accomplish this, the program and the specification are stated in precise mathematical notation, such as propositional logic. This is generally done by specifying an invariant together with pre- and post-conditions for a given program. An invariant is a condition that must hold during the entire (or some part) execution of the program, while pre- and post-conditions must hold before and after the execution respectively.

Achieving fault tolerance for distributed programs require not only that the individual components of which it is comprised of are fault tolerant, but also that the guarantee holds under composition of those components. It is thus not sufficient to use model checking to assert fault tolerance on the individual components of a distributed system, but also on their interaction. Unfortunately, model checking techniques require that the state space in a distributed system — which increases exponentially with the number of individual components — be iterated exhaustively. For large systems, the state space gets vast to the point of which it becomes infeasible to enumerate the entirety of it.

The attempts to circumvent this issue include the usage of heuristics to guide the model checker through the search space [22], or use model checkers concurrently with execution on the individual components and provide conjectures (not guarantees) on how the system holds under composition [23]. These methods are, however, incapable of providing
CHAPTER 3. RELATED WORK

guarantees in terms of fault tolerance with regards to the systems they examine.

3.2 Fault Injection

Fault injection techniques are used to explore the space of possible failures within a given system. As the name implies, the techniques deliberately inject faults into programs to determine whether the outcome of the program changes in response to said failure input. Common fault injection techniques make use of one or a combination of heuristic, random and brute-force strategies [9, 12]. Section 3.2.1 details a specific randomized fault injection strategy. In section 3.2.2 a more sophisticated (in terms of choosing injections) strategy is introduced, and lastly, details of an implementation of LDFI are given in section 3.2.3.

3.2.1 Simian Army

After Netflix transitioned from traditional monolithic system architectures to distributed ones, they looked for ways to test their resilience [24]. The first attempt was to introduce what they call the Simian Army. The Simian Army is composed of various mechanisms that are used to simulate failures in different production systems to measure their impact. One of these mechanisms, the Chaos Monkey, disables different in-production services at random. The disparate outcomes are thereafter carefully monitored and any unknown bugs that arise can consequently be addressed. Another mechanism, Latency Monkey, induces large delays in the network to simulate node or service downtime — without physically shutting them down — and consequently tests the system’s ability to survive it.

3.2.2 Failure Injection Testing

Although the Simian Army was successful at finding bugs or potential failure input, randomized failure injection techniques are unlikely to discover complex failures that arise from combinations of different inputs that are rarely explored. Failure Injection Testing (FIT) is a platform to insert failure injection with higher precision in comparison to the randomized fault injection technique employed by the Simian Army mechanism [12]. FIT uses an internal tracing system to extract the path that a certain execution takes within the system. By continuing to insert failures at different points on the path, it can see how the system reacts by monitoring how the outcome changes in response. As a result, paths that are rarely explored within typical execution can be seen and evaluated accordingly.

3.2.3 LDFI at Netflix

FIT is more precise than Chaos Monkey, but the precision comes at a cost: the injection points must be identified by humans. This identification can consequently only be done by engineers who are acquainted with the systems topology and the expected behaviour of it following the disparity in inputs. This process of human identification is expensive in
3.2. FAULT INJECTION

regards to time and resources. Netflix was thus looking for ways to automate this process: is there a way to find bugs without the intuition of experienced testing engineers? The answer was found in LDFI, as it does not require humans to find the potential failure scenarios in a given distributed system. Conveniently, the implementation of LDFI could make use of much of the groundwork that had been laid with the implementation of FIT. For instance, FIT uses a tracing system to record the execution path that the implementation of LDFI made use of to extract the data lineage that is required to perform its analysis. By implementing LDFI, Netflix found 11 critical failures that could prevent the users from using their video streaming services.
Chapter 4

Research Method

This chapter includes details on how this study was conducted scientifically. Section 4.1 details the overall strategy that was chosen. The method is described in section 4.2 while section 4.3 include specifics about the various phases in the study.

4.1 Research strategy

The field that this study covers is broad and intensively researched. Consequently, an efficient research strategy was chosen in order to optimize the study. The research phases consist of the initial steps that were conducted in the research process to facilitate the use of the research instruments. The research method was thereafter chosen in order to process the research instruments. Lastly, the validity threats were determined to evaluate the research method.

4.2 Method

In this section, a description and more importantly, a justification of the chosen research types is given. Moreover, the section also contains details about the research approach chosen for the study.

4.2.1 Research type

As the field of fault tolerance, fault injection and chaos engineering is an extensively researched area, large number of research articles and publications was evaluated. These articles and publications in turn contain numerical, mathematical and statistical data that would need to be analyzed and scrutinized in order to make comparisons between alternating methods and theories.

Qualitative research is defined as an interpreting research method that is applied on various academic disciplines, such as social sciences and natural sciences, but also in market analysis and other relating areas. Its main benefits lies in its ability to answer why and
how questions in academic research. Quantitative research is described as an empirical research method of observable phenomena via statistical, mathematical or computational methods. It is often used to make objective analyses of data or to quantify problems that can be converted to useful statistics.

A qualitative research method was determined to be most suitable with regards to this particular study. It was employed to gather information, interpret and evaluate alternating theories and methods within the field of fault tolerance, fault injection and chaos engineering. Although a qualitative research method was predominantly employed, it is not suitable when it comes to analyzing statistical and mathematical data. As such, a quantitative research was utilized, as it is designed to make empirical investigations of such data. Therefore, a mixture of both qualitative and quantitative research was needed to conduct the research for this study.

4.2.2 Research approach

The research approach chosen for this study is deductive. In essence, correctness and soundness of the implementation is assumed, due to its foundation in an already formalized and proved theory. As such, this study need only reason about in which the proof might break with regards to the extensions that this implementation makes to the theory, if such extensions are made.

4.3 Research phases

This study consisted of two research phases: a literature study and a practical study. The phases were not necessarily consecutive events, i.e., the practical study was done while the literature study was in progress and vice-versa.

4.3.1 Literature study

The literature study mostly consisted of studying Lineage-driven fault injection, which was invented by Alvaro et al. First, the article “Lineage-driven Fault Injection” by Alvaro, Rosen et al. was studied, as it covers all of the necessary knowledge prerequisites in order to conduct this study. Afterwards, the literature study was complemented by studying “Automating Failure Testing Research at Internet Scale” and “Abstracting the Genius”. Moreover, to conduct the implementation of this study, in-depth knowledge of Dedalus was required, and as a result article ”Dedalus: Datalog in Time and Space” was necessary for the literature study.

Furthermore, alternating methods and theories were studied to gain further insight into the field, i.e., understanding solutions to previous problems that existed prior to the research of Alvaro, Rosen et al. in “Lineage-driven Fault Injection”. The related work includes the implementation of various chaos engineering techniques by Netflix, such as the Simian Army and Failure Injection Testing.
4.3.2 Practical study

The practical study began with the creation of a conceptual framework as a theoretical basis on which the implementation could be founded upon. LDFI as a technique is the combination of various disciplines, such as distributed systems, fault tolerance, chaos engineering and data replication. Not only did this study inherit the non exhaustive list of previous mentioned fields, but also researched other fields such as the actor model and the Scala language. As a consequence, part of creating the conceptual framework was to incorporate the fields such that their utility would be optimized. After the conceptual framework had been established, the implementation, acting as a proof of concept, was made. The process of the implementation can most easily described as iterative. The conceptual framework was initially broken down into various parts, which were then implemented consecutively. For instance, the first step of the implementation was to write Akka equivalent Dedalus programs. LDFI was only performed conceptually on these programs: only when the concept held under the new circumstance did the implementation proceed with the subsequent step, and so on and so forth.

4.4 Validity

With regards to the quantitative data in the evaluation: the validity threat is non-existent, since the quantitative data that this study provides in terms of statistical and numerical data are proven valid by their reproducibility. The calculations are exclusively based on mathematical principals which in turn makes them reproducible, regardless of who carries out the calculations. When it comes to qualitative research, validity is defined as the relevance of the data for a given problem and the proper representation of an observed phenomena. Reliability is the precise measurement of what is aspired to be measured, and the assertion that repeated measurements of the same phenomena yield the same or similar results. Naturally, it is not possible to prove that an inherent subjective research method such as the qualitative research method is objectively sound with respect to validity. However, this and other studies are subjected to peer-review which at least strengthens the validity. On the other hand, the reliability of a particular study can be strengthened by performing the same measurement multiple times.
Chapter 5

LDFI for Actor-based Programs

In this chapter, a novel conceptual framework for how to apply Lineage-driven fault injection for actors is presented. More concretely, section 5.1 gives insight on how actor programs are encoded in CNF and subsequently solved. In section 5.2 a novel approach for employing logical clocks to actor programs is detailed. Lastly, an extension to Molly’s approach of constraining fault injections is given in section 5.3 whereas section 5.4 includes a description of the evaluator in addition to the components it is composed of.

5.1 Boolean Encoding

A key step of the LDFI technique is to use boolean encoding of a given run of a program with the purpose of deriving injections hypotheses for the next run. The purpose of formatting the logs in a structured manner is thus to simplify the process of encoding the execution of a program in CNF. Section 5.1.1 details Scala representation of formulas in CNF while section 5.1.2 includes brief explanations of how the formulas are solved.

5.1.1 Formulas as Paths

The boolean formulas of a given run of a distributed program represent the distinctive paths that lead to a successful outcome, i.e., outcomes that do not violate the correctness specification after the program has terminated. A boolean formula in CNF consists of clauses and literals, and as such, the implementation illustrated in Listing 5.1 was used.

A boolean converter, depicted in Listing 5.2 is used to translate the behavior of a given actor program — stored as rows in FormattedLogs — to a boolean formula in CNF. The

\[^1\]Note that many fields and methods have been omitted due to space limitations.
class Formula {
  var clauses: List[Clause] = List.empty
  def addClause(clause: Clause): Unit = {
    clauses = clause :: clauses
  }
}

class Clause(formula: Formula) {
  var literals: List[Literal] = List.empty
  def addLiteralToClause(literal: Literal): Unit = {
    literals = literal :: literals
  }
}

sealed trait Literal
final case class Node(node: String, time: Int) extends Literal
final case class MessageLit(sender: String, recipient: String, time: Int) (
  val message: String)
  extends Literal

object CNFConverter {
  def convert(formattedLog: FormattedLogs, formula: Formula): Unit = {
    val clause = new Clause(formula)
    for (line <- formattedLog.rows){
      addRowToClause(clause, line)
    }
    formula.addClause(clause)
  }

  def addRowToClause(clause: Clause, line: Row): Unit = {
    val messageLiteral = getMessage(line)
    val senderNodeLiteral = getSenderNode(line)
    val recipientNodeLiteral = getRecipientNode(line)
    if(!clause.literalExistsInClause(senderNodeLiteral))
      clause.addLiteralToClause(senderNodeLiteral)
    if(!clause.literalExistsInClause(recipientNodeLiteral))
      clause.addLiteralToClause(recipientNodeLiteral)
    clause.addLiteralToClause(messageLiteral)
  }
}

Listing 5.1: Scala representation of literals and clauses

Listing 5.2: Conversion of the formatted to logs to boolean formula in CNF.
5.1. BOOLEAN ENCODING

`FormattedLogs` case class was intentionally made such that the conversion would be trivial. Before iterating the rows in the `FormattedLogs`, an empty clause is created. The clause is then filled with literals corresponding to the `Rows`. After the iteration, the clause, now filled with literals, is added to the boolean formula. The boolean formula therefore consists of a list of clauses, each representing the parsed messages and node activities sent within an actor system previously stored in `FormattedLogs`.

5.1.2 Minimal Solutions

The purpose of encoding the behavior of a given actor program in CNF is to solve it, and use the solutions as hypotheses for the next run of that program. The boolean satisfiability problem (SAT) is known to be a NP-complete decision problem and is as such a research topic of its own. Consequently, an external SAT-solver was used. Considering that the core of this implementation is written in Scala, a natural choice was to use SAT4J (SAT for Java), a java library for solving boolean problems [13]. Prior to using SAT4J, however, an algorithm mapping the boolean formula depicted in previous sections to a format understandable to the solver is needed.

Initially, all of the clauses in the formula are passed to the solver. Recall however that lineage-driven fault injection uses backward reasoning in order to make targeted failure injections, as opposed to injecting failures at random. Therefore, the resulting solutions from unconstrained SAT-solving would be of little interest. In order to constrain the failure injections, we use the previously introduced concept, failure specifications. Thus, we look at the maximum number of crashes, `maxCrashes`, and with a given number of nodes `n`, we add a clause with the constraint that at least `n - maxCrashes` nodes do not crash. The process is however encumbered by the fact that a node can be active, and as such, crash at different times. For that reason, each node activity has its own literal. As our model assumes no crash recoveries however, a node can not crash at two disparate times. Therefore, an additional encompassing literal is added: the never crashed (nv) literal, with the added constraint that a node either crashes once, or not at all. In essence, exactly one of the never crashes literal in disjunction with all the node activity literals can be true at a time. Furthermore, suppose that we have assumed that some nodes are crashed and messages are omitted, i.e., we have crashes and omission from previous iterations that lead to this particular formula. In that case, the SAT4JSolver’s API is used to add unit clauses for each of those literals representing the crashes and omissions. More specifically, additional clauses are added containing precisely the negation of those literals. As a boolean formula in CNF is comprised of conjunctions of disjunctions, those literals must be false as otherwise the formula would be impossible to satisfy. As a result, the solver uses unit propagation, a rule that performs two simplifying operations. First, all of the clause containing the literal (i.e., the negation of that literal) are discarded as they now can be deduced to be satisfied, and second, the negation of the literal (the literal) are removed from the formula, as they no longer can contribute to any clause being satisfied.

The algorithm for the solver used for this project reused many of the ideas from corresponding solver implementation used in Molly. For that reason, the algorithm is omitted,
but is publicly available on Github.  

*Example.* Suppose we are given a formula, $\varphi$ with two clauses, $\varphi_{c_1}$ and $\varphi_{c_2}$, each comprised of node literals $l_n$ and message literals $l_m$.

$$
\varphi = \varphi_{c_1} \land \varphi_{c_2}
$$

$$
\varphi_{c_1} = l_{m_1} \lor l_{n_1} \lor l_{n_2}
$$

$$
\varphi_{c_2} = l_{m_2} \lor l_{n_1} \lor l_{n_3}
$$

Let’s assume that one crash failure and one message omission is allowed. Furthermore, $l_{m_1}$ is assumed to be omitted from a previous iteration. Then, in accordance with the algorithm described above, we would make the following transition (we can use unit propagation for the unit clause, but it has been retained for illustratory purposes):

$$
\varphi = \varphi_{c_1} \land \varphi_{c_2} \land \varphi_{c_3} \land \varphi_{c_4} \land \varphi_{c_5} \land \varphi_{c_6} \land \varphi_{c_7} \land \varphi_{c_8} \land \varphi_{c_9} \land \varphi_{c_{10}} \land \varphi_{c_{11}} \land \varphi_{c_{12}}
$$

$$
\varphi_{c_1} = l_{m_1} \lor l_{n_1} \lor l_{n_2}
$$

$$
\varphi_{c_2} = l_{m_2} \lor l_{n_1} \lor l_{n_3}
$$

$$
\varphi_{c_3} = -l_{m_1}
$$

$$
\varphi_{c_4} = l_{n_1} \lor \text{nv}(l_{n_1})
$$

$$
\varphi_{c_5} = -l_{m_1} \lor -\text{nv}(l_{n_1})
$$

$$
\varphi_{c_6} = l_{n_2} \lor \text{nv}(l_{n_2})
$$

$$
\varphi_{c_7} = -l_{n_2} \lor -\text{nv}(l_{n_2})
$$

$$
\varphi_{c_8} = l_{n_3} \lor \text{nv}(l_{n_3})
$$

$$
\varphi_{c_9} = -l_{n_3} \lor -\text{nv}(l_{n_3})
$$

$$
\varphi_{c_{10}} = \text{nv}(l_{n_1}) \lor \text{nv}(l_{n_2})
$$

$$
\varphi_{c_{11}} = \text{nv}(l_{n_1}) \lor \text{nv}(l_{n_3})
$$

$$
\varphi_{c_{12}} = \text{nv}(l_{n_2}) \lor \text{nv}(l_{n_3})
$$

Note however, that if a program fails to meet the correctness specification by one omission or crash, then all additional cuts are superfluous. Thus, the solutions of interest are the minimal solutions, i.e, solutions that are not contained within other solutions. More formally, for the solution set $S$, a solution $s \in S$ is minimal if $\nexists s'$: $s' \subseteq s$, for some other solution $s' \in S$. As such, the solutions are passed to a procedure that removes all super sets for each solution after being retrieved from the SAT-solver; the procedure of which is depicted in Algorithm [1]. In the above example, the minimal solution would be \{l_{n_1}\}, as the dummy variables $\text{nv}(l_{n_2})$ and $\text{nv}(l_{n_3})$ are discarded.

### 5.2 Logical Time

A critical feature of performing LDFI is to incorporate the notion of logical time. Keeping track of the logical time is trivial for a run of program without any injections: increment the clock for every witnessed message. Thus, we can uniquely identify each message with the time in which it occurred. With the introduction of message injections however, the task of keeping track of the logical clock is encumbered.

---

5.2. LOGICAL TIME

Consider Figure 5.1 illustrating a typical interaction with message omissions. A program, consisting of three actors, A; B and C, is initially run with no interference. We witness two messages being passed, one between actors A and B, and the other between actors B and C, resulting in an initial boolean formula with a single clause: $M(A, B, 1) \lor M(B, C, 2)$. With no specific information, an initial assumption that the programs correctness relies on a message being successfully delivered to C is made. As such, we start our backward reasoning and attempt to omit the message between B and C. As a direct consequence, we witness different events. The parser reads logs stating that three messages were passed: one between A and B, another between B and some other actor $R_1$, and finally one between actors $R_1$ and C. The omitted message between B and C is not logged, and is therefore not part of the lags. Therefore, a (false) inference that the message between B and $R_1$ took place at logical time 2 is made, and as a consequence all messages taking place afterwards have their time inferred to “correct” time - 1. As illustrated in the figure, this pattern continues for future omissions in different runs.

The resulting time discrepancies have to be consistent in order for the analysis to work properly. In essence, the controller attempts to omit messages based on the injection hypotheses provided by the solver. At the last iteration in the example, the controller has the following injection hypothesis: \(M(B, C, 2), M(R_1, C, 3)\). The controller has to omit two messages coming from different clauses, and as such, they have the logical time that correspond to the events that occurred in their respective program iteration. If the controller would naively increment the logical time for each message it witnessed: the logical time would be 4 at the time $R_1$ sends a message to actor C. The controller would consequently not find $M(R_1, C, 4)$ among the injections. We have now arrived at an inconsistency.

Algorithm 2 solves the problem with inconsistent clocks across the clauses within the formula. In essence, whenever the controller witnesses a message: Algorithm 2 is called with the sender actor, recipient actor, all clauses within the formula and a map that keeps track of the current clock for each clause in the formula. The algorithm then updates the

---

Algorithm 1

```
1: procedure getMinimalSolutions(current, allsolutions)
2:     if current = ∅ then
3:         return ∅
4:     else
5:         tail ← current(1) \cap current
6:         if \(\forall sol \in allsolutions : sol \neq current(1) : \exists sol \subseteq current(1)\) then
7:             return getMinimalSolutions(tail, allsolutions)
8:         else
9:             return current(1) \cup getMinimalSolutions(tail, allsolutions)
10:     end if
11: end if
12: end procedure
```
clock map according to the message that is witnessed by the controller. For each clause, it initially checks whether the message exists in the clause. If it does, then the logical clock is incremented for that clause in the clock map. If the message does not exist in that clock, then the algorithm checks whether the sender has an activity at \( curTime + n \), where \( curTime \) refers to the currently recorded time for that clause in the clock map and \( n = EOT \) (end of time) - \( curTime \). Essentially, if the actor has indeed been active at some later time, then it must mean that message passing new to this particular clause has taken place that has lead to to this actor being active again. More specifically, another route leading to this particular actor receiving the message has taken place. Therefore, the logical time is updated to the sender actors first activity, \( act \) s.t. \( act > curTime \). If the sender active is never active again, the clock is not updated.

5.3 Failure Specification

Given the disparity in logical time across the clauses within the formula, a single failure specification is no longer sufficient as a global constraint. Therefore, the current notion of a failure specification needed to be extended to a Failure Specification Box (henceforth abbreviated as \( fspecbox \)). Similar to the clause clock map (described in the previous section) keeping track of the logical time for each clause, the \( fspecbox \) consists of a failure specification map that maps all of the clauses to their respective failure specification. As
Figure 5.1: Some of the many possibilities that can arise with injections. The letters represent messages and the cross represent message omissions.
the EOT is different across the clauses, as a direct consequence from the disparity in logical
time across the clauses, said extension is paramount to the correctness of the analysis as
the EFF can no longer be absolute for all clauses, but must be relative to each clause.
Thus, the constraint that the EOT must be greater than EFF can only hold if the EOT
is specific for each clause. Moreover, the fspecbox keeps track of the globalCrashes, i.e.,
the maximum number of global crashes allowed. While the Crashes defined in the original
failure specification constraints the maximum number of crashes allowed for each clause
(and by extension each iteration), the globalCrashes also encompasses assumed crashes
from previous iterations. More specifically, if a clause is added in the formula as a direct
result of some previous crash(es) for some previous clauses, then the number of crashes
allowed for the newly added clause’s failure specification must be smaller than the maxi-
imum crashes allowed for said previous clauses. Lastly, the fspecbox defines the initialEff
which simply refers to the initial EFF set by the user or the sweep mechanism.

With the above definitions, the fspecbox must set up some well defined constraints.
First, we must keep the constraint that the EOT must be greater than the EFF for all
failure specifications. Given that EOT, EFF ∈ fspec, we define it as:

∀fspec ∈ fspecMap : EOT > EFF

(5.1)

Second, we must add a constraint on the number of crashes of respective clause’s failure
specification. As such, for globalCrashes ∈ fspecbox and assumedCrashes, Crashes ∈ fspec,
we must uphold:

∀fspec ∈ fspecMap : globalCrashes ≥ num(assumedCrashes) + Crashes

(5.2)

Third, the globalCrashes must be greater or equal to all of the total assumed crashes
across all failure specifications. Thus, we assert:

globalCrashes ≥ num(⋃ \forall fspec assumedCrashes).

(5.3)

5.4 Evaluator

The concrete evaluator can be described as the only stateful entity of LDFI. It controls
the process of finding and injecting failures, i.e., running the programs, extracting the
formula, solving the formulas and then running the programs with the failure injections.
This process can be broken down into two major components, the backward and forward
step, which are shown in greater detail in the following sections.

5.4.1 Backward Step

The backward step is a core step of the LDFI analysis, as it converts the outcome of a given
run of a program to a CNF-formula, which is thereafter passed to a SAT-solver in order
to obtain hypotheses. It is procedural and stateless, i.e., it performs the same operations when called upon, regardless of the state of the LDFI analysis. It is therefore important that the backward step is only called at the correct step of the evaluation, as it performs its procedure on the last run of the program. The procedure is depicted in Algorithm 3.

After the run of a given program has terminated: it is logged, parsed and formatted as described in previous sections. The formatted behavior of the program is then encoded in CNF, which subsequently is passed to the SAT4J solver, giving the solutions, representing the hypotheses (possible failure injections) for the next run of the program, which are ultimately then returned.

Algorithm 3

```
1: procedure backwardStep(formula, fspecbox, fpm, hypothesis)
2:     input ← get input from logs
3:     formattedLogs ← parse(input)
4:     newClause, existsInFormula ← CNFConverter.convert(formattedLogs, formula)
5:     updatedFSpecBox ← ∅
6:     if existsInFormula then
7:         updatedFSpecBox ← fspecbox
8:     else
9:         fSpecForClause ← createFSpecForClause(newClause)
10:        updatedFSpecMap ←
11:            fspecmap ∪ (newClause.Id → fSpecForClause)
12:        updatedFSpecBox ←
13:            fspecbox(initialEff, globalCrashes, updatedFSpecMap)
14:     end if
15:     hypotheses ← SAT4JSolver.solve(formula, updatedFSpecBox)
16:     return (hypotheses, updatedFSpecBox)
17: end procedure
```

5.4.2 Forward Step

The forward step is most easily described as the step that runs a program with a given injection hypothesis. Naturally, the first run of the program is done with no injection hypothesis: the good outcome is obtained with the objective of extracting the lineage and perform the backward step. The forward step is then run with the hypotheses given by the backward step with the purpose of collecting real solutions, i.e., injections that lead to a violation of the correctness specification. The injections are given to the controller to be inserted at the run-time execution of the program. The program is then subsequently run with these injections. The forward step then returns the correctness of the program, i.e., true in case the program held under the failure injections or otherwise false, in case the correctness specification was violated. Due to space limitations in combination with the triviality of the forward step, the algorithm that implements it has been omitted.
5.4.3 Concrete Evaluator

The concrete evaluator — which is broken down into three separate algorithms — makes use of both the forward and the backward step. Algorithm 4 takes as input a program (that includes a list of messages that are not part of the analysis, `freepassmessages`), a boolean formula, the `fspecbox` and all previously tried injection hypotheses. Initially, the evaluator function is called where the empty sets represent no hypotheses or solutions. If the evaluator does not find any solutions (i.e., fault injections that violates the correctness specification), then there is a check to see whether there exists a failure specification that could have its `EFF` or `Crashes` incremented, with a recursive call to the concrete evaluator, if such exists.

Algorithm 5 begins with updating the current injection hypothesis with the hypotheses already assumed in the failure specification and adding the current hypothesis to the tried hypotheses set. It proceeds to call the forward step with the hypothesis. If the program’s correctness specification is violated, the hypothesis is deduced to be a real solution. Otherwise, the failure specification is updated with the current hypothesis, such that it is already assumed for the next iteration. Thereafter, the backward step is called, updating the formula with the newly generated successful outcome which is subsequently passed to the solver such that additional hypotheses are retrieved. The procedure then continues by evaluating each of the newly retrieved hypotheses.

The hypotheses are each evaluated by the function illustrated in Algorithm 6. If there are no hypotheses to evaluate, the current solutions and all tried hypotheses are returned. Otherwise, the first hypothesis is evaluated if and only if it has not already been tried in a previous iteration. The results of the evaluation is then added to the current sets, and the iteration continues recursively for each hypothesis in the hypotheses list.
Algorithm 4

1: procedure CONCRETE_EVALUATOR(prog, formula, fspecbox, triedHypos)
2:   (solutions, resTriedHypotheses, resFspecbox) ←
3:       evaluator(prog, formula, fspecbox, triedHypos, ∅, ∅)
4:   if solutions = ∅ then
5:       allTriedHypo ← triedHypos ∪ resTriedHypotheses
6:       if ∃fspec ∈ fspecmap : EFF < EOT - 1 then
7:           updatedFspecbox ← resFspecbox with incremented EFF
8:           return concreteEvaluator(prog, formula, updatedFspecbox, allTriedHypo)
9:       else if ∃fspec ∈ fspecmap : Crashes + assumedCrashes < globalCrashes then
10:          updatedFspecbox ← resFspecbox with incremented Crashes
11:          return concreteEvaluator(prog, formula, updatedFspecbox, allTriedHypo)
12:       else
13:           return ∅
14:   end if
15: end procedure

Algorithm 5

1: procedure EVALUATOR(prog, formula, fspecbox, triedHypo, hypo, sols)
2:   updatedHypothesis ← hypo ∪ fspec.cut ∪ fspec.crashes
3:   incTried ← triedHypo ∪ hypo
4:   if forwardStep(prog, updatedHypothesis, formula) = true then
5:       newHypos, updatedFspecbox ← backwardStep
6:       if newHypos ≠ ∅ then
7:           return evalHypotheses(prog, formula, updatedFspecbox, incTried, newHypos, sols)
8:       else
9:           return (∅, incTried, updatedFspecbox)
10:   end if
11:   else
12:       return (sols ∪ (hypo → fspec), incTried, fspecbox)
13:   end if
14: end procedure
Algorithm 6

1: procedure evalHypotheses(prog, formula, fspecbox, triedHypos, hypos, sols)
2:   if hypos = ∅ then
3:     return sols, triedHypos, fspecbox
4:   else
5:     tail ← hypos(1) ∩ hypos
6:     if hypos(1) ∈ triedHypos then
7:       (resSolutions, resTriedHypotheses, resFSB) ←
8:         evaluator(prog, formula, fspecbox, triedHypos, hypos(1), sols)
9:       sumTriedHypos ← resTriedHypotheses ∪ triedHypos
10:      allSols ← sols ∪ resSolutions
11:      return evalHypotheses(prog, formula, resFSB, allTriedHypos, tail, allSols)
12:   else
13:     return evalHypotheses(prog, formula, fspecbox, triedHypos, tail, sols)
14:   end if
15: end procedure
Chapter 6

LDFI for Akka

Recall the toolkit Akka, which is based on the actor model, from section 2.5. In Akka, all of the communication between the actors is done by explicit message passing: an actor can therefore not be influenced (e.g., mutate its internal state or change its behaviour) by a different mean. Thus, the behavior of a given Akka program can be deduced by analyzing the messages sent within the actor system after it has finished executing. This is crucial to the objective of this thesis, as this would imply that by scrutinizing the execution of the program, its outcome can be inferred, and more importantly, the cause of that particular outcome can be known.

The problems that are addressed in the subsequent sections mainly focus around resolving the following subproblems, whose solutions are paramount to successfully enabling LDFI on Akka programs:

i) Log the execution traces of an Akka program and extract the data lineage from them.

ii) Control the run-time execution of an Akka program.

iii) Given an arbitrary Akka program, find a way to rewrite it so that it is possible to enable LDFI.

This chapter includes detailed descriptions of the novel general conceptual framework which were applied on the actor-based framework Akka, in addition to new solutions to above subproblems. In section 6.2 the first subproblem is addressed by showing how Akka programs can be logged and what can be deduced from such logs. Section 6.3 includes details on the methods used to parse these logs, and thus how the data lineage can be extracted. Section 6.4 details specifics on how the second subproblem is addressed. Lastly, section 6.5 is dedicated to tackling the third subproblem.

6.1 Simple-Deliv in Akka

Recall the naive simple best-effort broadcast Dedalus program, simple-deliv, from section 2.4. Given a failure specification and a starting fact, Molly could find fault injections
that would violate the post-condition. Namely, Molly could inject faults such that there would exist a node that has not logged the messages of its neighbors after the program had terminated. In Dedalus, the nodes and their respective neighbors were represented with simple rules. In the corresponding implementation in Akka however, each node process represents an actor. In the case of simple-deliv, we would have three actors; A, B and C, where A starts the initial broadcast by sending a message to its neighboring actors. Figure 6.1 illustrates the Akka equivalent of simple-deliv. Thus, when A broadcasts a message, A, B and C should all have a log entry of that message when the program has finished executing, provided that they are neighbors.

Listing 6.1 shows the implementation of the Node actors, i.e., the nodes that broadcasts and logs messages. The Node class — extends the Akka actor trait. If the message received is a Broadcast object, then the payload is logged. Otherwise, the message received is a Start object, and as such, some payload is broadcasted to all of the receiving actors neighbors. Afterwards, the payload is logged, and lastly the actor system terminated.

Listing 6.2 implements the logs and relations. The relations are stored in a global singleton object with a single field consisting of an immutable map. The actor names — which are fit to be keys, since they are unique for each actor — acts as keys that are mapped to a set of actor references; actors that are neighbors to the concerned actor. The logs are implemented in a similar fashion, apart from the set that is within the map being mutable as the actors need to append logs that they receive. This carries no risk of race conditions as every actor only mutate its own set of logs.
6.2. Data Lineage in Akka

Recall from section 2.4 — that the state of the art implementation of LDFI — Molly takes as input distributed programs written in Dedalus. Moreover, Molly performs an analysis that finds all possible failure scenarios for such programs. These failure scenarios are found, mostly by leveraging the fact that all computations in Dedalus programs are derived from deductive or inductive rules. Furthermore, a Dedalus program is simply data or relationships among the data elements. For that reason, obtaining the data lineage in Dedalus programs is trivial when compared to obtaining it for languages in other programming paradigms. For instance, it is not possible to deduce with certainty how a given Scala program behaves at run-time by analyzing it at compile-time. Moreover, if it is also distributed and concurrent, this task becomes increasingly difficult. A prerequisite of applying LDFI to a given distributed program is that it is possible to extract the data lineage. If the distributed program is implemented using the actor model, then the communication between the minimal computational entities, the actors, is done entirely by explicit message passing. Therefore, the key to extracting the data lineage is to analyze the messages. In Akka, this can be done by logging the communication (the messages) within the actor system. After the messages has been logged, they can be analyzed and as a result, their origin and life cycle can be determined. In other words, the data lineage can be extracted from the logs. Section 6.2.1 and 6.2.2 includes specifics of the necessary steps.
CHAPTER 6. LDFI FOR AKKA

Listing 6.3: Logging pattern used to extract vital information about actor activity.

```text
%$ level%[thread] %X(akkaSource) $-% $msg%n
```

Listing 6.4: Part of the resulting logs from running an Akka implementation of simple-deliv

required to extract the logs from an Akka program.

6.2.1 Logging Configuration

Logging in Akka is printed to STDOUT by default, but comes with the possibility of using a custom or Simple Logging Facade for Java (SLF4J) logger [15]. Keeping the default and printing the logs to STDOUT would not be fruitful, since the purpose of logging the execution traces was to extract the data lineage: the logs need to be persisted. In order to persist the logs, the default configuration needs to be modified so that the logging is directed to, in this case, a file. Furthermore, in order to obtain detailed logging, the configuration is changed to "DEBUG" level logging. Moreover, the format of the logging is highly customizable and was thus adjusted to fit the needs of this project. The logging pattern used to extract the required information is illustrated in Listing 6.3. In the pattern, `level` refers to the level of logging, e.g., "INFO" or "DEBUG", while `thread` refers to the current dispatcher that the activity was processed on. The sender actor (with full path) is given by `akkaSource`, whereas `msg` logs the message sent from said actor together with the full path of the receiving actor. Listing 6.4 shows parts of the resulting logs (the other parts have been omitted) after running an Akka equivalent of simple-deliv. It is trivial to tell that the above pattern corresponds to the resulting logs. The level of the logging is at "DEBUG", dispatcher 5 was used to run the program, and two actors, B and C, received the message “Broadcast(Some payload)” from some actor A.

6.2.2 Actor Logging

Modifying the configurations is necessary but not sufficient to acquire logs from a run of an Akka program. The logging in Akka is in fact two-fold: first, the debug level logging has to be enabled in the configurations and second, the actor classes must be extended with the `ActorLogging` trait while also setting the receive method to the `LoggingReceive` method 1. Listing 6.5 illustrates this extension. First, the `Node` class is extended with

---

1This is one way of enabling logging in the actors, there are others — but to my knowledge — more tedious ways of doing it.
the ActorLogging trait. Second, the receive method is set to the LoggingReceive method to ensure that all messages received by the Node actor are properly logged. As a consequence, all messages received to an instantiated Node actor are logged by the SLF4J logger.

6.3 Parsing

The information needed to perform the LDFI analysis is dependant upon knowing three key components of every activity. First, given a message, the sender actor must be known, and second, the recipient of said message must be known. Third, it is of paramount importance to know the logical time in which said activity took place. Note that it is not important what is being sent in order to perform the analysis, but rather “who” sent and received the message and when it was sent. Thus, the resulting logs from running a given program must be parsed and formatted to facilitate the procedure of retrieving the key components of every activity. In order to structure the logs in a meaningful way based on the three key components — excluding the message —, the setup illustrated in Listing 6.6 was implemented. FormattedLogs is simply a list of Rows, which in turn is comprised of the above components.

Recall that LDFI uses a synchronous execution model: it evaluates an asynchronous distributed program in a synchronous simulation. As a result, the following inference can be made: the activities in the program took place in the order in which they were logged. This is crucial when setting the logical time for every activity.

By making use of this very fact, we implement a parser that iterates over every line in the logs. The above logging pattern is then leveraged using regular expressions to retrieve the sender, recipient and message. Moreover, for each message witnessed, the logical clock is incremented. Finally, after necessary information has been retrieved, a Row is constructed to be added to the list of Rows contained within the FormattedLogs.

After running a given Akka program, on a single dispatcher together with the above

---

2It is not important for the analysis, but it makes it easier for the user to understand what is going on.
logging configuration, its behavior can be deduced by analyzing the formatted logs. For instance, by running the Akka implementation of simple-deliv, the following `FormattedLogs` would be retrieved: `FormattedLogs(List(Row(A,B,1, Broadcast(some payload)), Row(A,C,2, Broadcast(some payload)))`. It is therefore trivial to conclude that the Akka program consisted of the message passing between three actors, A; B; C, where A sent two messages to the others at logical time 1 and 2 respectively.

## 6.4 Controller

In order to implement a lineage-driven fault injector for Akka programs, it must either be possible to make an analysis of said programs to the extent of which it would be possible to deduce all possible run-time outcomes, or else find a way to control the run time execution. For programs written in general-purpose, object-oriented languages, the former approach can be reduced to the halting problem and is therefore not feasible. More specifically, it is impossible to deduce how some program or function, or in this specific case, actor, reacts to all possible inputs at compile time. Therefore, the latter approach was chosen by default.

Following from above conclusions, we model a controller that controls the run-time execution of a given Akka program by intercepting all messages that passes within the actor system. Thus, the controller acts as a gateway between the communication for a given pair of actors. More concretely, before sending a message, all actors communicate with the controller and awaits a permission to pass a message. If the controller rejects a message, the actor proceeds to perform some other activity. Thus, the message cuts or crashes are not actually performed by killing the processes behind the actors, but rather simulated by forcing such actors to be idle by continuously prohibiting any activity they attempt to engage in.

Consider Algorithms 7 and 8 implementing above algorithm. The `greenLight` function interfaces with the Akka actors. In essence, before sending a message, each actor passes their own and the sender actor name in addition to the message they are sending. The `greenLight` function retrieves the set of injections and the updated clock map for all clauses, and proceed to check if there exists a message that is not allowed for each given clauses’s respective time. If such a message exists, then the resulting boolean is inversed: in other words, the permission is rejected.

## 6.5 Program Rewrite

A prerequisite of performing LDFI on Akka programs is that they interact with the external controller described in the previous sections. In other words, the actors are not free to perform any activity without the explicit permission of the controller. Naturally, Akka programs are not written such that they can interact with the controller. Furthermore, it

---

3The actual message bears no importance to the analysis of the program, but is saved for illustration and debugging purposes.
6.5. PROGRAM REWRITE

Algorithm 7

1: procedure ISINJECTED(sen, rec, msg, time, injections)
2: \text{message} \leftarrow \text{MessageLit}(\text{sender}, \text{rec}, \text{time})(\text{msg})
3: \text{senderCrashed} \leftarrow \exists n \in \text{injections}: n.\text{sender} = \text{sen} \land n.\text{time} \leq \text{time}
4: \text{recipientCrashed} \leftarrow \exists n \in \text{injections}: n.\text{recipient} = \text{rec} \land n.\text{time} \leq \text{time}
5: \text{msgCut} \leftarrow \exists \text{message} \in \text{injections}
6: \text{return } \text{senderCrashed} \lor \text{recipientCrashed} \lor \text{msgCut}
7: end procedure

Algorithm 8

1: procedure GREENLIGHT(sen, rec, msg)
2: \text{clauseClock}, \text{injections} \leftarrow \text{get injections and clause clock map}
3: \text{newClauseClk} \leftarrow \text{manageClock}(\text{sen}, \text{rec}, \text{msg}, \text{clauses}, \text{clauseClock})
4: \text{if } \text{injections} \neq \emptyset \text{ then}
5: \text{existsInjection, injection} \leftarrow \exists t \in \text{newClauseClk} : \text{isInjected}(\text{sen}, \text{rec}, \text{msg}, t, \text{injections})
6: \text{if } \text{injection} \neq \emptyset \text{ then}
7: \text{injections} \leftarrow \text{injections} \cap \text{injection}
8: \text{end if}
9: \text{return } \text{!existsInjection}
10: \text{else}
11: \text{return false}
12: \text{end if}
13: end procedure
is unrealistic to expect the designers and developers for Akka programs to rewrite their program such that it interacts with the controller. Another prerequisite of performing LDFI on Akka programs is that they are logged. Otherwise, no inferences about the system’s behavior can be made. Moreover, it is necessary that every single message that is passed is logged or else the analysis is made on false and insufficient information, rendering the analysis useless. Again, it would be cumbersome to force developers to rewrite their program in a very specific manner for the sole reason of making this analysis possible. The solution is found in Scalafix, a rewriting tool for Scala programs.

Scalafix provides an API for the user to write rewriting rules for any given Scala program. Scalafix does this by hooking into the Scala compiler after the source code is parsed, and lets the user invoke their rules directly on to the abstract syntax tree (AST). By modifying the AST directly, it is possible to embed logging as well as the controller interaction into Akka programs that initially do not contain those.

6.5.1 Embedding Logging

The logging of Akka programs can be split into two major components. The first is to extend the Actor trait with the ActorLogging trait, in addition to ensuring that the receive methods are set to the LoggingReceive method. Moreover, Akka has over 10 different modules in their toolkit, and depending on which module is used, and especially how it is used: the rewrites can differ. The most common usages of Akka actors (as shown in the examples in the previous sections) is the use of the Actor trait and the standard receive method, but sometimes users of Akka might decide to override the built-in receive method with their own. Moreover, for some modules, for instance the Akka Persistence module, one might use the receiveCommand declared as a val instead of the standard def. What they have in common however is that they are both of the Receive type, and this very fact plays a large role in the rewrite implementation.

Listing 6.7 implements the rule of adding LoggingReceive using the Scalafix API. To reiterate, Scalafix lets the user hook into the AST directly, and as such it is possible to collect on all of the subtrees within the AST and pattern match on those of interest. In

```scala
def addExtendsWithActorLogging(ctx: RuleCtx): Patch = {
  ctx.tree.collect {
    case parent @ Defn.Class(_, _, _, _, template) =>
      if(isActorClassWithNoLogging(template)){
        template.inits.lastOption match {
          case Some(parent) => ctx.addRight(parent, " with ActorLogging")
          case None => Patch.empty
        }
      } else Patch.empty
  }.asPatch
}
```

Listing 6.7: Scalafix rule for extending the Actor trait with ActorLogging.
6.5. PROGRAM REWRITE

```scala
def addLoggingReceive (ctx: RuleCtx): Patch = {
  ctx.tree.collect {
    case Defn.Def(_, name, _, _, tpe, body) if (name.value == "receive" || tpe.toString == "Some(Receive)") && !hasLoggingReceive(body) =>
      ctx.addLeft(body, "LoggingReceive ")
    case Defn.Val(_, _, tpe, body) if tpe.toString == "Some(Receive)" && !hasLoggingReceive(body) =>
      ctx.addLeft(body, "LoggingReceive ")
    case _ => Patch.empty
  }.asPatch
}
```

Listing 6.8: Scalafix rule for rewriting the receive methods and vals.

In this case, only the classes that are extended by the `Actor` trait and is not already extended by the `ActorLogging` trait are of interest. If such a class is found — regardless of how many traits are already extended —, the `ActorLogging` trait is added.

Consider Listing 6.8 implementing the rule for setting `LoggingReceive`. As previously mentioned, there are two points of interest: first, all receive methods, and second, all of the `vals` that might be used for receiving messages. In the first case, it is possible to simply look for all methods that are named “receive”. For the methods and `vals` that are not named “receive” however, it is essential that they have an explicitly declared type. If they have an arbitrary name and rely on Scala’s type inference then they are not be found by this rule, the consequences of which is discussed in the ensuing chapters.

6.5.2 Incorporation of Fault Injections

The validity of the external controller is based on the information it receives from the Akka program at run time. It is therefore a necessity that the information being passed to the controller is sufficient in order for it to make a valid decision. For this reason, every time a message is sent, the controller needs to know the name of the sender and receiving actor and the message that is being passed. In Akka, messages are sent by using the bang (“!”) operator or the built-in `tell` method.

Consider Listing 6.9 illustrating the goal of the rewrite of the original code depicted in Listing 6.10. Essentially, wherever the bang operator or the `tell` method is found in the source code, they are encompassed by an if statement with a call to the controller by the rewrite rules. The parameters are the sender actor, recipient actor and the message respectively; but how they are extracted vary based on where the bang operator or the `tell` method are found. If they are found within a class that extends the `Actor` trait, then the `self` and `sender` actor references are in scope and can thus be used. In the event where the they are found outside an Actor trait, it is assumed that the messages are being sent from the actor system. The actor system is denoted as “deadLetters” in Akka, and as such, the string “deadLetters” represents the sender actor whenever the actor system sends messages.
class NodeActor(helpActor: ActorRef) extends Actor with ActorLogging {
  def receive = LoggingReceive {
    case "hello" =>
      if (Controller.greenLight(self.path.name, helpActor.path.name, "hello")) {
        helpActor ! "hello"
      }
    case "howdy" =>
      if (Controller.greenLight(self.path.name, helpActor.path.name, "howdy")) {
        helpActor.tell("howdy", self)
      }
  }
}
class SimpleDeliv {
  if (Controller.greenLight("deadLetters", nodeActor.path.name, "hello")) {
    nodeActor ! "hello"
  }
  if (Controller.greenLight("deadLetters", nodeActor.path.name, "hello")) {
    nodeActor.tell("hello", system.deadLetters)
  }
}

Listing 6.9: Example program after Scalafix rewrite.

class NodeActor(helpActor: ActorRef) extends Actor {
  def receive = {
    case "hello" => helpActor ! "hello"
    case "howdy" => helpActor.tell("howdy", self)
  }
}
class SimpleDeliv {
  nodeActor ! "hello"
  nodeActor ! "howdy"
}

Listing 6.10: Example program prior to Scalafix rewrite.
def addControllerGreenLight(ctx: RuleCtx): Patch = {
  ctx.tree.collect {
    case Template(_, inits, _, stats)
    if isExtendedWithActor(inits) && !hasGreenLight(stats) =>
      getControllerPatch(ctx, "self", stats)
    case Template(_, inits, _, stats) if !hasGreenLight(stats) =>
      getControllerPatch(ctx, "\deadLetters\", stats)
  }.asPatch
}

Listing 6.11: Scalafix rule to incorporate the controller

Listing 6.11 depicts the implementation of rewrite rule to incorporate the external controller. The `getControllerPatch` method has been omitted due to space limitations, but its functionality is trivial. In essence, it collects on all of the terms where the bang operator or the `tell` method is used, and proceeds to construct an if statement enwrapping the term and ultimately replaces the original tree with a new tree which includes the newly constructed if statement.
Chapter 7

Results

In this chapter, the results of using the LDFI approach to find bugs in distributed systems written in Akka, henceforth abbreviated as ldfi-akka is presented. In section 7.1, ldfi-akka is used to analyze the Akka equivalent of simple-deliv, whereas section 7.2 is dedicated to evaluating retry-deliv. In sections 7.3, 7.4 and 7.5, ldfi-akka is employed on existing programs created as illustratory samples for Akka, whereas section 7.6 details the interaction between ldfi-akka and a consistency protocol.

7.1 Simple-Deliv

The Akka equivalent version of simple-deliv was implemented in approximately 200 lines of code, as an initial proof of concept for ldfi-akka. To reiterate, simple-deliv consists of three actors, one of which sends two messages to the others. The correctness specification is that all messages must be logged by each actor separately. In this particular case, actor A sends the message “Broadcast(Some payload)” to actors B and C. Thus, after the actor system terminates, all actors should have logs with said message.

Ldfi-akka perform its analysis on simple-deliv in Akka and immediately find both minimal solutions, with \( \text{EFF}=2 \), that violates the correctness specification\(^1\). In order to determine whether each run of the program is correct, additional methods need to be added that check whether any violation occurred in each run. The verifyCorrectness and missing_log methods illustrated in Listing 7.1 implements the correctness check. The methods functionality is trivial: each actor’s logs is compared with its neighbors. If a log is defined as missing, i.e., some actor has it saved but it neighbor does not, then the method returns false.

Consider Figure 7.1 depicting the omitted messages that result in the violation of the correctness specification. Note that crashing one of the recipient actors would also lead to a violation, but since a message failure is considered to be more likely than a crash failure: the crash failures are omitted in case message failures are sufficient to cause violations.

\(^1\text{EFF}=1\) is indeed sufficient to violate the correctness specification, but \( \text{EFF}=2 \) was used to be consistent with simple-deliv in Dedalus.
def verifyCorrectness(): Boolean = {
  actors.map { actor =>
    val node = actor.path.name
    Logs.logs.get(node) match {
      case Some(logs) =>
        logs.map { log =>
          missing_log(log.pload, node)
        }.exists(b => !b)
      case None => false
    }
  }
}

def missing_log(pload: String, node: String): Boolean = {
  Relations.relations.get(node) match {
    case Some(neighbors) =>
      neighbors.map { neigh =>
        Logs.logs.get(neigh.path.name) match {
          case Some(logs) => !logs.contains(Log(pload))
          case None => true
        }
      }.exists(b => b)
    case None => false
  }
}

Listing 7.1: Method needed to verify the correctness of *simple-deliv*.

Figure 7.1: Depiction of the omitted messages in *simple-deliv* resulting in violation of the correctness specification.
7.2 Retry-Deliv

In section 7.1, a naive best-effort broadcast protocol, simple-deliv was subjected to an evaluation by ldfi-akka. As previously mentioned, ldfi-akka quickly found failure injections that violated the correctness specification of the protocol. Using the same reasoning as in the paper written by Alvaro et. al [3], one can use the fact that Molly, and in this case ldfi-akka, exposes a potentially weak protocol weaknesses to strengthen it. By realizing that simple-deliv fails since the messages are only naively broadcasted once, one can improve the protocol to make repeated broadcast attempts. Thus, with the introduction of repeated attempts, a single broadcast failure no longer causes the correctness specification to be violated. The protocol that repeats broadcasts, retry-deliv, has the same correctness specification: all messages must be logged by all nodes.

In Dedalus, it is possible to express the notion of logical time with the help of inductive facts and rules. As such, a rule or a fact can, in theory, be expressed such that they incorporate the notion of infinity. In Akka, it is not possible to model such behavior. Therefore, the Akka equivalent of retry-deliv, which repeats broadcast attempts, is implemented with a pre-defined limit of maximum number of attempts. For simplicity, the number of repeated attempts is set to 1: we call it retry-deliv-1.

Similar to simple-deliv, ldfi-akka initiates the analysis by generating a clause, representing a successful outcome in the boolean formula. Starting backwards, ldfi-akka then proceeds to omit the last message sent between actor A and C, as illustrated in Figure 7.2. As opposed to simple-deliv, a single omission is not sufficient to violate the correctness specification for retry-deliv-1, as the node has already previously received the broadcast. Using the same reasoning, ldfi-akka tries omitting both messages sent to actor C. Now, the correctness specification is violated since actor C never receives the broadcast, as actor A only makes a single repeated attempt to broadcast.

Therefore, ldfi-akka concludes that retry-deliv-1 is capable of withstanding a single failure injection, and that it is possible to violate its correctness specification by allowing for two message omissions, i.e., setting $EFF=2$ in the failure specification.

7.3 Hello World

The creators of Akka have a number of illustratory samples of the Akka API in a Github repository. The samples makes use of the different API in a straightforward manner, and as such are ideal initial targets for ldfi-akka. The hello-world sample consists of 4 actors (including the actor system) implemented in roughly 60 lines of code. In essence, a hello world actor initiates the message passing by creating a greeter actor and sending it an initial case object, Greet. The greeter actor then subsequently responds with a Done (another case object) after printing “Hello World” to the console. After receiving the Done message, the hello world actor stops itself. The terminator actor, implementing a

---

2https://github.com/akka/akka-samples
Figure 7.2: Omissions that violate the correctness specification of retry-deliv-1.
7.4. PERSISTENCE

superiority strategy on the hello world actor, responds to the hello world actor stopping by terminating the actor system. The entire interaction can be summarized into three messages: the initial Greet message passed from the hello world actor to the greeter actor; the subsequent Done message from the greeter to the hello world actor; and lastly the Terminated (an inbuilt message in Akka) from the hello world actor to its supervisor, the terminator actor.

The additional code added in order to make this sample viable for ldfi-akka consisted of 4 lines, depicted in Listing 7.2. The verifyCorrectness method is invoked after every run of the program to verify the program’s correctness. Essentially, the contents of the logs are checked: in case the Greet message is absent from the logs, then it must mean that the greeter could not possibly have printed ”Hello World” to the console.

Ldfi-akka encodes the system’s behavior leading to a successful outcome in a boolean formula: M(helloworld, terminator, 3) ∨ M(greeter, helloworld, 2) ∨ M(helloworld, greeter, 1). It is trivial to see that the minimal solutions are each of the literals. In contrast to simple-deliv however, all three of the hypotheses do not lead to a violation of the correctness specification. Only the omission of the initial message from the hello world actor to the greeter actor leads to a true solution, illustrated in Figure 7.3. Ldfi-akka first tries to omit the message sent from the helloworld actor to the terminator, then proceed to omit the message from the greeter to the helloworld actor. The next step is to omit both at the same time. Realizing that this does not result in a violation of the correctness specification, ldfi-akka is forced to increment the EFF (initially at 2) to allow for cuts earlier than logical time 2. As a consequence, the initial message between the helloworld actor and the greeter actor is omitted, which ultimately results in a violation of the correctness specification.

7.4 Persistence

Another sample that can be found in the akka-sample repository is the persistence sample, illustrating the Akka persistence API. Akka persistence allows actors to keep persist internal state, thus enabling them to recover the state in the event that the actor is restarted, JVM crashes or if some other external event causes the state to be lost. Additional data that reaches the actor never mutates, or else changes the storage but is appended to it.


\[
\text{Listing 7.2: Added code in order to make the hello-world sample viable for ldfi-akka.}
\]

```scala
def verifyCorrectness(): Boolean = {
  val input : Source = Source.fromFile("ldfi-akka/logs.log")
  input.getLines().exists { line => line.contains("Greet") }
}
```

\[^3\]Omitting the other messages however causes the system to crash. If, for instance, the Done message is omitted: the hello world actor never stops and as such, the system never terminates.
In essence, the sample consists of two actors (including the actor system), where the actor system sends 6 messages to the persistent actor after creating it. Two of the messages are not part of the samples functionality, but there for illustration purposes\(^4\). The other four of the messages consists of data that is intended to be persisted by the persistent actor. The persistent actor persists these messages coupled with a duplicate and an added counter to the end. For instance, the first time “foo”, “baz”, “bar” and “buzz” is sent to the persistent actor, the resulting persisted data is “foo-0”, “foo-1”, “baz-2”, “baz-3”, “bar-4” and so on, up to “buzz-7”. The next time the actor system is run, the counter starts at “foo-8”, which is appended to the end of the data from the previous run. Therefore, to verify the correctness of this actor system, a method that checks this pattern must be created for ldfi-akka to invoke.

The method is illustrated in Listing 7.3. Essentially, the initial state data and the end state data is retrieved (modeled as lists) and grouped as lists of 8. The last list, is then verified with a control list, containing the elements “foo-”, “foo-”, etc.

The verification of the simple-deliv or hello-world samples in previous sections could be made with no interaction with the actor system. In this sample however, the data is persisted as the actors internal state. Granted, the data is stored locally on the file system, and as such can be retrieved and read, but it would be cumbersome to open those files, especially since they are encoded in special characters and not in plain text. Therefore, a workaround was used: the data can be retrieved using blocking futures. A future is part of the Scala library and used to retrieve results of some concurrent event, using the

\(^4\)One is a message that causes the actor to print its internal state, and the other is to save a snapshot of the state.
def verifyCorrectness(): Boolean = {
    val correctList = List("foo-", "foo-", "baz-", "baz-", "bar-", "bar-", "buzz-", "buzz-")
    val pattern = "(?<=\-).*"
    initState match {
        case Nil =>
            val splitEndState = endState.reverse.flatMap(str => str.split(pattern).lift(0))
            compareLists(splitEndState, correctList)
        case _ =>
            val groupedEndState = endState.grouped(8).toList
            val lastEndState = groupedEndState.lift(0) match {
                case Some(list) => list.reverse.flatMap(str => str.split(pattern).lift(0))
                case None => sys.error("Could not get last end state.")
            }
            compareLists(lastEndState, correctList)
    }
}

def compareLists(eventList: List[String], correctList: List[String]): Boolean =
    (eventList, correctList) match {
        case (Nil, Nil) => true
        case (ehead :: etail, chead :: ctail) => compareLists(etail, ctail) && ehead == chead
        case _ => false
    }

Listing 7.3: Verify method for the persistence sample.

case object getEvents

class ExamplePersistentActor extends PersistentActor with ActorLogging {
    ... 
    val receiveCommand: Receive = LoggingReceive {
        ... 
        case 'getEvents' => sender() ! state.events 
    }
    ... 
}

object PersistentActorExample extends App {
    ... 
    implicit val timeout = Timeout(5 seconds)
    val initFuture = persistentActor ? getEvents
    val initState = Await.result(initFuture, timeout.duration).asInstanceOf[List[String]]
    /* Messages are sent */
    val endStateFuture = persistentActor ? getEvents
    val endState = Await.result(endStateFuture, timeout.duration).asInstanceOf[List[String]]
}

Listing 7.4: Using futures to retrieve the internal data from the actor using the getEvents case object.
“?” operator. A blocking future simply refers to the fact that the message passing does not proceed until a response is received from the actor. To allow for the data retrieval, an additional case object `getEvents` in addition to a corresponding case in the `receiveCommand` pattern match was added. The implementation is shown in Listing 7.4.

Similar to the previous examples, a single message cut is enough to force the violation of the correctness specification. Note that except for the "print" message, there is little importance to which message is cut, as all message cuts results in the same violation. For this particular example, the EFF is initially set to the absolute minimum, i.e., allowance of a single message cut, which in turn yielded a real solution. Therefore, by starting backwards, `ldfi-akka` cuts the last message sent from the actor system, which in this case corresponds to “buzz”.

## 7.5 Dining Philosophers

Akka also provides a module that models actors as finite state machines (FSM). An FSM can be described as:

\[
\text{State} (S) \times \text{Event} (E) \rightarrow \text{Actions} (A), \text{State} (S')
\]

which informally reads, given a current state S and an event E, we perform actions A and arrive at the new state S'. The FSM module is illustrated by implementing a resolution to the known synchronization problem, dining philosophers. The problem consists of five philosophers and five chopsticks, where the goal for each philosopher is to try to eat the food on their plate using exactly two chopsticks. The chopsticks are placed between each pair of adjacent philosophers, and as such, the philosophers have to coordinate their usage of the chopsticks. The philosophers can either wait for available chopsticks, think or eat. In order to avoid live- and deadlocks, the philosophers must implement a successful "eating" strategy. This problem is easily modeled in Akka using the FSM trait. The chopsticks and philosophers each represent actors. The philosopher actors can be in three states: waiting, thinking or eating; whereas the chopstick actors can be in two states: available or taken.

In the implementation that can be found at the `akka-samples` repository each philosopher is initially in a waiting state, and after some duration transitions to the thinking state. There, a philosopher waits for some other duration to pass after which it ask the left and right chopsticks actors for their status. If one chopstick is available it asks for the other, and if it is busy the philosopher relinquishes the one it holds and transitions back to the thinking state. If both chopsticks are busy the philosopher transitions to the thinking state. If both chopsticks are available the philosopher transitions to the eating state, and stays in that state for some finite duration, until it once more transitions to the thinking state. This implementation had the philosophers eating and thinking forever. In order to run `ldfi-akka` on this program however, it was modified to run for a finite time duration. More specifically, each philosopher would now stop whenever they have eaten once, instead of thinking and then trying to eat again. Also, a terminator actor implementing a supervision strategy was added, so that it is possible to deduce when or if all philosophers
object DiningHakkersOnFsm {
    var terminatedActors = mutable.Set.empty
    def main(args: Array[String]): Unit = run()
    def run(): Unit = {
        ...
        system.actorOf(
            Terminator.props(hakkers.toSet)
            .withDispatcher(CallingThreadDispatcher.Id),
            "terminator")
        Thread.sleep(60000)
        system.terminate()
    }

    def verifyCorrectness(): Boolean = {
        terminatedActors.size == 5
    }
}

class Terminator(actors: Set[ActorRef]) extends Actor with ActorLogging {
    actors.foreach(act => context watch act)
    def receive = LoggingReceive {
        case Terminated(actor) =>
            terminatedActors += actor
            println("Actor " + actor.path.name + " has finished eating and has stopped.")
            if(actors == terminatedActors) println("All actors have finished eating.")
    }
}

Listing 7.5: Added code to make dining philosophers a viable program for ldfi-akka.
stops.

Consider the added code shown in Listing 7.5. The terminator actor executes a supervision strategy on the philosopher actors: whenever they are terminated their actor references are appended to the initially empty set, initialized as a field in the *Dining-HakkersOnFsm* object. Without any interjections, the system takes around 25 seconds to terminate. Therefore, if it takes any longer than 60 seconds to run we assume that the system is prevented from terminating, or more specifically, one or more of the philosophers are not allowed to eat. After 60 seconds, the system is forcefully shut nevertheless, allowing *ldfi-akka* to invoke the *verifyCorrectness* method to deduce the outcome of the run.

The ”eating” strategy implemented to resolve the dining philosophers problem in the samples repository is naive. Consequently, it requires over 3500(!) messages to be passed before the philosophers can eat even once. Presumably, this sample is only meant to be illustratory of the FSM module, and as such, it is reasonable that it has not been optimized. For this reason, *ldfi-akka* could not perform its analysis as the SAT-solver was overwhelmed with the sheer number of messages literals alone (not counting the added literals for crashes and constraints). A more in-depth and detailed analysis of the cause is given in the ensuing chapters.

7.6 Observable Atomic Consistency Protocol

Conflict free replicated data types (CRDT) are data types used in distributed computing that can be spread across different nodes and be updated concurrently, and yet still eventually converge to a non-inconsistent state between the nodes. The data type was first introduced by Marc Shapiro et al. in a paper in 2011 and is currently used in popular distributed databases such as Redis and Cassandra. There are two types of CRDTs: operation based CRDTs (CmRDTs) and state based CRDTs (CvRDTs). CvRDTs (and CmRDTs) do not have support for consistent non-monotonic operations, and to address that issue, an extension of CvRDTs, observable atomic consistency protocol (OACP) were conceptualized in a 2018 paper by Zhao and Haller [25].

The OACP uses two distinct operations: CVRDT operations (CvOps) and Totally-ordered operations (TOps). CvOps are commutative, and as such, the order of such operations do not affect the final state. On the other hand, TOps, require that the ordering is preserved, and therefore, OACP uses the reliable total order broadcast (RTOB) protocol, to guarantee the ordering. RTOB, and by extension, OACP employs the *Raft* consensus algorithm. Raft guarantee that as long as $n + 1$ nodes do not fail in a system comprised of $2n$, the nodes reach consensus [19].

For the two operations, OACP must consider four alternatives. First, in the trivial event where two CvOps are invoked in succession, the operations function as defined in CvRDTs, i.e., with no strict ordering. Second, if a TOp follows a CvOp, then OACP guarantees that no more CvOps are allowed in the system (freeze) until the elected leader in the RAFT protocol makes sure that all of the replicas keep the same log. Third, if a CvOp follows a TOp, then the system notifies all replicas to revert to normal activity.
(melt) after which the CvOp is treated as defined in CvRDTs. Last, if two TOps are invoked in sequence, then the system employing OACP behaves according to the RAFT protocol.

An OACP implementation based on Akka, written by Zhao can be found at Github\[5\] and serves as a basis for the analysis that ldfi-akka performs. Thus, ldfi-akka prepares for the analysis by rewriting the program such that it is compatible for the evaluation. After initiating the analysis, ldfi-akka is unable to make any progress due to the occurrence of a deadlock when the program is run. The deadlock is caused by the encroachment performed by ldfi-akka in order to force the program to run in a synchronous execution model, with the purpose of achieving deterministic ordering of the messages, and therefore by extension, deterministic execution. This particular constraint is discussed in greater detail in the following chapters. With that being said, in theory, ldfi-akka should be able to realize that the OACP implementation would fail to meet its correctness specification by crashing more than half of the nodes.

### 7.7 Evaluation

In this section we evaluate the analysis that ldfi-akka performs on the subject programs with regards to different metrics. The number of actors is displayed to give a sense of the complexity of the program, as the LOC is not a sufficient metric in this regard. The failure specification illustrates the minimal failure specification required to violate the subject program’s correctness specification, if one was found. This information is the very purpose of ldfi-akka, i.e., to illustrate the general robustness of the program it is analyzing. Furthermore, the number of executions is important to highlight, as it refers to the number of iterations required for ldfi-akka to find a minimal failure specification needed to violate the correctness specification of the subject program.

As seen in Table 7.1 the number of executions increase as it becomes harder to find the correct failure injections. For instance, for simple-deliv which is a naive and simple broadcast program, only one execution is required to find the proper failure injection whereas five are required to find failure injections that violate the correctness specification of hello-world. Although the sample size is relatively small, the number of actors or lines of code do not correlate with the number of executions needed to find failure injections that violate the subject program’s correctness specification. This result is unsurprising, as a complex program with a large number of actors still might have a single point of failure, or vice versa. Moreover, we consider Crashes to be an expensive injection, therefore, failure specifications without any node crashes needed are given precedence. As failure injections could be found without crashing any nodes for the samples used, Crashes is set to 0. As simple-deliv and retry-deliv are Akka programs written specifically for the purpose of this study, there are no added lines of code. The code needed to verify the programs could be considered ALOC, and in such case, the ALOC is 50. The metric ALOC does seem

\[5\]https://github.com/CynthiaZ92/OACP
Table 7.1: For every program, the numbers of actors and the lines of code (LOC) is given. Also, we illustrate the failure specification (EOT, EFF and Crashes). ALOC refers to the added lines of code to the original implementation, typically consisting of the correctness verification code. Lastly, Exe refers to the number of concrete evaluations performed until a solution is found.

to correlate with the complexity of the program, i.e., number of actors and LOC. Again, this makes sense intuitively: if the complexity if the program is increased, then it is more difficult to verify that everything went correctly.
Chapter 8

Correctness

In this chapter, an informal exploration of the correctness of ldfi-akka is discussed. More concretely, this chapter details how the ordering of the messages, the occurrence of large formulas and the rewrites affect the analysis in sections 8.1, 8.2 and 8.3, respectively.

8.1 Ordering

Ldfi-akka assumes a synchronous execution model, i.e., it assumes that all messages are eventually delivered and with a deterministic ordering. Thus, we do not explore bugs that arise from different schedules and by extension different orderings of message delivery. There are however, several factors to take into consideration, even with this simplification.

Since different ordering of the messages sent between actors normally yield different results, we must first make sure that a given actor system always have a deterministic ordering. Akka only guarantees that the messages sent between a pair of actors arrive at the recipient actor in the order they were sent in. As a consequence, multiple actors sending multiple messages might result in different message ordering. The Akka API does however allow execution of the actor systems using different dispatchers. One of the dispatchers, the Calling Thread dispatcher, provides convenient features for the purposes of ldfi-akka. Namely, given that all actors are run using this dispatcher, deterministic execution is guaranteed [16]. Unfortunately, this guarantee is limited to deterministic program behavior, and as such, it is only upheld given that two crucial constraints are upheld: the actors’ message passing or the selection of actors can not depend upon some non deterministic event. For instance, if some actor depends on a random event and sends a message based on it: ldfi-akka makes the false assumption that this random event is repeated for every run of the program. Similarly, if any actor is selected to perform some activity based on some random event: ldfi-akka makes a similar assumption. Therefore, an adversary could construct a program such that ldfi-akka would either make no progress or else come to incorrect conclusions.
8.2 Large Formulas

A major limitation of ldfi-akka is the increase in the duration for solving large formulas that occur as a consequence of analyzing programs that consist of a large amount of message passing within the actor system before termination. As seen with the dining-philosophers sample, ldfi-akka was unable to perform its analysis due to over 3500 messages being passed before the philosophers ate even once. One message sent between two actors generates three literals in a clause (the message, the recipient actor and the sender actor), thus for 3500 messages we expect to get 10500 literals in the formula. Moreover, the solver has to take constraints into consideration — some messages might already be omitted and some nodes might already be crashed — which makes the analysis even less feasible. For Molly, this is not an issue, because it can directly access the logic of the program by parsing the Dedalus source code. This is not possible for ldfi-akka as it must run the program in order to make inferences about its behavior. If the subject program is written such that it generates a vast number of messages, then there is no option but to encode all of the messages in the boolean formula, as it otherwise would not be possible to arrive at correct conclusions.

8.3 Rewrite

A prerequisite of ldfi-akka is that the controller is aware of every single interaction within the actor system in order to perform a valid and correct analysis. Therefore, it is paramount that the program rewrite correctly targets all message passing within the actor system, in terms of logging and also in terms of controlling the message passing. Otherwise, the controller make assumptions based on insufficient information and draw incorrect conclusions. As such, the program rewrite rules (Scalafix rules) are essential to the validity of the analysis: they must have a 100% accuracy in the targeting of the message passing. Unfortunately, with the increase in LOC and number of used modules in Akka, the risk of the rewrite missing to have a complete coverage increases. This can only be corrected by a slow, iterative process where rules are added whenever new edge cases are found. Moreover, whenever the Akka API changes in areas that affect ldfi-akka: it must be updated accordingly.
Chapter 9

Discussion

In this chapter, some of the concepts and implementations used in ldfi-akka are discussed. Section 9.1 details a discussion on the conceptual difference between logical clocks in ldfi-akka and Molly. In section 9.2, the rewrites that ldfi-akka performs on existing programs is discussed whereas section 9.3 explores the difficulty of handling large formulas. Lastly, the correctness of the analysis that ldfi-akka performs is scrutinized in section 9.4.

9.1 Logical Clock

The notion of logical clocks in ldfi-akka are inherently different from its equivalent in Molly. In Molly, each process keeps track of its own local time, and enforces consistency with the usage of Lamport clocks. Trivially, when sending messages, each process sends its own timestamp alongside the message. If the recipient process internal clock is at an earlier time then the senders, it updates it clock to match the senders. Otherwise, it does not perform any action with regards to its clock. For ldfi-akka, all messages are simply parsed and assigned a logical time as they naturally occur in the logs. Therefore, the only requirement is that the controller are aware of all of the possible arising inconsistencies across multiple iterations and implements an algorithm that takes those into consideration.

9.2 Rewrite

To write a corresponding Akka program of e.g. simple-deliv is trivial. However, as a result of Scala as a language, and therefore Akka by extension, being fundamentally different from Dedalus: the discrepancy in the number of ways to write a given program is large. For instance, simple-deliv was written in Dedalus in 7 lines of code (LOC), whereas the same implementation in Akka was around 200 LOC. Naturally, with the lines of code increasing, the number of ways to write a given program rises accordingly. It is thus reasonable to assume that with the increase in LOC, the disparity in LOC between two similar programs Akka programs might be large. Thus, depending on the way the program is written, the
difference of LOC might be even larger in relation to the Dedalus equivalent. As a result, the total rewrite time depends on how efficiently the code was originally written.

9.3 Large Formulas

One major limitation that \textit{ldfi-akka} has is that it is difficult to perform an analysis on poorly constructed Akka programs. Namely, programs that are written in such a way that an unnecessary vast amount of message passing is needed to accomplish trivial functionality creates too much complexity for \textit{ldfi-akka} to handle. There is no simple resolution to this issue, given that the an unnecessary complex problem does not necessarily imply slow performance to such an extent that it becomes an issue for the user of the program. More concretely, if a task can be accomplished with 10 messages being passed, but instead uses 100 messages: it is not clear that the program becomes so slow that it is useless. For \textit{ldfi-akka} however, it can be the difference between an analysis that can be done in a feasible amount of time or not possible at all due to overwhelming the underlying SAT-solver.

Thus, there are several potential resolutions to this issue, some of which are more realistic than others. The most apparent solution is to have developers construct programs that avoid unnecessary complexity, which of course is more easily said than done. Another possibility, albeit more long-term, is to create a framework along-side \textit{ldfi-akka} that lets the user provide an actor contract with some domain-specific language that would detail the actors’ behavior. For instance, in addition to implementing the actors in Akka, the developer would also provide details on how the actors respond to messages they receive. That would allow \textit{ldfi-akka} to perform a compile time analysis as opposed to a run time one.

9.4 Correctness

The correctness of the analysis depends on a variety of factors discussed in the previous sections. One of those, are the program rewrites that are needed for \textit{ldfi-akka} to perform its analysis. In order for the rewrites to have a 100\% accuracy, the rules must cover all cases of which messages are passed between actors. Indeed, it is possible to be confident that all messages passed using the built-in bang operator or the tell method are correctly targeted by the rewrite rules. Akka is however comprised of more than 10 modules, some of which include specific ways of passing messages. For instance, in the \textit{FSM} module, it is possible to use the \textit{replying} method. The \textit{replying} method allows for sending a message while declaring the state transition. More specifically, given a message \( M \), a state \( S \), a sender \( Snd \) and a recipient \( Rec \); \( S.replying(M) \) is equivalent to \textit{tell}(\( M, Snd \)); \( S \). Without a specific rewrite rule for this particular case, \textit{ldfi-akka} would make false inferences about the programs behavior. Naturally, the resolution to this particular problem is an exhaustive search of all means of passing messages while writing corresponding rewriting rules such that they are correctly accounted for. Said process is however tedious while also being
vulnerable to any API changes within the Akka framework.

Moreover, *ldfi-akka* places the constraint that all programs must be deterministic in their nature. Trivially, the program must exhibit the same behavior across multiple runs. Given that the actor model was created to tackle non-determinism that was caused by concurrency problems: this particular constraint should not be exceptionally difficult to uphold. Naturally, it is not always the case that developers are aware of the built-in non-determinism of the programs (caused by whatever reason), and as such lets *ldfi-akka* perform its analysis all the same. In such eventuality, it is possible that the analysis performed by *ldfi-akka* is valid as long as the non-determinism is either consistent across all runs or by chance does not happen at all.
Chapter 10

Conclusion

In this thesis, ldfi-akka, a lineage-driven fault injector for Akka programs written in Scala has been presented. In an intensively studied area, ldfi-akka distinguishes itself by using a novel and proven sound and complete approach to find bugs in programs written in a standard and widely used language. Ldfi-akka is shown to be able to perform its analysis on Akka programs by finding failure injections that violate the programs correctness specification, or else conclude that the program is fault tolerant. The correctness of ldfi-akka relies upon a number of constraints that have to be upheld, one of which is deterministic program behavior. Furthermore, it is shown that the analysis conducted by ldfi-akka depends on the complexity of the subjected program. More specifically, the number of actors and messages passed within a single iteration is paramount to the analysis, and as they grow, the analysis suffers, and at times becomes infeasible.

The entire project is implemented in around 2.5k lines of Scala code, not including the Akka implementation of simple-deliv that was written to serve as an illustratory comparison to its Dedalus equivalent.

The research questions presented in chapter 1 are considered in section 10.1 whereas section 10.2 explores future possibilities for ldfi-akka.

10.1 Research Questions

This study has shown that it is possible to employ lineage-driven fault injection on actor-based programs by a) introducing a conceptual framework and b) implementing it on the actor-based framework Akka. The lineage-driven fault injector for Akka programs, ldfi-akka is shown to successfully and precisely pinpoint the weaknesses of the sample programs that could be evaluated in a feasible amount of time. In this study, a novel approach to extracting data lineage from the execution traces of Akka programs is conceptualized and implemented, which is then made use of by ldfi-akka. Moreover, an external controller, capable of controlling the run-time execution of Akka programs is presented.

In order to show that the analysis is sound, it would be necessary to prove that for any given Akka program that the resulting boolean formula generated by ldfi-akka perfectly
CHAPTER 10. CONCLUSION

encompasses the behavior of the program it is representing. Thus, it would be required to show that the rewrites have a perfect accuracy and that given such accuracy, the resulting logs details all message passing. With such an assumption, it would follow that the analysis is sound given that the rest of the analysis is proven sound, i.e., the lineage-driven fault injection approach itself. As for the proven completeness guarantees that states that Molly finds a failure injection that violates the correctness specification of a given program (if one exists): it follows that the analysis performed by ldfi-akka is complete, given above assumptions. As the assumptions are false: it is not possible to show that the analysis is sound or complete.

10.2 Future Work

Ldfi-akka could be improved in various ways. The rewriting rules do not cover all Akka programs, especially the ones written using various modules. Also, the sample size used in this study is relatively small. With an increased sample size, the reliability of the analysis performed by ldfi-akka would increase.

To our knowledge, ldfi-akka and Molly are the only two lineage-driven fault injectors. One of the main criteria that LDFI places upon the languages is that it must be possible to extract the data lineage from the execution. Fortunately, the topic of data-lineage and provenance is an intensively studied area, and as a result there is a formally proved programming model for lineage-based distributed computation [10], as well as language-integrated provenance in Haskell [20], which opens up for a future lineage-driven fault injector for Haskell programs. Moreover, Akka is loosely based on Erlang, and therefore, many ideas used for ldfi-akka could be used to construct an Erlang equivalent.
Bibliography


