Degree Project in Computer Science

Second cycle, 30 credits

Abstracting Failures Away From Stateful Dataflow Systems

ALEKSEY VERESOV

Abstracting Failures Away From Stateful Dataflow Systems

ALEKSEY VERESOV

Master's Programme, Computer Science, 120 credits
Date: December 16, 2024

Supervisors: Philipp Haller, Jonas Spenger
Examiner: Roberto Guanciale
School of Electrical Engineering and Computer Science
Swedish title: Abstrahera Fel Från Tillståndfull Dataflöde

Abstract

Systems distributed across several computers are essential for modern infrastructure, and their reliability is reliant on the correctness of the constituent computers' failure-handling protocols. Correctness in such systems is often understood as *failure transparency*, a property that enables to use a system as if no failures occur in it; in other words, it states that there is a high-level model of the system, from which the failures are abstracted away. This work proves that failure transparency is provided by the Asynchronous Barrier Snapshotting protocol used in Apache Flink, a prominent distributed stateful dataflow system. This protocol is formalized in operational semantics for the first time in this thesis. As no prior definition of failure transparency is suitable for this formalization, a novel definition is proposed, applicable to systems expressed in small-step operational semantics with explicit failure-related rules. The work demonstrates how failure transparency can be proven by reasoning about each execution as a whole, presenting a proof technique convenient for proofs about checkpoint-recovery protocols. The results are a first step towards a verified stateful dataflow programming stack.

Keywords:
Failure Transparency, Stateful Dataflow, Operational Semantics, Checkpoint Recovery

Sammanfattning

System fördelade över flera datorer är väsentliga för den moderna infrastrukturen, och deras tillförlitlighet är baserad på korrektheten i protokollen som hanterar fel i de ingående datorerna. Riktigheten förstås ofta som *failure transparency*, en egenskap som gör det möjligt att använda ett system som om inga fel uppstår i det; med andra ord står det att det finns en högnivåmodell av systemet, från vilken misslyckandena abstraheras bort. Detta arbete bevisar att feltransparens tillhandahålls av protokollet Asynchronous Barrier Snapshotting som används i Apache Flink, en framträdande representant för distribuerade system med stateful dataflöde. Den första operativa semantiken för protokollet presenteras; Dessutom, eftersom det inte fanns någon definition av feltransparens för modeller i småstegsoperativ semantik, föreslås en ny definition, tillämplig på system uttryckta i småstegsoperativa semantik med explicita felrelaterade regler. Beviset visar hur misslyckandetransparens kan bevisas genom att resonera om varje exekvering som helhet, vilket gör det praktiskt i bevis om protokoll för återställning av checkpoints. Resultaten är ett första steg mot en verifierad stack för stateful dataflödesprogrammering.

Nyckelord:
Feltransparens, Tillståndfull Dataflöde, Operationell Semantik, Kontrollpunktsåterställning

Acknowledgments

I am deeply grateful to all people who have been supporting me throughout this work and who made it possible to be done. Due to the limits of the thesis' language and this page's size, I cannot fully express my gratitude to everyone who deserves it, but this is my best attempt to do so.

First, I thank my supervisors, Philipp Haller and Jonas Spenger for their support and guidance. This work was a great pleasure to do because of their efforts, and I feel incredibly lucky to have enjoyed the opportunity to work with them. They gave me the warmest welcome into the world of research and science. Furthermore, I am grateful to Paris Carbone, who was essentially my third supervisor, albeit unofficially, providing me with valuable insights into the world of distributed systems and stateful dataflow programming.

Second, I thank my professors at KTH, especially Dilian Gurov, David Broman, Karl Palmskog, Stefan Nilsson, Tomas Ekholm, Musard Balliu, and Roberto Guanciale, who is also the examiner of this thesis. I have enjoyed the courses I have taken, and I am sure I will use the knowledge I gained in them for decades. I also thank KTH itself for awarding me with a scholarship, without which I might have missed the wonderful opportunity to learn in this awesome place and community.

Third, I thank my fellow students at KTH. Although I don't think it is appropriate to mention anyone by name, I express my gratitude to all of them for making my experience here bright and pleasant, and I am especially grateful to the people who gifted me their friendship.

Last but not least, I thank my friends and family. You are my strongest support, and I wouldn't be who I am without you.

Thank you! Спасибо!

Contents

1 Introduction

This work started from the desire for a fully verified distributed programming stack. However, it soon became clear that complete verification of such a system is a far more demanding task than it was hoped to be: in practice teams of computer scientists spent years formalizing and verifying each of the existing sufficiently complex verified systems. Therefore, it was decided to focus in this work (1) on formalizing an essential part of a popular stateful dataflow system, the failure-handling protocol [Carbone et al. 2015] of Apache Flink [Carbone 2018], and (2) on proving correctness of the protocol itself instead of verifying its implementation. The results of this thesis are a first step towards the goal of a fully verified stateful dataflow programming stack.

1.1 Motivation

Distributed programming is the backbone of modern computing infrastructure, as it enables the creation of high performant, geographically separated, and reliable systems [Fu and Soman 2021; Mao et al. 2023]. High performance is beneficial for computationally intense tasks, such as scientific computations or big data processing; geographical separation is integral for some applications, users of which are located on different sides of the globe; while reliability is what makes the systems usable in the first place, even in spite of failures of individual computers.

Due to the characteristic large scale and longevity of distributed systems [Armstrong 1996; Fragkoulis et al. 2024], practically every error present in their failure-handling protocols is doomed to occur. This presents a challenge for reasoning about them, as the customary way of ensuring reliability, namely testing, is not sufficient. Testing is capable of discovering particular errors, and testing techniques are developed with the goal of finding the most frequently occurring errors. However, testing is not suited for eliminating all errors in a given program. As such, application of traditional testing techniques to distributed systems, while being helpful at eliminating some errors, still leaves a gap for the untested errors to occur, resulting in relatively rare but disastrous distributed infrastructure outages known today [Lianza and Snook 2020; Madory 2021; Satariano 2020].

An alternative prominent approach for ensuring reliability and correctness is

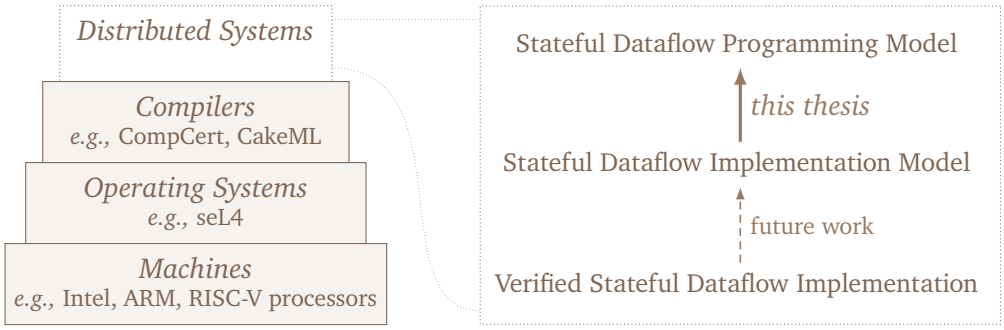


Figure 1.1. This work in the context of a fully verified stack for distributed programming.

formal verification. In formal verification, a program is mathematically proven to be correct according to some specification; in other words, it is proven not to contain any errors. The approach is known to be successfully applied in compilers [Kumar et al. 2014; Leroy 2009], operating systems [Klein et al. 2009], as well as processors [Choi et al. 2017; Kaivola et al. 2009; Reid et al. 2016]. However, there is an apparent lack of verified distributed systems, for example, there is no verified stateful dataflow system yet despite their high impact and wide adoption by industry [Fragkoulis et al. 2024].

Stateful dataflow systems enable processing large and continuously increasing amounts of data with high reliability. A prominent representative of stateful dataflow systems is Apache Flink [Carbone 2018], which is used on a large scale by some companies, *e.g.,* ByteDance [Mao et al. 2023] and Uber [Fu and Soman 2021]. The failure-handling mechanism of this system is not yet proven to be correct. This work aims to increase confidence in the correctness of Apache Flink, particularly its failure-handling mechanism, by using methods commonly used in formal verification.

The grand goal behind this thesis is to make a first step towards a fully verified stack for stateful dataflow programming, as shown in Figure 1.1. The abovementioned prior work has dealt with the verification of the lower layers of the stack, while this work aims to contribute to the analysis of distributed systems themselves, particularly stateful dataflow systems. Concretely, this thesis is focused on proving correctness of the failure-handling protocol of Apache Flink, while its full verification is left for future work. The proof essentially abstract failures away from the implementation model, providing a high-level programming model of the system, justifying the use of the system by programmers who assume that failures do not occur in it.

1.2 Research Question

While the wide and successful use of Apache Flink hints at the correctness of its underlying failure-handling protocol, and the protocol is proven to make causally-consistent snapshots; it is not yet proven that it handles failures correctly. In other words, it is not yet proven to provide failure transparency. Thereby, the main research question of this thesis is the following:

How to define and prove failure transparency of stateful dataflow systems?

1.3 Delimitations

While being a first step towards complete formalization and verification of Apache Flink, this work is not targeted to achieve these grand goals. Instead, this work is focused only on the essential part of the failure-handling protocol of Apache Flink; for example, partitioning is not covered in the devised model. Moreover, the model is not intended to be used for reasoning about the performance of the system, but only about its correctness.

Although the proposed definition of failure transparency and proof technique are intended to be reusable for other stateful dataflow systems than Apache Flink, for example Portals [Spenger et al. 2022], Google MillWheel [Akidau, Balikov, et al. 2013], IBM Streams [Jacques-Silva et al. 2016] and Microsoft Trill [Chandramouli et al. 2014], such systems are out of the scope of this thesis.

Last, despite its popularity, the small-step operational semantics is not the only way to formalize a distributed system, other approaches include Communicating Sequential Processes [Hoare 1978] and TLA+ [Lamport 2002]. However, this work is focused only on reasoning in small-step operational semantics. One of the reasons for choosing it is its wide adoption in verification [Klein et al. 2009; Kumar et al. 2014; Leroy 2009]. The choice of operational semantics is further motivated in Section 3.1.

1.4 Contributions

This thesis makes the following contributions.

– Chapter 4 presents the first small-step operational semantics of the *Asynchronous Barrier Snapshotting* protocol used in Apache Flink.

– Chapter 5 provides a novel definition of *failure transparency* for systems expressed in small-step operational semantics with explicit failure rules.

– Chapter 6 presents a technique for proving such failure transparency. The technique is based on reasoning about the whole execution traces, which makes it suitable for proving failure transparency of checkpoint-recovery protocols.

– Chapter 7 provides a full proof of the failure transparency. Besides ensuring the correctness of the failure-handling protocol of Apache Flink, the proof demonstrates applicability of the proposed definition and proof technique.

– The definitions, theorems, and models are mechanized in Coq.*

To be noted is that the results of this thesis are also presented in a paper at ECOOP 2024 [Veresov et al. 2024a] and in a companion technical report available in arXiv [Veresov et al. 2024b]. This thesis goes into greater details than the published paper and the report, as well as presents a different perspective on the completed work. However, there is an unavoidable intersection of the ideas and formalisms presented; in such cases references to the paper and the report are omitted.

1.5 Outline

The rest of the thesis is structured into the following chapters: (2) Background, which summarizes the preliminary knowledge from the related prior work; (3) Methods, which describes the techniques used in this work, as well as justifies the choice of them; (4) Stateful Dataflow Model, which describes the devised semantics of a stateful dataflow system; (5) Failure Transparency Definition, which introduces and motivates the proposed definition of failure transparency; (6) Trace-Mapping Proof Technique, which presents the main developed proof technique used in the proof of failure transparency of the formalized stateful dataflow system; (7) Evaluation, which evaluates the results from Chapters 4 to 6 by providing the full proof of the abovementioned failure transparency and presents a discussion

*https://github.com/aversey/abscoq/

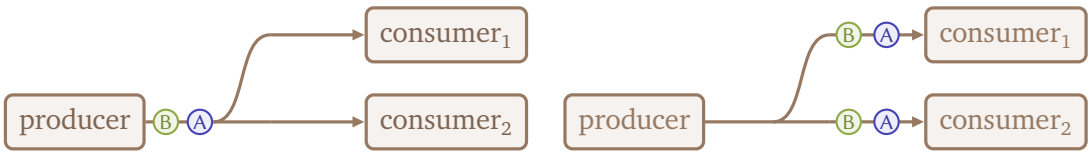
of them; and, finally, (9) Conclusions, which concludes the thesis and contains reflections on ethical and societal aspects of the work, as well as thoughts on the possible future work.

2 Background

This chapter summarizes the background knowledge necessary for understanding the rest of the thesis. There are three main areas covered: (1) a brief introduction is made to stateful dataflow systems and the way they are used, (2) the Asynchronous Barrier Snapshotting protocol commonly used for handing failures in stateful dataflow systems is explained, and (3) an introduction into small-step operational semantics is provided.

2.1 Stateful Dataflow

Stateful dataflow programming [Carbone 2018] is used to create applications distributed across several computers. A stateful dataflow program is expressed as a logical graph, where nodes are called *processors* and represent a stateful data processing function, while edges are called *streams* through which data "flows" from one task to another. A message sent to a stream is broadcast to all processors that consume the stream, as shown in Figure 2.1; the order of messages received from a stream is the same for all its consumers. In the context of stateful dataflow programming, messages are often called *events*.



(a) Before broadcast. **(b)** After broadcast.

Figure 2.1. Broadcast of messages through a stream.

As a recurring example, a program calculating the incremental average of a data stream of numbers is used throughout this thesis. It consists of a single processor which maintains the sum and the count of all numbers received from the stream and produces the average of the numbers. The numbers are sent to the task through a data stream. An ability to reset the state of the task is provided via a separate control stream with RESET events on it, as shown in Figure 2.2.

In practice, the input and output streams are not just given, but have to be defined by the programmer. Therefore, in stateful dataflow systems, there are

Figure 2.2. The incremental average task.

three basic types of processors: tasks, sources, and sinks. The "averages" processor from the example is considered a *task*, as it consumes, processes, and produces events on streams. *Sources* are used to define input streams, while *sinks* are used to define output streams. They can be seen as adapters, which convert the external data into the internal format of the system and vice versa. For the incremental average example, a more practical representation including its sources and sinks is shown in Figure 2.3; here the arrows coming into the sources "controls" and "measurements" and from the sink "results" represent the data flows outside the system, different from in-system streams.

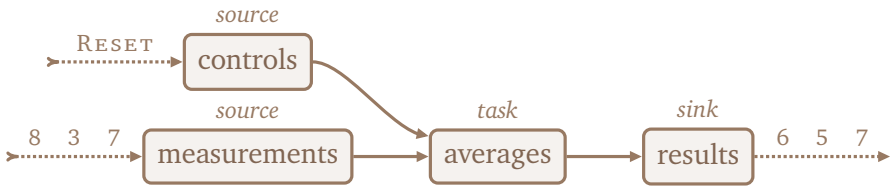


Figure 2.3. The incremental average example in a more realistic representation.

An example of the incremental average program written in Scala is shown in Listing 2.1. For simplicity, here and in the rest of the thesis, each of the tasks and the sources is assumed to have a single output stream. In the code, the results are output at the sink via the `println` function; while the input streams are read from external sources via `controller.read()` and `measurer.read()`, which, for example, may read the data from a TCP socket, blocking until new data is available. The event here is either a `Reset` event, which resets the state of the task with nullified state, or a number in 64-bit floating-point representation, which is used to calculate the average. The state of the task is a tuple of two elements, the first stores the sum of all measurements up to the current moment from the last reset, and the second stores the count of the measurements. The task is defined in a functional style, with a pure function which takes the new event and the current state as input and returns the new state of the task. Another pure function is used to define the produced output of the task; it is triggered after the processing function and returns the list of output events.

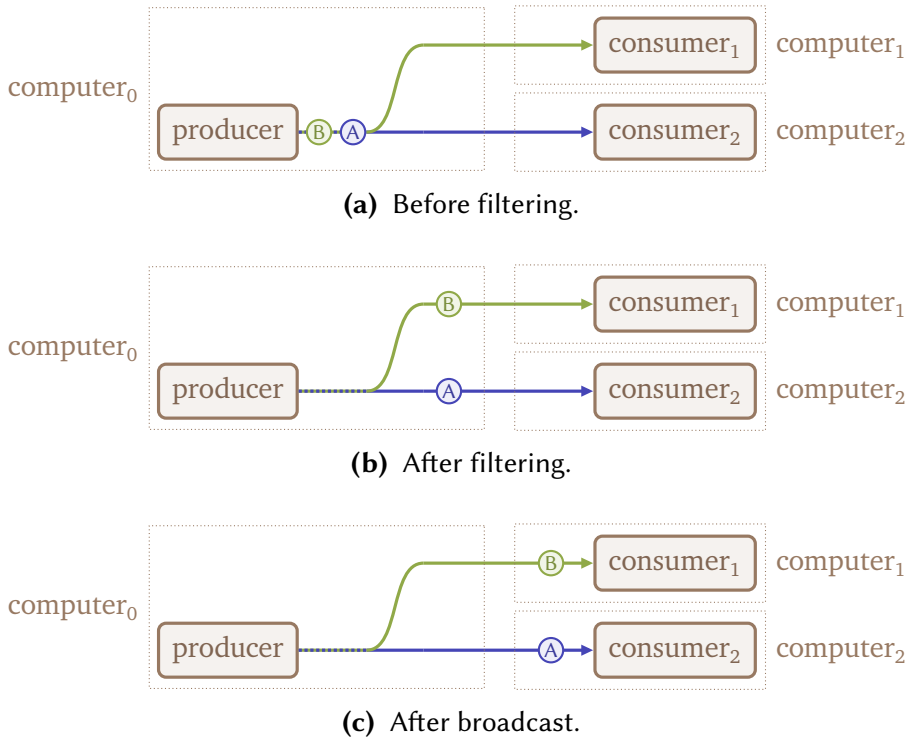
Listing 2.1. The incremental average example program in Scala.

```
1 val measurements = Source().produce(_ => List(measurer.read()))
2 val controls = Source().produce(_ => List(controller.read()))
3 val averages = Task().input(measurements).input(controls)
4   .init(_ => (0.0, 0)).process((event, s) => event match
5       case measure: Double => (s._1 + measure, s._2 + 1)
6       case reset: Reset => (0.0, 0))
7   .produce(s => if s._2 != 0 then List(s._1 / s._2) else List())
8 Sink().input(averages).consume(average => println(average))
```

Physical Graphs. Up until now we discussed logical graphs only, describing how to *program* a stateful dataflow system; however, a logical graph does not show a direct way to *execute* a program on several computers distributed over a network. For this reason, the logical graph is transformed into a physical graph by the stateful dataflow engine of choice. This transformation may happen in advance, or on the fly, as the program is executed. The physical graph itself is still described essentially in the same way as the logical graph, as a number of processors and streams between them. However, a physical graph usually carries additional information, such as the allocation of the processors to specific computers or the methods used to form each of the streams. For example, a stream between two processors executing on the same computer may use shared memory, while a stream between two processors executing on two different computers may use TCP.

As a vital optimization enabling stateful dataflow systems to scale efficiently, a way to split a logical processor into pieces executable on individual computers is needed. This is done by introducing the concept of *key*, which is a value associated with each event. Keys are used to split and group the events of the system into *partitions*, and, ideally, each of the partitions is processed independently by its single corresponding task. For example, for a system processing the data of users, the key may be the user ID and the partitions may be formed by grouping them modulo the number of physical processors corresponding to the original logical processor, thus enabling storing user data locally in each of the physical processors. In other words, keys and partitions are used to introduce data parallelism into stateful dataflow systems. This is achieved by an alternation of the stream broadcast rules, i.e., by adding key-based filtering to the broadcast; for efficiency, the filtering is done by the producer before sending the events out, as illustrated

by Figure 2.4.



(a) Before filtering.

(b) After filtering.

(c) After broadcast.

Figure 2.4. Broadcast of events with two keys from different partitions through a stream.

However, the peculiarities of physical graphs are not considered in this thesis, as they are not crucial for reasoning about the failure transparency provided by the Asynchronous Barrier Snapshotting protocol; and no distinction between logical and physical graphs is made in the rest of the thesis. The assumptions made in the model of the system are discussed in details in Section 4.4.

2.2 Asynchronous Barrier Snapshotting

The Asynchronous Barrier Snapshotting protocol, abbreviated as ABS, is a distributed algorithm for obtaining causally consistent snapshots of a system, which was introduced as a part of Apache Flink [Carbone 2018; Carbone et al. 2015]. The snapshots are used as checkpoints of the system, and the recovery is done to the most recent global snapshot in case of a failure.

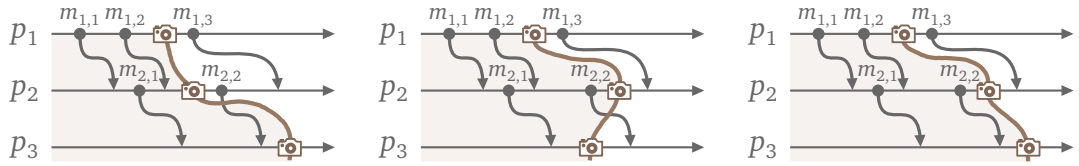
To request a distributed snapshot via ABS a special event is introduced to the system, called *barrier* or *epoch border*; in this thesis the latter term is preferred. The event is then propagated through the system, asynchronously collecting local

snapshots of tasks. While it is easy to obtain a snapshot of a single task, the challenge is to provide a sensible global snapshot of the whole system. ABS ensures this by preserving causal relationships of the events in the system, in other words, by being causally consistent.

Causal Consistency. A distributed snapshotting protocol is considered *causally consistent* if it captures snapshots that do not violate causality, captured as causal order on events [Chandy and Lamport 1985]. The causal order is defined by the happens-before relation; informally, an event a happens before another event b if either (1) a was processed before b on the same data processor, or (2) a sends a message received by b, or (3) there is an event which happens after a and before b; the causal order is captured formally later in the thesis by Definition 6.5. Figure 2.5 illustrates the concept by showing three different snapshots of a dataflow program with three data processors.

A naïve and causally inconsistent implementation of snapshotting may simply make a local snapshot of each of the processors at any time after the snapshot request is made, without any interprocessor coordination. This may result in an inconsistent snapshot violating causality, as in Figure 2.5a. The source of the violation is that the snapshot captures that $m_{2,2}$ is received by p_3, however, the snapshot does not have any information about the source of the message, in other words, it does not capture the fact that it was sent by p_2. From the point of view provided by the snapshot, the message $m_{2,2}$ just emerges out of nothing, which is clearly a violation of causality. In a practical sense, recovery from this specific snapshot will result in a resending of $m_{2,2}$, which in the end will result in a duplicate processing of the message by p_3. Supposing the message was a command to withdraw money from a bank account, the result of the recovery would be a double withdrawal, which is clearly not acceptable.

In contrast, causally consistent snapshotting protocols do not violate causality,



(a) Inconsistent snapshot **(b)** Chandy-Lamport snapshot **(c)** ABS snapshot

Figure 2.5. Examples of snapshots of a system with three data processors $p_1 \rightarrow p_2 \rightarrow p_3$.

which intuitively tells that they can be used as proper checkpoints in failure recovery protocols; an insight into the formal reasons why it is so are provided in this thesis by the proof of failure transparency of an ABS-based system. In the example, two causally consistent snapshots are shown.

The first causally consistent snapshot, depicted by Figure 2.5b, is captured by the classic Chandy-Lamport asynchronous snapshotting protocol [Chandy and Lamport 1985], which ensures causal consistency by separating pre-snapshot and post-snapshot messages via special marker messages. The protocol is notorious for its elegance and simplicity as well as ability to work on any strongly connected dataflow graph. However, a Chandy-Lamport snapshot may contain in-flight events; for example, according to the snapshot shown in Figure 2.5b, $m_{2,2}$ is sent but not processed, in other words, $m_{2,2}$ is in flight and has to be captured by the snapshot. This means, that the size of the snapshot is highly dynamic and hard to predict, as it is dependent on the size of each message and their arrangement during the snapshot acquisition.

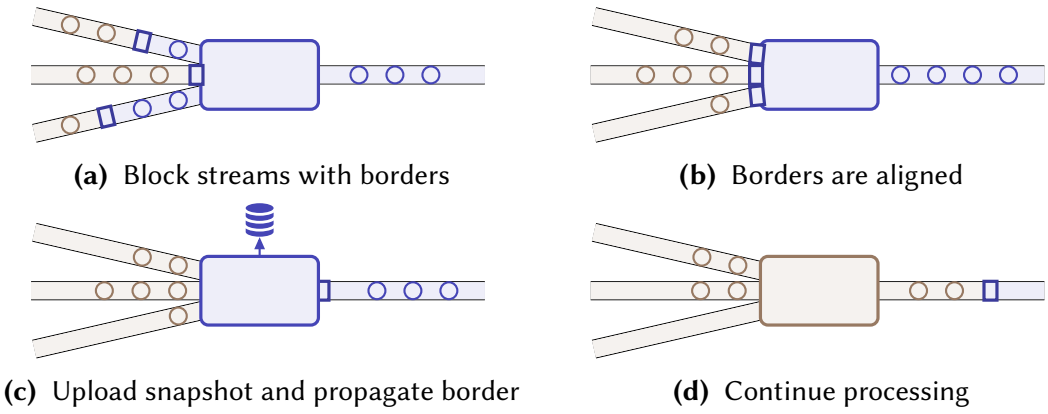
The second causally consistent snapshot, depicted by Figure 2.5c, is captured by the Asynchronous Barrier Snapshotting protocol [Carbone 2018; Carbone et al. 2015], abbreviated as ABS, which is inspired by the Chandy-Lamport protocol and is an answer to its issue of capture of in-flight events. As Chandy-Lamport protocol, ABS is based on markers, is simple and works for a wide range of dataflow graphs; more than that, it ensures that an obtained snapshot does not contain in-flight events; thus making the size of the snapshot predictable and stable. This thesis focuses on ABS: an informal introduction into its algorithm is made in the next paragraph.

ABS Algorithm. ABS works by introducing special events, called *epoch borders*, on each of the input streams of the system, and enforcing local synchronization rules on border processing to obtain a causally-consistent global snapshot. The protocol ensures that the snapshot is not influenced by any of the events that happened after the borders were introduced to the system, and, on the other hand, that the effects of all the events which happened to be introduced before the border are included in the snapshot. An epoch is a group of all such effectful events.

The introduction of epoch borders to the system determines which events affect the resulting snapshot and plays the role of request of a snapshot, thus initializing the whole protocol. A border event has to be sent to all input streams of the system; however, it does not mean that they have to be sent simultaneously. Due to the

assumed asynchronicity of message passing, a situation in which they are introduced at different times is indistinguishable from a simultaneous introduction. In practice, it means that to ensure progress of the system, it is enough to periodically emit a border event on each of the input streams; the periods can then be adjusted dynamically to avoid limiting the throughput of the system.

The border is then propagated through the system, and, on each task, local synchronization of borders happens, as is illustrated by Figure 2.6. First, for each input stream of a task, it is blocked as soon as the border is reached, thus preventing events of the next epoch from being processed before taking a snapshot of the current one (as in Figure 2.6a). The processing on the rest of the streams continues until all input streams of the task are blocked, in other words, until the current epoch is completely processed (as in Figure 2.6b). Finally, after fully processing the current epoch, the task needs to save its state to a local persistent storage, thus obtaining its local snapshot, and to propagate the epoch border (as in Figure 2.6c). After this, the processing on the task continues for the next epoch (as in Figure 2.6d).



(a) Block streams with borders

(b) Borders are aligned

(c) Upload snapshot and propagate border

(d) Continue processing

Figure 2.6. Epoch border alignment protocol.

Since for all processors this algorithm enforces their local snapshots corresponding to an epoch to be affected by all events of this epoch and by no events of the consequent epochs, the collection of these local snapshots forms a global snapshot of the system with the same property. This property leads directly to the causal consistency of the global snapshot.

A causally consistent global snapshot obtained by ABS can be used in the following way for failure recovery. In case of a detected failure, the system's coordinator calculates the greatest common fully processed epoch and asks all the processors

to restore their state to the one saved immediately after processing this epoch. To calculate the greatest common epoch, the coordinator organizes a two-phase commit: in the first phase, it collects "precommit" messages about epoch processing from all processors, and then, in the second phase, it sends a "commit" message back, which is especially important for sinks. Therefore, a committed message is processed by all processors and included in the global snapshot. Any event which is not committed can be aborted, and therefore should not be sent to the user by the sinks, since the goal is to provide a failure-transparent view of the system. A result of an epoch processing cannot be aborted after it was "committed" and thus it is safe to be output by the sinks.

2.3 Small-Step Operational Semantics

Small-step operational semantics [McCarthy 1960; Plotkin 1981] is an approach to capture meaning of a program, coming from programming languages theory and commonly used in formal verification. It consists of providing a set of rules, with each of them describing a single type of step of a program execution. The reasoning about the program is then made in terms of all the executions of it which are possible according to these rules.

For example, we can formalize a simple calculator, capable of summing integers by performing the add function on pairs of them. We do so by providing the Add, AddL, and AddR rules:

$$\textsc{Add}\ \frac{a \in \mathbb{Z} \qquad b \in \mathbb{Z} \qquad c = a + b}{\mathsf{add}(a, b) \to c}$$

$$\textsc{AddL}\ \frac{a \to a'}{\mathsf{add}(a, b) \to \mathsf{add}(a', b)} \qquad\qquad \textsc{AddR}\ \frac{b \to b'}{\mathsf{add}(a, b) \to \mathsf{add}(a, b')}$$

Each of the rules is marked by its label on the left; everything above the line to the right from the label is the premises, all of which have to be satisfied in order to apply the rule; and the statement below the line is its conclusion. In case of small-step operational semantics, the conclusion is usually a statement of the form $a \to b$, where a is the current state of the program, and b is the state after taking an execution step captured by the rule.

To note is that here we had to introduce two similar rules for propagation of

evaluation to arguments of add. *Evaluation contexts* [Felleisen and Friedman 1987; Felleisen, Friedman, et al. 1987] are often used to simplify description of such cases; however, the particular stateful dataflow semantics introduced in this thesis can not be simplified this way, and therefore the technique is not used.

For example of an evaluation, let's consider program $\mathsf{add}(1,2)$. The only rule applicable to it is Add, which results in a conclusion $\mathsf{add}(1,2) \to 3$. As there are no further rules applicable to 3, the execution of the program is finished after this single step, and integer 3 is the result.

For a more complex program $\mathsf{add}(\mathsf{add}(1,2),\mathsf{add}(3,4))$, the result 10 is reached after three steps, however there are two ways to reach it: as a first rule to apply we can choose either AddL or AddR, resulting in two different derivations:

$$\mathsf{add}(\mathsf{add}(1,2),\mathsf{add}(3,4)) \xrightarrow{\textsc{AddL}} \mathsf{add}(3,\mathsf{add}(3,4)) \xrightarrow{\textsc{AddR}} \mathsf{add}(3,7) \xrightarrow{\textsc{Add}} 10$$

$$\mathsf{add}(\mathsf{add}(1,2),\mathsf{add}(3,4)) \xrightarrow{\textsc{AddR}} \mathsf{add}(\mathsf{add}(1,2),7) \xrightarrow{\textsc{AddL}} \mathsf{add}(3,7) \xrightarrow{\textsc{Add}} 10$$

This nondeterminism does not have an effect on the results achieved by applying this sample semantics, however in more complex cases it can be crucial. Particularly, the ability to easily formulate nondeterministic systems in small-step operational semantics can be used to model asynchronicity of communication in distributed systems, which is essential for accurately capturing the behavior of the Asynchronous Barrier Snapshotting protocol.

3 Methods

In this chapter, the methods and techniques used to carry out this research are motivated by their suitability for addressing the specific challenges posed by the task of ensuring correct failure handling in stateful dataflow systems.

3.1 Operational Semantics of Distributed Systems

There are several approaches to capturing semantics of programs. This work uses small-step operational semantics, and the choice is motivated in this section.

Types of Semantics. In programming languages theory, three groups of semantics are distinguished: denotational, axiomatic, and operational [Pierce 2002].

In *denotational semantics*, the programming language is translated to mathematical notation directly. For example, programs are commonly understood as state-transforming functions. The loop "`while` c `do` b" is commonly understood as the least fixed point w, such that it solves the following equation with id standing for the identity function, $\mathbb{T}(c)$ standing for the translation of the condition c into a mathematical function taking a state and returning the boolean value of the conditional expression, and $\mathbb{T}(b)$ standing for the translation of the loop body b into a mathematical function taking a state and returning the new state after performing the loop body:

$$w \;=\; \text{if}(\mathbb{T}(c), w \circ \mathbb{T}(b), \text{id}) \;=\; \lambda s. \begin{cases} w \circ \mathbb{T}(b) & \text{if } \mathbb{T}(c)(s) \text{ is } \texttt{true} \\ \text{id} & \text{if } \mathbb{T}(c)(s) \text{ is } \texttt{false} \end{cases}$$

In *axiomatic semantics*, a program is understood as a transformer of assertions about its state. The programming language is described as a set of axioms, prescribing how each of the possible syntactic structures of the language affects any given assertions about program state. For example, in Hoare logic [Hoare 1969], which is the most well-known example of axiomatic semantics, the loop "`while` c `do` b" is defined via the following rule, which requires that some loop invariant I is preserved by the loop body b and ensures that the loop condition c is false after the

loop finishes while the invariant I stays true:

$$\textsc{WhileA}\ \frac{\{I \wedge c\}\ b\ \{I\}}{\{I\}\ \texttt{while}\ c\ \texttt{do}\ b\ \{I \wedge \neg c\}}$$

Finally, in *operational semantics*, a program is understood via a set of rules, each of which describes a single step of the program execution. There are two major types of operational semantics: big-step and small-step.

In *big-step operational semantics*, the conclusion about the whole execution of the program is made in one, "big" step, backed by a large derivation tree. In this respect, it is very similar to denotational semantics, providing another way to define a function corresponding to the program. For example, the meaning of the loop "`while` c `do` b" can be defined by two rules, WhileBT, corresponding to a new iteration of the loop, and WhileBF, corresponding to the loop termination:

$$\textsc{WhileBT}\ \frac{s \vdash c \Downarrow \texttt{true} \qquad s \vdash b \Downarrow s' \qquad s' \vdash \texttt{while}\ c\ \texttt{do}\ b \Downarrow s''}{s \vdash \texttt{while}\ c\ \texttt{do}\ b \Downarrow s''} \qquad \textsc{WhileBF}\ \frac{s \vdash c \Downarrow \texttt{false}}{s \vdash \texttt{while}\ c\ \texttt{do}\ b \Downarrow s}$$

In *small-step operational semantics*, also known as *structured operational semantics*, execution of the program is modeled as a sequence of execution steps, going from one state of the program to the next one. This is the key difference of this type of semantics from the other ones, since it emphasizes the process of the execution. For example, the loop "`while` c `do` b" can be understood via six rules, WhileS unwraps an iteration of it, IfS, IfST and IfSF define the behavior of the conditional statement, and SeqS and SeqSE define the behavior of a sequence of statements:

$$\textsc{WhileS}\ \frac{}{\langle \texttt{while}\ c\ \texttt{do}\ b,\ s\rangle \to \langle \texttt{if}\ c\ \texttt{then}\ (b\,;\ \texttt{while}\ c\ \texttt{do}\ b),\ s\rangle}$$

$$\textsc{IfS}\ \frac{\langle c,\ s\rangle \to \langle c',\ s\rangle}{\langle \texttt{if}\ c\ \texttt{then}\ b,\ s\rangle \to \langle \texttt{if}\ c'\ \texttt{then}\ b,\ s\rangle}$$

$$\textsc{IfST}\ \frac{}{\langle \texttt{if true then}\ b,\ s\rangle \to \langle b,\ s\rangle} \qquad \textsc{IfSF}\ \frac{}{\langle \texttt{if false then}\ b,\ s\rangle \to \langle \varepsilon,\ s\rangle}$$

$$\textsc{SeqS}\ \frac{\langle b_1,\ s\rangle \to \langle b_1',\ s'\rangle}{\langle b_1\ ;\ b_2,\ s\rangle \to \langle b_1'\ ;\ b_2,\ s'\rangle} \qquad \textsc{SeqSE}\ \frac{}{\langle \varepsilon\ ;\ b,\ s\rangle \to \langle b,\ s\rangle}$$

Evaluation of the Approaches. All the abovementioned types of semantics are widely used, and are valuable for different purposes. However, some of them have seen larger application in certain areas. In particular, a suitable approach should be able to handle the characteristic nondeterminism of distributed system easily. Moreover, taking into account the motivational goal of this work, i.e., construction of a verified stateful dataflow programming stack, the approach should be known to be successfully applied in verification.

Denotational semantics is convenient to use when formalizing languages close to mathematical notation, such as Haskell [Jones et al. 1999]. Nondeterminacy can be handled by representing programs as nondeterministic functions, taking an initial program state and returning the set of all possible resulting states. However, as well as big-step operational semantics, the approach is not very well suited for reasoning about distributed systems, as it becomes hard to keep track of a constantly growing set of all possible results. Moreover, execution traces are opaque in these approaches, thus complicating the reasoning about internal details of executions.

Axiomatic semantics is widely used in verification: firstly because it is convenient to provide program specification in it via pre- and post-conditions [Meyer 1992]; and secondly because Hoare logic [Hoare 1969] can be reformulated into predicate transformer semantics [Dijkstra 1976, 1975] used in automated verification, as well as influenced early efforts in it [King 1971]. However, the application of this approach to distributed systems, known as the Owicki-Gries method [Owicki and Gries 1976], is estimated to complicate reasoning about global invariants of the programs under consideration, and therefore to not be widely adopted in formal verification of distributed systems [Lamport 1993].

Finally, small-step operational semantics is widely used in the programming language theory, and in formal verification. It may seem not suitable for reasoning about distributed systems at first, as it describes program execution as a sequence of steps, while distributed systems are essentially concurrent and parallel. However, the approach is known to be successfully applied to distributed systems, and in particular, to model of asynchronous message passing, which is essential for capturing the behavior of the Asynchronous Barrier Snapshotting protocol.

For example, small-step operational semantics is successfully used to formalize a

lineage-based distributed programming model [Haller et al. 2018], which includes asynchronous message passing and failure recovery. As another example, Azure's Durable Functions programming model is also formalized in small-step operational semantics [Burckhardt et al. 2021]. That model is asynchronous and incorporates failure recovery too. These examples, as well as a number of others [Kallas et al. 2023; Mukherjee et al. 2019; Tardieu et al. 2023], serve as evidence that small-step operational semantics can be used to reason about complex distributed systems.

Moreover, operational semantics is used in many notable verification projects, such as seL4 [Klein et al. 2009], CompCert [Leroy 2009], and CakeML [Kumar et al. 2014]; this serves as evidence that it may be possible to use the model developed in this work in the future as a part of formal verification of stateful dataflow systems.

Therefore, in this work, the small-step operational semantics is chosen as the most suitable approach to model stateful dataflow systems.

3.2 Prior Definitions of Failure Transparency

The concept of failure transparency is not new, and it is widely used in the context of distributed systems. However, there is a lack of a general formal definition of failure transparency, in previous work, theorems about it are commonly stated in a way tailored to a specific system. Moreover, this lack hinders the development of proof techniques which could be applied in reasoning about the correctness of failure handling of a wide range of systems. As Henri Poincaré [1905] put it: *"There is no science but the science of the general."* That being said, it is not implied here that the most general definition is always the best one to use, since a system-specific definition may be simpler, and thus easier to reason about.

For example, in a recent paper [Mogk et al. 2019], failure transparency is defined as a property of a system, that after a rollback recovery to an exact global state which occurred previously in the execution, the system reproduces exactly the same effects as it did before the recovery. Although the approach is suitable for the system under consideration, it is not general enough to be applied to failure handling as provided by Asynchronous Barrier Snapshotting, since the recovery may be done to a state which never occurred in the past execution.

An earlier work focused on reasoning about observational equivalence of executions [Lowell and Chen 1999], and largely inspired the definition devised in this thesis. Nonetheless, the work is not defining failure transparency itself, as a

property of systems, and therefore is not suitable to be used to reason about whole systems in contrast to reasoning about individual executions.

A common approach is to use *simulation* to reason about the correctness of failure handling, it is used, for example, in [Burckhardt et al. 2021], the work which greatly influenced the definitions provided in this thesis. A relation R is a simulation if:

$$p\,R\,q \implies \exists p'.\ p \xrightarrow{l} p' \implies \exists q'.\ q \xrightarrow{l} q' \wedge p'\,R\,q'$$

However, it is hard to find such a simulation relation in case events can be discarded. Particularly, checkpoint recovery, which is performed in Asynchronous Barrier Snapshotting, discards non-committed events. The problem is that simulation relations are, by definition, inductive, and therefore in order to capture the effect of event discards, the relation has to keep track of the execution history, and it is done by auxiliary means. A similar inductive technique is the use of refinement mappings [Abadi and Lamport 1988], and it faces the same need to keep track of the execution history via auxiliary variables [Marcus and Pnueli 1996]. Although feasible, the approach is not straightforward and may complicate the reasoning about the system.

Therefore, it is reasonable to provide a new definition of failure transparency, which is general enough to be applied to a wider range of systems, particularly a definition suitable for reasoning about checkpoint recovery as it is performed in Asynchronous Barrier Snapshotting failure handling protocol.

3.3 Evaluation Procedure

The results of this work are mostly theoretical, moreover, a significant part of the work is the formalization of stateful dataflow systems and the development of the definitions. Although these results can be evaluated on their own by checking that the reasoning behind them is sound and seems to agree with reality and intuitions, it is also important to evaluate the results in a more practical and objective way. The suitability of the proposed failure transparency definition and trace-mapping proof technique is evaluated by applying them to the stateful dataflow model, resulting in a formal proof described in Section 7.2. Correctness of the proof and its relative conciseness are the main criteria for the evaluation, testifying suitability of the

devised definitions and technique to the task of ensuring reliable failure handling of distributed systems, particularly stateful dataflow systems.

4 Stateful Dataflow Model

This chapter introduces the stateful dataflow model, a small-step operational semantics capturing the essence of stateful dataflow systems, *i.e.,* the model of message-passing systems of directed acyclic graphs of processors which may fail and recover using Asynchronous Barrier Snapshotting protocol.

4.1 Basic Notation

The thesis follows standard mathematical notation for logical statements, quantifiers, and sets, including set-builder notation. Furthermore, to make the notation more concise and readable, the following representations of functions and sequences are used, which are inspired by the usual set-builder notation and the approach to sequences taken in TLA+ [Lamport 2002].

Sets. The set-building notation looks like $\{x \mid x \in X\} = X$, where $\{x \mid x \in X \wedge P(x)\}$ is the subset of X for all elements of which P holds. Set constructor lists the elements in curly braces separated by commas, for example, $\{1, 2, 3\}$. The empty set is denoted as $\emptyset$.

Functions. A function f is denoted as $[k \mapsto v \mid k \in \mathrm{dom}(f)]$. The part after the bar defines the domain of the function. The part before the bar defines the value of the function at point k as the expression v. The expression v may capture all variables defined on the right side of the bar, including k. A function with only one element in its domain is represented as $[k \mapsto v]$, where k and v are concrete values. We denote function update as $f\ g$, such that:

$$(f\ g)(x) = \begin{cases} g(x) & \text{if } x \in \mathrm{dom}(g) \\ f(x) & \text{if } x \notin \mathrm{dom}(g) \end{cases}$$

Combined, the one-element function notation and functional update notation can be used to conveniently denote update of a function at a single point in the usual way as $f[k \mapsto v]$.

Sequences. A sequence f is represented as a function with domain $\{i \mid i \in \mathbb{N} \wedge i < |f|\}$. The length of the sequence is represented by $|f|$ and may be infinite. If a variable f stands for a sequence, then the representation f_i is used instead of $f(i)$. To simplify the analysis of sequences, $\left[x\right]_i^n$ is used as a shorthand for $\left[i \mapsto x \mid i \in \mathbb{N} \wedge i < n\right]$, where x is an expression possibly containing i. Therefore, any sequence f is equal to $\left[f_i\right]_i^{|f|}$.

The usage of indices for variables standing for sequences may differ from other variables. If f stands for a sequence, then f_i corresponds to the i-th element of f. If, in contrast, f is not a sequence, then f_i is an independent variable and is not connected to f or any f_j. To avoid confusion, sets and sequences are named using uppercase and individual elements are named using lowercase.

Sequence concatenation can be used to extend or shrink existing sequences. Concatenating f with g is denoted as $f : g$ and is a shorthand for $\left[i \mapsto f_i \mid i \in \mathbb{N} \wedge i < |f|\right]\left[j + |f| \mapsto g_j \mid j \in \mathbb{N} \wedge j < |g|\right]$. To simplify extraction and addition of single elements, single-element sequences are denoted as $\left[x\right]$ meaning $\left[x\right]_i^1$. The empty sequence is represented as ε.

Quantifiers. $\forall x \in X \wedge P(x).\ Q(x)$ means $\forall x.\ x \in X \wedge P(x) \implies Q(x)$, and $\exists x \in X.\ Q(x)$ means $\exists x.\ x \in X \wedge Q(x)$.

4.2 Streaming Semantics

The streaming model is based on processors that communicate via streams. A processor is a stateful entity that may consume an event from an incoming stream, process it, and produce events to its outgoing stream. Streams, in turn, transport the events between processors in a first-in first-out order. A stream may be produced by at most one processor, but consumed by multiple processors, in which case the processors will all receive the same events, it can be seen as a form of broadcast. The execution consists of a sequence of steps, where each step is a consumption of an event from a stream by a processor together with the production of events to its outgoing stream. The asynchronicity of the message passing is captured by nondeterministic consumption of the events from streams. This section focuses on the general streaming model, leaving the implementation of processors abstract, to be detailed in the next section.

Syntax. Figure 4.1 shows the syntax of the streaming model. A configuration $c = \langle \Pi, \Sigma, N, M, D \rangle$ represents a point in an execution of a streaming program and captures all the information about the program and its state. The processors Π indexed by identifiers p represent processor definitions, and Σ represents the corresponding states of the processors, so that Σ_p corresponds to Π_p. The processor definitions and their states remain opaque in this section. The messages M are modeled as a sequence of all messages, and a message m is a tuple of its sequence number n in its stream, the stream's name s, and the message data d. The current sequence number from which a processor p reads from or writes to a stream s is represented by $N_p(s)$, and N_p is the map of sequence numbers of all stream which are consumed by the processor p. The sequence number maps for all processors are kept in sequence N.

When a processor takes a step, it may consume and produce messages. To keep track of and manage this behavior, the actions X are used. A production action producing message with data d to stream s has the form $+s\,d$, and similarly a consumption action has form $-s\,d$. The auxiliary data D is kept unspecified and can be used by the particular models based on this streaming semantics to store global and additional execution information. For example, in the next section, it is used to keep track of initial input messages of the executing system, and this information is used in the failure recovery to roll back properly.

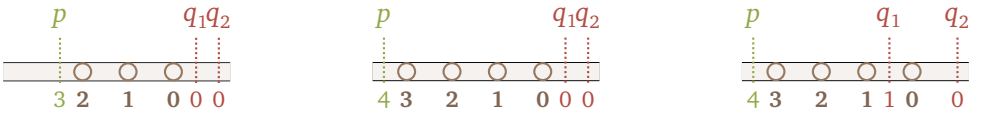
In this section, the processor π, state σ, message data d and auxiliary global data D are seen as atomic values, that is, no information about their internal structure is provided. These limitations permit reusing the same syntax and rule for

$p,\ q$ processor ID $\quad s,\ o$ stream name $\quad n \in \mathbb{N}$ sequence number

π processor $\quad \sigma$ state $\quad d$ message data $\quad D$ auxiliary data

$$
\begin{array}{lll}
\Pi ::= \left[\pi\right]_p^{|\Pi|} & \text{processors} & \\
\Sigma ::= \left[\sigma\right]_p^{|\Pi|} & \text{states} & \\
M ::= \left[m\right]_i^{|M|} & \text{messages} & \\
N ::= \left[N_p\right]_p^{|\Pi|} & \text{sequence numbers} & \\
N_p ::= \left[s \mapsto n \;\middle|\; s\right] & \text{sequence numbers of } p &
\end{array}
\qquad
\begin{array}{lll}
X ::= \left[x\right]_i^{|X|} & \text{actions} & \\
x ::= & \text{action} & \\
\quad +s\,d & \textit{production} & \\
\quad |\, -s\,d & \textit{consumption} & \\
m ::= n\,s\,d & \text{message} &
\end{array}
$$

Figure 4.1. Streaming syntax.



(a) Initial state **(b)** Production by p **(c)** Consumption by q_1

Figure 4.2. Operation of a stream with producer p and consumers q_1 and q_2.

different instantiations of π, σ, d and D.

Figure 4.2 illustrates a stream s as a sequence of messages with index numbers. When producing a message to a stream, as is shown in transition from Figure 4.2a to Figure 4.2b, the message is appended to the stream with an incremented index number; that is, a message m with index number 3 on the stream s is added to M. The producer's index number for the stream is also incremented from 3 to 4, to keep the right numbering of consequently produced messages. Similarly, the consumer's index number points to the next message to be consumed. Figure 4.2c shows how the next message is consumed by the consumer q_1. In this process, the consumer processes the message with sequence number 0 and increments its index number for the stream, preparing correct consumption of the next message. Thus, although executing in sequential steps, consumers and producers process streams independently and asynchronously. The production of a message is a kind of broadcast, in the sense that all processors will have to consume it before consuming a newer message.

Step Rule. The streaming model essentially consists of a single rule (S-STEP) which describes the processing of messages. Intuitively, a streaming step from configuration $\langle \Pi, \Sigma, N, M, D \rangle$ can be taken if there is a local step with actions X, such that the actions are applicable to the current configuration of the system. A local step describes how the processor Π_p changes its current state Σ_p to its next state Σ'_p using actions X. The local step is to be defined in the next section, since peculiarities of the local processor behavior are not essential for capturing message passing, which is the goal of this section. The actions X are applicable to N_p and M if all messages consumed by X are available on the input streams of the processor. This restriction is fruitful for formalization of the local steps of the processors, as it enables to formulate them in a way which tells how to consume any message, and then the applicability restriction rules out the consumptions which do not agree with the real state of affairs. An application of actions X results in the incremented sequence numbers N'_p and the updated sequence of messages M', extended with

the newly produced messages. The consumed messages are *not* removed from the message sequence, as they may be consumed by other processors. Moreover, since each stream has at most one producer, we do not need to synchronize sequence numbers across processors.

Taking a streaming step results in the configuration transition to the new, updated configuration $\langle \Pi, \Sigma[p \mapsto \Sigma'_p], N[p \mapsto N'_p], M', D \rangle$. To note is that the processors' definitions Π and the auxiliary data D remain unchanged; M' can be only extended or remain unchanged; and Σ and N are updated only for the processor over which the step takes place. In summary, the result of the streaming step is an update of the local state of the processor according to the nondeterministically chosen local step, and an update of the sequence numbers and messages according to the actions X. To simplify the analysis of streaming steps, auxiliary information about the processor ID, its sequence numbers, and the actions of the step is placed on the arrow of the execution step. This information can be omitted when it is not needed by applying abstraction steps S-AbsX and S-AbsP.

$$\textsc{S-Step}\ \frac{\Pi_p \Vdash \Sigma_p \xrightarrow{X} \Sigma'_p \qquad X(N_p, M) = (N'_p, M')}{\langle \Pi, \Sigma, N, M, D \rangle \underset{p}{\overset{N_p, X}{\Longrightarrow}} \langle \Pi, \Sigma[p \mapsto \Sigma'_p], N[p \mapsto N'_p], M', D \rangle}$$

$$\textsc{S-AbsX}\ \frac{c \underset{p}{\overset{N_p, X}{\Longrightarrow}} c'}{c \underset{p}{\Rightarrow} c'} \qquad\qquad \textsc{S-AbsP}\ \frac{c \underset{p}{\Longrightarrow} c'}{c \underset{p}{\Rightarrow} c'}$$

The streaming rule can be applied if there exists a derivation of a local step over a processor Π_p of the form $\Pi_p \Vdash \Sigma_p \xrightarrow{X} \Sigma'_p$. They are called local steps since they have access only to the local data of a processor, *i.e.,* its definition, state and directly accessible input messages. These rules describe a local step of a processor, in which the processor may produce and consume messages via actions X, and update its local state to Σ'_p. The to-be-performed actions X are checked for their applicability, and, if it is the case, they are used to modify the sequence numbers N_p of the processor and the messages M in the system. The check and update are captured by the action application definition. Action application has form $X(N_p, M)$ and is either undefined, in which case the actions X are not applicable to the N_p and M, or results in the new sequence numbers N'_p and messages M' used for the

next configuration.

Action Application. Here application of actions is defined, particularly in which cases they are applicable and how they modify the sequence numbers and messages. A production action $+s\,d$ increases the sequence number of the stream s of the processor, and adds a message on stream s with data d and the correct sequence number to the sequence of messages. A consumption action $-s\,d$ increases the sequence number of the stream s of the processor, but does not remove it from the sequence of messages, as there may be other consumers waiting to consume the message. To note is that the consumption action application is only defined if the consumed message is present in the sequence of messages in the correct position. In other words, a processor can only consume a message if it is present in the sequence of messages, it should consume it only once, and it should consume a message with a lower sequence number before consuming a message with a higher sequence number. The remaining cases of the definition recursively propagate the definition to sequences of actions.

Definition 4.1. (Action Application)

$$
\begin{aligned}
(+s\,d)(N_p, M) &= (N_p\left[s \mapsto N_p(s)+1\right], M \cup \left\{N_p(s)\,s\,d\right\}) \\
(-s\,d)(N_p, M) &= (N_p\left[s \mapsto N_p(s)+1\right], M) \textit{ if } N_p(s)\,s\,d \in M\textit{, undefined otherwise} \\
(\left[x\right] : X)(N_p, M) &= X(x(N_p, M)) \\
\varepsilon(N_p, M) &= (N_p, M)
\end{aligned}
$$

According to the definition, it is not always possible to apply an action. This may be the case if, for example, a message for some sequence number is not yet available on its stream. This enables indirectly “passing” messages to the local step rules. Whereas the local step rule is defined for all possible steps for all messages that it may consume, cases in which the message consumption is not applicable by the action application definition are ruled out by the streaming global step rule. This leaves only messages which are consumable in the steps, thus “passing” the message to the rule.

$$
\begin{array}{rll}
 & v,\ w & \text{value} \\
\pi & ::= \mathtt{TK}\langle f, [S_i]_i^{|S|}, o \rangle & \text{task} \\
a & ::= [e \mapsto v \mid e] & \text{snapshot archive} \\
\sigma & ::= \langle a, \sigma_{\mathrm{V}} \rangle & \text{state} \\
\sigma_{\mathrm{V}} & ::= & \text{volatile state} \\
 & \quad \mathtt{fl} & \textit{failed state} \\
 & \mid \langle e, v \rangle & \textit{normal state}
\end{array}
\qquad
\begin{array}{rll}
 & e \in \mathbb{N} & \text{epoch number} \\
d & ::= \langle e, d_{\mathrm{C}} \rangle & \text{message} \\
d_{\mathrm{C}} & ::= & \text{message cases} \\
 & \quad \mathtt{EV}\langle w \rangle & \textit{event} \\
 & \mid \mathtt{BD} & \textit{epoch border} \\
D & ::= M_0 & \text{initial input messages}
\end{array}
$$

Figure 4.3. Stateful dataflow syntax.

4.3 Stateful Dataflow Semantics

The presented stateful dataflow model consists of processing tasks, sources, and sinks. A processing task consumes messages from a set of input streams, and produces messages on its output stream. The task's behavior is defined by a function f which processes the messages. The function f takes the task's state and an input message, and produces a new state and a sequence of output messages: $f(v, w) = v', \left[W_i'\right]_i^n$. The presented formal model does not provide a syntax and semantics for functions; they can be expressed using any suitable formalism. The sources of the model are emulated by streams which are initialized in the first configuration to contain all the messages which are to be consumed from the source. That is, each source is represented by its output stream, which in turn becomes an input to one of the tasks of the computational graph. Sinks are also emulated as streams, however, in contrast to sources, they are initially empty. The computation of the system, informally, takes inputs from the sources, processes them in the processing graph, and produces outputs to the sinks.

Syntax. The syntax of the implementation model (Figure 4.3) extends the shared streaming syntax and semantics (Figure 4.1) by providing concrete instances of processors/tasks, messages, and state definitions. A task $\mathtt{TK}\langle f, S, o \rangle$ is a three-tuple of its processing function f, sequence of input streams S, and its output stream o. Tasks process messages which are tuples of an epoch number e and the message data d_{C}. There are two kinds of messages: normal events $\mathtt{EV}\langle w \rangle$ and epoch borders $\mathtt{BD}$. The epoch border messages are markers used for the snapshotting algorithm, whereas the events are the actual data processed by the tasks. When processing, the tasks manipulate state which consists of a persistent *snapshot archive* a, *i.e.,* a

map from epoch numbers to the corresponding local snapshots, and some *volatile state* σ_{V}. The snapshot archive is a map from epoch numbers e to the state v of the processor at the end of the epoch. The volatile state is either a *failed state* `fl` or a *normal state* $\langle e, v \rangle$, consisting of the current epoch number and the state data value v of the processor. The normal state is fed to the processing function f of the task. As with the messages, normal states are tagged by epoch numbers. A processor is in a failed state if it has crashed and lost its volatile state. The auxiliary data D used for this model consists of the initial input messages for the system. As we may need to restore the messages which are yet to be consumed, we keep track of all the initial input messages as the global auxiliary data of the system.

Derivation Rules. The semantics of the model consists of five rules. Three of the rules, I-Event, I-Border, and F-Fail, are local rules which enable deriving a local step of the form $\pi \Vdash \sigma \xrightarrow{X} \sigma'$. Whereas the I-Event and I-Border rules model the processing of the system, the F-Fail rule models nondeterministic crash-failures of a processing task within the system. These rules, together with the streaming rule S-Step, are used for deriving global steps. The fourth rule, F-Recover, is a global rule used for recovering the state of all processors after a failure.

Event Rule. The first rule, I-Event, models tasks processing events:

$$\text{I-Event}\ \frac{f(v,w) = v', \left[W_i'\right]_i^n}{\mathsf{TK}\langle f, S, o \rangle \Vdash \langle a, \langle e, v \rangle \rangle \xrightarrow{\left[-S_j \langle e, \mathsf{EV}\langle w \rangle \rangle\right] : \left[+o \langle e, \mathsf{EV}\langle W_i' \rangle \rangle\right]_i^n} \langle a, \langle e, v' \rangle \rangle}$$

The rule can perform a local step for a task $\mathsf{TK}\langle f, [S_i]_i^{|S|}, o \rangle$, if the current state of the task is a normal state $\langle e, v \rangle$, and the task can consume an event $\mathsf{EV}\langle w \rangle$ from one of its inputs S_j. Applying a task's function f to its current state v and the consumed event w results in the task's next state v' and a sequence of output events $\left[W_i'\right]_i^n$. The rule updates the state of the task to the new state $\langle e, v' \rangle$ and produces the output event $\mathsf{EV}\langle w' \rangle$ on the output stream o. The local step produces the actions which are the concatenation of the consumed and produced events. For example, $\left[-S_j \langle e, \mathsf{EV}\langle w \rangle \rangle\right] : \left[+o \langle e, \mathsf{EV}\langle w_i' \rangle \rangle\right]$ is the action of consuming the event $\mathsf{EV}\langle w \rangle$ with epoch number e from the input stream S_j and producing the event $\mathsf{EV}\langle w_i' \rangle$ with epoch number e on the output stream o.

Border Rule. Whereas the event rule consumes a single event from a stream, the border rule (I-Border) consumes one border event BD from *every incoming stream*:

$$\text{I-Border}\ \frac{}{\mathsf{TK}\langle f, [S_i]_i^n, o\rangle \Vdash \langle a, \langle e, v\rangle\rangle \xrightarrow{[-S_i\langle e,\mathtt{BD}\rangle]_i^n:[+o\langle e,\mathtt{BD}\rangle]} \langle a\big[e \mapsto v\big], \langle e+1, v\rangle\rangle}$$

This consumption is enabled for a task if the next event to be consumed on every one of its incoming streams is a border event. In other words, the event rule consumes events up until all streams are aligned by the border events, at which point the border rule consumes the border events from all its incoming streams. The rule is a local step which, in addition to consuming border events from all incoming streams and producing a border event on its outgoing stream, stores the current state v for epoch e to the snapshot storage a (by setting the new snapshot archive to $a[e \mapsto v]$), as well as incrementing the current epoch number.

Epochs are a key concept of Asynchronous Barrier Snapshotting. Each epoch is a sequence of data-bearing *events*, ending with an *epoch border*, and are used to define the boundaries of state snapshots. After regular processing for which some streams are blocked by border events (Figure 2.6a), the rule aligns the streams by the borders (Figure 2.6b), takes a copy of the current state of the processor storing it to the snapshot archive (Figure 2.6c), and propagates the epoch border message downstream and increments the epoch number, ready to process events from the next epoch (Figure 2.6d). The effect of this is that epochs of events are separated by the border events throughout the whole processing graph.

Failure Rule. Failures are introduced nondeterministically by the F-Fail rule:

$$\text{F-Fail}\ \frac{}{\mathsf{TK}\langle f, S, o\rangle \Vdash \langle a, \sigma_{\mathrm{V}}\rangle \rightarrow \langle a, \mathtt{fl}\rangle}$$

The failure rule sets the task's state to failed $\langle a, \mathtt{fl}\rangle$, thus losing the task's volatile state. Once a task is failed, it is no longer able to apply the steps I-Event and I-Border, and will remain idle until the F-Recover rule has been applied.

Failure Recovery Rule. The last rule, F-RECOVER, is a global rule which recovers the state of all failed tasks:

$$\text{F-RECOVER}\ \frac{\langle a, \texttt{fl} \rangle \in \Sigma}{\langle \Pi, \Sigma, N, M, M_0 \rangle \Rightarrow \text{lcs}(\langle \Pi, \Sigma, N, M, M_0 \rangle)}$$

The rule may be triggered nondeterministically if there exists a task in a failed state, and will reset the state of the system to the latest common snapshot. The full details of how the latest common snapshot (lcs) is computed is discussed further below, as it depends on additional definitions.

The latest common snapshot is constructed by: (1) calculating the greatest common epoch for which a snapshot has been taken by all processors in the system; (2) restoring the state of all processors to their local snapshots at the greatest common epoch; and (3) restoring sequence numbers and messages to undo any messages that were produced or consumed for epochs greater than the greatest common epoch. The greatest common epoch is calculated by finding the minimum (*common*) of the maximum (*greatest*) epoch numbers of the local snapshots of all the processors.

Definition 4.2. (Greatest Common Epoch Number) *The greatest common epoch number of a configuration* $c = \langle \Pi, \Sigma, N, M, D \rangle$ *is:*

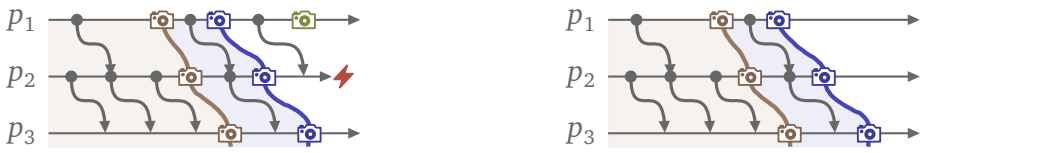
$$\text{gce}(c) = \min\left\{\max(\text{dom}(a)) \;\middle|\; \Sigma_p = \langle a, \sigma_{\text{V}} \rangle\right\}$$

The persistent *output messages* of the system consist of all messages produced up to and including the greatest common epoch. These messages can be identified by comparing their epoch number e to the greatest common epoch number $e \leq \text{gce}(c)$. The recovery purges any messages which are not part of this set, bar the initial input messages M_0, thereby making these output messages (identified by out) persistent.

Definition 4.3. (Output Messages) *For a configuration* $c = \langle \Pi, \Sigma, N, M, D \rangle$*, its output messages are:*

$$\text{out}(c) = \left\{ns\,\langle e, d \rangle \;\middle|\; (ns\,\langle e, d \rangle) \in M \wedge e \leq \text{gce}(c)\right\}$$

Definition 4.4. (Messages on a Stream) *The subset* $M \downarrow s$ *of messages on a particular*



(a) An execution with a failure **(b)** Snapshot-view of the execution

Figure 4.4. Executions viewed through the latest common snapshot.

stream is defined as:

$$M \downarrow s = \left\{ n' s' d' \;\middle|\; (n' s' d') \in M \wedge s' = s \right\}$$

The *lcs* function computes the latest common snapshot of a configuration for use as a recovery point in the F-RECOVER rule. Its computation makes use of the greatest common epoch number (gce), and the output messages (out). The states Σ' are restored by removing any stored snapshots with an epoch number larger than the gce, and the volatile states are restored to the states captured by the snapshot of the gce. The messages are updated to only keep the stable output messages $\text{out}(c)$ and the messages which are yet to be consumed M_{in}. The sequence numbers N' are updated accordingly, setting the sequence number of a processor p for a stream s to the number of messages that the processor has either produced or consumed on the stream: $|\text{out}(c) \downarrow s|$. Its complete definition is given below.

Definition 4.5. (Latest Common Snapshot) *The latest common snapshot of a configuration* $c = \langle \Pi, \Sigma, N, M, M_0 \rangle$ *is a configuration described by* $\text{lcs}(c)$*:*

$$\text{lcs}(c) = \langle\ \Pi,\ \Sigma',\ N',\ M_0 \cup \text{out}(c),\ M_0\ \rangle, \textit{where}$$

$$\begin{aligned}
\Sigma' &= \left[p \mapsto \langle \text{A}(a),\ \langle\ \text{gce}(c) + 1,\ a(\text{gce}(c))\ \rangle \rangle \;\middle|\; \Sigma_p = \langle a,\ \sigma_{\text{V}} \rangle \right] \\
\text{A}(a) &= \left[e \mapsto a(e) \;\middle|\; e \in \text{dom}(a) \wedge e \leq \text{gce}(c) \right] \\
N' &= \left[p \mapsto \left[s \mapsto |\text{out}(c) \downarrow s| \;\middle|\; s \in \text{dom}(N_p) \right] \;\middle|\; p \in \text{dom}(N) \right]
\end{aligned}$$

Viewing computations through the lens of the latest common snapshot shows configurations which are caused by failure-free executions. Figure 4.4a shows an execution with a failed processor p_2 and an incompletely processed epoch (green). In contrast, the latest common snapshot view of the same execution (Figure 4.4b) shows only the two completed epochs (red, blue), masking the failed epoch. The snapshot is emulating an execution such that all the steps on epochs after the

greatest common epoch are not taken, and all failed steps of incompletely processed epochs are ignored. This reasoning is further elaborated for the proof of failure transparency in the next section, where we show that the implementation model is failure transparent when viewed through the lens of the output messages function.

4.4 Assumptions

We make the following assumptions as a means to distill the essential mechanism of the failure recovery protocol. We assume that the message channels are FIFO ordered, a common assumption for snapshotting protocols [Chandy and Lamport 1985]. With regard to failures, we make common assumptions to asynchronous distributed systems [Cachin et al. 2011]. Failures are assumed to be crash-recovery failures, in which a node looses its volatile state from crashing. Further, we assume the existence of an eventually perfect failure detector, which is used for (eventually) triggering the recovery. With regard to system components, we assume the following components which can be found in production dataflow systems. The implicit coordinator instance is assumed to be failure free; in practice it is implemented using a distributed consensus protocol such as Paxos [Lamport 1998]. The snapshot storage is assumed to be persistent and durable; a system such as HDFS [Shvachko et al. 2010] would provide this. Further, the input to the dataflow graph is assumed to be logged such that it can be replayed upon failure. In practice, a durable log system such as Kafka [Kreps et al. 2011] would be used for this. For the stateful dataflow model, we make the following assumptions. The recovery is assumed to be an atomic, synchronous system-wide step. In practice, it may be implemented as an asynchronous atomic step, which can allow tasks to start processing before all have been recovered. Further, the task's processing functions are assumed to be pure, *i.e.,* free from side effects. A function f may be re-executed multiple times due to failures; a common assumption in related work [Burckhardt et al. 2021; Kallas et al. 2023].

The following simplifications are made in the model regarding real production systems. First, the presented formal model has no notion of "keys", whereas, in production systems, events are tagged by key, and tasks' states are partitioned by key. Further, tasks are not executed in parallel across partitioned instances in the model. These assumptions do not affect the assessment of the ABS protocol, as the "keyed" semantics are not relevant for the correctness of the snapshotting protocol.

The same applies to the various optimizations which are carried out by dataflow systems, such as task fusion and reordering.

5 Failure Transparency Definition

In this section, we define failure transparency such that it can be applied to systems described in small-step operational semantics. We first provide a rationale behind failure transparency, followed by its formalization.

5.1 Rationale

The purpose of failure transparency is to provide an abstraction of a system which hides the internals of failures and failure recovery. In particular, we would like to be able to show that the implementation model presented in the previous section is failure transparent. In concrete terms, this entails showing that executions of the implementation model can be "explained" by failure-free executions, something which we explore in this section.

Consider the task of computing incremental average as it was presented in Chapter 2 (Figure 2.3). The task consumes numeric data events i, reset events, and border events BD. For this example, we will consider a partial execution of the task in which it processes the events: $[1, \texttt{BD}, 3, \text{fail}, \text{recover}, \textsc{Reset}, 3, 5, \texttt{BD}, \ldots]$. The task's configurations consist of the task's current average value av, and its snapshot archive, a. Figure 5.1 shows at the top an execution of the task with a failure and subsequent failure recovery as the fourth and fifth events. After the recovery step, in its sixth configuration, the task's state is reset to its state for the snapshot $a_1(1)$, at which point it had the average value 1.

The question we ask is whether we can rely on the behavior of the task? More specifically, can we use the average value $av = 2$ in the fourth configuration (after

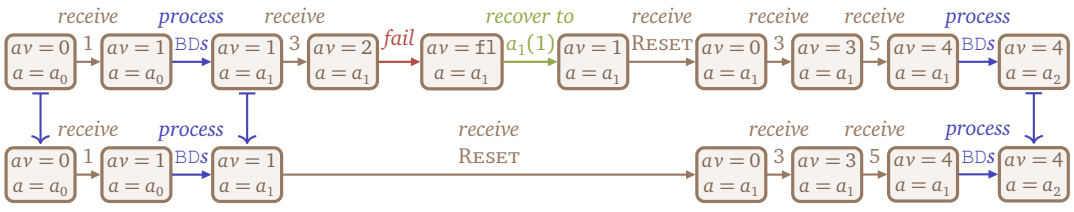


Figure 5.1. Execution of the incremental average task (Figure 2.3). Top: an execution with a failure and subsequent recovery. Bottom: a corresponding failure-free execution. The arrows between the executions indicate a mapping of the observed outputs. The snapshot archives are defined as: $a_0 = [0 \mapsto 0]$, $a_1 = a_0[1 \mapsto 1]$, $a_2 = a_1[2 \mapsto 4]$.

receiving the event 3) without further thought. The problem is that the task will fail in its next step, and recover to a state in which the receiving of the event has been undone. Moreover, the task continues its execution after recovery by processing the reset event first, and does never reach a state again in which its average value is 2. For this reason, we cannot blindly rely on the observed behaviors of the task as we may observe things which are later undone. In more complex systems, failures may further result in duplications and reorderings of events, further complicating the reasoning about the system.

Dealing with these issues requires the observer of the system to reason about which events are effectful and which are to be discarded. In some sense, the observer should be able to reason about the observed execution as if it was an ideal, failure-free execution, *i.e.,* an execution in which all events are effectful. Put in another way, the solution is to find a corresponding failure-free execution, and reason about that one instead. Intuitively, the observer should find some failure-free execution which "explains" the execution. Considering the above example, a failure-free execution thereof would correspond to the bottom execution in Figure 5.1. Note that there are no failure or recovery steps in the failure-free execution, yet its state progresses similarly to the original execution.

Even though the failure-free execution on an intuitive level correspond to the original execution, we would like to have a formal notion for this. The idea is to lift the observed executions by means of "observability functions", to a level where failure-related events and states are hidden. For example, for the executions above, we could define an observability function which takes the configuration of the task and keeps only the snapshot storage. After this transformation, applying this function to every configuration in the executions, we will not be able to distinguish the two executions by observing the system at any point in time. That is, common to both executions, we will first observe a_0, then a_1, and finally a_2. On a technical level, for every configuration of the original execution, we can find a configuration in the failure-free execution which, after application of the observability functions, is equal to it (*e.g.,* the mapping from top to bottom configurations in Figure 5.1); this is what we mean by "observable explainability". Thus, we can explain the original execution by the failure-free execution using the provided observability function.

The essence of the definition of failure transparency is derived from the notion of explaining the original executions by failure-free executions using observability

functions. Instead of reasoning about executions, we can reason about the observable output of executions at any given moment. Using observability functions effectively hides the internals of the model and allows the user to focus on the output of the system. That is, the user can reason about failure-free executions instead of faulty executions.

This informal introduction highlights three essential parts of failure transparency: the execution system, failures within the system, and the observability of the system. The goal of the rest of this section is to define these terms and to provide a formal definition of *failure transparency*.

5.2 Executions

The execution system for the failure transparency analysis is modelled as a transition system for which the transition relation is provided as a set of inference rules. In particular, we provide a formal definition for executions as a means to discuss the execution of systems. With this notion, distributed programs can be formally modelled in small-step operational semantics, and consequently formally verified. Although it may seem unintuitive to model distributed systems as transition systems for which the transition relation is defined over the global state, this is in fact commonly done in other formal frameworks such as TLA$^+$ [Lamport 2002].

Definition 5.1. (Execution Step) *A statement $c \Rightarrow c'$ is called an execution step from c to c'. We denote the derivability of an execution step in the set of rules R by $R \vdash c \Rightarrow c'$.*

We reason about systems in terms of their executions. An execution is a sequence of configurations C, connected by execution steps derivable in a set of rules R, and starting from some initial configuration C_0.

Definition 5.2. (Executions) *A sequence of configurations $\left[C_i\right]_i^n$ is called an* execution *in a set of rules R, if $\forall i < n.\ R \vdash C_{i-1} \Rightarrow C_i$. The set of all possible executions starting from C_0 in R is denoted as $\mathbb{E}_{C_0}^R$.*

The set of rules R of an execution specifies its reducibility relation by providing $c \Rightarrow c'$ as a conclusion of some of its rules. This approach is commonly known as *small-step operational semantics*. In this representation, the set of rules is explicit,

whereas commonly it is implicit. This is due to the need to explicitly distinguish between separate execution systems. This allows us, for example, to separate an execution system into two parts: one with failures R *s.t.* the failure-related rules are a subset thereof $F \subseteq R$, and one without failures $(R \setminus F)$.

5.3 Observational Explainability

The observability function represents the observer's view of the system. It notably differs from the plain configurations in the following two ways: the observer may not observe all internal details of configurations, *i.e.,* some parts of the configuration are *hidden* from the observer (*e.g.,* hiding commit messages [Burckhardt et al. 2021]); and the observer may observe some derived views of the configuration.

Definition 5.3. (Observability Function) *An* observability function O *of an execution system is a function which maps configurations to their observable outputs. It is required to be monotonic with respect to execution steps possible in the set of rules* R *for some partial order* $\sqsubseteq_O$*,* i.e.*, :*

$$\forall c, c'.\ (R \vdash c \Rightarrow c') \implies O(c) \sqsubseteq_O O(c')$$

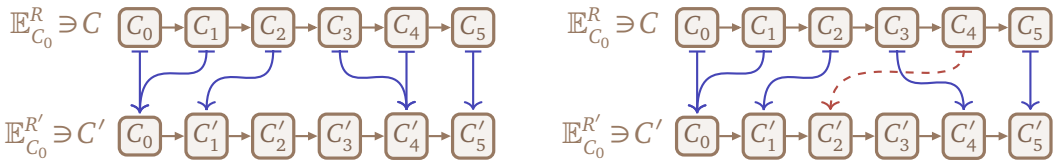
We say that an implementation's execution is observably explained by a specification's execution, if the observer cannot distinguish the two executions. This is the case when, for every configuration in the implementation's execution, there is a corresponding configuration in the specification's execution, such that their observed values are equal after application of the respective observability functions.

Definition 5.4. (Observational Explanation) *A sequence of configurations* C *of length* n *is explained by a sequence of configurations* C' *of length* m *with respect to observability functions* O *and* O'*, denoted as* $C \overset{O}{\rightleftharpoons}{}^{O'} C'$*, if:*

$$\forall n' < n.\ \exists m' < m.\ O(C_{n'}) = O'(C'_{m'})$$

An implementation's system, in turn, is observably explainable by the specification's system, if for each execution of the implementation there exists an explaining execution in the specification.

Definition 5.5. (Observational Explainability) *The set of rules* R *is* observationally



(a) Monotonic mapping **(b)** Non-monotonic mapping

Figure 5.2. Monotonic and non-monotonic mapping of configurations.

explainable *by R' with respect to their observability functions O and O' and the translation relation T, denoted as $R \overset{T}{}^{O}\!\rightleftharpoons^{O'} R'$, if:*

$$\forall\, c' \in \mathrm{dom}(T).\ \forall c.\ c' T c \implies \forall C \in \mathbb{E}^{R}_{c}.\ \exists\, C' \in \mathbb{E}^{R'}_{c'}.\ C\ {}^{O}\!\rightleftharpoons^{O'} C'$$

Monotonicity of Observational Explainability.. Observability functions are required to be monotonic, since observations should be regarded as stable. That is, once a value has been observed, then it should remain observable in the future. The system should not be able to undo something that has been observed, otherwise the observer would not be able to rely on the output. The reason for this is twofold. First, an observer may observe the system multiple times, and newer observations should provide more up-to-date views. Second, the sequence of observations should correspond to a valid explanation with respect to the higher-level specification, this is explored next.

If Definition 5.5 was not restricted to monotonic observability functions, then we would have the following problem. Consider two sequences of configurations: on the abstract level we have the sequence $[a, b, c, d]$ and on the implementation level the sequence $[a, c, b, d]$. Further, let us use the identity function as the observability function for both levels. For every element in the implementation-level sequence, we would be able to find a corresponding element in the abstract-level sequence with the same observability value. Therefore, if the definition allowed non-monotonic observability functions, we would consider $[a, c, b, d]$ to be an explainable execution of the sequence $[a, b, c, d]$. This, however, is counterintuitive, as the observer would expect to have observed some execution $[a, c, b, d]$ when the actual execution was $[a, b, c, d]$.

In the general case, it is desirable to have a monotonic mapping of configurations between the abstract-level and implementation-level executions. Figure 5.2a shows a monotonic mapping of configurations between two executions. Figure 5.2b, on

the other hand, shows a non-monotonic mapping, as seen by the red line which cross the other lines. Thus, we should not use non-monotonic mappings for the explainability of executions. We capture this notion in the definition of monotonic observational explanation.

Definition 5.6. (Monotonic Observational Explanation) *An observational explanation is* monotonic *if it is a monotonic mapping of configurations. That is,* $[C_i]_i^n$ *is* monotonically *explained by* $[C'_j]_j^m$ *w.r.t. O and O′ if:*

$$\exists \left[h_k\right]_k^n.\ (\forall k < n.\ \forall k' \leq k.\ h_{k'} \leq h_k) \wedge$$
$$\forall n' < n.\ \exists m' = h_{n'} < m.\ O(C_{n'}) = O'(C'_{m'})$$

The following lemma explicitly shows that the presented definition of *observational explainability* is equivalent to the definition of *monotonic observational explainability*. That is, the definition does not have the problem with non-monotonic mappings of configurations. For this reason, we will not distinguish between the two definitions in the following sections.

Lemma 5.7. (Monotonicity) *If R is observationally explainable by R′ w.r.t. T and monotonic O and O′, then it is also* monotonically *observationally explainable:*

$$\forall\, c' \in \mathrm{dom}(T).\ \forall c.\ c'Tc \implies \forall C \in \mathbb{E}_c^R.\ \exists\, C' \in \mathbb{E}_{c'}^{R'}.$$
$$C \text{ is monotonically explained by } C' \text{ w.r.t. } O \text{ and } O'$$

Proof. Section 7.1 § Proof of Lemma 5.7 **QED**

Composability of Observational Explainability. To further aid the use of these definitions within future proofs, we also show that the definition of observational explainability is transitive, as well as a compositionality lemma on the observability functions.

Lemma 5.8. (Transitivity) $R \overset{O}{\rightleftharpoons}{}^{T}_{O'} R' \wedge R' \overset{O'}{\rightleftharpoons}{}^{T'}_{O''} R'' \implies R \overset{O}{\rightleftharpoons}{}^{T \circ T'}_{O''} R''$

Proof. Section 7.1 § Proof of Lemma 5.8 **QED**

Lemma 5.9. (Composition of Observability Functions)

$$\forall O''.\ R \overset{O}{\rightleftharpoons}{}^{T}_{O'} R' \implies R \overset{O'' \circ O}{\rightleftharpoons}{}^{T}_{O'' \circ O'} R'$$

Proof. Trivial from the fact that $\forall x, y.\ x = y \implies O''(x) = O''(y)$. **QED**

The main idea of these properties is to enable reusing results about observational explainability in proofs detailing the low level or abstracting the high level further than the previous proofs. Imagine that there is a proof of observational explainability of R by R' with respect to observability functions O and O', $R \overset{T}{\underset{O}{\rightleftharpoons}}_{O'} R'$. Now, let's say someone wants to elaborate the low level which is observationally explainable by R' to a more detailed version of R, namely $\hat{R}$. Instead of proving $\hat{R} \overset{\hat{T}}{\underset{\hat{O}}{\rightleftharpoons}}_{O'} R'$ from scratch, one can use the transitivity lemma to separate $\hat{R} \overset{\hat{T}}{\underset{\hat{O}}{\rightleftharpoons}}_{O'} R'$ into $\hat{R} \overset{\tilde{T}}{\underset{\hat{O}}{\rightleftharpoons}}_{O} R \wedge R \overset{T}{\underset{O}{\rightleftharpoons}}_{O'} R'$, where $\hat{T} = \tilde{T} \circ T$ and the second claim is already proven. Now, to handle the first one, composability can be used to adjust the observability function to the desired form. Namely, instead of proving $\hat{R} \overset{\tilde{T}}{\underset{\hat{O}}{\rightleftharpoons}}_{O} R$, where O may be a too abstract observability function, we can move to a detailed enough observability function $\check{O}$ and find a hiding function O'' such that $O'' \circ \check{O} = O$, and then prove $\hat{R} \overset{\tilde{T}}{\underset{\tilde{O}}{\rightleftharpoons}}_{\check{O}} R$, with $\hat{O} = O'' \circ \tilde{O}$. In total, in order to prove $\hat{R} \overset{\tilde{T} \circ T}{\underset{O'' \circ \tilde{O}}{\rightleftharpoons}}_{O'} R'$ reusing $R \overset{T}{\underset{O}{\rightleftharpoons}}_{O'} R'$ it is enough to prove $\hat{R} \overset{\tilde{T}}{\underset{\tilde{O}}{\rightleftharpoons}}_{\check{O}} R$, where $O = O'' \circ \check{O}$. Plainly speaking, it is possible to reuse observational explainability results as long as the high-level observability function of one of the theorems can be mapped onto the low-level observability function of the other theorem; in other words, the high level of one theorem is more detailed than the low level of the other one.

This approach enables step-by-step refinement of observational explainability theorems. In this thesis, the failure transparency of stateful dataflow systems is proven as an observational explainability theorem. Using the reasoning above, it is possible to go down from the relatively high low level used in the main theorem of this thesis to a more and more detailed low levels, hopefully achieving a theorem stating observational explainability of an actual source code implementation of a stateful dataflow system by the failure-free part of the stateful dataflow model described in Section 4.3.

5.4 Defining Failure Transparency

The general goal of failure transparency is to provide an abstraction of a system which masks failures from the users. We can express this notion using observational explainability between the implementation and its failure-free part. That is, the implementation should be observationally explainable by the implementation

without failures. By considering explicit sets of rules which prescribe executions, such that some of them can be considered explicitly as failure-related rules, we can separate the execution system into two separate systems. The implementation system with all rules, *i.e.,* R, and another system with all rules except the failure rules, *i.e.,* $R \setminus F$. To fully instantiate the observational equivalence, we further use the same observability function O on both the low and high levels, and as a translation relation we use the identity relation on the set of initial configurations.

Definition 5.10. (Failure Transparency) *A set of rules R is* failure-transparent *with respect to failure-related rules $F \subseteq R$ for a monotonic observability function O and a set of initial configurations K, denoted as $R \mathbin{\backslash\!\backslash}^{O}_{K} F$, if:*

$$R \ {}^{O}\!\xrightleftharpoons{\{(c,c)\ \mid\ c \in K\}}{}^{O} (R \setminus F)$$

6 Trace-Mapping Proof Technique

This chapter introduces the trace-mapping proof technique. First, traces and causal order of steps are defined; next, the trace-mapping proof technique is explained and executed to sketch out a proof that the presented stateful dataflow model is failure transparent.

6.1 Traces and Causal Order

Traces. Traces are sequences of steps, providing a different view on executions, which are sequences of configurations. We will use the definition of traces to discuss permutations of traces which will preserve certain properties. In particular, we will define a causal order relation on traces, and show that all causal-order preserving permutations are "valid" permutations.

Definition 6.1. (Trace) *A trace Z is a sequence of trace steps z. A trace step z can be: either $\langle$I-Event$, p, N_p, X\rangle$, or $\langle$I-Border$, p, N_p, X\rangle$, or $\langle$F-Fail$, p\rangle$, or $\langle$F-Recover$\rangle$. Here* I-Event*,* I-Border*,* F-Fail*, and* F-Recover *play the role of the discriminant, where the trace step is a tagged union.*

A trace is a sequence of steps, where each step is a compact representation of the derivation of a transition from one configuration to another. That is, for the execution step from the ith to the $i+1$th configuration, *i.e.,* $R \vdash C_i \Rightarrow C_{i+1}$, if F-Recover was the root rule of the derivation tree, then this would correspond to the step $\langle$F-Recover$\rangle$ in the trace. To link traces with executions, we use the following definition.

Definition 6.2. (Trace Application) *A trace Z of length n applied to a configuration c results in a sequence of configurations C of length $n+1$, $Z(c) = C$, if $\forall i < n$:*

$$
\begin{aligned}
&(\exists p, N_p, X.\ Z_i = \langle \text{I-Event}, p, N_p, X\rangle \quad \wedge \quad \{\text{I-Event}, \text{S-Step}\} \vdash C_i \xRightarrow[p]{N_p, X} C_{i+1}) \\
\vee\ &(\exists p, N_p, X.\ Z_i = \langle \text{I-Border}, p, N_p, X\rangle \wedge \quad \{\text{I-Border}, \text{S-Step}\} \vdash C_i \xRightarrow[p]{N_p, X} C_{i+1}) \\
\vee\ &(\exists p.\ \quad Z_i = \langle \text{F-Fail}, p\rangle \quad \wedge \{\text{F-Fail}, \text{S-AbsX}, \text{S-Step}\} \vdash C_i \xRightarrow[p]{} C_{i+1}) \\
\vee\ &(\quad\ Z_i = \langle \text{F-Recover}\rangle \quad \wedge \quad \{\text{F-Recover}\} \vdash C_i \Longrightarrow C_{i+1})
\end{aligned}
$$

Traces can be generated from executions. However, not every trace corresponds to an execution. This may be the case if a trace has been constructed on its own, or reordered in some way. For this reason, we will define valid traces, which are traces that correspond to executions.

Definition 6.3. (Valid Trace) *A trace Z is valid from configuration c if it is applicable to it,* i.e., *if there exists an execution $C \in \mathbb{E}_c^I$ such that $Z(c) = C$. Note, that by definition of execution and trace application, any sequence of configurations produced by a trace application is an execution.*

It is easy to show that any prefix of a valid trace is also a valid trace. Still, this is a useful property, so we state it as a lemma.

Lemma 6.4. (Validity of Prefix) *For a trace valid from c, all of its prefixes are traces valid from c.*

Proof. Trivial. **QED**

Causal Order. Using the notion of traces, we can define a causal order on steps similar to the classic happens-before relation [Lamport 1978].

Definition 6.5. (Causal Order) *There are four cases in which a step Z_i happens before Z_j with $i < j$:*

1. *$Z_i = \text{F-RECOVER} \vee Z_j = \text{F-RECOVER}$ then Z_i happens before Z_j, by total order on the recovery steps;*
2. *if they are not recovery steps,* i.e., *for some p and p', N_p, N_p', X, X':*

   $$Z_i = \langle \text{I-EVENT}, p, N_p, X \rangle \vee Z_i = \langle \text{I-BORDER}, p, N_p, X \rangle \vee Z_i = \langle \text{F-FAIL}, p \rangle, \text{ and}$$

   $$Z_j = \langle \text{I-EVENT}, p', N_p', X' \rangle \vee Z_j = \langle \text{I-BORDER}, p', N_p', X' \rangle \vee Z_j = \langle \text{F-FAIL}, p' \rangle$$

   and $p = p'$ then Z_i happens before Z_j, by intraprocessor order;
3. *if they are action-producing steps,* i.e., *for some p and p', N_p and N_p', X and X':*

   $$Z_i = \langle \text{I-EVENT}, p, N_p, X \rangle \vee Z_i = \langle \text{I-BORDER}, p, N_p, X \rangle, \text{ and}$$

   $$Z_j = \langle \text{I-EVENT}, p', N_p', X' \rangle \vee Z_j = \langle \text{I-BORDER}, p', N_p', X' \rangle$$

and there is a message m produced by Z_i, that is

$$\exists M, M'', N_p''.\ X(N_p, M) = (N_p'', M'') \wedge m \in (M'' \setminus M')$$

which is consumed by Z_j, that is

$$\forall M'.\ m \notin M' \implies (X'(N_p', M') \textit{ is undefined})$$

then Z_i happens before Z_j, by interprocessor order;

4. *if there exists a step Z_k such that Z_i happens before Z_k and Z_k happens before Z_j, then Z_i happens before Z_j, by transitivity.*

Now we capture reorderings of steps as permutations of traces:

Definition 6.6. (Trace Permutation) *A trace Z' of length n is a permutation of another trace Z of length n if there is a bijection f from* $\mathrm{dom}(Z)$ *to itself such that* $\forall i \in \mathrm{dom}(Z).\ Z_i = Z'_{f(i)}$.

Definition 6.7. (Trace Permutation Preserving a Relation) *A trace Z' of length n is a permutation of another trace Z of length n preserving relation R if, for a bijection f defining the permutation, $\forall i, j.\ R(Z_i, Z_j) \implies R(Z_{f(i)}, Z_{f(j)})$.*

Finally, we state a lemma about the validity of permutations preserving causal order. Intuitively, it follows from the fact that causally unrelated steps should not influence each other.

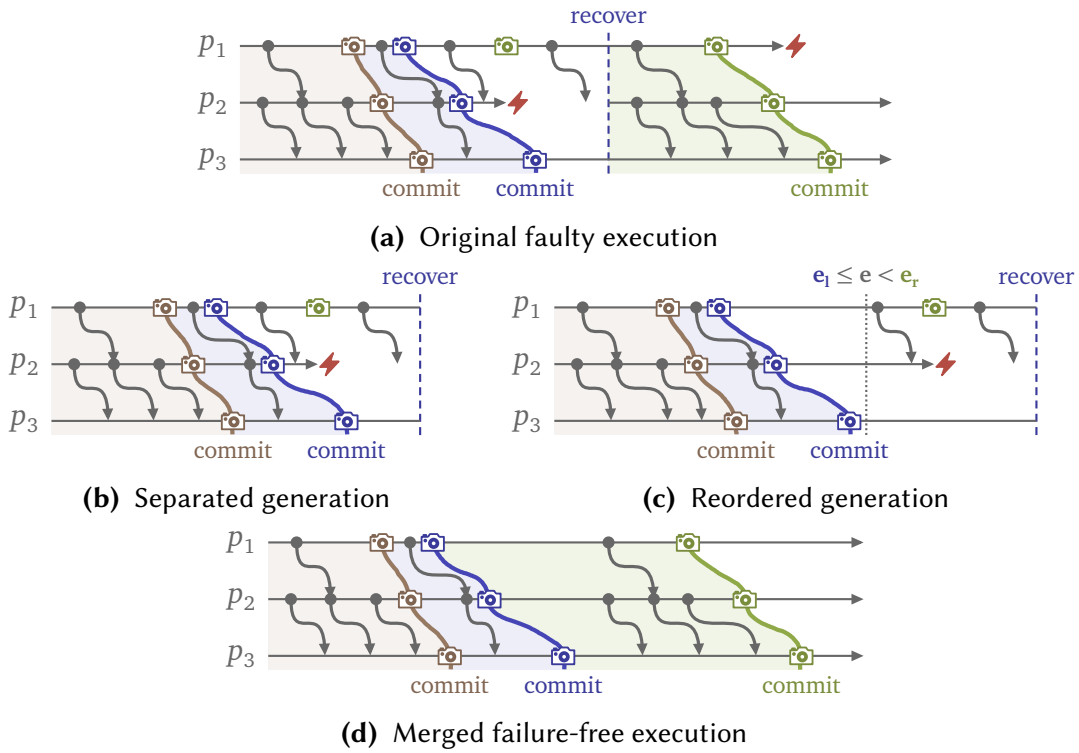
Lemma 6.8. (Validity of Causal Permutations) *For any trace valid from c, all of its causality-preserving permutations are traces valid from c.*

Proof. Section 7.1 § Proof of Lemma 6.8. **QED**

6.2 Proving Failure Transparency

As it is required by the definition of failure transparency, we will first define the sets of rules, namely I, F, and $(\mathrm{I} \setminus \mathrm{F})$; and the set of valid initial configurations K.

Sets of Rules. The semantics of the model consist of five rules, defining two separate sets of rules. The set of rules with failures consists of all five rules that have



(a) Original faulty execution

(b) Separated generation

(c) Reordered generation

(d) Merged failure-free execution

Figure 6.1. The step-wise construction of a failure-free execution trace from an execution with failures.

been defined for the stateful dataflow implementation model. It is named I as it corresponds to the implementation model presented in Section 4.3. The set of failure-related rules F within the implementation model consists of the two rules F-Fail and F-Recover. This way, the rules with failures are defined as the set I, and the rules without failures are defined as the set (I \ F).

Definition 6.9. (Implementation Model Rules)
$\mathrm{I} = \{\text{S-Step}, \text{S-AbsX}, \text{S-AbsP}, \text{I-Event}, \text{I-Border}\} \cup \mathrm{F}$

Definition 6.10. (Failure-Related Rules) $\mathrm{F} = \{\text{F-Fail}, \text{F-Recover}\}$

Initial Configurations. The sets of initial configurations which are considered are any acyclic graph structures which are properly initialized.

Definition 6.11. (Valid Initial Configurations) $K = \langle \Pi, \Sigma, N, M, M_0 \rangle$ *such that:* Π *is acyclic and weakly connected;* Σ *are the initial states;* N *are sequence numbers initialized to 0 for the streams;* M *consists of the well-formed inputs to the streams;* $M_0 = M$.

Theorem 6.12. (Failure Transparency of the Implementation Model) $I \lessdot\!\!\lessdot^{\text{out}}_{K} F$, i.e., *the set of rules* $\mathrm{I} = \{\text{S-Step}, \text{S-AbsX}, \text{S-AbsP}, \text{I-Event}, \text{I-Border}\} \cup \mathrm{F}$ *is failure transparent with respect to the failure rules* $\mathrm{F} = \{\text{F-Fail}, \text{F-Recover}\}$ *for the observability function* out *and the set of initial configurations* K.

Proof. Section 7.2 § Proof of Theorem 6.12, illustrated by Figure 6.1. **QED**

The proof is done by constructing a failure-free trace of a given arbitrary faulty execution (Figure 6.1a), as shown in Figure 6.1, and proving by construction that it is valid, and its corresponding execution is an observational explanation (Definition 5.4) of the faulty execution.

First, the faulty execution is transformed into a trace, and is split at the points of the recovery steps into so-called "generations" (Figure 6.1b). Each generation is a valid trace, as each of them is a continuous part of a valid trace. We observationally explain each of the steps by the most recent committing border step before it (marked by dotted line with "commit" label in Figure 6.1). The equality of observations holds by Lemma 7.1, since only a committing border step can change output.

Second, each generation is reordered, so that all the steps of epochs above the greatest common epoch of the generation are placed after the last border step of the greatest common epoch (Figure 6.1c). The reordered generation is still valid by Lemma 6.8 and the fact that no message from a future epoch can happen before a message from a past epoch (Lemma 7.2). The outputs of the committing border steps are not changed by the reordering, by Lemma 7.3, since we keep the total order of the steps in earlier epochs, and none of the moved steps happens before a committing border step. Therefore, the previously outlined explanation is still valid; it is to note that the most recent committing border step is defined in terms of the original, not reordered generation, as is shown by the bending of the second dotted commit-line in Figure 6.1c.

Next, each reordered generation is stripped to keep only the steps which are captured by the snapshot of the generation. The result is still a valid trace by Lemma 6.4; the outputs of committing border steps also stay the same, since no step before any of them is moved or changed.

Finally, the stripped generations are merged into one trace (Figure 6.1d). The resulting trace is valid, since, by Lemma 7.4, the recovery is done exactly to the configuration obtained after executing the stripped trace (*cf.* Figure 4.4). This means that the last configuration of each stripped generation is also the first con-

figuration of the next generation. Therefore, a step applied in the beginning of a generation is also applicable after the last step of the preceding stripped generation. The output is not changed by the merge, and hereby we have constructed a failure-free observational explanation of the faulty execution, which means that the implementation model is observationally explainable (Definition 5.5) by its failure-free version, or, in other words, it is failure transparent (Definition 5.10).

7 Evaluation

In this chapter the definitions, stateful dataflow model and the trace-mapping proof technique are evaluated by performing a full proof of Theorem 6.12.

7.1 Proofs of Lemmas

Here the proofs of the lemmas from the previous sections are provided, namely monotonicity and transitivity of observational explainability and validity of causal permutations. While the first two lemmas are not used in the proof of failure transparency, they are useful for deeper understanding of observational explainability and further motivated in Section 5.3. All three lemmas can be seen as an evaluation of the related definitions.

Lemma 5.7. (Monotonicity) *If R is observationally explainable by R' w.r.t. T and monotonic O and O', then it is also* monotonically *observationally explainable:*

$$\forall\, c' \in \mathrm{dom}(T).\ \forall c.\ c'Tc \implies \forall C \in \mathbb{E}^{R}_{c}.\ \exists\, C' \in \mathbb{E}^{R'}_{c'}.$$
$$C \textit{ is monotonically explained by } C' \textit{ w.r.t. } O \textit{ and } O'$$

Proof. Comparing the definitions of the observational explanation and its monotonic version, we see that the only difference is that the monotonic observational explanation requires the sequence of all m', namely h', such that m' corresponding to n' is equal to $h'_{n'}$, to be monotonic. In other words, by the definition of observational explanation, given c and c' such that $c'Tc$ and executions C and C' of lengths n and n' in R and R' from c and c' respectively, we have:

$$\forall m < n.\ \exists m' = h'_m < n'.\ O(C_m) = O'(C'_{m'})$$

Next, we construct sequence h, such that for all $k < n$ we take $h_k = h'_m$, where m is the smallest index of h' such that $O(C_k) = O'(C'_{h'_m})$. For it, we have:

$$O(C_k) = O'(C'_{h_k}) \ \wedge\ \forall k' < n.\ O(C_k) = O(C_{k'}) \implies h_k = h_{k'}$$

Now, let's assume that for certain k and k' such that $k' \leq k$ we have $h_{k'} > h_k$.

We have $O(C_k) = O'(C'_{h_k})$ and $O(C_{k'}) = O'(C'_{h_{k'}})$ by construction of h. Then, by monotonicity of O and O', from $k' \leq k$ we know that $O(C_{k'}) \sqsubseteq_O O(C_k)$ and from $h_{k'} > h_k$ we know that $O'(C'_{h_k}) \sqsubseteq_O O'(C'_{h_{k'}})$. Replacing $O(C_k)$ with $O'(C'_{h_k})$ in the first and $O'(C'_{h_{k'}})$ with $O(C_{k'})$ in the second, we get:

$$O(C_{k'}) \sqsubseteq_O O'(C'_{h_k}) \wedge O'(C'_{h_k}) \sqsubseteq_O O(C_{k'})$$

By antisymmetry of $\sqsubseteq_O$, we have $O(C_{k'}) = O'(C'_{h_k})$, which in turn is equal to $O(C_k)$. By construction of h, we have that $h_k = h_{k'}$, which contradicts the assumption, particularly that $h_k < h_{k'}$. Therefore, we have that $\forall k < n.\ \forall k' \leq k.\ h_{k'} \leq h_k$. In other words, the sequence h of all explaining prefixes m' has to be non-decreasing.

QED

Lemma 5.8. (Transitivity) $R \overset{O}{\rightleftharpoons} \overset{T}{O'} R' \wedge R' \overset{O'}{\rightleftharpoons} \overset{T'}{O''} R'' \implies R \overset{O}{\rightleftharpoons} \overset{T \circ T'}{O''} R''$

Proof. By definition of observational explainability, we have:

$$\begin{aligned}(\forall c' \in \mathrm{dom}(T).\ \forall c.\ c'Tc \implies \forall C \in \mathbb{E}^R_c.\ \exists C' \in \mathbb{E}^{R'}_{c'}.\ C \overset{O}{\rightleftharpoons} \overset{O'}{} C') \wedge \\ \forall c'' \in \mathrm{dom}(T').\ \forall c'.\ c''T'c' \implies \forall C' \in \mathbb{E}^{R'}_{c'}.\ \exists C'' \in \mathbb{E}^{R''}_{c''}.\ C' \overset{O'}{\rightleftharpoons} \overset{O''}{} C''\end{aligned}$$

By rearranging the terms, we get:

$$\begin{aligned}\forall c'' \in \mathrm{dom}(T').\ \forall c' \in \mathrm{dom}(T).\ \forall c.\ c''T'c' \wedge c'Tc \implies \\ \forall C \in \mathbb{E}^R_c.\ \exists C' \in \mathbb{E}^{R'}_{c'}.\ \exists C'' \in \mathbb{E}^{R''}_{c''}.\ C \overset{O}{\rightleftharpoons} \overset{O'}{} C' \wedge C' \overset{O'}{\rightleftharpoons} \overset{O''}{} C''\end{aligned}$$

Expanding the definition of observational explanation and rearranging quantifiers, we get:

$$\begin{aligned}\forall c'' \in \mathrm{dom}(T').\ \forall c' \in \mathrm{dom}(T).\ \forall c.\ c''T'c' \wedge c'Tc \implies \\ \forall C \in \mathbb{E}^R_c.\ \exists C' \in \mathbb{E}^{R'}_{c'}.\ \exists C'' \in \mathbb{E}^{R''}_{c''}. \\ \forall m < |C|.\ \exists m' < \left|C'\right|.\ \exists m'' < \left|C''\right|.\ O(C_m) = O'(C'_{m'}) \wedge O(C_{m'}) = O''(C''_{m''})\end{aligned}$$

From the last line it follows that:

$$\forall m < |C|.\ \exists m'' < \left|C''\right|.\ O(C_m) = O''(C''_{m''})$$

Contracting this according to the definition of observational explanation and applying the definition of composition of relations, we get:

$$\forall\, c'' \in \mathrm{dom}(T').\ \forall c.\ c''(T \circ T')c \implies \forall C \in \mathbb{E}^R_c.\ \exists\, C'' \in \mathbb{E}^{R''}_{c''}.\ C\ {}^{O}\!\rightleftharpoons^{O''} C''$$

Which is exactly the definition of $R\ {}^{O}\!\overset{T\circ T'}{\rightleftharpoons}{}^{O''}\ R''$. **QED**

Lemma 6.8. (Validity of Causal Permutations) *For any trace valid from c, all of its causality-preserving permutations are traces valid from c.*

Proof. The proof is done in two steps, In the first step we prove the validity of the permutations; and in the second one we prove their similarity.

Validity. Given a trace Z valid from c, and any causality-preserving permutation Z' thereof such that $Z_i = Z'_{f(i)}$, we have to show that Z' is valid from c. This is done by induction over the length of prefixes of Z'.

The base case is $\left[Z'_i\right]^0_i$. Since $\left[Z'_i\right]^0_i = \varepsilon = \left[Z_i\right]^0_i$ which is a prefix of Z, by Lemma 6.4, it is valid from c.

In the induction step we have that $\left[Z'_i\right]^j_i$ is valid from c and we have to show that $\left[Z'_i\right]^{j+1}_i$ is valid from c. Since $\left[Z'_i\right]^j_i$ is a valid trace from c, there exists an execution C' such that $\left[Z'_i\right]^j_i(c) = C'$. We proceed by case-analysis on the $(j+1)$'th step Z'_j to show that Z'_j can be applied to configuration C'_j.

1. In case $Z'_j = \langle \text{I-}\textsc{Event}, p, N_p, X\rangle$, the reasoning is as follows. By assumption, Z' is a causality-preserving permutation of Z. Therefore, by the third clause of the causality Definition 6.5, any of the consumed messages in X have been produced by previous steps and are thus available for consumption in C_j. By the second clause, all steps Z_k with $k < f(j)$ on the same processor p as Z'_j, appear once in the prefix of Z' before Z'_k. That is, the intraprocessor order is preserved. For this reason, the state Σ_p of p in configuration C'_j is the same as the state of Σ_p in $Z(c)_{f(j)}$. For these reasons, the step Z'_j can be applied to configuration C'_j for actions X on processor p, as the consumed messages are available in C'_j.

2. In case $Z'_j = \langle \text{I-}\textsc{Border}, p, X\rangle$, the reasoning is analogous.

3. In case $Z'_j = \langle \text{F-}\textsc{Fail}, p\rangle$, the step can be applied to C'_j as can be seen from the rule F-Fail, which does not have premises, and therefore is always enabled.

4. In case $Z'_j = \langle$F-Recover$\rangle$, the reasoning is as follows. By assumption, Z' is a causality-preserving permutation of Z. By the first clause of the causality Definition 6.5, for all other steps Z'_k with $f(k) < f(j)$, we have that $k < j$. As the step is enabled in the trace Z, we know that some fail step $\langle$F-Fail$, p\rangle$ must have occurred in one of these other steps before the recovery step. For this reason, the recovery step is enabled and Z'_j can be applied to C'_j.

Similarity. It suffices to show that, given any trace Z of length n valid from c, with the last two steps not causally ordered, swapping these two steps does not change the final configuration obtained by application of the trace to c. That is, given $Z = Z_{prefix} : [Z_{n-2}] : [Z_{n-1}]$, we need to show that for $Z' = Z_{prefix} : [Z_{n-1}] : [Z_{n-2}]$ we have $Z(c)_n = Z'(c)_n$. In simpler terms, it suffices to show that $[Z_{n-2}] : [Z_{n-1}](Z_{prefix}(c)) = [Z_{n-1}] : [Z_{n-2}](Z_{prefix}(c))$. The proof is done by case analysis on the last two steps. By assumption, Z_{n-2} and Z_{n-1} are not causally ordered, therefore: by the first clause of the Definition 6.5 of the casual order, neither of them can be an F-Recover step; by the second one, the steps cannot be on the same processor; by third one, one of the steps may not be consuming messages produced by the other step; and by the fourth one, there are no transitive causal dependencies between the steps. This leaves the following two cases.

First, if one of them is an F-Fail step, then reordering the two steps will not affect the final configuration. This is the case because the steps are on different processors, therefore the steps will not be influenced by the local state effect of either step, further, the F-Fail step does not consume or produce any messages, therefore it cannot influence the other step. For this reason, the application of these two steps will produce the same final configuration.

Second, we have the case that the two last steps are I-Event or I-Border steps on different processors. As the steps are on different processors, they do not influence each other in terms of the local processor state. Moreover, by the third and the fourth clauses of the causality Definition 6.5, a step on one processor may not produce a message which is consumed by the other processor. Therefore, both steps are enabled and consume and produce the same messages regardless of the order of the steps. Thus, we have the same final configuration regardless of the ordering. **QED**

7.2 Full Proof of the Failure Transparency Theorem

In this section, we are going to prove the main theorem of the thesis, i.e., that the stateful dataflow model is failure transparent. In order to do it rigorously, we first prove a series of auxiliary lemmas, which are tailored to be and are used in the proof of the main theorem, culminating with the failure transparency proof itself.

Lemma 7.1. (Only Committing Border-Steps Affect Output) *In* I*, except for* S-Step *with local step* I-Border *such that* $\mathrm{gce}(c) \neq \mathrm{gce}(c')$*, all steps are not affecting the output:* $\mathrm{out}(c) = \mathrm{out}(c')$.

Proof. The proof is done by case analysis of the rules in I. In context of $\forall c = \langle \Pi, \Sigma, N, M, M_0 \rangle$, $c' = \langle \Pi', \Sigma', N', M', M_0' \rangle$:

Failure. The fail-rule itself is not introducing any change of output, since it does not change the epoch number and any messages, and the output cannot be changed without changing the epoch number or the messages.

The goal is to show that:

$$(\{\text{S-Step}, \text{F-Fail}\} \vdash c \Rightarrow c') \implies \mathrm{out}(c) = \mathrm{out}(c')$$

First, let's show that:

$$(\{\text{S-Step}, \text{F-Fail}\} \vdash c \Rightarrow c') \implies \mathrm{gce}(c) = \mathrm{gce}(c') \wedge M = M'$$

Since the only local step is F-Fail, the only possible execution step is:

$$\frac{\begin{array}{ccc} & \Pi_p = \mathsf{TK}\langle f, S, o \rangle & \Sigma_p = \langle a, \sigma_{\mathrm{V}} \rangle \\ X = \varepsilon & \Sigma_p' = \langle a, \mathtt{fl} \rangle & (N_p', M') = X(N_p, M) \end{array}}{\langle \Pi, \Sigma, N, M, D \rangle \Rightarrow \langle \Pi, \Sigma\big[p \mapsto \Sigma_p'\big], N\big[p \mapsto N_p'\big], M', D \rangle}$$

By the definition of action application, $\varepsilon(N_p, M) = (N_p, M)$, therefore $M = M'$. $\Sigma' = \Sigma\big[p \mapsto \langle a, \mathtt{fl} \rangle\big]$ where $\Sigma_p = \langle a, \sigma_{\mathrm{V}} \rangle$. From this follows that:

$$\forall q, a, a', \sigma_{\mathrm{V}}, \sigma_{\mathrm{V}}'.\ \Sigma_p = \langle a, \sigma_{\mathrm{V}} \rangle \wedge \Sigma_p' = \langle a', \sigma_{\mathrm{V}}' \rangle \implies a = a'$$

Therefore:

$$\begin{aligned}\mathrm{gce}(c) &= \min\left\{\max(\mathrm{dom}(a)) \;\middle|\; \Sigma_p = \langle a, \sigma_V \rangle\right\} \\ &= \min\left\{\max(\mathrm{dom}(a)) \;\middle|\; \Sigma'_p = \langle a, \sigma_V \rangle\right\} = \mathrm{gce}(c')\end{aligned}$$

So, we have derived that:

$$(\{\text{S-STEP}, \text{F-FAIL}\} \vdash c \Rightarrow c') \implies \mathrm{gce}(c) = \mathrm{gce}(c') \wedge M = M'$$

Now let's show that:

$$\mathrm{gce}(c) = \mathrm{gce}(c') \wedge M = M' \implies \mathrm{out}(c) = \mathrm{out}(c')$$

The conclusion directly follows from the premises and the definition of out:

$$\begin{aligned}\mathrm{out}(c) &= \left\{ns\,\langle e, d\,\rangle \;\middle|\; (ns\,\langle e, d\,\rangle) \in M \wedge e \leq \mathrm{gce}(c)\right\} \\ &= \left\{ns\,\langle e, d\,\rangle \;\middle|\; (ns\,\langle e, d\,\rangle) \in M' \wedge e \leq \mathrm{gce}(c')\right\} = \mathrm{out}(c')\end{aligned}$$

Combined, the proved implications give us the desired result.

Event-step. They do not change gce and do not produce messages with $e \leq \mathrm{gce}$.

Ordinary border-step. By definition, it is not increasing gce, and no new messages with $e \leq \mathrm{gce}(c)$ can be produced (since all processors have $e > \mathrm{gce}(c)$, and the actions have e in messages).

Recovery. The recovery rule is not changing the output. Here we should reason about the lcs, particularly, that it preserves the output messages (which is done in its definition quite directly).

The goal is to show that:

$$(\{\text{S-STEP}, \text{F-RECOVER}\} \vdash c \Rightarrow c') \implies \mathrm{out}(c) = \mathrm{out}(c')$$

By applying the recovery rule, we get:

$$c' = \mathrm{lcs}(c)$$

By definition of lcs, gce will stay the same, since $\forall \langle a', \sigma'_V \rangle \in \Sigma'.\ \mathrm{dom}(a') =$

$\{\text{gce}(c)\}$; and $M' = M_{in} \cup out(c)$, where $\forall (n\,s\,\langle e, d\rangle) \in M_{in}.\ e > \text{gce}(c)$, which means that they will be not part of $out(c')$. Combined, these two points show that out$(c) =$ out(c'). **QED**

Lemma 7.2. (Order of Steps by Epochs) *For a trace Z of* S-Steps *valid from c in* I*; a step on a processor with epoch number e cannot happen before a step on the same processor with epoch number e' such that $e' < e$.*

Proof. By definitions of local steps, epoch number of a processor can only grow, particularly, for event step and failure step it stays the same, and for border step it is incremented. Therefore, by the second clause of Definition 6.5, any step on a processor of epoch $e' < e$ happens before all steps on the same processor of epoch e. Causal order is partial order, therefore the step of e cannot happen before e'. **QED**

Lemma 7.3. *For a trace Z valid from a configuration c and its permutation Z', such that total order on epochs below e is preserved, and causal order is preserved throughout the permutation, the outputs of corresponding configurations obtained by committing border steps of epochs below e are equal.*

Proof. The output of a committing border step is solely defined by its epoch number e and by the subset of existing messages with epoch number below e. Steps above e do not produce messages below e. **QED**

Lemma 7.4. *For a trace Z of* S-Steps *valid from c, $Z(c) = C$, such that the first n steps are on epochs below or equal to* gce$(C_{|C|-1})$ *and all the later steps are on epochs above* gce$(C_{|C|-1})$*, it is true that* lcs$(C_n) =$ lcs$(C_{|C|-1})$.

Proof. By induction on steps above gce$(C_{|C|-1})$.

The base case is that there are no such steps, that is, the length of prefix $i = n$, therefore lcs$(C_n) =$ lcs(C_i).

In the induction step, we have this property for a trace prefix of length i, and we have to show that it is also true for the trace prefix of length $i+1$. By definition, the latest common snapshot is affected only by the steps below the greatest common epoch and the greatest common epoch number. The new step does not change gce, and an S-Step can only produce messages. Since the steps are on epochs above gce$(C_{|C|-1})$, the messages produced by such a step are also of epochs above the

greatest common epoch, and therefore their production does not affect the latest common snapshot. **QED**

Theorem 6.12. (Failure Transparency of the Implementation Model) $I \Vdash_K^{\text{out}} F$, i.e., *the set of rules* $\mathrm{I} = \{\text{S-Step}, \text{S-AbsX}, \text{S-AbsP}, \text{I-Event}, \text{I-Border}\} \cup \mathrm{F}$ *is failure transparent with respect to the failure rules* $\mathrm{F} = \{\text{F-Fail}, \text{F-Recover}\}$ *for the observability function* out *and the set of initial configurations* K.

Proof. Expanding the definitions, we need to prove that:

$$\forall\, c \in K.\ \forall [C_i]_i^n \in \mathbb{E}_c^{\mathrm{I}}.\ \exists\, [C'_j]_j^m \in \mathbb{E}_c^{\mathrm{I}\setminus\mathrm{F}}.\ \forall n' < n.\ \exists m' < m.\ \mathrm{out}(C_{n'}) = \mathrm{out}(C'_{m'})$$

In other words, for all executions with failures, we have to construct explanations of them in the failure-free model. Let's choose arbitrary $c \in K$ and $[C_i]_i^n \in \mathbb{E}_c^{\mathrm{I}}$, and construct a $[C'_j]_j^m$ such that:

$$[C'_j]_j^m \in \mathbb{E}_c^{\mathrm{I}\setminus\mathrm{F}} \wedge \forall n' < n.\ \exists m' < m.\ \mathrm{out}(C_{n'}) = \mathrm{out}(C'_{m'})$$

We do the construction in two steps, gradually *mapping* the original trace to the explaining failure-free trace.

The key idea of construction of the failure-free explanatory execution is to keep only effectful steps from the original execution, that is, the steps which are not discarded by a recovery. Thus, we first construct a trace, then we prove that the trace is valid from c, by which we get the execution $[C'_j]_j^m$. Finally, we prove that this execution is an observable explanation of the original execution.

Trace Construction. First, let's take the valid trace Z of steps used to produce C, *i.e.,* $Z(c) = C$.

Now, let's split the trace by recovery rules into subtraces which we call generations, so that we have a sequence G of generations. Each generation is a sequence of S-Steps ending with a F-Recover; in case of the last generation there may be no recovery in the end.

Now, for each generation G_i we construct a new reordered trace $G'_i = \mathrm{filter}(x \in G_i.\ \mathrm{epoch}(x) \le e) : \mathrm{filter}(x \in G_i.\ \mathrm{epoch}(x) > e)$, where e is the epoch used as gce in the recovery step of the generation. In it, we move all discarded steps to be after the last committing border step in the generation. By discarded step, we mean a step with an epoch number larger or equal to the epoch number to which the

recovery is done. We can do so, preserving validity of trace, since no step from epoch e' can happen before a step from epoch e such that $e < e'$ (Lemma 7.2). A step can be discarded only if it was not checkpointed, by which we get that all the moved steps have an epoch number larger than the epoch number of the last committing border step. In other words, we use Lemma 7.3, since by construction of G'_i, total order is preserved for steps below epoch e and causal order is preserved throughout the reordering.

Now, for each G'_i, we construct a new trace G''_i by removing the discarded steps and the recovery step. The trace is still valid by trace prefix validity, Lemma 6.4.

Finally, we concatenate all G''_i to get the trace Z'. Here, we need to provide for each generation a derivation of a step from the last committed configuration to the first configuration of the next generation. This is possible since recovery is done exactly to the last committed configuration, Lemma 7.4, and since there is such a step from it in the original valid trace.

Execution Construction. By the reasoning above, we have a valid trace Z', so there is a corresponding execution which we will use as $[C_i]^n_i$. The last statement we have to prove is that it is an explanation of the original execution, i.e.:

$$\forall n' < n.\ \exists m' < m.\ \mathrm{out}(C_{n'}) = \mathrm{out}(C'_{m'})$$

Now we have to provide a map from n' to m', such that the outputs are the same.

The output is only changed by a committing border-step I-Border* by Lemma 7.1.

Second, the used reorderings are not changing the output of any committed epoch e since the steps are on epoch e' such that $e' > e$, by Lemma 7.3.

Therefore, we can say that for each generation, the output of corresponding committing border-steps are the same, and not changed by other steps. We map n' to m' of the configuration immediately after the most recent committing border-step, and the outputs at n' and m' will be the same. **QED**

7.3 Mechanizaiton

The definitions and the stateful dataflow model presented in this thesis are captured formally in Coq, the most well-known proof assistant capable of automatic

check of the proofs expressed in it. The definitions and the model are available in a git repository*, and are formulated in a hopefully reusable way, so that it is easier to plug the failure transparency definition or the streaming model to a different type of systems than the ones based on the Asynchronous Barrier Snapshotting protocol. Concretely, the definitions and the model are parametrized so that it is possible to provide a different model, definitions of processors, their states, local steps, etc.

However, in contrast to the original idea of fully formalizing the proofs in Coq, the proofs were done in a more informal way. The task of their complete mechanization turned out to be more challenging than expected. The presented proof is based on reasoning about possibly infinite sequences, particularly it transforms them by splitting them, reordering their elements, dropping parts of them and concatenating them. Although this type of proof is possible to formalize in Coq, the existing automation tools in Coq are tailored more to inductive reasoning, and it seems especially hard to handle infinite sequences in it the same way it is done in the presented proof. The author believes that the proofs could be formalized relatively easily in Coq or another proof assistant, once suitable utilities are provided. Their development is, however, a task beyond the scope of this thesis.

7.4 Discussion

The presented proofs were carried out by a single Master student with some help from his two supervisors. This testifies to the feasibility of the presented proof technique, definitions and the stateful dataflow model. Moreover, the mechanization efforts show that the definitions and the model are rigorous enough to be captured in a proof assistant. Overall, the evaluation shows that the definitions, the stateful dataflow model and the trace-mapping proof technique can be successfully applied together to raise confidence in reliability of failure handling of complex real-world systems.

*https://github.com/aversey/abscoq/

8 Related Work

In this chapter, the related work is discussed and acknowledged. Although the list of the related works is not exhaustive, the author believes that it covers the most relevant papers and books that are related to the topics discussed in this thesis; and includes all works which influenced this thesis directly. The chapter is structured in two parts, the first part focuses on the formal approaches to failure transparency and the second part focuses on the practices of failure handling in distributed and especially stateful dataflow systems.

Failure Transparency and Observational Explainability. This thesis started with a search of a rigorous definition of the idea of failure transparency. One of the first discoveries was the work of Lowell and Chen [1999], in which they discussed failure transparency in the context of consistent failure recovery protocols. The work provides a way to reason about relations of faulty and failure-free executions with respect to certain equivalence functions, a concept which inspired the observability functions of this thesis. Equivalence functions enable, for example, to filter out the events duplicated due to failures. They seem to not be flexible enough, however, to conveniently express the complexities of the failure handling of stateful dataflow systems.

Around the same time as Lowell and Chen, Gärtner [1999] discussed general models for fault-tolerant computing. Similar to this thesis, Gärtner explicitly marks the rules of normal operation and the rules related to failures. He then discussed various properties and forms of fault-tolerant programs, providing a taxonomy of failure-handling models. In the context of Gärtner's work, the definition of failure transparency presented in this thesis corresponds to *fail-safe fault tolerance*. He furthermore claims that in order to be "truly fault-tolerant", *i.e., failure-masking*, such a fail-safe system should be also proven to have liveness [Alpern and Schneider 1985; Lamport 1977]. Although liveness is not discussed in this thesis, it was proven for the presented stateful dataflow model in [Veresov et al. 2024b].

Other sources of inspiration for the devised failure transparency definition, particularly for the observational explainability part of it, are the previous definitions of refinement (*e.g.,* Temporal Logic of Actions [Lamport 2002, 1994], Compositional Compiler Correctness [Patterson and Ahmed 2019]), implementation (*e.g.,* I/O Automata [N. Lynch and Tuttle 1989; N. A. Lynch and Stark 1989]), and sim-

ulation. In simplified terms, one set of executions implements another if it is a subset thereof, modulo stuttering and multistep executions; the idea is similar to refinement mappings. The provided definition of observational explainability, in some sense, extends the notion of refinement to directly include a refinement mapping [Abadi and Lamport 1988] on both sides via observability functions. It resembles notions from related work such as observational equivalence [Burckhardt et al. 2021] and observational refinement [Kallas et al. 2023]. The terms are used in these works without capturing them in definitions, as it is done in this thesis; however, as is obvious from the names, these papers are one of the major inspirations of this work. One of the main limitations of the related work is that the known approaches are mostly inductive; and while induction is an essential way of mathematical reasoning, its application to systems with complex history manipulations seems to be complicated. In practice, this need to reason about past or future events makes it necessary to use ghost variables [Marcus and Pnueli 1996], also known as auxiliary variables [Lamport and Merz 2017], in the proofs of such inductive properties. In contrast, the presented definition of observational explainability is based on direct mappings of traces, enabling easier reasoning about whole histories of executions.

Simultaneously with the search for a suitable definition of failure transparency, a search for an appropriate proof technique was also conducted. Mukherjee et al. [2019] propose a failure transparency theorem for their system of reliable state machines: an execution of the implementation is a refinement of an execution without failures “with respect to its observable behavior”, reminiscent of the failure transparency definition of this thesis. The work also presents the reliable state machines model in two parts, as a collection of global step rules and a collection of local step rules, which is similar to the way the stateful dataflow model is presented here; a similar layered presentation of a programming model can be observed in [Burckhardt et al. 2021]. The analysis of other works resulted in distinguishing certain inspiring proof ideas. For example, the use of simulation relations [Burckhardt et al. 2021; Kallas et al. 2023] and explicit failure modeling [Kallas et al. 2023; Mukherjee et al. 2019; Tardieu et al. 2023] provided valuable insights. The devised failure transparency proof is also inspired by the proof of causal consistency of the snapshots obtained by the Asynchronous Barrier Snapshotting [Carbone 2018], particularly in the key insight on using a causal relation to reason about executions.

Resilient Programming Models and Failure Handling. Resilient programming models are ones which provide automatic failure-handling mechanisms, thus freeing their users from the need to react to failures themselves. Stateful dataflow systems provide this resiliency and consist of a number of systems, out of which Apache Flink [Carbone et al. 2015] and Portals [Spenger et al. 2022], especially influenced this thesis, as they are the primary systems for which the stateful dataflow model was developed. However, there are other notable resilient programming models and systems not based on dataflow programming. Namely, Durable Functions [Burckhardt et al. 2021] are presented in three models with different abstraction levels, while Reliable State Machines [Mukherjee et al. 2019] are formalized in global and local semantics, similarly to the stateful dataflow model.

Some models, in contrast, provide the users with manual failure-handling constructs. For example, this is the case with the actor model. The telecom industry successfully uses the failure-handling constructs of Erlang, such as actor monitors and supervision [Armstrong 1996; Armstrong et al. 1993]. Moreover, other programming models such as Argus [Liskov 1988] and transactors [Field and Varela 2005] provide constructs for transactions, which in turn can be used for building reliable services.

A general overview of rollback-recovery protocols was given by Elnozahy et al. [Elnozahy et al. 2002], comparing between checkpointing-based and logging-based protocols. Stateful dataflow systems use either checkpointing, or a combination of the two [Akidau, Bradshaw, et al. 2015; Balazinska et al. 2005; Carbone et al. 2015; Dean and Ghemawat 2004; Shah et al. 2004; Silva et al. 2016; Wang et al. 2019; Zaharia et al. 2012]. The MapReduce system performs failure recovery by detecting failed nodes, and replaying the computation from sources or from persisted intermediate results [Dean and Ghemawat 2004]. Apache Spark, in contrast, improves the recovery by replaying from the sources through what is called lineage recovery [Zaharia et al. 2012]. A similar idea is used in a dynamic dataflow system within Ray [Wang et al. 2019].

This thesis is focused on the Asynchronous Barrier Snapshotting protocol used in Apache Flink, which, in contrast to the previous works, uses an asynchronous checkpointing technique [Carbone et al. 2015]. It has been proven to provide high performance and has since been widely adopted [Siachamis et al. 2024]. The current version of Apache Flink's runtime offers an opt-in feature for "unaligned checkpoints", which allow the checkpoint markers to be treated at a higher priority,

decreasing the end-to-end latency at the cost of some overhead as buffered events may become part of the snapshots [The Apache Software Foundation 2020]. Other adaptations of the Flink protocol include Clonos [Silvestre et al. 2021], which logs the nondeterminism to facilitate faster partial recovery after failures. Failure recovery remains an open research topic, as it has great impact on the performance characteristics of fault-tolerant systems [Siachamis et al. 2024].

9 Conclusions

The research question of this thesis, i.e.:

How to define and prove failure transparency of stateful dataflow systems?

is answered by providing a novel definition of failure transparency and a novel trace-mapping proof technique. The answer is evaluated by performing a detailed proof of failure transparency of a model of stateful dataflow systems, particularly the ones with failure-handling protocols based on Asynchronous Barrier Snapshotting.

Moreover, in this thesis, the first small-step operational semantics of the Asynchronous Barrier Snapshotting protocol is provided under the name of the stateful dataflow model. The novel definition of failure transparency is applicable to a wide range of systems expressed in small-step operational semantics with explicit failure rules. The novel trace-mapping proof technique is estimated to be more suitable for reasoning about systems with rollback checkpoint recoveries than the classic inductive approaches. Finally, the devised definitions and models are formalized in Coq in a parametrized way, so that it should be possible to reuse them in other works.

9.1 Reflections

The work is mostly theoretical, and as such the immediate practical implications of it are limited. However, the work is a step towards a fully verified stateful dataflow programming stack. In this context, the work could benefit society by ensuring reliability of a range of distributed systems.

Distributed systems are the backbone of modern society, backing free access to information, thus contributing to spread of education. They are also essentially the only way to perform large-scale computations, such as aerodynamics modeling used in the design of airplanes, DNA sequencing used in medical research, or for training AI systems which are used in wide range of research, engineering and customer servicing. Distributed systems are also essential for building highly reliable systems controlling critical infrastructure, such as nuclear power plants or air traffic control systems. Ensuring reliability of distributed systems therefore is

beneficial for the society as it reduces the amount of the mental effort needed to build and control these systems.

However, distributed systems can also be used to perform massive scale surveillance, to collect and analyze large amounts of personal data. Despite that, the author firmly believes that these negative effects can be mitigated by developing better understanding of and higher control over distributed systems. Although the work presented in this thesis focuses on failure handling, it seems possible to reuse the formalization of stateful dataflow systems done in this thesis to formally reason about privacy and security of these systems. For example, in order to ensure compliance of a system to GDPR, it can be useful to prove that a request to delete all data related to a specific user is fully propagated through a stateful dataflow system. In turn, it should be easier to do once it is known that failures are handled correctly and the request cannot disappear because of them.

Overall, the author believes that the work presented in this thesis will make a positive impact on the society and environment.

9.2 Future Work

The work presented in this thesis opens up several lines of future work. In this section some of them are highlighted.

Machine-checkable Proofs. The proofs presented in this thesis are not machine-checkable. However, the state-of-the-art in computer science is to provide machine-checkable proofs using one of the available proof assistants, the most well-known of them being Coq. It would be interesting and useful to construct a machine-checkable proof for the failure transparency theorem, preferably by developing utilities which can be later reused in other work. This would in turn enable easier propagation of the failure transparency theorem to other systems.

End-to-End Failure Transparency. Another line of future work is to extend the failure transparency theorem to cover composite systems, parts of which are using different failure-handling mechanisms; or, in other words, to make the failure transparency property composable. Composability here means that if two systems are independently proven to be failure transparent, then it should be possible to derive failure transparency of a composite system, in which these two subsystems interact. This would be a significant step towards formal verification of real-world

distributed systems, which are often heterogeneous.

Further Abstraction. This work mainly focuses on correctness of the failure handling mechanisms; however, it is also important to provide feasible and easy-to-understand programming models of complex real-world systems. It would be interesting to provide an even cleaner model of stateful dataflow systems in which no trace of failure-handling mechanism is left and to prove that the stateful dataflow model presented in this thesis is observationally explainable by this even cleaner model. In particular, the failure-free part of stateful dataflow model still contains the notion of epochs, barriers and local persistent storages of tasks; however, none of these details are necessary in the failure-free context. The failure transparency theorem presented in this thesis could be reused by combining it with the new results using Lemma 5.8 on transitivity and Lemma 5.9 on composability of observational explainability.

Prove Failure Transparency for More Systems. The failure transparency definition presented in this thesis is applicable to a wide range of systems expressed in small-step operational semantics with explicit failure rules. Therefore, it would be interesting to prove failure transparency for other systems, which still lack such a proof.

Bibliography

Martín **Abadi** and Leslie **Lamport**. **1988**. *The Existence of Refinement Mappings*. In: Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988. IEEE Computer Society, 165–175. DOI: 10.1109/LICS.1988.5115.

Tyler **Akidau**, Alex **Balikov**, Kaya **Bekiroğlu**, Slava **Chernyak**, Josh **Haberman**, Reuven **Lax**, Sam **McVeety**, Daniel **Mills**, Paul **Nordstrom**, and Sam **Whittle**. **2013**. *MillWheel: fault-tolerant stream processing at internet scale*. Proc. VLDB Endow., 6, 11, (Aug. 2013), 1033–1044. DOI: 10.14778/2536222.2536229.

Tyler **Akidau**, Robert **Bradshaw**, Craig **Chambers**, Slava **Chernyak**, Rafael **Fernández-Moctezuma**, Reuven **Lax**, Sam **McVeety**, Daniel **Mills**, Frances **Perry**, Eric **Schmidt**, and Sam **Whittle**. **2015**. *The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing*. Proc. VLDB Endow., 8, 12, 1792–1803. DOI: 10.14778/2824032.2824076.

Bowen **Alpern** and Fred B. **Schneider**. **1985**. *Defining Liveness*. Inf. Process. Lett., 21, 4, 181–185. DOI: 10.1016/0020-0190(85)90056-0.

Joe **Armstrong**. **1996**. *Erlang–a Survey of the Language and its Industrial Applications*. In: Proc. INAP. Vol. 96, 16–18.

Joe **Armstrong**, Robert **Virding**, and Mike **Williams**. **1993**. *Concurrent programming in ERLANG*. Prentice Hall. ISBN: 978-0-13-285792-5.

Magdalena **Balazinska**, Hari **Balakrishnan**, Samuel **Madden**, and Michael **Stonebraker**. **2005**. *Fault-tolerance in the Borealis distributed stream processing system*. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005. Ed. by Fatma Özcan. ACM, 13–24. DOI: 10.1145/1066157.1066160.

Sebastian **Burckhardt**, Chris **Gillum**, David **Justo**, Konstantinos **Kallas**, Connor **McMahon**, and Christopher S. **Meiklejohn**. **2021**. *Durable functions: semantics for*

stateful serverless. Proc. ACM Program. Lang., 5, OOPSLA, 1–27. DOI: 10.1145/3485510.

Christian **Cachin**, Rachid **Guerraoui**, and Luís E. T. **Rodrigues**. **2011**. *Introduction to Reliable and Secure Distributed Programming (2. ed.)* Springer. ISBN: 978-3-642-15259-7. DOI: 10.1007/978-3-642-15260-3.

Paris **Carbone**. **2018**. *Scalable and Reliable Data Stream Processing*. Ph.D. Dissertation. Royal Institute of Technology, Stockholm, Sweden. https://nbn-resolving.org/urn:nbn:se:kth:diva-233527.

Paris **Carbone**, Gyula **Fóra**, Stephan **Ewen**, Seif **Haridi**, and Kostas **Tzoumas**. **2015**. *Lightweight Asynchronous Snapshots for Distributed Dataflows*. CoRR, abs/1506.08603. arXiv: 1506.08603.

Badrish **Chandramouli**, Jonathan **Goldstein**, Mike **Barnett**, Robert **DeLine**, Danyel **Fisher**, John C. **Platt**, James F. **Terwilliger**, and John **Wernsing**. **2014**. *Trill: a high-performance incremental query processor for diverse analytics*. Proc. VLDB Endow., 8, 4, (Dec. 2014), 401–412. DOI: 10.14778/2735496.2735503.

K. Mani **Chandy** and Leslie **Lamport**. **1985**. *Distributed Snapshots: Determining Global States of Distributed Systems*. ACM Trans. Comput. Syst., 3, 1, 63–75. DOI: 10.1145/214451.214456.

Joonwon **Choi**, Muralidaran **Vijayaraghavan**, Benjamin **Sherman**, Adam **Chlipala**, and **Arvind**. **2017**. *Kami: a platform for high-level parametric hardware specification and its modular verification*. Proc. ACM Program. Lang., 1, ICFP, 24:1–24:30. DOI: 10.1145/3110268.

Jeffrey **Dean** and Sanjay **Ghemawat**. **2004**. *MapReduce: Simplified Data Processing on Large Clusters*. In: 6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004. Ed. by Eric A. Brewer and Peter Chen. USENIX Association, 137–150. http://www.usenix.org/events/osdi04/tech/dean.html.

Edsger W. **Dijkstra**. **1976**. *A Discipline of Programming*. Prentice-Hall. ISBN: 013215871X. https://www.worldcat.org/oclc/01958445.

Edsger W. **Dijkstra**. **1975**. *Guarded Commands, Nondeterminacy and Formal Derivation of Programs*. Commun. ACM, 18, 8, 453–457. DOI: 10.1145/360933.360975.

E. N. **Elnozahy**, Lorenzo **Alvisi**, Yi-Min **Wang**, and David B. **Johnson**. **2002**. *A survey of rollback-recovery protocols in message-passing systems*. ACM Comput. Surv., 34, 3, 375–408. DOI: 10.1145/568522.568525.

Matthias **Felleisen** and Daniel P. **Friedman**. **1987**. *Control operators, the SECD-machine, and the λ-calculus*. In: Formal Description of Programming Concepts - III: Proceedings of the IFIP TC 2/WG 2.2 Working Conference on Formal Description of Programming Concepts - III, Ebberup, Denmark, 25-28 August 1986. Ed. by Martin Wirsing. North-Holland, 193–222.

Matthias **Felleisen**, Daniel P. **Friedman**, Eugene E. **Kohlbecker**, and Bruce F. **Duba**. **1987**. *A Syntactic Theory of Sequential Control*. Theor. Comput. Sci., 52, 205–237. DOI: 10.1016/0304-3975(87)90109-5.

John **Field** and Carlos A. **Varela**. **2005**. *Transactors: a programming model for maintaining globally consistent distributed state in unreliable environments*. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005. Ed. by Jens Palsberg and Martín Abadi. ACM, 195–208. DOI: 10.1145/1040305.1040322.

Marios **Fragkoulis**, Paris **Carbone**, Vasiliki **Kalavri**, and Asterios **Katsifodimos**. **2024**. *A survey on the evolution of stream processing systems*. VLDB J., 33, 2, 507–541. DOI: 10.1007/S00778-023-00819-8.

Yupeng **Fu** and Chinmay **Soman**. **2021**. *Real-time Data Infrastructure at Uber*. In: SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021. Ed. by Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava. ACM, 2503–2516. DOI: 10.1145/3448016.3457552.

Felix C. **Gärtner**. **1999**. *Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments*. ACM Comput. Surv., 31, 1, 1–26. DOI: 10.1145/311531.311532.

Philipp **Haller**, Heather **Miller**, and Normen **Müller**. **2018**. *A programming model and foundation for lineage-based distributed computation*. J. Funct. Program., 28, e7. DOI: 10.1017/S0956796818000035.

C. A. R. **Hoare**. **1969**. *An Axiomatic Basis for Computer Programming*. Commun. ACM, 12, 10, 576–580. DOI: 10.1145/363235.363259.

C. A. R. **Hoare**. **1978**. *Communicating sequential processes*. Commun. ACM, 21, 8, (Aug. 1978), 666–677. DOI: 10.1145/359576.359585.

Gabriela **Jacques-Silva**, Fang **Zheng**, Daniel **Debrunner**, Kun-Lung **Wu**, Victor **Dogaru**, Eric **Johnson**, Michael **Spicer**, and Ahmet Erdem **Sariyüce**. **2016**. *Consistent regions: guaranteed tuple processing in IBM streams*. Proc. VLDB Endow., 9, 13, (Sept. 2016), 1341–1352. DOI: 10.14778/3007263.3007272.

Simon L. Peyton **Jones**, Alastair **Reid**, Fergus **Henderson**, C. A. R. **Hoare**, and Simon **Marlow**. **1999**. *A Semantics for Imprecise Exceptions*. In: Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, May 1-4, 1999. Ed. by Barbara G. Ryder and Benjamin G. Zorn. ACM, 25–36. DOI: 10.1145/301618.301637.

Roope **Kaivola**, Rajnish **Ghughal**, Naren **Narasimhan**, Amber **Telfer**, Jesse **Whittemore**, Sudhindra **Pandav**, Anna **Slobodová**, Christopher **Taylor**, Vladimir A. **Frolov**, Erik **Reeber**, and Armaghan **Naik**. **2009**. *Replacing Testing with Formal Verification in Intel CoreTM i7 Processor Execution Engine Validation*. In: Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings (Lecture Notes in Computer Science). Ed. by Ahmed Bouajjani and Oded Maler. Vol. 5643. Springer, 414–429. DOI: 10.1007/978-3-642-02658-4_32.

Konstantinos **Kallas**, Haoran **Zhang**, Rajeev **Alur**, Sebastian **Angel**, and Vincent **Liu**. **2023**. *Executing Microservice Applications on Serverless, Correctly*. Proc. ACM Program. Lang., 7, POPL, 367–395. DOI: 10.1145/3571206.

James C. **King**. **1971**. *A Program Verifier*. In: IFIP Congress. https://api.semanticscholar.org/CorpusID:6021364.

Gerwin **Klein**, Kevin **Elphinstone**, Gernot **Heiser**, June **Andronick**, David A. **Cock**, Philip **Derrin**, Dhammika **Elkaduwe**, Kai **Engelhardt**, Rafal **Kolanski**, Michael **Norrish**, Thomas **Sewell**, Harvey **Tuch**, and Simon **Winwood**. **2009**. *seL4: formal verification of an OS kernel*. In: Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009. Ed. by Jeanna Neefe Matthews and Thomas E. Anderson. ACM, 207–220. DOI: 10.1145/1629575.1629596.

Jay **Kreps**, Neha **Narkhede**, Jun **Rao**, et al.. **2011**. *Kafka: A distributed messaging system for log processing*. In: Proceedings of the NetDB. Vol. 11. Athens, Greece, 1–7.

Ramana **Kumar**, Magnus O. **Myreen**, Michael **Norrish**, and Scott **Owens**. **2014**. *CakeML: a verified implementation of ML*. In: The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014. Ed. by Suresh Jagannathan and Peter Sewell. ACM, 179–192. DOI: 10.1145/2535838.2535841.

Leslie **Lamport**. **1977**. *Proving the Correctness of Multiprocess Programs*. IEEE Trans. Software Eng., 3, 2, 125–143. DOI: 10.1109/TSE.1977.229904.

Leslie **Lamport**. **2002**. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley. ISBN: 0-3211-4306-X. http://research.microsoft.com/users/lamport/tla/book.html.

Leslie **Lamport**. **1998**. *The Part-Time Parliament*. ACM Trans. Comput. Syst., 16, 2, 133–169. DOI: 10.1145/279227.279229.

Leslie **Lamport**. **1994**. *The Temporal Logic of Actions*. ACM Trans. Program. Lang. Syst., 16, 3, 872–923. DOI: 10.1145/177492.177726.

Leslie **Lamport**. **1978**. *Time, Clocks, and the Ordering of Events in a Distributed System*. Commun. ACM, 21, 7, 558–565. DOI: 10.1145/359545.359563.

Leslie **Lamport**. **1993**. *Verification and Specifications of Concurrent Programs*. In: A Decade of Concurrency, Reflections and Perspectives, REX School/Symposium, Noordwijkerhout, The Netherlands, June 1-4, 1993, Proceedings (Lecture Notes

in Computer Science). Ed. by J. W. de Bakker, Willem P. de Roever, and Grzegorz Rozenberg. Vol. 803. Springer, 347–374. DOI: 10.1007/3-540-58043-3_23.

Leslie **Lamport** and Stephan **Merz**. **2017**. *Auxiliary Variables in TLA+*. CoRR, abs/1703.05121. arXiv: 1703.05121.

Xavier **Leroy**. **2009**. *Formal verification of a realistic compiler*. Commun. ACM, 52, 7, 107–115. DOI: 10.1145/1538788.1538814.

Tom **Lianza** and Chris **Snook**. **2020**. *A Byzantine failure in the real world*. (2020). https://blog.cloudflare.com/a-byzantine-failure-in-the-real-world/.

Barbara **Liskov**. **1988**. *Distributed Programming in Argus*. Commun. ACM, 31, 3, 300–312. DOI: 10.1145/42392.42399.

David E. **Lowell** and Peter M. **Chen**. **1999**. *The Theory and Practice of Failure Transparency*. Tech. rep. University of Michigan.

Nancy **Lynch** and Mark **Tuttle**. **1989**. *An introduction to Input/Output automata*. CWI-Quarterly, 2, 3, 219–246. Also available as MIT Technical Memo MIT/LCS/TM-373, Laboratory for Computer Science, Massachusetts Institute of Technology.

Nancy A. **Lynch** and Eugene W. **Stark**. **1989**. *A Proof of the Kahn Principle for Input/Output Automata*. Inf. Comput., 82, 1, 81–92. DOI: 10.1016/0890-5401(89)90066-7.

Doug **Madory**. **2021**. *Facebook's historic outage, explained*. (2021). https://www.kentik.com/blog/facebooks-historic-outage-explained/.

Yancan **Mao**, Zhanghao **Chen**, Yifan **Zhang**, Meng **Wang**, Yong **Fang**, Guanghui **Zhang**, Rui **Shi**, and Richard T. B. **Ma**. **2023**. *StreamOps: Cloud-Native Runtime Management for Streaming Services in ByteDance*. Proc. VLDB Endow., 16, 12, 3501–3514. DOI: 10.14778/3611540.3611543.

Monica **Marcus** and Amir **Pnueli**. **1996**. *Using Ghost Variables to Prove Refinement*. In: Algebraic Methodology and Software Technology, 5th International Conference, AMAST '96, Munich, Germany, July 1-5, 1996, Proceedings (Lecture Notes in Com-

puter Science). Ed. by Martin Wirsing and Maurice Nivat. Vol. 1101. Springer, 226–240. DOI: 10.1007/BFB0014319.

John **McCarthy**. **1960**. *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I*. Commun. ACM, 3, 4, 184–195. DOI: 10.1145/367177.367199.

Bertrand **Meyer**. **1992**. *Applying "Design by Contract"*. Computer, 25, 10, 40–51. DOI: 10.1109/2.161279.

Ragnar **Mogk**, Joscha **Drechsler**, Guido **Salvaneschi**, and Mira **Mezini**. **2019**. *A fault-tolerant programming model for distributed interactive applications*. Proc. ACM Program. Lang., 3, OOPSLA, 144:1–144:29. DOI: 10.1145/3360570.

Suvam **Mukherjee**, Nitin John **Raj**, Krishnan **Govindraj**, Pantazis **Deligiannis**, Chandramouleswaran **Ravichandran**, Akash **Lal**, Aseem **Rastogi**, and Raja **Krishnaswamy**. **2019**. *Reliable State Machines: A Framework for Programming Reliable Cloud Services*. In: 33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom (LIPIcs). Ed. by Alastair F. Donaldson. Vol. 134. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 18:1–18:29. DOI: 10.4230/LIPICS.ECOOP.2019.18.

Susan S. **Owicki** and David **Gries**. **1976**. *Verifying Properties of Parallel Programs: An Axiomatic Approach*. Commun. ACM, 19, 5, 279–285. DOI: 10.1145/360051.360224.

Daniel **Patterson** and Amal **Ahmed**. **2019**. *The next 700 compiler correctness theorems (functional pearl)*. Proc. ACM Program. Lang., 3, ICFP, 85:1–85:29. DOI: 10.1145/3341689.

Benjamin C. **Pierce**. **2002**. *Types and Programming Languages*. (1st ed.). The MIT Press. ISBN: 0262162091.

Gordon D **Plotkin**. **1981**. *A structural approach to operational semantics*.

Henri **Poincaré**. **1905**. *Science and Hypothesis*. Trans. by William John Greenstreet. The Walter Scott Publishing Co., Ltd. https://en.wikisource.org/wiki/Science_and_Hypothesis.

Alastair **Reid**, Rick **Chen**, Anastasios **Deligiannis**, David **Gilday**, David **Hoyes**, Will **Keen**, Ashan **Pathirane**, Owen **Shepherd**, Peter **Vrabel**, and Ali **Zaidi**. **2016**. *End-to-End Verification of Processors with ISA-Formal*. In: Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II (Lecture Notes in Computer Science). Ed. by Swarat Chaudhuri and Azadeh Farzan. Vol. 9780. Springer, 42–58. DOI: 10.1007/978-3-319-41540-6_3.

Adam **Satariano**. **2020**. *Google's apps crash in a worldwide outage*. (2020). https://www.nytimes.com/2020/12/14/business/google-down-worldwide.html.

Mehul A. **Shah**, Joseph M. **Hellerstein**, and Eric A. **Brewer**. **2004**. *Highly-Available, Fault-Tolerant, Parallel Dataflows*. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004. Ed. by Gerhard Weikum, Arnd Christian König, and Stefan Deßloch. ACM, 827–838. DOI: 10.1145/1007568.1007662.

Konstantin **Shvachko**, Hairong **Kuang**, Sanjay **Radia**, and Robert **Chansler**. **2010**. *The Hadoop Distributed File System*. In: IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2012, Lake Tahoe, Nevada, USA, May 3-7, 2010. Ed. by Mohammed G. Khatib, Xubin He, and Michael Factor. IEEE Computer Society, 1–10. DOI: 10.1109/MSST.2010.5496972.

George **Siachamis**, Kyriakos **Psarakis**, Marios **Fragkoulis**, Arie van **Deursen**, Paris **Carbone**, and Asterios **Katsifodimos**. **2024**. *CheckMate: Evaluating Checkpointing Protocols for Streaming Dataflows*. CoRR, abs/2403.13629. arXiv: 2403.13629. DOI: 10.48550/ARXIV.2403.13629.

Gabriela Jacques da **Silva**, Fang **Zheng**, Daniel **Debrunner**, Kun-Lung **Wu**, Victor **Dogaru**, Eric **Johnson**, Michael **Spicer**, and Ahmet Erdem **Sariyüce**. **2016**. *Consistent Regions: Guaranteed Tuple Processing in IBM Streams*. Proc. VLDB Endow., 9, 13, 1341–1352. DOI: 10.14778/3007263.3007272.

Pedro F. **Silvestre**, Marios **Fragkoulis**, Diomidis **Spinellis**, and Asterios **Katsifodimos**. **2021**. *Clonos: Consistent Causal Recovery for Highly-Available Streaming Dataflows*. In: SIGMOD '21: International Conference on Management of Data, Vir-

tual Event, China, June 20-25, 2021. Ed. by Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava. ACM, 1637–1650. DOI: 10.1145/3448016.3457320.

Jonas **Spenger**, Paris **Carbone**, and Philipp **Haller**. **2022**. *Portals: An Extension of Dataflow Streaming for Stateful Serverless*. In: Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2022, Auckland, New Zealand, December 8-10, 2022. Ed. by Christophe Scholliers and Jeremy Singer. ACM, 153–171. DOI: 10.1145/3563835.3567664.

Olivier **Tardieu**, David **Grove**, Gheorghe-Teodor **Bercea**, Paul **Castro**, Jaroslaw **Cwiklik**, and Edward A. **Epstein**. **2023**. *Reliable Actors with Retry Orchestration*. Proc. ACM Program. Lang., 7, PLDI, 1293–1316. DOI: 10.1145/3591273.

The Apache Software Foundation. **2020**. *Unaligned Checkpoints FLIP-76*. https://issues.apache.org/jira/browse/FLINK-14551. Accessed on 2024-03-28. (2020).

Aleksey **Veresov**, Jonas **Spenger**, Paris **Carbone**, and Philipp **Haller**. **2024a**. *Failure Transparency in Stateful Dataflow Systems*. In: 38th European Conference on Object-Oriented Programming, ECOOP 2024, September 16-20, 2024, Vienna, Austria (LIPIcs). Ed. by Jonathan Aldrich and Guido Salvaneschi. Vol. 313. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 42:1–42:31. DOI: 10.4230/LIPICS.ECOOP.2024.42.

Aleksey **Veresov**, Jonas **Spenger**, Paris **Carbone**, and Philipp **Haller**. **2024b**. *Failure Transparency in Stateful Dataflow Systems (Technical Report)*. CoRR, abs/2407.06738. arXiv: 2407.06738. DOI: 10.48550/ARXIV.2407.06738.

Stephanie **Wang**, John **Liagouris**, Robert **Nishihara**, Philipp **Moritz**, Ujval **Misra**, Alexey **Tumanov**, and Ion **Stoica**. **2019**. *Lineage stash: fault tolerance off the critical path*. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019. Ed. by Tim Brecht and Carey Williamson. ACM, 338–352. DOI: 10.1145/3341301.3359653.

Matei **Zaharia**, Mosharaf **Chowdhury**, Tathagata **Das**, Ankur **Dave**, Justin **Ma**, Murphy **McCauly**, Michael J. **Franklin**, Scott **Shenker**, and Ion **Stoica**. **2012**. *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing*.

In: Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012. Ed. by Steven D. Gribble and Dina Katabi. USENIX Association, 15–28. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia.

TRITA-EECS-EX-2025:82
Stockholm, Sweden 2025

www.kth.se