A Programming Paradigm for Reliable Applications in a Decentralized Setting

Ragnar Mogk

November 7, 2021

A Programming Paradigm for Reliable Applications in a Decentralized Setting

vom Fachbereich Informatik
der Technischen Universität Darmstadt

zur Erlangung des akademischen Grades eines
Doktor-Ingenieurs (Dr.-Ing.) genehmigte

Dissertation von
Ragnar Mogk

Erstreferentin	Prof. Dr. Mira Mezini
Zweitreferent	Prof. Dr. Volker Markl

Darmstadt, 2021

Ragnar Mogk:
A Programming Paradigm for Reliable Applications in a Decentralized Setting
Darmstadt, Technische Universität Darmstadt
Hochschulkennziffer: Darmstadt – D 17
Tag der Einreichung: 27.01.2021
Tag der mündlichen Prüfung: 10.03.2021
Jahr der Veröffentlichung auf TUprints: 2021
URN: urn:nbn:de:tuda-tuprints-194035

"Reliability can be purchased only at the price of simplicity" – Hoare [105].
Said in the context proofs of program correctness that solve real problems, but are difficult to give for programs with complex designs.

"What would you tell your grandmother when she asks what you study?" – Sergio [13]

Programming languages were conceived to make it easier for humans to provide instructions to the machines. In the early days, computers essentially executed a sequence of instructions changing some internal state to compute an output. Machines were big and slow, and many of us shared a single machine. Languages were designed to take advantage of those machines. However, the machines have changed. They are small and fast, and each of us has many machines, they outnumber us by a quickly increasing amount. Programming languages have not changed. We still talk to machines as if they were big and slow, this makes it hard for the machines to execute programs efficiently, and makes it hard for us to express what we want. The world has changed. Applications must – above all else – become reliable. In this thesis, we study how to talk about applications that interact with humans, and how to do so in a way that is both good for machines and for people.

Abstract

The use of applications has changed together with the underlying computing platform. The modern computer is no longer a big piece of office equipment that is booted to execute a single task producing a single output. Instead, we have many interconnected devices – smartphones, laptops, routers, Internet of Things gadgets, and even some venerable desktop computers we still use to get that heavy work done. Furthermore, ubiquitous connectivity with the Internet, and thus collaboration with other people and their set of devices has drastically changed how people expect applications to work.

For such distributed, interactive, and collaborative applications, we currently lack a declarative fault-tolerant programming paradigm with easy-to-reason high-level guarantees. We want to empower developers from organizations of all sizes to be able to create reliable applications that solve their users needs. Thus, the central question in this thesis is: How to automate fault tolerance for such applications?

To answer this question, we present a novel approach to automatic fault tolerance using a high-level programming paradigm. Our goal is to provide future developers with a paradigm that reduces the challenge posed by failures in interactive applications similar to how a garbage collector reduces the challenge of managing memory. To do so, our programming paradigm abstracts from the notion of changes in data, thus removing the need to handle failure cases differently and providing developers a single set of properties to always rely on.

Zusammenfassung

Die Nutzung von Anwendungen hat sich zusammen mit der zugrunde liegenden Computerplattform verändert. Der moderne Computer ist nicht länger ein großes Stück Büroausstattung das gestartet wird, um eine einzelne Aufgabe auszuführen und eine einzelne Ausgabe zu produzieren. Stattdessen haben wir viele miteinander verbundene Geräte – Smartphones, Laptops, Router, Internet-der-Dinge-Gadgets und sogar einige altgediente Desktop-Computer, die wir immer noch verwenden, um umfangreiche Arbeiten zu erledigen. Darüber hinaus hat die allgegenwärtige Konnektivität mit dem Internet und damit die Zusammenarbeit mit anderen Menschen und deren Geräten die Erwartungen an die Funktionsweise von Anwendungen drastisch erweitert.

Für solche verteilten, interaktiven und kollaborativen Anwendungen fehlt uns derzeit ein deklaratives, fehlertolerantes Programmierparadigma mit einfach zu verstehenden Garantien. Wir wollen Entwicklern aus verschiedenen Organisationen die Möglichkeit geben, zuverlässige Anwendungen zu erstellen, die die Anforderungen ihrer Benutzer erfüllen. Die zentrale Frage in dieser Arbeit lautet daher: Wie kann man die Fehlertoleranz für solche Anwendungen automatisieren?

Um diese Frage zu beantworten, präsentieren wir einen neuartigen Ansatz zur automatischen Fehlertoleranz unter Verwendung eines Programmierparadigmas. Unser Ziel ist es, zukünftigen Entwicklern ein Paradigma an die Hand zu geben, das die Herausforderung von Fehlern in interaktiven Anwendungen reduziert, ähnlich wie ein Garbage Collector die Herausforderung der Speicherverwaltung reduziert. Um dies zu erreichen, abstrahiert unser Programmierparadigma wie sich Daten über die Zeit ändern. Dadurch entfällt die Notwendigkeit, Fehlerfälle unterschiedlich zu behandeln und Entwickler können sich immer auf einen festen Satz von Eigenschaften verlassen.

Acknowledgments

My first thanks go to Mira who used her experience as a PhD advisor and spared no effort to provide feedback on many versions of this document. Mira, I really enjoyed working with you on my dissertation and many other documents before, and I am very happy for how much better the result is thanks to that. Many thanks to Volker Markl – I could not have hoped to find for anyone more fitting as a second reviewer, and I am very glad that you seem to have enjoyed this work. Finally, thanks to Carsten Binnig, Matthias Hollick, and Kristian Kersting for being the rest of my committee and helping to make the whole process as enjoyable as it was.

Naturally, this thesis is the result of my whole research career and there are too many people that helped with that to thank them all, so here is just a small selection. First, thank you, Gudrun! You make the life of so many PhD students so much easier, and I can't even count the number of times you helped me with all sorts of problems, thank you! Also thanks to my office mates – Matthias, Pascal, and Mirko – who always had to listen to me talking about whatever ideas I had at the moment and yet never threw me out of the office. It has always been fun working with you. Thank you to Sylvia and Joscha from whom I learned so much about all the challenges faced during a PhD and who always seemed so much better at so many things than I am – I also made it! Thanks for your help. Thank you to Guido, who has been my inofficial second supervisor – a luxury that I think more students should be able to enjoy. And finally, thank you to everyone at STG, to my friends, and my family. You only get this blanket statement here, but I hope you know you are appreciated.

Contents

Chapter 1

Introduction

The use of applications has changed together with the underlying computing platform. The modern computer is no longer a big piece of office equipment that is booted to execute a single task producing a single output. Instead, we have many interconnected devices – smartphones, laptops, routers, Internet of Things gadgets, and even some venerable desktop computers we still use to get that heavy work done. Furthermore, ubiquitous connectivity with the Internet, and thus collaboration with other people and their set of devices has drastically changed how people expect applications to work.

The core tenets of such applications are dictated by the needs of their users: *distribution* because tasks span multiple devices, *interactivity* to react to the users and their environment, and *collaboration* to facilitate users working together. Think about your favorite web-based office suite as an example. Distribution exists because of the execution platform. Parts of the office application are executed on my smartphone or desktop to provide me with the functionality to edit documents and other parts of the application are executed a server in the cloud to make use of available storage, processing power, and central coordination. In general, distribution is necessary because different resources are available on different devices. Interactivity exists to serve the user. I decide when I edit a document, when I want to share it, or if I want to shut off my computer and do something else. The behavior of an application is driven by these external interactions, thus the application must always be ready to react timely and with useful results. Collaboration between multiple users requires applications to communicate but also to function independently. I want feedback on a document thus I share the document with my reviewer. I continue working on the document while my reviewer adds comments and eventually we synchronize our work again. For the remainder of this document, we call such distributed, interactive, and collaborative applications *dicApps*.

The importance of the niche addressed by dicApps has been discussed in the literature before [207] and their value to users and companies has steadily increased. Users of dicApps consume, play, create, share, and collaborate on a wide range of digital goods. Many users own multiple devices (smartphone, tablet, laptop) and it is important to them to work on any device, share content with other users, and

not lose any of their work. Companies in all sectors use internal applications to manage their employees, their inventory, and their business processes. Due to current trends such as cloud computing, the Software-as-a-Service model, and the easier maintenance model of thin clients (i.e., applications are just websites) many of those internal applications have become dicApps. For companies, it is important that dicApps are always available to allow their employees to be productive and to not lose data. For both users and companies the increase of remote working due to the Corona pandemic will accelerate the trend towards dicApps, while at the same time diversifying the network and devices that those dicApps are deployed on. Some popular examples of dicApps that most users know include messaging (instant messengers, chats), activity streams (social networks), data visualization applications (e.g., Calendars, Jupyter notebooks), multi-user collaborative applications (e.g., Trello, Google Docs, Microsoft Office), and multi-player online games. Private dicApps of companies have less widely known specific instances, but many people have worked with internal room booking, time management, contract management, task planning, resource scheduling, and many other similar systems.

This thesis proposes a programming paradigm to *democratize* the development and deployment of dicApps.

1.1 Problem Statement

While dicApps are very popular, most of the big examples of dicApps are currently organized and controlled by centralized entities such as Google and Microsoft. In many cases, these organizations create both the software and provide their execution environment and data storage. With new development in cloud offerings, such as serverless computing, these centralized entities are also trying to seize control over data and applications of companies. While we do believe that the trend towards dicApps is desirable, we wish to avoid the issues and limitations of centralization. One issue is the rising concern over privacy as evident through wide-reaching regulations such as the General Data Protection Regulation. Another issue is the inefficiency of transferring large amounts of data between the central storage and the user causing latency and making the user dependent on the network. Privacy, latency, and reliability will always make it more desirable to keep data as close to the user as possible. Moreover, we believe that centralization is detrimental to the usability of dicApps, because the business model often of centralized services relies on keeping users within their ecosystem. Still, many features of centralized dicApps are useful: users and companies wish to have their data processed on more powerful devices than they own and wish to be able to easily collaborate on shared data. We believe that users and companies should be free to choose where to store data and where to execute computations, thus allowing them to optimize for their choice of latency, network traffic, reliance on the network, cost, and privacy. Others have also noted the need to take back control over data [36] and the necessity to free processing of data from vendor lock-in [194]. These solutions focus on data

and data processing – solutions we complement by improving the development of dicApps.

To overcome the issues of centralization we believe it is necessary to democratize the development of dicApps. First, the development of dicApps must become independent of the runtime providers to solve the issues of centralization and the lack of user choice. Specifically, it must become cost-efficient to develop dicApps whose design is independent of the challenges of the underlying deployment. Second, the development of dicApps must become cheaper – optimally even cheaper than the development of traditional applications. The reasoning is that there will be constantly new business use cases and specific user needs that should be addressed by individual dicApps, but currently the required development resources only make dicApps for large groups of people profitable.

As a means for the democratization, we believe the right choice is a programming paradigm tailored to the development of dicApps. A programming paradigm is both flexible enough to implement diverse dicApps but also restricted enough to abstract away specific issues and automatically solve challenges faced by dicApps. In the remainder of this section, we discuss the technical challenges faced by dicApps to understand why existing solutions are insufficient and require reinventing individual solutions for each application.

Developing dicApps is challenging – even without distribution. Their control flow is inverted ("inversion of control") compared to non-interactive applications, because dicApps react to external events and have no control over what happens next. Reactions are traditionally modeled in some form of continuation-passing style where callbacks are registered to be executed when events happen. Designs using continuation-passing style are fragile, hard to maintain, and hard to reason about – bad enough to become known as callback hell [72]. Supporting collaboration in dicApps worsens the problems of callback-based solutions, because instead of a single user interacting with the system there are now multiple users leading to more permutations of interactions that must be handled [121]. In particular, callback-based communication makes handling of exceptional conditions and failures during the execution of an application challenging [170].

Existing dicApps have – by necessity – a distributed architecture with some components running on cloud platforms and others running on the connected (end-user) devices. A distributed architecture – compared to only running locally and accessing remote data, or to only running the application remotely – reduces network traffic, improves latency, enhances privacy, and enables offline usage [205]. In turn, however, a distributed architecture introduces several failure modes that need to be addressed. First, individual devices may fail: mobile devices running out of battery, mobile and web applications are regularly terminated by the runtime when the user switches context, cloud servers may crash and recover again by restoring persisted state. Second, communication between devices may be temporarily disrupted, causing messages to get lost or be duplicated. In a distributed architecture, where the control flow is spread across several devices, disconnects and crashes lead to partial failures, where half of an application is still running

and the other half is crashed. Exposing the reasoning about such a partial failure state to the programmer leads to fragile designs, because it is nearly impossible to manually predict all the possible interactions.

In summary, the challenge with dicApps is that they combine failure modes of distributed systems, with the complexity of control flow for interactive and collaborative applications. To address these challenges a programming paradigm must be able to express reactive control flow in a way that lends itself to automatic fault tolerance.

1.2 State of the Art

To get a better understanding of the solution space, we first discuss the state of the art for fault tolerance and for handling interactive applications. However, as we will see, there are no solutions that handle everything at once.

Actors-based Programming Paradigms

Actor-based approaches [15, 17, 25, 118] are the state of the art in programming dicApps. However, they only ensure fault tolerance of individual actors [33]. Communication failures and crashes affecting multiple actors are left to be handled by the application logic using only low-level message passing, thus merely extending cumbersome and error-prone continuation-passing style solutions. Actor supervision only replaces crashed or disconnected actors with fresh instances, but recovery of accumulated local state and synchronization thereof with the overall application state is left to be programmed in user-defined handlers. Moreover, there is no automated support for re-establishing application consistency after disconnects. Crashed or unreachable actors are replaced by fresh instances. By default, they lose any accumulated state in the process, but user-defined handlers can be used to, e.g., recover state. This must be implemented manually and tailored towards each individual actor, though, and is therefore cumbersome and fragile. Orleans [33] addresses this issue, by providing dedicated means for actors to declare persistent state. However, application programmers still need to implement when to persist and restore that state so that consistency is upheld. There are, indeed, solutions to save the state across an actor restart (e.g., storing state in the Mnesia [140] database), but state saving and restoration is completely left to the programmer.

Programming Paradigms for Data Processing Applications

Reasoning about high-level properties of distributed applications when device failure may occur is generally challenging [90]. Significant progress is made in this respect for certain types of distributed applications thanks to specialized programming models offering fault-tolerant programming abstractions. When distributed applications are developed using these abstractions, then the runtime automatically takes care of handling (certain) failure cases for the applications – what we refer

to as automatic fault tolerance. We look at two common directions for solutions providing automatic fault tolerance.

Automated fault tolerance is currently best covered by the approaches targeting data streaming and processing on controlled server clusters such as Spark [206], Flink [19], and Kafka [124]. With these systems, developers declare how data should be processed usually in a system specific domain specific language and the runtime then takes of correct deployment of functionality and the management of data regardless of failures. These systems typically provide many specific building blocks each tuned for fault-tolerant and efficient execution of a specific task or integration with a specific external system. However, for dicApps these systems have several shortcomings. First, these systems focus on applications that process data without user interaction, thus their programming model does not include any way to immediately react to external interactions. Seconds, the runtime of these systems must be deployed on a cluster architecture and controlled by a central managing service, thus making it unsuitable for applications running on a user's device.

Another more flexible approach are abstractions for structured data [58, 128, 143] which restrict the programming model in order to tolerate unreliable network conditions (c.f., CALM [103]). Compared to data processing systems, the building blocks of these approaches are flexible enough to support a wider variety of application use cases, although nothing specific to interactive applications. While these solutions can work well, they do not help to build complete dicApps, because it remains the developer's responsibility to ensure that combining these building blocks results in fault-tolerant applications.

Programming Paradigms with Declarative Interactivity

Unfortunately, there is no declarative fault-tolerant solution for the whole spectrum of dicApps. Actors cannot provide the same level of automated fault tolerance as cluster approaches, nor can they automatically connect shared state of multiple devices, because they lack knowledge of the overall dataflow in the application. The above approaches with automatic fault tolerance do not target applications, where external events and user inputs determine how computations unfold (inversion of control) and individual application components own data and operate on data independently.

A current declarative approach for designing interactive applications is functional reactive programming [59, 75]. Functional reactive programming provides a declarative, modular, and composable programming paradigm [179, 182] that deals with continuous inputs of new data and events. These languages usually sidestep the issue with fault tolerance and delegate the responsibility for handling both distribution and connectivity to the developer and the embedding language paradigm.

However, the declarative nature allows the programmer to state the intent of the application instead of specifying concrete execution behavior. Declarative definitions not only improve code clarity, but also leave concrete execution behavior unspecified – the underlying runtime can freely change as long as the intended

semantic is kept intact. In our solution we will exploit this freedom provided by declarative programming paradigm to automatically make dicApps reliable.

The declarative model of functional reactive programming has been used before to automate coordination of message propagation between multiple devices [63, 69, 138]. However, none of the existing approaches provide fault tolerance.

1.3 How to Automate Fault Tolerance for Distributed, Interactive, and Collaborative Applications?

For distributed, interactive, and collaborative applications, we lack a declarative fault-tolerant programming paradigm with easy-to-reason high-level guarantees akin to those available for data processing applications. The widespread use of some dicApps (e.g., Google Docs) shows that it is possible to develop correct and useful solutions. However, each example is a one-of-a-kind and has to yet again solve the challenges faced by dicApps. We believe that there are many more future use cases for dicApps, which are currently hindered by the complexity caused by the diverse impact of failures on interactive use. We want to empower developers from organizations of all sizes to be able to create reliable dicApps that solve their users needs. Thus, the central question in this thesis is: How to automate fault tolerance for dicApps?

1.4 Sketch of the Proposed Solution

To answer this question, we present a novel approach to automatic fault tolerance using a high-level programming paradigm. Our goal is to provide future developers with a paradigm that reduces the challenge posed by failures in interactive applications similar to how a garbage collector reduces the challenge of managing memory. To do so, our programming paradigm abstracts from the notion of changes in data, thus removing the need to handle failure cases differently and providing developers a single set of properties to always rely on. Specifically, we propose Cvtr, REScala, and $\mathcal{F}_r$. Cvtr refers to the programming paradigm that enhances existing paradigms to be suitable for dicApps. REScala is a library that embeds the Cvtr paradigm into Scala – a language that already supports functional and object-oriented paradigms. $\mathcal{F}_r$ is the core calculus that models Cvtr and that enables formal reasoning about applications' dataflow within and across individual devices.

Cvtr, the Programming Paradigm

To tackle the challenges faced by developers of dicApps, we design a programming paradigm which provides *convergence, transactions, and reactive abstractions* and refer to this paradigm as Cvtr (pronunciation suggestion: *cuter*). As a programming paradigm, Cvtr determines how applications are designed using convenient abstractions within the limitations of what is efficient and correct on all target platforms. Moreover, for such application Cvtr automates fault tolerance.

We reinterpret the functional reactive programming paradigm as the development model of Cvtr. At the core of Cvtr is a dynamic *flow graph* that describes how an application reacts to external inputs. Each node in the flow graph is called a reactive and describes a single reaction. A declarative API allow to specify these reactions without implying a global sequential execution as typical in imperative code. The resulting declarative specification of the application model provides the runtime enough flexibility to automate correctness without the prohibitive cost.

Reasoning about applications with unstructured reactions is hard. Thus, transactions allow bundling individual low-level reactions into what a user would consider single operation or unit of change. Transactions ensure that all reactions within seem to take effect at the same time, that is, developers never have to worry about intermediate states.

Convergence refers to low-latency distributed applications with offline capability and eventual consistency. Each device has its own flow graph and may always execute transactions locally, thus diverging the state in their flow graph. Once two devices are able to communicate, then their states eventually become consistent. Using transactions for the granularity of convergence aligns distributed consistency guarantees with user expectations, because transactions represent single logical operations for the user. The result is, that while a user may observe effects of a remote transaction in a different order they will only observe all or none of the reactions of a transaction. This model of consistency aligns with a natural understanding that things happening far away may require time to become known locally, but once things settle down the world is still in a consistent state.

Cvtr provides a single set of guarantees that are always provided even when faults occur. This simplifies reasoning for users and developers. Compared to imperative paradigms fault tolerance can be automatically achieved because it is completely up to the runtime how the reactions described by the flow graph are executed within the limits of a transaction. In addition, it is also up to the runtime when remote transactions are applied. Concretely, the flow graph is enhanced with automated crash recovery of device-local dataflow and enhanced with connectivity to remote devices that tolerates disconnects while providing causal consistency. The crash recovery is based on ideas behind consistent snapshots [19] but simplified by the use of transactions. The connectivity is based on eventually consistent replicated data types [46, 96, 186]. As a result of the declarative nature of the programming paradigm, our solution relieves programmers of handling intricate details of achieving reliability for dicApps.

REScala, the Implementation

REScala as discussed in this thesis is an extension of the core language design proposed by Salvaneschi et al. [181]. REScala is a fault-tolerant reactive programming language for developing dicApps. REScala has first-class abstractions for *events* and *signals* that are concrete instances of reactives in the flow graph. An event is a reactive that produces distinct occurrences of values at certain times, e.g., an

event corresponding to an input field produces text when the user submits the input. Events can be derived from each other using operations such as filters or transformations, and they can be aggregated into signals. Signals represent time-changing values, such as the latest text a user submitted. Signals resemble spreadsheet cells where the value of a cell is derived from the values of other cells and a change causes updates of all derived values.

REScala extends the declarative style of reactive programming with syntax for replicated reactives and developer defined snapshots. These extensions are tightly integrated with the local execution of transactions. REScala provides application-wide fault tolerance with little overhead in terms of both performance and syntactic clutter. Furthermore, REScala provides language abstractions for propagating and handling errors at the application level to enable developers to handle faults when the default behavior of REScala is undesirable and to seamlessly integrate application-level fault handling into the flow graph.

Technically, REScala is realized as a shallow domain specific language for Scala. Thus, REScala is fully compatible with arbitrary complex domain model expressed in Scala. It is fully interoperable with the Scala type system to ensure that the application is consistently type checked across all the programming paradigms available in Scala. REScala can be integrated into the APIs of objects in Scala enabling flexible modularization of large applications. The computations over domain objects are expressed as ordinary Scala functions with minimal syntactic overhead – additional syntax serves the purpose of making the boundaries between paradigms clear to developers. REScala works well with all existing Scala tooling including IDEs, compilers to JavaScript, profilers, testing frameworks, and GraalVM native image.

$\mathcal{F}_r$, the Formalization

The goal of $\mathcal{F}_r$ is twofold. First, it provides a core calculus of Cvtr, thus providing a precise definition of what the paradigm entails. Second, $\mathcal{F}_r$ is used to state and prove the guarantees of the paradigm. $\mathcal{F}_r$ is defined as an operational semantics that specifies the behavior of dicApps, their interactions with the external world and the embedding programming language, the process of executing transactions on the flow graph, and the communication of state between devices.

1.5 Contributions of this Thesis

The key contribution of this thesis is to demonstrate that automatic fault tolerance can be provided for dicApps. Along the way, this thesis makes several individual contributions. We provide an introduction to REScala, its API usage from a developers point of view, the internal design of the system, how it is embedded into existing applications, and how REScala can be extended by developers. The implementation supports the development of dicApps by providing automatic fault tolerance in a

distributed setting, abstractions that allow causally consistent connectivity, and a programming paradigm suited for interactive applications.

We formalize the model based on our core calculus $\mathcal{F}_r$ and prove transactional guarantees for local applications, causal consistency for connected devices, and correct restoration after partial crashes for distributed applications. The solutions for transactions, causal consistency, and restoration solve individual issues, but combined provide fully automatic fault tolerance for all failures in our system model.

In the second part of this thesis, we provide a wide range of extensions and use cases for REScala to demonstrate the wide applicability of the programming paradigm. Specifically we include the following extensions and use cases:

- We show that exceptions (in the style of Java) can be seamlessly integrated with the flow graph.
- We demonstrate that $\mathcal{F}_r$ is a suitable foundation to compile a restricted form of a flow graph to embedded devices. The resulting executable has no additional runtime overhead compared to an imperative implementation.
- We present an initial prototype for advanced debugging and live programming of REScala applications.
- We discuss how Cvtr both adds a new view on existing research on state-based convergent replicated datatypes (CRDTs) and solves issues with composing multiple CRTDs.
- We show how to support strong consistency in dicApps by implementing the Raft consensus algorithm as a replicated data type in REScala.

We evaluate our solution from multiple angles. We implement case studies based on common specifications for interactive applications, thus showing that REScala is capable of representing what developers consider typical challenges of interactive applications. Our performance experiments show that REScala is fast for non-distributed applications and has comparable performance to available alternatives for applications involving distribution. We measure that transactions in REScala have overheads as low as 0.4 µs and application performance is completely dominated by the implementation of the business logic. We evaluate the performance overhead of providing fault tolerance against crashes. The results show that the throughput is reduced by less than 20% in common cases, which we deem acceptable, especially since most of the cost is related to serializing in-memory data structures for reliable storage, and existing solutions with automatic fault tolerance exhibit a similar overhead. We compare the performance of an application implemented in REScala with state-of-the-art solutions. The results indicate that performance of connectivity in REScala is dominated by sending network messages, a result we hope to improve by minimizing network messages in the future. However, REScala still beats state-of-the-art systems with strong consistency, because eventual consistency removes the need to wait for other devices.

1.6 Thesis Structure and Relation to Published Papers

Research is a collaborative process. The thesis supersedes (parts of) existing publications that include the work of co-authors. The core of all parts that have been included in this thesis are my own work, but the text, concepts, and solutions have been shaped and improved by my collaborators. Thus, I will state which parts have been included from which publication. Refer to Section 1.7 an overview of my publications.

The thesis is separated into two parts, the first part – Chapter 2 to Chapter 6 – introduce the core programming paradigm, its implementation, and formalization. Chapters in the first part depend on each other in linear order and form the basis for the second part. The second part provides evidence of the wide applicability of the programming paradigm by providing extensions to the core model to support common and specialized use-cases, by providing case studies and experience reports asserting the solution, and by providing empirical evaluation of the performance behavior of the implementation. The chapters in the second part can be read in any order.

Chapter 2 provides an introduction to developing applications in REScala by example and gives a first glimpse at the syntax used for fault tolerance. The examples are a refined version of the introductory examples from prior work [155, 157]

Chapter 3 gives a detailed overview of the potential faults and provides a high-level overview and description of how automatic fault tolerance is achieved given those faults. The text is an updated version of ideas published at ECOOP [155].

Chapter 4 formalizes the programming paradigm and is based on the work published at OOPSLA [157]. Compared to the high-level introduction, the formalization in this chapter uses a simplified syntax and focuses on the internal concepts that must be understood by developers in order to use the programming paradigm efficiently.

Chapter 5 states and proves the property of the programming paradigm. The proved properties include glitch freedom, maximally parallel execution of transactions, and complete and correct restoration of snapshots. These properties are taken from the OOPSLA publication [157], but updated for the changes made to the formalization. In addition, we added proofs for causal consistency that have not been published before.

Chapter 6 bridges between the formal model and the actual implementation. This chapter provides the high-level explanation of the structure of the source code of REScala and how the core packages of REScala interact. The concepts are still the same as in the formalization, but the implementation has better modularity, enabling just parts of the system to be replaced. Only the description of ParRP in Section 6.6 is based on published work [151], but the collaboration with Drechsler et al. [70] was a strong driver for the design.

Chapter 7 introduces error propagation to REScala and thus allows the programming paradigm to be embedded into languages that use exceptions – such as Scala. This extension is based in parts on work published at ECOOP [155] but has been expanded to include a thorough discussion of alternative designs.

Chapter 8 describes how the Cvtr paradigm can be ahead-of-time compiled to enable the use of the paradigm on embedded devices. This work is based on the collaboration with Sterz et al. [192] to which the compiler and language design was my contribution.

Chapter 9 shows the flexibility of REScala by enabling debugging and a limited form of live programming we call live tuning. The original implementation of the debugger was done by Salvaneschi and Mezini [178] but I contributed the support for the debugger in REScala. I also contributed the modified version of the debugger supporting live tuning published at the Live workshop [158] which this chapter is based on.

Chatper 10 discusses alternatives and extensions to the replication model that has been introduced in the first part of the thesis. In particular, this chapter discusses the challenges faced by alternative solutions to implement dicApps and why Cvtr does not suffer from the same problems. Yet, using this analysis we can take many of the advantages of existing ongoing research and apply those to Cvtr.

Chapter 11 describes several case studies. This chapter includes a case study from the collaboration with Baumgärtner et al. [32]. I contributed the implementation and the description of that case study as found in this chapter.

Chapter 12 summarizes our experience with programming using REScala and some similar languages and was first published at the programming experience workshop [156]. The chapter argues why REScala is such a promising approach to developing dicApps even when not worrying about faults and contrast various alternative approaches and their shortcomings.

Chapter 13 provides performance experiments using REScala. This chapter includes reasoning and evaluation that were done in the context of the collaboration with Drechsler et al. [69, 70], but the content in the form presented here has never been published before.

Chapter 14 discusses related work. We try to use this opportunity to provide the historic context of the research areas that have influenced our work and that we build on. The content is an aggregation of pieces published before, but all the text of the pieces and the resulting overall discussion are entirely my own work.

1.7 List of Publications

The full list of publications is as follows. All publications have been peer reviewed, though not all of the venues provide a formal publication process. The publications are sorted by year ascending.

Conference and Journal Publications

1. Joscha Drechsler, Guido Salvaneschi, Ragnar Mogk, and Mira Mezini. Distributed REScala: An update algorithm for distributed reactive programming. In Andrew P. Black and Todd D. Millstein, editors, *Proceedings of the 2014*

ACM International Conference on Object-Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014, pages 361–376. ACM, 2014. doi: 10.1145/2660193.2660240. URL https://doi.org/10.1145/2660193.2660240.

2. Artur Sterz, Lars Baumgärtner, Ragnar Mogk, Mira Mezini, and Bernd Freisleben. DTN-RPC: remote procedure calls for disruption-tolerant networking. In *2017 IFIP Networking Conference, IFIP Networking 2017 and Workshops, Stockholm, Sweden, June 12-16, 2017*, pages 1–9. IEEE Computer Society, 2017. doi: 10.23919/IFIPNetworking.2017.8264848. URL https://doi.org/10.23919/IFIPNetworking.2017.8264848.

3. Joscha Drechsler, Ragnar Mogk, Guido Salvaneschi, and Mira Mezini. Thread-safe reactive programming. *Proc. ACM Program. Lang.*, 2(OOPSLA):107:1–107:30, 2018. doi: 10.1145/3276477. URL https://doi.org/10.1145/3276477 (Conceptual relations to parts of Chapter 6 and Chapter 13).

4. Ragnar Mogk, Lars Baumgärtner, Guido Salvaneschi, Bernd Freisleben, and Mira Mezini. Fault-tolerant distributed reactive programming. In Todd D. Millstein, editor, *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*, volume 109 of *LIPIcs*, pages 1:1–1:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi: 10.4230/LIPIcs.ECOOP.2018.1. URL https://doi.org/10.4230/LIPIcs.ECOOP.2018.1 (Chapters 2, 3, and 7).

5. Ragnar Mogk, Joscha Drechsler, Guido Salvaneschi, and Mira Mezini. A fault-tolerant programming model for distributed interactive applications. *Proc. ACM Program. Lang.*, 3(OOPSLA):144:1–144:29, 2019. doi: 10.1145/3360570. URL https://doi.org/10.1145/3360570 (Chapters 4 and 5).

6. Lars Baumgärtner, Jonas Höchst, Patrick Lampe, Ragnar Mogk, Artur Sterz, Pascal Weisenburger, Mira Mezini, and Bernd Freisleben. Smart street lights and mobile citizen apps for resilient communication in a digital city. In *IEEE Global Humanitarian Technology Conference, GHTC 2019, Seattle, WA, USA, October 17-20, 2019*, pages 1–8. IEEE, 2019. doi: 10.1109/GHTC46095.2019.9033134. URL https://doi.org/10.1109/GHTC46095.2019.9033134 (Chapter 11).

7. Artur Sterz, Matthias Eichholz, Ragnar Mogk, Lars Baumgärtner, Pablo Graubner, Matthias Hollick, Mira Mezini, and Bernd Freisleben. ReactiFi: Reactive programming of Wi-Fi firmware on mobile devices. *The Art, Science, and Engineering of Programming*, 5(2):4, 2020. doi: 10.22152/programming-journal.org/2021/5/4. URL https://doi.org/10.22152/programming-journal.org/2021/5/4 (Chapter 8).

Workshop Publications

1. Ragnar Mogk. Reactive interfaces: Combining events and expressing signals. In *Workshop on Reactive and Event-based Languages & Systems (REBLS)*, 2015. URL `https://2015.splashcon.org/track/rebls2015#event-overview`.

2. Ragnar Mogk, Guido Salvaneschi, and Mira Mezini. Reactive programming experience with REScala. In Stefan Marr and Jennifer B. Sartor, editors, *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming, Nice, France, April 09-12, 2018*, pages 105–112. ACM, 2018. doi: 10.1145/3191697.3214337. URL `https://doi.org/10.1145/3191697.3214337`. (Chapter 12).

3. Ragnar Mogk, Pascal Weisenburger, Julian Haas, David Richter, Guido Salvaneschi, and Mira Mezini. From debugging towards live tuning of reactive applications. In *Workshop on Live Programming Systems*, LIVE, 2019 (Chapter 9).

4. David Richter and Ragnar Mogk. Turning unobservable into unreachable: Dynamic reactive programming without leaks. In *Workshop on Reactive and Event-based Languages & Systems (REBLS)*, 2019. URL `https://2019.splashcon.org/details/rebls-2019-papers/3/Turning-Unobservable-into-Unreachable-Dynamic-Reactive-Programming-without-Leaks`.

Abstracts, Posters, and Demos

1. Ragnar Mogk. Concurrency control for multithreaded reactive programming. In Jonathan Aldrich and Patrick Eugster, editors, *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity, SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 77–78. ACM, 2015. doi: 10.1145/2814189.2815374. URL `https://doi.org/10.1145/2814189.2815374` (Section 6.6).

2. Christian Meurisch, The An Binh Nguyen, Martin Kromm, Andrea Ortiz, Ragnar Mogk, and Max Mühlhäuser. Disvis 2.0: Decision support for rescue missions using predictive disaster simulations with human-centric models. In *26th International Conference on Computer Communication and Networks, ICCCN 2017, Vancouver, BC, Canada, July 31 - Aug. 3, 2017*, pages 1–2. IEEE, 2017. doi: 10.1109/ICCCN.2017.8038474. URL `https://doi.org/10.1109/ICCCN.2017.8038474`.

3. Ragnar Mogk and Joscha Drechsler. REScala – principled distributed reactive programming. In *Poster <Programming>*, 2017. URL `http://tubiblio.ulb.tu-darmstadt.de/101440/`.

1.8 Commonly Used Terms

The terms that are used throughout the thesis are:

Cvtr The convergent transactional reactive paradigm we discuss in this thesis. Refer to Section 1.4.

REScala Our implementation of Cvtr as a Scala domain specific language. Visit `www.rescala-lang.com` for the code repository. Refer to Section 1.4.

$\mathcal{F}_r$ Our core calculus. Refer to Section 1.4.

dicApps Distributed, interactive, and collaborative applications. Refer to Chapter 1.

Flow graph The conceptual model of applications when using Cvtr. Refers to the flow of data between reactives (the nodes of the graph). Refer to Section 1.4

Reactive Abstraction that represent the concept of reactions to some kind of change. Reactives declare what their sources of change are and thus form the flow graph. Refer to Chapter 2.

CRDT Usually refers to state-based convergent replicated data types. The term may also refer to commutative or conflict-free replicated data types, but we will note if the distinction is important. Refer to Section 10.1.

Part I

The Paradigm and its Realization

Chapter 2

An Overview of REScala by Example

In this chapter we introduce REScala by example. We focus on the surface syntax used by developers and thus only marginally cover fault tolerance which is mostly automatic and not visible in the syntax. At its core, the Cvtr programming paradigm is about composing many small declarative reactions to external or internal changes into complex applications. The result of the composition is the flow graph – a declarative model of how an application reacts to external changes. REScala implements the Cvtr programming paradigm as a Scala library. This embedding provides an object-oriented API to interact with the flow graph of REScala.

From a developer's perspective REScala most prominently provides the *event* and *signal* APIs that provide the small building blocks out of which the flow graph is composed. In addition, there are APIs for bundling changes into transactions, storing state into persistent snapshots, and connecting parts of the flow graph to other devices. Readers familiar with the general syntax of REScala may want to skip to Section 2.4 for a more involved example that serves as a quick overview of the syntax while highlighting the most recent additions for fault tolerance.

We do not consider Cvtr as a replacement for existing programming paradigms that are better suited for implementing algorithms, modeling the domain data, and organizing the application into modules. Instead, REScala embeds itself into its host language as shown in Figure 2.1. The green block (on the right) of the figure shows the different parts of REScala and the white blocks of the figure shows how the embedding environment and host language. The overall application is defined in Scala, using packages, objects, and functions to organize the code. The flow graph

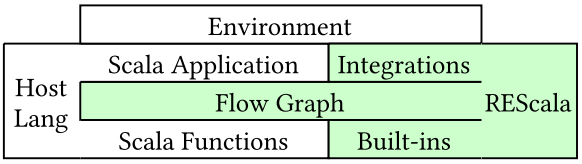

Figure 2.1: Integration of REScala and the host language.

seamlessly integrates with other abstractions in the language, but lives at an intermediate layer, as the graph itself relies on functions written in the host language. Allowing the use of Scala where possible, provides developers the flexibility to integrate existing libraries and programs with REScala. For convenience, REScala also provides integration with the external environment (e.g., UI libraries) and built-in operators that do not require developers to write custom functions. If the scope of the target domain is limited enough to not need custom integrations with the environment, then it is possible to fully remove the host language and write Cvtr programs by only declaring the dataflow with integrations and built-ins (c.f., Chapter 8). However, in the case of REScala, the declaration of the structure of the flow graph always uses Scala syntax. In the rest of this section, we first introduce the concepts of transactions and reactives, and then we discuss a concrete example of how a developer would use REScala, i.e., how the Scala API looks and behaves.

2.1 Transactions and Reactives – the Basic Concepts

First, we need two discuss two general concepts are essential to understand both REScala and Cvtr: transactions and reactives.

Cvtr uses transactions to define its semantics especially regarding time. With dicApps there is often the question if two things occurred at the same time and transactions provide the framework to answer such questions. In Cvtr, everything that is part of a single transaction is considered to happen "at the same time". This leads to a semantics that simplifies the mental model for developers: within a transaction state does not change. In particular, any declared dependency between values is always correct within a transaction. In addition, transactions provide intuitive behavior for users, because users associate reactions with any action that happened at the same time. In REScala, each method of the Scala API causes a single transaction and there is rarely a need to explicitly declare transactions. If required, the `transaction` method bundles multiple interactions into a single transaction.

Reactives represent interdependent discrete time-changing values. Reactives are the nodes of the flow graph. *Sources* are reactives that represent external input from outside the flow graph, such as user interactions or sensor data. *Derived reactives* allow forming complex behavior by deriving behavior from other reactives (including sources). When a source changes, then all derived reactives change (as part of the same transaction) according to their *operator*. The operator is to a reactive is what a class is to an object – an abstract definition of the behavior of many concrete reactives. For example, a reactive with the map operator changes at the same time as its input, but the value is transformed by applying a user-defined function. REScala provides a range of operators including those that combine multiple reactives, aggregate state, and change the flow graph dynamically.

We distinguish between two categories of reactives: events and signals. The distinction is based on the value that a reactive represents at a given point in time. A signal always holds a value, and represents state in the program, e.g., the current text of an input field. An event only has a value while the reactive is active,

```
1  val name: Var[String] = Var("")
2  val text: Evt[String] = Evt()
3  val message: Event[String] = text.map{ l => name.value + ": " + l)}
4  type Room = List[String]
5  val room1: Signal[Room] =
6        message.fold(List.empty){ (history, msg) => msg :: history}
7  val room2: Signal[Room] = Var(List("Me: a constant message"))
8  val roomList: Signal[List[Room]] = Var(List(room1, room2))
9  val index: Var[Int] = Var(0)
10 val selectedRoom: Signal[Signal[Room]] =
11       Signal { roomList.value(index.value) }
12 val roomContent: Signal[Room] = selectedRoom.flatten
13 val contentWidget: Signal[Widget] =
       roomContent.map(makeListWidget)
14 displayUI(name, text, contentWidget)
```

Figure 2.2: Complete chat example code for a single device.

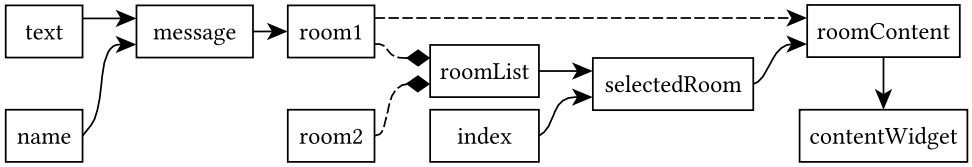

Figure 2.3: The flow graph of the chat example.

and it represents actions that occur in the program, e.g., when a button is clicked. The separation into events and signals is so significant that it is represented in the type system – an event reactive will be of type Event and a signal reactive of type Signal. Signals and events thus each provide their own API that allows to derive new reactives with fitting operators. For example, the signal API generally produces reactives that combine the value of many input signals, while the event API produce reactives that select the value of one of multiple events.

2.2 The Chat Example – REScala Step by Step

Figure 2.2 shows the complete code of a simple chat example we use to introduce the core concepts of the API step by step. The figure includes type annotations showing which reactive is a signal or an event, and what type of value the reactive has. Figure 2.3 shows the flow graph of the running example, with each box representing a reactive with their name taken from the variable they are stored in. Arrows point in the direction of dataflow, i.e., the inputs are at the tail of an arrow and the derived

reactives are at the tip. Reactives with no incoming arrows are sources. The dashed lines represent dynamic edges, and dashed lines with diamonds represent nesting of reactives in other reactives. In the following, we will incrementally introduce and discuss the application code that builds that graph – starting with the following snippet.

```
15 val name = Var("Alice")
16 val text = Evt[String]()
17 val message = text.map{ txt => name.value + ": " + txt) }
```

The code snippet creates a source signal (`Var`) that represents the name of the current user, and a source event (`Evt`) whose occurrences represent chat messages. A `Var` is subtype of a signal and supports the full signal API, but in addition provides an API for imperative code to set the value of the signal, thus providing an integration with imperative Scala code. An `Evt` is analogous to `Var` but extends events. The values of the two source reactives are combined by using the *map* method, which returns a new event reactive (`message`). The operator of the reactive returned by map is specialized with a user-defined function that appends the name and text strings. When we refer to a reactive with a specific type and operator such as `message`, we often use the name of the defining method and say that `message` is a *map event*. Note that it is generally easy to combine signals and events in REScala. The example includes all basic abstractions of REScala: events, signals, sources, and derived reactives. These abstractions are discussed next.

Events

An event is a "container" for occurrences at a discrete point in time. Each occurrence of an event has an associated value and there is no value when the event does not occur. For example, the user sending a new message would be an occurrence of an event, with the text of the message as its value.

Most event operators focus on reacting to occurrences of events. For example, the `map` in Line 17 derives a new event, by applying the passed function every time the input event occurs. Other common operations are filtering of events, where the derived event only has those occurrences that pass a filter function, and the choice `||` operator that occurs whenever the right or the left input occurs, with a bias on the left one if both inputs occur at the same time. Due to the abstract notion of time used by Cvtr "at the same time" means that two occurrences happen in the same transaction, which typically means they are (indirectly) derived from the same input. Correlating event occurrences based on real time – as is common in event processing scenarios – is rare for dicApps, and currently not provided by REScala.

Signals

Values that change over time are represented as signals. Signals always have a value and reactives derived from signals access that value using the `.value` syntax as seen in Line 17. In the example, the derived message is an event reactive,

because `.map` is called on an event. The most common form of deriving signals are signal expressions using the `Signal` keyword. Accessing dependencies is always explicit in REScala – using the `value` method to syntactically mark accessed dependencies. The current value of a derived signal is updated automatically by executing its computation whenever its dependencies change, similar to how formulas in spreadsheets are reevaluated when one of the values they use is changed. Changes of a signal are automatically propagated to *derived* signals that use the signal in their definitions.

Imperative Sources

Imperative sources (`Var` and `Evt` in REScala) allow imperative code to make changes to values in the flow graph. For example, a message is sent using `text.fire("Hello Bob!")`. The typical use for imperative sources is as part of integration libraries. For example, the Swing UI library executes a callback when a user presses a button and we would use the `.fire` method as the callback.

Changing the value of an imperative source is the simplest form of starting a new transaction. When a transaction changes any reactive, then the derived reactive changes accordingly resulting in a chain of reactions – a process called *propagation* (see Section 2.3).

Folds

Fold signals (or simply *folds*) aggregate individual event occurrences – similar to folding over (infinite) lists. A fold, such as in Line 20, creates a signal with an initial value and updates it according to the parameter function every time the event occurs.

```
18 type Room = List[String]
19 val room1: Signal[Room] =
20       message.fold(List.empty){ (history, msg) => msg :: history}
```

We represent chat rooms as their history of messages – implemented as a list with the newest message at its head. The `room1` fold signal accumulates all messages into the history using `::` to add an element to the head of the chat history. Folds are central to the Cvtr paradigm, because their value depends on the values and order of all occurrences of the input event – this is how state is encoded in the flow graph. Or, changing the viewpoint, all operations that depend on value and order of occurrences can be encoded as a fold, thus supporting folds allows us to support all such operations. We will use this to our advantage later.

Signal Expressions

In contrast to events that may or may not have an occurrence in any given transaction, signals always have a current value. Thus, the only way to combine multiple signals is to apply a function to the values of the input signals. The result of the

function is the value of the derived signal. In our chat example we use an index to select one of multiple chat rooms to be then displayed to the user.

```
21 val roomList = Var(List(room1, room2)) // list of two rooms
22 val index = Var(0)
23 val selectedRoom = Signal { roomList.value(index.value) }
```

The `selectedRoom` uses a signal expression to select the one at the position denoted by `index` (Line 23). Signal expressions are blocks of Scala code, but inside the block the `.value` method of input signals is available, similar to the `.map` method of events. In the example, `roomList.value` returns the list of two rooms (at least initially). The function application syntax (the parenthesis) is not part of the `.value` method, but instead is an index-based selection of elements of the list. The plain Scala equivalent of the signal expression would be `roomList(index)`. Signal expressions allow developers to program in a familiar direct coding style, only requiring the use of `.value` to unwrap signals. Using `Signal{}` to delimit the scope of signal expressions and explicitly using `.value` to unwrap signals is a design choice in REScala to make the use of reactives explicit. Using explicit methods to convert between types (signals and their values) is generally considered best practice in Scala and similar typed languages to make it easier for readers of the code to understand what is happening.

Flatten and Dynamic Dependencies

REScala allows for creating new reactives at runtime and add them to the flow graph. New reactives may use any existing reactive as an input, thus adding arbitrary new outgoing edges to the flow graph. Incoming edges can only be changed by the reactive itself and we call reactives with changing inputs *dynamic reactives*. Dynamic reactives are typically derived from reactives that contain other reactives as their values – such as the `selectedRoom` in Line 23. We call reactives that are contained in the value of another a *nested reactive* (or more specifically nested signal or nested event). Both rooms in Line 21 are nested signals inside the `roomList`. Nested reactives on their own are not special, they are values flowing through the flow graph. The value of directly nested signals is accessed using the `.flatten` method.

```
24 val roomContent = selectedRoom.flatten
```

When a nested signal is flattened, then a dynamically changing dependency is created. In this example, `roomContent` depends on either `room1` or `room2`. In Figure 2.3 the dynamic edge is shown with a dashed arrow and nested reactives are shown as dashed lines with diamonds. The `roomContent` signal has two inputs, the first is always the `selectedRoom` signal, the second is the dynamic one and defined by the first input. Signal expressions also support dynamic edges and the flatten operation could be rewritten using a double unwrapping of signals.

```
25 val roomContent = Signal { selectedRoom.value.value }
```

Accessing nested signals using signal expressions produces a dynamic flow graph in the same way as flatten does, but using signal expressions enables directly selecting nested signals inside some other data structure or event multiple levels of nested signals. However, it is always possible to rewrite signal expressions with dynamic access instead use only static access and (multiple) flatten reactives. For example, assume we have a a function `f` that dynamically computes a reactive based on the value of reactive `r` and we want to pass the value of the dynamic reactive to a function `g`. Then the two lines below compute the same result.

```
26 Signal { g(f(r.value).value) }
27 r.map(f).flatten.map(g)
```

Thus, we limit our later formal analysis to flatten reactives without loss of functionality.

Observers

Observers provide a way to integrate with imperative Scala APIs. Observers execute a callback function every time the observed signal changes its value or the observed event has a new occurrence. The callback function is applied to the current value of the reactive. For example, to print the whole chat history to the console every time a new chat message is sent we use the `println` function as an observer.

```
28 roomContent.observe(println)
```

Observers are an API meant to integrate with existing imperative APIs of UI libraries. The completed example in Figure 2.2 uses such an integration with a UI library instead of directly using observers. In that case, the sources and the view on the chat are simply passed to a UI library that understands reactives.

2.3 Peek Behind the Scenes

While developers do not need to understand all the internal details of REScala, it is beneficial to understand the basics parts of the internal design that have an influence on how REScala behaves. We greatly expand on these details in the rest of the thesis.

Propagation

An application that uses REScala combines reactives using map, fold, flatten, etc to build acyclic flow graphs in preparation to react to external stimuli as soon as they arrive. Until now, we have given an intuition that the sources of the flow graph can be directly changed by external code starting a transaction that also changes all derived reactives. In a transaction, the set of *active* reactives – those directly or indirectly derived from the sources changed in the transaction – change their values to reflect the changes in their inputs. We say that active reactives *reevaluate* to compute new values in response to the changes. The process of *reevaluating*

all active reactives is called *propagation*. Conceptually all reevaluations in a single propagation occur at the same time, which defines the semantics of propagation to be synchronous. Synchronous semantics simplifies reasoning about program behavior [35, 182]. However, in a real implementation, the process of propagating new values to all derived reactives is not instantaneous. External factors such as other threads starting a concurrent transaction may occur during another transaction. There is also the potential of a device crash or a disconnect to occur during the processing of a transaction. REScala, our formal model, and the Cvtr paradigm itself are all designed to face these challenges. We formally show that the step wise propagation behaves equivalently to a synchronous system (Chapter 5).

Space and Time Leaks

Some implementations of folds suffer from a problem of infinitely growing memory requirements and computation time [125]. These leaks exist because those implementations require that when a new fold is created, all event occurrences that happened prior to that fold – since the program was started – are visible in the current state of the fold. This is necessary to ensure that each fold that uses the same user-defined function, starting state, and input event also has the same value – a property required by side-effect-free languages such as Haskell. REScala follows the more pragmatic approaches of the Scala language where folds including occurrences starting from the transaction in which the fold is created. Thus, `.fold` is to be considered as having side effects.

Cycles

Dynamic edges allow the creation of cycles in the flow graph. Cycles in the graph can cause unsatisfiable constraints to the flow graph such as $A = B+1$ and $B = A+1$. A greedy propagation algorithm would keep increasing A and B never terminating, and any form of early termination of a transaction would leave a constraint from a derived reactive to be violated. Thus, cycles are not allowed in Cvtr nor in REScala. Dynamic changes that add cycles can be detected by a straightforward graph traversal at the time a dependency changes. However in practice this issue is rare, because the Scala language does not allow cyclic definitions and cyclic graphs require contrived setups involving dynamic reconfigurations that does not occur in normal programs, thus our current stance is that those checks are unnecessary.

2.4 The Shared Calendar Example

We discuss a second case study to show a more involved domain model for an interactive application, but also to show the syntax for snapshots and replication used in REScala. A user of the shared calendar application can create new calendar entries and select the displayed week. The calendar will be synchronized with other users when a connection is available. Figure 2.4 shows our implementation. We

```
29 case class Entry(title: Signal[String], date: Signal[Date])
30
31 val newEntry: Event[Entry] = App.newEntryUI()
32 val automaticEntries: Event[Entry] = App.nationalHolidays()
33 val allEntries = newEntry || automaticEntries
34
35 val selectedDay: Signal[Date] =
36   Storage.persist(id = "day", default = Date.today){
37     init => App.selectedDayUI(init) }
38 val selectedWeek = Signal { Week.of(selectedDay.value) }
39
40 val entrySet: Signal[ReplicatedSet[Entry]] =
41   Storage.persist("entryset", default = ReplicatedSet.empty){
42     init => allEntries.fold(init) {
43       (state, entry) => state.add(entry) } }
44
45 Network.replicate(id = "entryset", reactive = entrySet)
46
47 val selectedEntries = Signal {
48   entrySet.value.toSet.filter { entry =>
49     selectedWeek.value == Week.of(entry.date.value)
50   }
51 }
52
53 selectedEntries.observe(Ui.displayEntryList)
```

Figure 2.4: Excerpt of REScala source code for the shared calendar application.

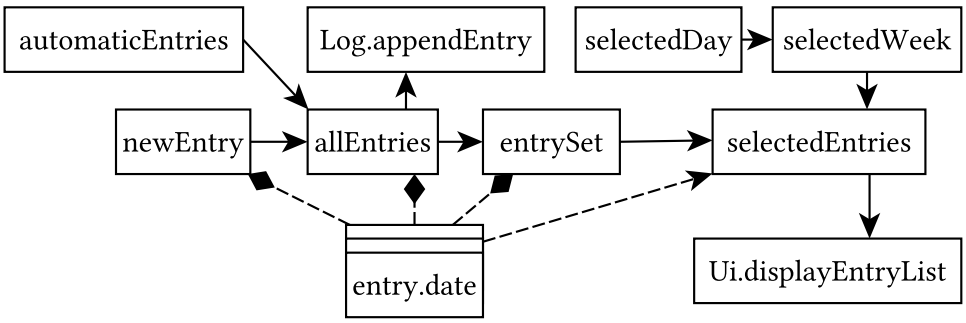

Figure 2.5: The flow graph for the calendar application.

refer to it as we discuss how events and signals are relevant for representing fault-tolerant application state and communication. The flow graph of the application is depicted in Figure 2.5.

The calendar applications manages a list of calendar entries. The entry class in Line 29 consists of two nested signals representing the title and date of the entry. New entries enter the calendar either by user input (Line 31) or from the network (Line 32). Both cases are modeled as events. The two source events are combined into a single derived event using the choice `||` method in Line 33.

Lines 35 to 37 show a source signal that represents the day the user has selected in the UI which is persisted to permanent storage using a given ID. Section 3.2 discusses how persisting values work. The currently selected week is derived from that source as normal (Line 38). Lines 40 to 43 define a list of all calendar entries as the signal `entrySet` by folding over the calendar entry events and collecting them into a set. Similar to the selected day the application also declares that the set of calendar entries must be persisted to storage.

The value type of the set of entries in Line 41 is a `ReplicatedSet` which is a special data type that enables REScala to transparently replicate the signal to other devices over the network. Replication requires a network runtime and also an ID the syntax for which is shown in Line 45. Details for how replication works are given in Section 3.3.

Lines 47 to 51 show a signal expression that accesses the set of calendar entries, and uses Scala's built-in filter function on sets to select only those entries of the current week. The date of each calendar entry is a signal itself, thus this signal expression accesses many nested signals dynamically. Note that the set of entries is a replicated reactive, thus the entries themselves – including the nested signals – are replicated on the network.

Finally, we use an explicit observer (Line 53) to notify the UI of changes to the set of selected calendar entries.

2.5 Conclusion

REScala is a Scala library that allows developers to define applications that react to changes in the external world. Compared to existing paradigms, REScala enables the composition of many small declarative reactions into a flow graph that represents a complete application. The declarative definition of the application allows the runtime of REScala to automatically update the flow graph whenever any input changes, thus the state of application is always consistent with the external changes that are observed. In addition, REScala provides integrations with imperative code to interface with other paradigm provided by the Scala language. These integrations are used to provide high-level integration libraries that make certain features (such as UI toolkits) also available in a declarative style. Moreover, REScala also enables automatic snapshots and restoration, and provides replication of reactive to other devices in the network. We will see both features in more detail in the next section.

Chapter 3

Faults and Resiliency

In a world with fast and reliable networks the design of dicApps would not need to distinguish between distributed and non-distributed cases. Edges in the flow graph would simply span over multiple devices and network boundaries would have no impact on application design. However, in reality networks and devices are slow and unreliable and the most common system environments (i.e., web, mobile) regularly shut down running applications. Yet, we do not wish to abandon the simplicity of transactions and strong consistency for operations that are not affected by these issues. Thus, the disparity of reliable operations and fault-prone communication must be explicitly considered during the development of dicApps. To help developers do so, Cvtr automatically provides a combination of strong consistency and causal consistency, and makes it explicit where the different levels of consistency interact.

We have already seen in the previous chapter what this looks like for the developers. Replicated reactives provide explicit interactions between the strong consistency of transactions on a local device. Explicit replication is crucial to make developers aware at which points the consistency model is different. Due to all dependencies in Cvtr being explicit it is then straightforward for developers to inspect which parts of their design is affected by replication.

This chapter explains what remains invisible to developers: how fault tolerance is automated to achieve the developer friendly consistency. Automation is enabled by taking some control of the state of the application away from developers. In return, developers may always assume that the state of their application is consistent independently of the occurrences of faults.

Specifically, there are two parts to fault tolerance: persistent storage and synchronization between devices. Both cases are solved and explained individually and provide individual value, but they both complement each other if used together. The rest of this chapter extends the intuition given in Chapter 2 of how these two solutions in combination solve a large class of errors when applications are executed in a distributed environment. But first, we discuss the types of addressed faults in more detail.

3.1 Addressed Types of Faults

We use the term *fault* to refer to the origin of a failure and *error* to refer to the representation of a fault in the language [134]. The types of faults tolerated by REScala are crashes and disconnects.

We use the term crash to refer to any situation where a part of a distributed application running on a single device terminates. This may include the physical device crashing because of some unrelated failure in the host system, because of the device running out of battery or another form of power outage, or because the device reboots after an update terminating the application forcefully. The crash may also only apply to the application not to the whole device. The execution environments of dicApps often terminate applications for various reasons. Example reasons for termination include the environment running out of memory, compute clusters ofter stop less important tasks when more important tasks must be executed, mobile operating systems terminate applications arbitrarily while they are not the active focus of the user, and web applications are terminated by a simple reload of the containing browser tab. In all of these cases the application is considered crashed. Cvtr assumes that in all cases the application will be restarted by the device at some point and then the state of the application must be restored. Permanent failures are not addressed, because they are not very interesting in the context of dicApps. Some cluster systems restart crashed applications on a spare device, but that does not make much sense for mobile phones or web browsers – when your mobile phone runs out of battery, you usually do not expect to be replaced by some other user that happens to still be available.

Disconnects between devices are due to crashes of remote devices (for all the above reasons) or due to broken network links. Disconnects cause messages to get lost or reordered, resulting in an inconsistent state across devices. An important design of Cvtr is that the dicApps on each device work even when the device is completely offline. While for some types of dicApps functionality may be vastly limited without connectivity all local operations are still available and when connection is restored then consistency between the reconnected devices is established again. This makes Cvtr especially useful for all dicApps where users are not simply consuming remote content, but rather productively work with the application.

Our solutions do not address data corruption (malicious or accidental). The latter is usually handled at a lower layer using checksums for both stored state and send messages. We also do not explicitly consider active attacks. We usually assume that devices authenticate themselves before synchronizing their state with other devices to prevent attackers modifying user data. However, because of how data in Cvtr is stored, it is easier to restore data after incorrect modifications, and because applications work offline users are less susceptible to denial-of-service type attacks – users can continue working even if communication is unavailable.

3.2 Fault-tolerant Application State

Crashes of individual devices during the execution of an application may result in a loss of the state of the flow graph hosted on these devices. Loss of local device data is problematic since such data often contains important private or unsynchronized information. To address this issue, REScala provides automatic snapshots and recovery. We have already discussed the developer API for restoration in Section 2.4. In summary, developers give names to reactives in the flow graph when they consider the state of the reactive essential to the application. Note that REScala provides the option to statically enforce that all state of an application is restored after a crash, but that option is not the default, because it turns out that developers prefer to opt in to what state should be stored. The rest of this section discusses crash tolerant signals in detail, and then describes how snapshots are created and restored.

Crash-tolerant Signals

Signals hold state and are restored after a crash, either by loading the value from a snapshot, or by recomputation. For example, in the calendar application, each calendar entry must be restored from the snapshot, but derived information such as the layout of the entries on the screen can be recomputed. REScala is capable to automatically determine the minimum set of signals that are required to include in a snapshot such that from those signals all others can be recomputed. However, it turns out that developers want precise control over what state is restored. For example, the view of the calendar should always display the current week when the application is started, thus any changes to that view should not be persisted. Thus, REScala allows developers to explicitly use `Storage.persist` to specify which reactives to persist. An alternative would have been to allow opting out of restoration, but our case studies indicate that the application code is easier to understand the use of restoration is explicit – which aligns with our overall strategy for fault tolerance. For the remaining discussion, however, we generally assume that all state of an application should be restored and show how that process is automated.

Technically, persisting takes an `id` and a `default` parameter. The ID of a stored signal is used to identify which value in a stored snapshot corresponds to which signal in the application and the default is used when the current snapshot does not contain a matching ID (e.g., when the application is started for the first time, without a snapshot). We use the Scala type system to require that the values of signals that are included in snapshots are serializable (i.e. can be stored to disk). For example, a signal containing a calendar date of type `Signal[Date]` in requires that the `Date` type is serializable.

Each device has its own application code together with its own storage for state. When a device crashes, the runtime representation of the flow graph is lost, including all values of reactives. The flow graph itself can be reconstructed from the application code and values of reactives are restored from the snapshot. Every processed transaction updates the snapshot at the end of the transaction, thus losing

at most the latest interaction with the user (i.e., a single press of a button). Creating snapshots is incremental – only values changed by a transaction are updated in the snapshot. In the first place, only the few reactives with state essential to the application are included in the snapshot and the value of all other reactives is recomputed from those during restoration. We trade efficient frequent snapshots for more expensive but rare restorations. Section 5 elaborates on further details and proves the correctness of restoration from minimal snapshots.

Snapshot Anatomy

Conceptually, a snapshot is a mapping of the reactives to their current values. Neither the structure of the flow graph nor any other state is required for restoring a snapshot. Applications often store redundant derived state in memory for efficiency – this pattern is especially prevalent with derived signals which are basically all recomputable. For example, a histogram displayed to the user can be recomputed from database entries, but it would be expensive to repeat this process for every frame the application displays. Non-distributed REScala applications typically consist of many small derived parts of the state (i.e., single reactives) to take advantage of incremental updates. In such a setting, REScala detects derived state and excludes it from snapshots. Precisely, in REScala, the only reactives with values that cannot be recomputed are source signals (capturing external state) and fold signals (aggregating event occurrences), since their state depends on past user interactions. All other reactives are either stateless events or derived signals that can re-execute their user-defined functions to recompute their state. We say that sources and folds constitute the *essential state*, and REScala recovers the state of all reactives from the essential state.

Creating Snapshots

In traditional paradigms for interactive applications developers must carefully reason about when the application may create a snapshot, because snapshots should not include transitional state. However, transactions correspond to a single user perceived change and provide the boundaries for fault tolerance. Thus, snapshots are created as part of each transaction and each snapshot correspond to a user perceived change. Moreover, the transaction manager also protects the snapshot mechanism from any external control flow for free.

Storing a full snapshot of all the essential state after each transaction is wasteful, since only a subset of the reactives change. Full snapshots are especially problematic when considering modularity of applications, because composing modules would suddenly incur an overhead on unrelated transactions. Instead of full snapshots, REScala keeps track of all reactives that are changed during a transaction. Snapshots are then stored incrementally and only reactives changed by a transaction are updated in the snapshot. Thus, the cost of snapshots grows linearly with the size of transactions, not linearly with the size of the application.

Recovering State

For recovery, REScala re-executes the application to restore the structure of the flow graph, but restores the values of reactives from the snapshot instead of initializing them. During this recovery process, the value of each signal with essential state is restored to the state after the last completed transaction before the crash. Events do not have state, so no value is restored. Derived signals recompute their values from the restored values. As the flow graph is acyclic, it is guaranteed that input reactives are always restored before any derived reactives. To handle dynamically changing flow graphs, the recovery process is also incremental. Reactives are restored as soon as they are created during the re-execution of the application. Thus, REScala allows the restored parts of the application to already handle new interactions, while other parts are still recovering.

We make two arguments why recomputation is preferable over storing more values in the snapshot. First, a snapshot is created every time an update occurs, while restoration only happens when a device fails. Hence, storing only necessary state has performance benefits if the latter is only a small portion of the overall state (cf. Section 13.2 for an empirical evaluation). In a previous study [181], we reported that in a typical reactive application only 14% of the flow graph contains essential state. Second, and arguably more important than the performance, only the essential parts of the state need to be serializable, thus allowing the use of data types that cannot be (efficiently) serialized for the rest of the application. In general, serialization works on the level of individual reactives, thus performance is heavily dependent on the serialization performance and the size of state changed in each individual transaction.

Observers

REScala only restores state that is part of the flow graph. To ease integration with external libraries, REScala executes observers on signals during restoration. For example, when the list of selected entries of the calendar is restored, the observer that informs the UI about updates is executed. In general, observers allow developers to define a relationship between the changes of transactions – including restoration – and some external state. In the case of signal observers, the relationship is that the external state should always be computed by the observer function, using the current value of the signal. Executing the observers during recovery allows the application to uphold this relationship. However, it is ultimately the responsibility of the application to use correct handlers. Events have no state to be restored (because snapshots are only stored between updates), so the handlers on event observers are not executed during restoration.

3.3 Managing Distributed State

This section presents how REScala keeps the application responsive when network connections are not reliable and still ensures that the state of different replicas of

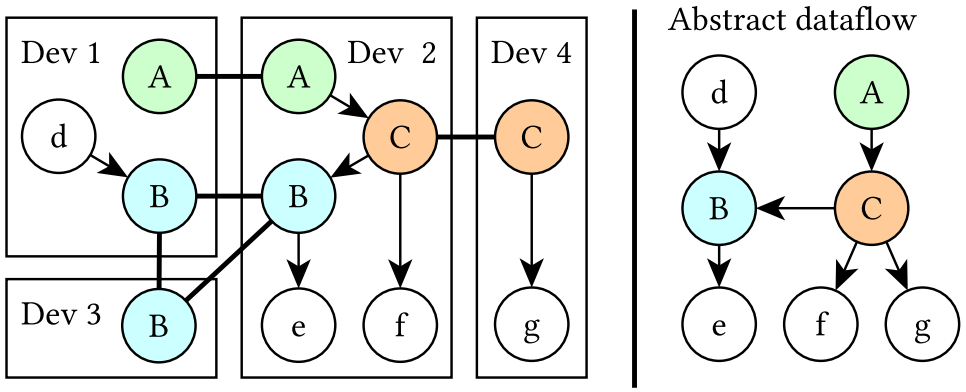

Figure 3.1: Full flow graph of a distributed application (left) and abstract dataflow (right).

the same application remains consistent. The key idea is to manage fault tolerance at the level of replicated data structures and ensuring their consistency instead of handling fault tolerance at the level of individual messages.

Replicated Reactives

The smallest unit of replication are individual reactives. *Replicated reactives* model a shared piece of state that is replicated to the flow graphs of multiple devices. Unlike derivation of reactives, replication creates a bidirectional connection between these reactives, thus extending the flow graph beyond device boundaries. For illustration, consider the left part of Figure 3.1, which shows four devices each having its own local flow graph. Reactives A, B, C are replicated (C is replicated in Dev 2 and Dev 4, A in Dev 1 and Dev 2, and B in Dev 1, Dev 2, Dev 3). The right of the figure shows the abstract dataflow of the overall distributed application. For example, even though reactives d (Dev 1) and e (Dev 2) are on different devices, a change of d will eventually reach e. However, because remote communication is slow and unreliable transactions are bounded within one device. The semantics for replicated reactives is as follows.

Replicated reactives behave like normal reactives with regard to device-local dataflow. Their state is computed from their inputs, propagated to their outputs, and stored in the local snapshot. Unlike the directed connections between reactives on the same device, connections between replicas work in any direction. Each device can change the state of its replica independently, even while being disconnected from the rest, thus transactions must be able to complete locally. Local transactions cause the state of the replicas to diverge.

To still provide a convenient user experience the devices use eventually consistent synchronization when communication is possible. Cvtr requires the existence of an eventually consistent synchronization protocol for the state of each individual replicated reactive. In REScala replicated reactives are implemented using

state-based conflict-free replicated data types (CRDTs) [186] for the state of those reactives. CRDTs provide automatic conflict-free merging of diverged state for a wide range of common data types [186]. A CRDT in REScala consists of a *lattice* and a set of local operations to modify the state of the CRDT. A lattice is a structure where every two values can be merged using an associative, commutative, and idempotent merge function that provides eventual consistency. While CRDTs usually have a limited set of possible operations, we argue in Chapter 10 why this is not an additional restriction for Cvtr.

The example in Figure 2.4 illustrates how `ReplicatedSet` is used as the value type for the replicated reactive `entrySet` which is then replicated in the network. The underlying lattice of `ReplicatedSet` is a simple set of values with set union as its merge function. The only local operation allows adding values to the set, which is used by the application to add new calendar entries. The set of entries is synchronized between all devices that use a replicated signal with the same name and type. Due to the properties of the merge function, the state of replicas can always be synchronized, and eventually all devices will converge to the same set containing all added elements.

For a set, once all replicas have been merged with each other, they all contain the same elements. In addition to `ReplicatedSet`, REScala currently supports replicated counters, last-writer-wins registers, ordered lists, and replicated data types that allow adding and removing elements from sets and lists. By using conflict-free data types – an existing technique already known to programmers – we provide simple and intuitive semantics for sharing state across devices.

Replication and Transactions

The local operations on CRDTs enable their use as values when deriving reactives, and their state-based nature enables their integration with snapshots and recovery. Both local changes or a local restoration are synchronized by merging the diverged states of different replicas. Every change to the value of a CRDT, both local and remote, is immediately propagated to local derived reactives. However, some transactional properties are lost at the device boundaries, because Cvtr prioritizes offline availability over strong consistency. Most users of interactive application prefer their applications to work offline, and they understand that their collaborators do not see their changes while they are offline.

Still, Cvtr keeps some transactional properties, most importantly, the causal consistency of changes to multiple reactives is preserved between two devices. For example, Dev 1 and Dev 2 in Figure 3.1 try to synchronize their state after every transaction. With slow network connections – a disconnect is a very slow connection – Dev 1 and Dev 2 may have different states for reactives A and B. However, all remote changes to replicated reactives are applied as a single transaction, thus still providing causal consistency because changes from one reactive do not become visible before earlier changes of another reactive. For example, when Dev 2 changes reactive A, then reactive B also changes in the same transaction because of the connection in the local flow graph. Dev 2 sends the most recent state of both reactive

A and reactive B to Dev 1. When the result of such a transaction from Dev 2 is received by Dev 1 the changes are applied as a single transaction on Dev 2, thus changing both reactives A and B and keeping the causal connection between their changes. Causal consistency aligns well with end user expectations. Causal consistency prevents situations where an answer (e.g., the value of reactive B) is visible before the original message (e.g., the value of reactive A) – a situation that is highly confusing to most users.

The first lost property of transactions is their identity. Only the most recent state is synchronized, thus Dev 2 may execute many transactions while offline, but Dev 1 will only apply the latest changes as a single transaction. REScala provides operators that are sensitive to the number of transaction, most notably many of user-defined functions for fold reactives execute their function once per transaction. It depends on the application, if the semantics of the user-defined function is actually dependent on the number of transactions. The order sensitive operators have to be inspected by the developer and depending on their semantics they must also be replicated over the network to ensure consistency. For example, the count operators counts the number of changes to its input reactive. If the changes to a single replicated reactive are counted on two different devices, the count may be different. Thus, to ensure consistency, the count reactive must also be replicated.

The second lost property is what is called distributed glitch freedom [68]. For example, consider again the graph in Figure 3.1. An update that affects reactive A on Dev 2 will immediately affect reactive B on Dev 2 because they are connected by the local flow graph. However, if A is updated on Dev 1, B is only indirectly affected and synchronization with Dev 2 is required to complete the update. In a local graph, being able to observe the changed state of reactive A without also observing the changed state of a derived reactive B is called a glitch. When the derived reactive is only connected over device boundaries, this is called a distributed glitch. Distributed glitches always require at least two network connections: either a cycle as in the example or a diamond shape with at least three devices. To prohibit distributed glitches, Dev 1 would have to wait for the update on B to arrive, whenever A is changed. REScala (as presented in this thesis) allows for distributed glitches in favor of availability (but Drechsler [68] provides serializable distributed transactions for REScala – losing availability). It is possible to analyze the flow graph and predict when a distributed glitch may occur, although it cannot be completely accurate with dynamic graphs and offline availability. In general, users usually appreciate when they are informed that their changes have not yet been confirmed by a remote connection and will understand that some changes or functionality thus is only applied later (e.g., a remote spell checker only highlighting mistakes after a couple of seconds).

Obscure Interactions: Published Signals, Replicated Events, and Nested Reactives

We have covered the core of distributed state in Cvtr. However, there are three additions covered here that make using replication in REScala more convenient.

First is a simplification when bidirectional communication is not required. Second is how replicated events are handled. Third is what happens when nested reactives are replicated.

Using CRDTs to implement replicated signals allows bidirectional communication, but the value of the reactive must be a CRDT. Alternatively, REScala allows to *publish* any signal – not only those based on CRDTs. Such a published signal is a replicated signals that may only be changed by the publishing device, thus precluding conflicting changes. Publishing is a special case of eventually consistent replication. To publish a signal, REScala creates a replicated signal with a last-writer-wins CRDT, a data type where the merge function always selects the most recent value. Since only one device is allowed to write, there are no races between writes.

Replication in Cvtr is based on synchronized state, but events do not have state, thus replication of events is not directly supported. Instead, events must be converted to signals first typically by using the `latest(n)` operator that creates a signal containing the latest `n` occurrences of the event. If more than `n` events occur when the device is disconnected, the oldest events will be lost. Similar operators can be used to define time or priority-based policies, allowing the application developer to tune the software behavior as necessary.

Replicated reactives may contain nested reactives. The value of nested reactives is synchronized independently of the containing reactive – that is, the containing reactive only synchronizes a reference to the nested reactive, but not the value. Depending on the network model and protocol, the transfer of values of individual replicated reactives may fail or be delayed. When the outer reactive is synchronized but the inner reactive is delayed, then the application may lose causality. To fix this issue, reactives nested in a replicated reactive must be considered as part of any transaction that changes the containing replicated reactive. Then the normal mechanism that ensures causal consistency also applies for nested reactives.

3.4 Replication and Restoration

When a replicated reactive aggregates state, then that state must both be synchronized with other devices and included in snapshots. For those reactives, all types of faults may disturb their operation leading to complex failure cases. For example, a device may leave the network, make some changes while offline, shut down and later restore from a snapshot, and finally reconnected to a different part of the network. Yet, as long as devices are at least sporadically connected to some other device in the network, then the network runtime of REScala will ensure that all replicas of the replicated reactive have the same value on all devices. This includes replicating state after recovering from a crash to ensure that all devices eventually see the same set of entries. Thus, no matter the combination of failures, the state of a Cvtr application will be fully recovered and consistent on all replicas that are connected.

3.5 Conclusion

Cvtr addresses crashes and disconnects in a way that simplifies the programming model two a two tiered system for the programmer. Local devices are considered consistent but may have to be restored after a crash, while communication is eventually consistent and done on a best-effort basis. Both parts of the solution are individually useful for their specific case (only addressing crashes, or only disconnects) but their strength is that they still work when both types of faults occur together. Even when a device crashes while it is offline the state is restored and will be correctly synchronized when connectivity is established again.

Chapter 4

The Formal Programming Paradigm

To precisely describe the Cvtr programming paradigm, this chapter introduces a core calculus of REScala called $\mathscr{F}_r$ that formalizes the syntax and semantics introduced in Chapter 2. While REScala is older than $\mathscr{F}_r$ the current implementation of REScala was co-developed together with $\mathscr{F}_r$. Chapter 6 discusses how the calculus is extended into the full implementation of REScala. However, the model is designed to be independent of any concrete embedding language. Chapter 5 formally states the assumptions on the embedding language and proves the resulting properties of the model.

4.1 Syntax

Figure 4.1 shows the syntax of $\mathscr{F}_r$. The outermost term is a set of communicating devices $(M; D_1|...|D_n)$ consisting of a mapping M that assigns each replicated reactive r a merge function $r \mapsto merge_r$ to model CRDTs, and a set of devices $D_1|...|D_n$ executing concurrently. Confer to Section 10.4 for a discussion how protocols other than classical CRDTs may be expressed using merge functions. Devices D have volatile state in the form of processes L, and durable state in the form of snapshots Σ – corresponding to a simplified view of RAM and disk storage. The current process is lost when the device crashes, but the snapshot is persisted. Each D_i starts with an initial process L_i and an empty snapshot.

The evaluation of processes L models either (a) the execution of application code $\langle\mu\rangle\, t$ with term t and store μ or (b) the propagation of changes during a transaction $\langle\mu\rangle\, t \triangleright (A; P)$. The propagation $\triangleright (A; P)$ contains the sets of active reactives A and processed reactives P and is used by the runtime for bookkeeping during transactions. The evaluation of processes uses the λ-calculus as the embedding language. Values v include functions $x \Rightarrow t$, the unit value, and reactives r. Terms t include function definitions and application, creating new reactives, and integrations of reactive and external code. In $\mathscr{F}_r$ all terms creating new reactives are capitalized. Reactives that are used as inputs or for activation are left of a dot, and user-defined functions for reevaluation are in braces. For the calculus, we assume an arbitrary

$$
\begin{array}{rll}
C & ::= (M; D|...|D) & \textit{Communication} \\
D & ::= \Sigma L & \textit{Device} \\
L & ::= \langle\mu\rangle\, t \mid \langle\mu\rangle\, t \triangleright (A; P) & \textit{Process} \\
v & ::= x \Rightarrow t \mid r \mid \text{unit} \mid ... & \textit{Value} \\
t & ::= v \mid x \mid t\ t \mid & \lambda\ \textit{Term} \\
 & \quad r \mid \bar{t}.\,\text{Map}\{f\} \mid t.\,\text{Fold}\,(t)\{f\} \mid t.\,\text{Flatten} \mid t.\,\text{Filter}\{f\} \mid & \textit{Reactive} \\
 & \quad \text{Source}(s) \mid t.\,\text{observe}(o) \mid t.\text{value} \mid t.\text{fire}(t) & \textit{Integration}
\end{array}
$$

$$
\begin{array}{rl}
r & \textit{Reactives (locations)} \\
s & \textit{Source definition} \\
o & \textit{Observer definition} \\
f : (\mu, \bar{r}) \to v & \textit{User-defined function} \\
M : r \mapsto ((v \times v) \to v) & \textit{Merge mapping} \\
\Sigma : r \mapsto v & \textit{Snapshot} \\
\mu : r \mapsto (in \subset r, val \in v, op \in f, act \in f, obs \subset o) & \textit{Local store}
\end{array}
$$

Figure 4.1: Syntax of $\mathscr{F}_r$.

$$
\begin{array}{rl}
E ::= & \mid [\cdot] \mid E\ t \mid v\ E \\
 & \mid (\bar{r}, E, \bar{t}).\,\text{Map}\{f\} \mid E.\,\text{Fold}\,(t)\{f\} \mid r.\,\text{Fold}\,(E)\{f\} \mid E.\,\text{Filter}\{f\} \mid E.\,\text{Flatten} \\
 & \mid \text{Source}(E) \mid E.\,\text{observe}(o) \mid E.\text{value} \mid E.\text{fire}(t) \mid r.\text{fire}(E)
\end{array}
$$

Figure 4.2: Evaluation context.

set of user-defined functions f to exists. Note that user-defined functions are functions on the meta-level and are not to be confused with functions ($x \Rightarrow t$) inside the calculus. In a typical implementation such as REScala, the set of user-defined functions will be the side-effect-free subset of functions from the host language.

We use a standard, left to right, call by value, evaluation context E [80] shown in Figure 4.2. The store μ maps reactives r to a 5-tuple with named values containing the set of input reactives ($in \subset r$), the current value ($val \in v$), the function used for computing new values during reevaluation ($op \in f$), the function to compute if a reactive activates after reevaluation ($act \in f$), and any observer definitions triggered by the reactive ($obs \subset o$). We write $\mu(r).val$ to access the value val of reactive r in store μ.

$$\textsc{Map}\ \frac{t_i : \text{Reactive}[\text{T}_\text{i}] \qquad f : \overline{\text{T}} \to \text{R}}{\bar{t}.\text{Map}\{f\} : \text{Reactive}[\text{R}]}$$

$$\textsc{Fold}\ \frac{t : \text{Reactive}[\text{T}] \qquad t_0 : \text{R} \qquad f : (\text{R}, \text{T}) \to \text{R}}{t.\text{Fold}\,(t_0)\{f\} : \text{Reactive}[\text{R}]} \qquad \textsc{Flatten}\ \frac{t : \text{Reactive}[\text{Reactive}[\text{T}]]}{t.\text{Flatten} : \text{Reactive}[\text{T}]}$$

$$\textsc{Filter}\ \frac{t : \text{Reactive}[\text{T}] \qquad f : \text{T} \to \text{Boolean}}{t.\text{Filter}\{f\} : \text{Reactive}[\text{T}]} \qquad \textsc{Source}\ \frac{s : \text{SourceDef}[\text{T}]}{\text{Source}(s) : \text{Reactive}[\text{T}]}$$

$$\textsc{Observe}\ \frac{t : \text{Reactive}[\text{T}] \qquad o : \text{ObserverDef}[\text{T}]}{t.\text{observe}(o) : \text{Observer}}$$

Figure 4.3: Typing rules of reactives.

4.2 Types

Figure 4.3 shows the typing rules of reactives. We use the types to give an intuition how syntactic elements in $\mathscr{F}_r$ are correctly combined, and how an implementation would embed and express the methods of $\mathscr{F}_r$ into the embedding language. In general, any language with a type system supporting generics should allow for type-safe embedding of $\mathscr{F}_r$. To simplify our presentation, we do not show the typing context for variables. All reactives are of type Reactive[T] and are parametric over the value they carry. A Reactive[T] corresponds to a signal carrying a value of type T, where an event of type T would be represented by a Reactive[Option[T]]. Many of the methods to create reactives allow for a user-defined function f to specialize the behavior of the created reactive. The type system ensures that the type of functions is compatible with the value type of reactives the function operates on. However, note that our type system for user-defined functions is specialized to include them in the flow graph. That is, in the dynamic semantics (Section 4.5) each user-defined function operates on the current store and the input reactives. For example, a user-defined function f of type $f : \text{A} \to \text{B}$ takes a store μ and a reactive $r : \text{Reactive}[\text{R}]$ as its parameter and returns a value of type B.

- Map has matching types for the i-th input reactive t_i and the i-th parameter T_i of the given function f. The function is applied to the values of the inputs.
- Fold uses t_0 as an initial value which is passed to the user-defined function f. The type of f matches the initial value and the value of the input. The result type of the function has the same type as the initial state.
- Flatten removes one level of nesting, by extracting the value of the nested reactive.

$$
\begin{aligned}
\mathit{lookup}(r, \Sigma, v) &= \begin{cases} \Sigma(r) & r \in \Sigma \\ v & r \notin \Sigma \end{cases} \\
\mathit{ready}(P, \mu) &= \{r \in \mu \mid r \notin P \wedge \mathit{inputs}(r, \mu) \subseteq P \cup \{r\}\} \\
\mathit{outdated}(A, \mu) &= \{r \in \mu \mid A \cap \mathit{inputs}(r, \mu) \neq \emptyset\} \\
\mathit{eval}(r, \mu) &= \mu(r).\mathit{op}(\mu, \mathit{inputs}(r, \mu)) \\
\mathit{filter}(r, \mu) &= \mu(r).\mathit{act}(\mu, \mathit{inputs}(r, \mu)) \\
\mathit{update}(\mu, r, v) &= (\mu, r \mapsto (\mu(r).\mathit{in}, v, \mu(r).\mathit{op}, \mu(r).\mathit{act}, \mu(r).\mathit{obs})) \\
\mathit{inputs}(r, \mu) &= \begin{cases} \{r_0, \mu(r_0).\mathit{val}\} & \mu(r).\mathit{in} = \mathit{dynamic}(r_0) \\ \mu(r).\mathit{in} & \text{otherwise} \end{cases} \\
\mathit{stateful}(r, \mu) &= (r \in \mu(r).\mathit{in} \vee \mu(r).\mathit{in} = \emptyset) \\
\mathit{flatten}(\mu, r) &= \mu(\mu(r).\mathit{val}).\mathit{val} \\
\mathit{location}(\mu) &= \text{fresh location in } \mu\text{, consistent between restarts} \\
\mathit{sources}(r) &= \text{source r is triggered externally} \\
\mathit{observe}(A, \mu) &= \text{execute all external observers } \{o \in \mu(r).\mathit{obs} \mid r \in A\}
\end{aligned}
$$

Figure 4.4: Auxiliary functions for the operational semantics.

- Filter has the same type, Reactive[T], as its single input t. The function f must take a parameter of the type T and return a Boolean. A filter reactives propagates the value of the input unchanged, if the filter condition $f(x)$ is true.
- A *source reactive* is of type Reactive[T], given that it is triggered by a source s of type SourceDef[T], i.e., the inner type T of the reactive is defined by the inner type of s (rule Source). Sources are how we model data arriving from somewhere outside the calculus.
- Similarly, an *observe reactive* has a single input t of type Reactive[T], given that they observe an o of type ObserveDef[T], i.e., the inner type of the input reactive must match the type that is consumed by o (rule Observe). Observers model how data is extracted from the calculus and passed to some external system.

4.3 Semantics Overview

We define a small step operational semantics for $\mathscr{F}_r$. Figure 4.4 contains auxiliary functions used throughout the semantics. Small step semantics allows modeling of message receives and crashes to nondeterministically occur between any two steps of a device. We first look at the stepping rules for individual devices (Section 4.4). Evaluation of a single device depends on the current process, and is separated into

$\boxed{\rightarrow_D \subset D \times D}$

$$\frac{\mu(r).in = \emptyset}{\Sigma\,\langle\mu\rangle\, r.\mathrm{fire}(v) \rightarrow_D \Sigma\,\langle update(\mu, r, v)\rangle\, r \triangleright (\{r\}; \{r\})} \text{ (FIRETX)}$$

$$\frac{\begin{array}{c} P = dom(\mu) \qquad observe(A, \mu) \\ \Sigma' = \Sigma, \{r \mapsto \mu(r).val \mid r \in A \wedge stateful(\mu, r)\} \end{array}}{\Sigma\ \langle\mu\rangle\, t_l \triangleright (A; P) \rightarrow_D \Sigma'\ \langle\mu\rangle\, t_l} \text{ (COMMIT)}$$

$$\frac{v = \mu(r).val}{\Sigma\,\langle\mu\rangle\, r.\mathrm{value} \rightarrow_D \Sigma\,\langle\mu\rangle\, v} \text{ (ACCESS)} \qquad \frac{t \rightarrow_\lambda t'}{\Sigma\,\langle\mu\rangle\, t \rightarrow_D \Sigma\,\langle\mu\rangle\, t'} \text{ (OUTER)}$$

$$\frac{\Sigma\,\langle\mu\rangle\, t \rightarrow_D \Sigma\,\langle\mu'\rangle\, t'}{\Sigma\,\langle\mu\rangle\, E[t] \rightarrow_D \Sigma\,\langle\mu'\rangle\, E[t']} \text{ (CONTEXT)}$$

Figure 4.5: Operational semantics for process behavior of devices.

two differently labeled stepping relations, $\rightarrow_D$ and $\rightarrow_p$, for presentation purposes. When evaluating application code ($\rightarrow_D \subset\ D \times D$) the flow graph is created (Section 4.5) and actions are received. An action starts a propagation ($\rightarrow_p \subset D \times D$) (Section 4.6). Propagation processes all reactives before execution of the application continues. A crash of the device during a propagation causes the triggering action to be lost. We extend the system to several communicating devices in Section 4.7.

Communication has its own stepping relation ($\rightarrow_C \subset\ C \times C$), which nondeterministically chooses to further evaluate one of the devices, to send messages, or to receive messages. We make the usual fairness assumptions that eventually all devices get the chance to evaluate and communicate.

4.4 Devices

A device $D = \Sigma L$ consists of a currently executing process L, and a snapshot Σ. A snapshot Σ is a mapping ($r \mapsto v$) from reactives to their value. Figure 4.5 shows the device evaluation relation except creating reactives.

Processes have two syntactic forms – one for the embedding application code and one for executing transactions in the flow graph: (1) $\langle\mu\rangle\, t$ executes the application code t with store μ to create and modify the flow graph, and (2) $\langle\mu\rangle\, t \triangleright (A; P)$ executes the runtime propagation of changes during a transaction, where $\langle\mu\rangle\, t$ is the application level code to continue with after the update, and $(A; P)$ contains the state of the runtime update propagation. The FIRETX rule (and other *TX rules we see later) switch from the embedding application to the transaction and the COMMIT ends the transaction and switches back.

The FIRETX rule evaluates the term $r.\text{fire}(v)$ to create a new transaction on source r with value v and activates r, i.e., marks it as having changed. The precondition ensures that r has no inputs, i.e., it is a source. The device switches to propagation by adding the runtime state $(\{r\};\{r\})$, which is read as r is *active* (it has changed its values) and r was *processed* (it will not change anymore in the current propagation). Section 4.6 explains how these sets are used for propagation. The current term of the process $r.\text{fire}(v)$ evaluates to just r, indicating that firing a source produces no new value. Finally, the value of the state $\mu(r)$ is changed to v – we write $\mathit{update}(\mu, r, v)$ to represent the updated store (c.f. Figure 4.4).

FIRETX could be extended to change multiple reactives at the same time, and REScala supports such transactions. However, changing multiple reactives in the same transaction can also be modeled by deriving all of them from a common source, which is the only form supported in the calculus.

The COMMIT rule ends the propagation and updates the snapshot to include all changes. When a propagation ends, the set of processed reactives P contains all reactives in the domain of μ. The COMMIT rule steps from a propagation that processed all reactives back to normal application execution by keeping the term and store $\langle \mu \rangle\, t_l$ from the propagation and producing a new snapshot Σ', which reflects the changes to folds and sources A that were active in the propagation. We write $\Sigma' = (\Sigma, r \mapsto v)$ to say that Σ' contains the same value assignments as Σ, except for r which is updated to v. The premise of COMMIT asserts that Σ' is updated to contain the current values of all active sources and folds $r \in A$. Sources and folds are computed by inspection of the inputs $\mu(r).\mathit{in}$, sources have no inputs, and folds have themselves as an input. The COMMIT rule also executes the observers of all active reactives using $\mathit{observe}(A, \mu)$. Similar to sources, the effect of observers is not specified in the calculus but repersents an abstract integration with an external system.

The ACCESS rule evaluates the term to the current value of the reactive, without modifying the store or the snapshot. Note that $r.\text{value}$ in $\mathscr{F}_r$ is not to be confused with the `.value` method in REScala. The version in $\mathscr{F}_r$ is for accessing the value of reactives from outside a transaction, while in REScala it is used only inside signal expressions.

The OUTER rule embeds the stepping relation of the λ-calculus outside reactive operators. The rules of the *lambda*-calculus are not shown but are standard call-by-value rules using substitution.

The CONTEXT rule evaluates nested device terms using a standard left-to-right evaluation context shown in Figure 4.2.

4.5 Creating and Restoring the Flow Graph

The reduction rules in Figure 4.6 are concerned with creating the flow graph. A reactive r is either a source without inputs, or is derived from its input reactives $\overline{x}$. The SOURCE, MAP, FOLD, FLATTEN, and FILTER rules each create a fresh identifier $r = \mathit{location}(\mu)$ and add the inputs, value, operator functions, observer functions,

$\boxed{\rightarrow_D \subset D \times D}$

$$\frac{r_0 = \mathit{location}(\mu) \qquad v_s = \mathit{lookup}(r_0, \Sigma, v) \qquad \mu' = \mu, r_0 \mapsto (\emptyset, v_s, \text{unit}, \emptyset, \mathit{true})}{\Sigma\,\langle\mu\rangle\,\text{Source}(v) \rightarrow_D (\Sigma, r_0 \mapsto v_s)\,\langle\mu'\rangle\,r} \text{ (SOURCE)}$$

$$\frac{r_0 = \mathit{location}(\mu) \qquad \mu' = \mu, r_0 \mapsto (\{\overline{r}\}, f(\mu, \overline{r}), f, \emptyset, \mathit{true})}{\Sigma\,\langle\mu\rangle\,\overline{r}.\,\text{Map}\{f\} \rightarrow_D \Sigma\,\langle\mu'\rangle\,r_0 \triangleright (\{r_0\}; \{r_0\})} \text{ (MAP)}$$

$$\frac{r_0 = \mathit{location}(\mu) \qquad v_s = \mathit{lookup}(r_0, \Sigma, v) \qquad \mu' = \mu, r_0 \mapsto (\{r_0, r_1\}, v_s, f, \emptyset, \mathit{true})}{\Sigma\,\langle\mu\rangle\,r_1.\,\text{Fold}\,(v)\{f\} \rightarrow_D (\Sigma, r_0 \mapsto v_s)\,\langle\mu'\rangle\,r_0} \text{ (FOLD)}$$

$$\frac{r = \mathit{location}(\mu) \qquad \mu' = (\mu, r_0 \mapsto (\text{dynamic}(r_1), \mathit{flatten}(\mu, r_1), \mathit{flatten}, \emptyset, \mathit{true}))}{\Sigma\,\langle\mu\rangle\,r_1.\,\text{Flatten} \rightarrow_D \Sigma\,\langle\mu'\rangle\,r_0} \text{ (FLATTEN)}$$

$$\frac{r_0 = \mathit{location}(\mu) \qquad \mu' = \mu, r_0 \mapsto (\{r_1\}, \mu(r_1).\mathit{val}, \mathit{identity}, \emptyset, f)}{\Sigma\,\langle\mu\rangle\,r_1.\,\text{Filter}\{f\} \rightarrow_D \Sigma\,\langle\mu'\rangle\,r_0} \text{ (FILTER)}$$

$$\frac{r_0 = \mathit{location}(\mu) \qquad \mu' = \mu, r_0 \mapsto (\{r_1\}, \mu(r_1).\mathit{val}, \mathit{identity}, \{o\}, \mathit{true})}{\Sigma\,\langle\mu\rangle\,r_1.\,\text{observe}(o) \rightarrow_D \Sigma\,\langle\mu'\rangle\,\mathit{unit}} \text{ (OBSERVE)}$$

Figure 4.6: Operational semantics for creating reactives.

and activation functions $r \mapsto (\mathit{in}, \mathit{val}, \mathit{op}, \mathit{obs}, \mathit{act})$ to the store μ. Restoration from a snapshot happens during the creation of sources and folds by loading the initial value from the snapshot if present.

Creating reactives always follows the same basic process. First, a new location is found to store the reactive. Then that location is initialized with the correct state depending on the method used to create the reactive. The state always includes the inputs and an initial value and in most cases a user-defined function for updating the value of the reactive. Most reactives have no observers and use *true* to compute their own activation resulting in those reactives to always activate when any of their inputs activate. The user-defined functions f used in many of the reactives are assumed to take a store μ and the input reactives as parameters. In REScala the signal expression macro converts a Scala expression to such a function, in $\mathscr{F}_r$ we just assume this to happen. Finally, the expression evaluates to the new store location.

The SOURCE rule creates a reactive r_0 with initial value v_s and no update functions. If the snapshot contains r_0, then the value is restored from the snapshot, i.e.,

$v_s = \Sigma(r)$. If the snapshot does not contain r_0, then the given value is used, i.e., $v_s = v$.

The MAP rule derives a new reactive from its inputs $\bar{r}$ using a user-defined function f. Map reactives compute their current value by applying f to the current value of their inputs. The process is the same for computing the initial value, where f is directly applied. Map reactives have no observer and they always activate if any of their inputs activates, thus the activation function is *true*. Map reactives are neither stored nor restored from the snapshots. Instead, the function f is applied to the restored values of $\bar{r}$ to compute the restored value.

The FOLD rule creates a reactive r_0 with user-defined function f. To model access of its state during reevaluation fold reactives have themselves as an input in addition to $\bar{r}$, thus passing the current value to the user-defined function f. The initial value v_s is restored in the same way as in the SOURCE rule.

The FLATTEN rule creates reactives which derive their value from a single input reactive r_1. The function $\mathit{flatten}(\mu_f, r_f) = \mu_f(\mu_f(r_f).\mathit{val}).\mathit{val}$ describes the indirection flatten creates. That is, the value of flatten is the value of the nested reactive. Like map reactives, flatten does not directly restore its values, but all inputs of flatten are assumed to be restored before the flatten reactive is created. Flatten reactives have a specially labeled input dynamic(r_1), instead of just r_1, to support dynamic detection of dependencies during propagation as described in Section 4.6.

The FILTER rule is the only rule that defines an activation function f to limit which values are propagated. The value of filter reactives is the value of their input.

Finally OBSERVE creates a new reactive that stores an observer. The value of observer reactives is also always the value of its input. The store location of observe reactives is not returned to the embedding application, thus there cannot be other reactives derived from observers.

Each of fold, flatten, filter, and observe require their own support from the calculus. Fold requires reactives to have themselves as inputs. Flatten requires dynamic inputs. Filter requires activation functions. Observe requires executing effects. Removing any combination of those four from the calculus still provides a useful subset of functionality. For example, removing filter essentially removes event semantics, removing fold results in stateless applications (except sources), and removing flatten makes flow graphs static.

We now have all constructs required to build a flow graph of an $\mathcal{F}_r$ application. Figure 4.7 shows store μ (which encodes the flow graph) after evaluating up to Line 13 of Figure 2.2. The application is structured in a way that each created reactive is assigned to a variable, and we have reuse these names to label the location of each reactive for better understanding.

Storing and Restoring Nested Reactives

When a reactive r_0 is included in a snapshot and contains a nested reactive r then r is also included in the snapshot. The value of r_0 in the snapshot will then contain a reference to r. This process is recursive.

$$
\begin{aligned}
\mu = (& \\
r_{name} &\mapsto (\emptyset, \texttt{"some name"}, \text{unit}, \emptyset, \mathit{true}), \\
r_{text} &\mapsto (\emptyset, \texttt{"some message"}, \text{unit}, \emptyset, \mathit{true}), \\
r_{message} &\mapsto \{r_{name}, r_{text}\}, \texttt{"some name: some message"}, \\
&\quad \texttt{(n, i) => n + ": " + i}, \emptyset, \mathit{true}), \\
r_{room1} &\mapsto (\{r_{room1}, r_{message}\}, \texttt{Nil}, \texttt{(history, msg) => msg :: history}, \emptyset, \mathit{true}), \\
r_{room2} &\mapsto (\emptyset, \texttt{List("Me: a constant message")}, \text{unit}, \emptyset, \mathit{true}), \\
r_{roomList} &\mapsto (\emptyset, (r_{room1}, r_{room2}), \text{unit}, \emptyset, \mathit{true}), \\
r_{index} &\mapsto (\emptyset, \texttt{0}, \text{unit}, \emptyset, \mathit{true}), \\
r_{selectedRoom} &\mapsto (\{r_{roomList}, r_{index}\}, r_{room1}, \texttt{(l, n) => l(n)}, \emptyset, \mathit{true}), \\
r_{roomContent} &\mapsto (\{\text{dynamic}(r_{selectedRoom})\}, \texttt{Nil}, \texttt{unit}, \emptyset, \mathit{true}) \\
&)
\end{aligned}
$$

Figure 4.7: Example of the store after evaluating the chat example up to constructing the content widget.

During restoration of r_0, the values of any nested reactive r are also restored, if r has not yet been restored otherwise. However, the operator and inputs of the nested reactive r are unknown when r_0 is being restored. Thus, we restore a placeholder reactive that is essentially constant. In the case that r is recreated later, the placeholder is replaced by the actual reactive transparently. We discuss in Section 11.1 how the restoration of nested reactives is achieved in a practical implementation.

4.6 Propagation of Changes During Transactions

The rules for propagation and reevaluation $\xrightarrow{r}_p$ of a reactive r are shown in Figure 4.8. We write just $\rightarrow_p$ if the specific reactive is irrelevant. Whenever a transaction changes the value of a reactive, then the runtime starts *propagation* of that change and all transitively derived reactives must *reevaluate*, i.e., compute their new value based on the inputs and on the operator. Syntactically, processes that perform a propagation are written $\langle \mu \rangle\, t \triangleright (A; P)$. With the state of the propagation expressed by the sets of active reactives A and processed reactives P. $\rightarrow_p$ evaluates devices with such processes. From the application developers' point of view, all reevaluations happen at the same time (synchronously) and use the most up-to-date value of their inputs. $\mathscr{F}_r$ models propagation as a stepwise process to reason about failure cases but guarantees synchronous semantics. At the beginning of a propagation, the transaction has changed the value of a reactive r, which is active $r \in A$ and processed $r \in P$. During the propagation, ready reactives $r' \in \mathit{ready}(\mu, P)$ are either reevaluated or skipped, until all reactives are processed.

$\boxed{\rightarrow_p \subset D \times D}$

$$\frac{r \in \mathit{ready}(P, \mu) \qquad r \notin \mathit{outdated}(A, \mu)}{\Sigma \langle \mu \rangle\, t \triangleright (A; P) \xrightarrow{r}_p \Sigma \langle \mu \rangle\, t \triangleright (A; P, r)} \text{ (SKIP)}$$

$$\frac{\begin{array}{c} r \in \mathit{ready}(P, \mu) \qquad r \in \mathit{outdated}(A, \mu) \qquad v = \mathit{eval}(r, \mu) \\ A' = A, r \text{ if } \mathit{filter}(r, \mu);\ A \text{ otherwise} \end{array}}{\Sigma \langle \mu \rangle\, t \triangleright (A; P) \xrightarrow{r}_p \Sigma \langle \mathit{update}(\mu, r, v) \rangle\, t \triangleright (A'; P, r)} \text{ (REEVALUATE)}$$

Figure 4.8: Operational semantics for propagation and reevaluation.

Reevaluation

A reevaluation (rule REEVALUATE) is the process of computing the current value of a reactive r by evaluating $\mathit{eval}(r, \mu)$ (Figure 4.4). That expression applies the operator function op to the current store μ and the inputs of r. In case r is a flatten reactive the nested reactive $\mathit{dynamic}(r)$ is treated as a normal input.

Propagation

The FIRETX rule (same for SOURCETX and SYNCHRONIZETX introduced later) writes a new value to the store μ and marks r as active and processed. A reactive r is ready ($r \in \mathit{ready}(P, \mu)$) if it has not been processed and all inputs of r are processed. Fold reactives have themselves as inputs and are ready if all other inputs are processed. Flatten reactives are ready when the outer and the nested inputs are processed. Additionally, a reactive r is outdated ($r \in \mathit{outdated}(A, \mu)$), if any of its inputs are active, i.e., the input is reevaluated and the activation function of that input returns true. Depending on whether a ready reactive is outdated the SKIP or REEVALUATE rule is applied. The SKIP rule marks a reactive as processed, if it is ready and not outdated. A skipped reactive is never active independent of its activation function. The REEVALUATE rule additionally causes a reevaluation of the reactive and marks the reactive as active if the activation function returns true. In all cases, the reactive is marked as processed. When all reactives in the store have been processed the COMMIT rule (Figure 4.5) ends the propagation. Because both sets, active reactives A and processed reactives P, only grow during a propagation, the process is guaranteed to terminate.

4.7 Communication Between Multiple Devices

Communication is modeled by sending state between devices without ordering or reliability of messages. Thus, the guarantees of $\mathscr{F}_r$ apply to most existing systems. The stepping rules for communicating devices $(M; D_1|...|D_n)$ are shown in Figure 4.9. The merge functions in M define the global behavior of the replicated reactives and they are fixed before any device starts – no central coordination is required.

$\boxed{\rightarrow_C \subset C \times C}$

$$\frac{D_i \rightarrow_D D_i'}{(M; D_1|...|D_i|...|D_n) \rightarrow_C (M; D_1|...|D_i'|...|D_n)} \text{ (DEVICE)}$$

$$\frac{v = \mathit{sources}(r) \qquad D' = \Sigma \langle \mathit{update}(\mu, r, v) \rangle\, t \triangleright (\{r\}; \{r\})}{(M; D_1|...|\Sigma \langle \mu \rangle\, t|...|D_n) \rightarrow_C (M; D_1|...|D'|...|D_n)} \text{ (SOURCETX)}$$

$$\frac{\begin{array}{c}\bar{r} = \mathit{dom}(\mu_s) \cap \mathit{dom}(M) \qquad \mathit{merge}_i = M(r_i) \\ v_i = \mathit{merge}_i(\mu_s(r_i).\mathit{val}, \mu_r(r_i).\mathit{val}) \\ D_s = \Sigma_s \langle \mu_s \rangle\, t_s \qquad D_r = \Sigma_r \langle \mu_r \rangle\, t_r \qquad A = \{r_i \mid \mu_r(r_i).\mathit{val} \neq v_i\} \\ D_r' = \Sigma_r \langle \mathit{update}(\mu_r, r_i, v_i) \rangle\, t_r \triangleright (A; A)\end{array}}{(M; D_1|...|D_s|...|D_r|...|D_n) \rightarrow_C (M; D_1|...|D_s|...|D_r'|...|D_n)} \text{ (SYNCHRONIZETX)}$$

Figure 4.9: Operational semantics for remote updates.

The DEVICE rule models any of the concurrent devices D_i taking a normal step in the device evaluation relation $\rightarrow_D$ (c.f., Section 4.4). This model allows for devices to execute at different speeds or pause execution for some time before resuming.

The SOURCETX starts a new transaction when any of the sources are activated externally, e.g., by a user interaction. When exactly this is the case is defined by the *sources*() function which returns the new value for the source. In our calculus this function is underspecified, but is understood to represent an input from an external system. The started transaction is the same as in the FIRETX rule.

The SYNCHRONIZETX rule models all communication between devices, including delayed, duplicated, and dropped messages. Conceptually the rule selects two devices one device D_s that sends the values of all replicated reactives in its store μ_s and another device D_r that receives and merges those values and starts a new transaction. Neither D_s nor D_r have an ongoing transaction – synchronization may only happen when the flow graph of the application is idle. The rule always merges all replicated reactives $\bar{r}$ of the sending device. The values of both replicas of each reactive are merged pairwise and the results are used as the new values for the store μ_r of the receiving device. If the receiving device does not have a value for a reactive, then merging returns the sent value. A new transaction is started on the receiving device with the changed replicated reactives A set as activating and processed. The transaction propagates the received changes to the local flow graph.

Dropped messages and networks with only partial conenctions between devices are modeled by not executing SYNCHRONIZETX for a pair of devices. Delayed and reordered messages use artificial devices as an indirection. Artificial devices are normal devices as far as the calculus is concerned, but server no purpose in the model of the application and are only used to synchronize state. A delayed message from a device A to a device B, for example, is modeled by first synchronizing from A to an artificial device A'. Then at any later point, A' synchronizes with B.

For consistency between devices, the merge function must form a semilattice (i.e., it is associative, commutative, and idempotent). Eventual consistency does require that all devices eventually do receive new messages for all reactives.

Sending Nested Reactives

A replicated reactive r_0 may contain other nested reactives r as values. When a device D adds r to the value of r_0 for the first time, there are two cases to consider.

First, if r is already a replicated reactive each device uses its local value of r inside r_0. The two reactives r and r_0 are synchronized as any other two replicated reactives. Second, if r is not a replicated reactive, only the local device d can cause r to change. In this case, d promotes r to a replicated reactive by providing an initial value and a merge function. The merge function for r selects the latest value according to a logical timestamp. This latest-writer-wins scheme is race condition free, because the original device is the only writer of r. The current value of r is sent along the value of r_0 when synchronizing, to allow remote devices to initialize a replica of r. Once initialized, r synchronizes independently of r_0.

To simplify the presentation of the formalization we always assume the first case, i.e., that only replicated reactives are nested into other replicated reactives. We can do so without loss of generality by assuming that the transformation explained for the second case is applied implicitly.

4.8 Conclusion

We have chosen an operational semantics to present $\mathcal{F}_r$ in this thesis because enables the reasoning about failures in the middle of execution. The calculus focuses on the core building blocks required to define a dynamic flow graph, but omits concrete concerns such as the distinction between signals and events, which are not necessary to reason about failure. Chapter 5 uses $\mathcal{F}_r$ as presented in this chapter to formally prove correctness. Chapter 6 then shows how REScala implements $\mathcal{F}_r$ in a modular way, such that different semantics for reactives (such as signals and events) can be implemented.

Chapter 5

Theory of Devices, Distribution, and Restoration

In this chapter, we present and proof the properties of $\mathscr{F}_r$. The properties apply to REScala and we make assumptions about the embedding language explicit, thus paving the path for other implementations of the Cvtr programming paradigm. In particular, we emphasize that we keep assumptions about the functionality of the embedding language minimal and have the Cvtr paradigm provide correct behavior even in corner cases.

We first prove properties of the evaluation of individual devices. We show that transactions on a single device are glitch-free, complete, deterministic, and isolated. The local guarantees constitute the foundation for fault tolerance.

Definition 5.1 (Syntax). We write $\rightarrow = \rightarrow_D \cup \rightarrow_p$, and $\rightarrow^*$ for the transitive closure. We write $\mu \in D$ to say that μ is the store of D, and likewise with other syntax. We write $D \in D_0 \rightarrow ... \rightarrow D_n$ and $D_i \rightarrow D_j \in D_0 \rightarrow ... \rightarrow D_n$ to say that D and $D_i \rightarrow D_j$, respectively, are contained in the sequence $D_0 \rightarrow ... \rightarrow D_n$.

5.1 Traces

Many proofs reason about program behavior in a single transaction or between transactions. We use *traces* to reason about sequences of steps of device evaluation while abstracting the concrete steps taken. We assume that traces for device evaluation are finite, i.e., that the evaluation of devices terminates when no external inputs are received. The proofs only reason about finite slices out of the full trace anyways, thus using only finite traces simplifies the presentation without affecting the proved properties.

Definition 5.2 (Trace). A trace of a device D_0, written $\mathit{trace}(D_0)$, is a sequence $D_0 \to^* D_n$, where D_n cannot be further reduced.

Definition 5.3 (Propagation). Given D_0, a propagation $p = \mathit{ptrace}(D_i, D_j)$ is any maximally long subsequence $D_i \to_p^* D_j$ of $\mathit{trace}(D_0)$, i.e., there is no longer sequence $D_i' \to_p^* D_j'$ that contains $D_i \to_p^* D_j$.

Definition 5.4 (Reevaluation). We call any $D \xrightarrow{r}_p D'$, which was produced by the REEVALUATE rule, a reevaluation of r.

Definition 5.5 (Transaction). Given a propagation $p = \mathit{ptrace}(D_i, D_j)$, the transaction trace $t = \mathit{transaction}(p)$ is $D_{i-1} \to_D p \to_D D_{j+1}$.

5.2 At-most-once Reevaluation of Reactives

We show that propagation ensures that reactives are reevaluated at-most-once. In a realistic implementation, multiple reevaluations of a single reactive are observable using side effects (directly in the reevaluation, or by observing execution time or energy consumption). Even in $\mathscr{F}_r$, multiple reevaluations would produce incorrect values for folds, i.e., the inputs are aggregated multiple times.

Lemma 5.6 (At-most-once evaluation). Each propagation $\mathit{ptrace}(D_0, D_n)$ contains at most one reevaluation $D \xrightarrow{r}_p D'$ of each reactive r.

Proof. By the premise of REEVALUATE it must hold that $r \in \mathit{ready}(P, \mu)$ which requires $r \notin P$. However, the REEVALUATE rule adds r to P, and no step during a propagation removes reactives from P. Thus, there can be at most one reevaluation of each reactive r.

□

5.3 Glitch-free Propagation

When transactions are incorrectly propagated, inconsistencies – called glitches – can arise. The concrete definition of a glitch varies in the literature, but corresponds to the situation where values that logically belong to different points in time are observed at the same time. In $\mathscr{F}_r$, a glitch happens precisely when the value of a reactive is written after a derived reactive has been reevaluated in the same propagation (c.f. Definition 5.7) A system is called glitch-free if there are no glitches.

Definition 5.7 (Glitch). For a propagation $p = ptrace(D_0, D_n)$ and a reevaluation $D_{j-1} \xrightarrow{r}_p D_j$ in p and for any input $r' \in inputs(r, \mu) \setminus \{r\}$ of r, any write $D_{k-1} \xrightarrow{r'}_p D_k$ with $k > j$ in p is called a glitch.

Lemma 5.8 (Glitch Freedom). There are no glitches in all propagations.

Proof. By contradiction. Assume there is a write on r' satisfying the conditions above. Because r is reevaluated it must be $r \in ready(P, \mu)$ for some μ and P, thus $r' \in P$ at the time of the reevaluation. Due to at-most-once reevaluation (Lemma 5.6), for $r' \in P$ to be true, the sole reevaluation of r' at $D_{i-1} \xrightarrow{r}_p D_i$ it must be $j > i$, i.e., r' is reevaluated before r. However, $k > j > i$ means that the reevaluation of r is between the reevaluation of r and the write of r, which by inspection of the rules is impossible.

□

5.4 Complete Propagation

At-most-once evaluation (Section 5.2), and glitch freedom (Section 5.3) are trivially fulfilled if transactions are not propagated at all hence, we require an additional liveness property. We show that after a propagation, all derived reactives reflect the changes to their inputs, given their operators. For folds applying their operator multiple times aggregates new values even without changed inputs, thus it is also incorrect to just reevaluate all reactives. We show that $\mathscr{F}_r$ reevaluates reactives that are reachable in the flow graph from the transaction (Lemma 5.12), except when processing is stopped by an activation function. To this end, we first show that there is always a *ready* reactive until all reactives become processed, and that exactly the reachable and not filtered reactives become active (Lemma 5.10).

Lemma 5.9. For any step during a propagation, either all reactives are processed $P = dom(\mu)$, or there is at least one $r \in ready(P, \mu)$.

Proof. By construction. Pick any unprocessed $r \in dom(\mu) \setminus P$, if r is ready, we found a candidate. Otherwise, there must be an unprocessed input $r_i \in inputs(r, \mu)$ from which we continue our search. This search must terminate, because the graph is acyclic.

□

Lemma 5.10. At the end of any propagation started by a transaction on reactive r with store μ the set of active reactives A is the set of transitively derived reactives of r excluding any paths on a reactive r' that is filtered $filter(\mu, r') \neq true$.

Proof. Inspection of the rules show that r is added to the active reactives A at the start of a propagation (*TX rules). Reactives are *outdated* only if they are derived from reactives in A. Exactly the *outdated* and *ready* reactives are processed by the REEVALUATE rule which adds them to A if they are not filtered. All other *ready* reactives are processed by the SKIP rule, which does not add them to A. Because of Lemma 5.9 there is always a ready reactive until all reactives are added to P and reachable ones are also added to A.

□

Definition 5.11 (Complete Reactives). Given a propagation $p = ptrace(D, D')$ with respective stores $\mu \in D$ and $\mu' \in D'$, and a new synthetic store $\mu'_r = update(\mu', r, \mu(r).val)$ where only the value of r is unchanged. We say a reactive r is $complete(r)$ in p if and only if evaluating r in μ'_r produces the value of the reactive in the final store $eval(r, \mu'_r) = \mu'(r).val$.

Lemma 5.12 (Complete Propagation). For any propagation p starting with a transaction setting r to v and ending in store μ', we call p complete if

- the transaction changes the correct reactive $\mu'(r).val = v$,
- all active derived reactives $\{r' \in A\}$ are complete,
- all non-active folds and sources keep their values.

Proof. First, the *TX rules cause a transaction to set the correct state and mark the reactive as processed. Because of at-most-once reevaluation (Lemma 5.6) we know those will not be changed again. Second, all active reactives are reevaluated (Lemma 5.10). Due to glitch freedom (Lemma 5.8) each reevaluated reactive is complete and because of at-most-once evaluation (Lemma 5.6) they do not change afterwards. Finally, the SKIP rule causes all non-active reactives to become processed without changing their value (Lemma 5.10).

□

5.5 Determinism

The execution of devices is deterministic with the following caveats.

- When embedding Cvtr into a language, user-defined functions may be non-deterministic. We thus assume that either the embedding language guarantees deterministic user-defined functions or otherwise developers are responsible for ensuring used functions are deterministic.
- The order of reevaluations during propagation is not deterministic. However, propagation is confluent, i.e., always produces the same result. Thus, when considering the overall result of a transaction, the execution is still deterministic.

Lemma 5.13 (Confluence). For any two propagations $p_1 = ptrace(D, D_1)$ and $p_2 = ptrace(D, D_2)$ starting at the same configuration D, the final states $\mu_1 \in D_1$ and $\mu_2 \in D_2$ are equal $\mu_1 = \mu_2$.

Proof. No reactives are created during a propagation, thus $dom(\mu) = dom(\mu_1) = dom(\mu_2)$. Due to Lemma 5.10 the set of active reactives $A_1 \in D_1$ and $A_2 \in D_2$ are the same $A_1 = A_2$, and by complete propagation (Lemma 5.12) they evaluate to the same values.

□

Lemma 5.14 (Determinism). For any device D all execution traces $trace(D)$ contain the same sequence of $\rightarrow_D$ steps, i.e., they are equal after removing all $\rightarrow_p$ steps.

Proof. Follows from the determinism of the λ-calculus, which we extend only with deterministic $\rightarrow_D$-rules, and the confluent propagation (Lemma 5.13).

□

5.6 Isolated Propagation

Isolation captures the final piece of the synchronous nature of $\mathcal{F}_r$ by stating that propagations do not interfere with each other. $\mathcal{F}_r$ itself executes only one propagation at a time, thus propagations are trivially isolated from each other. In a distributed $\mathcal{F}_r$ application, propagations are executed concurrently. However, because there is no shared state, propagations on different devices are naturally isolated from each other.

Lemma 5.15 (Isolation). The resulting configuration D' of a propagation $p = ptrace(D, D')$ is independently of any concurrent transaction.

Proof. Propagation is confluent (see Lemma 5.13) and there are no concurrent transactions by inspection of the rules. None of the *TX rules may trigger concurrent transactions when a propagation is in progress.

□

5.7 Optimal Parallelization of Propagation

$\mathcal{F}_r$ is designed for efficient implementations by allowing for high-level optimizations due to the use of managed propagation. Reevaluation of multiple reactives – which are *ready* – are parallelizable. In addition, we show that the algorithm used in $\mathcal{F}_r$ is optimal with regard to the number of reactives that are *ready* at any given

point of time. Thus, allowing for maximum parallelization. These results are equivalent to earlier work in SID-UP [69] and FElm [65] both of which are also optimal but proven in the original publication of this chapter for the first time [157]. We only consider configurations during propagation where the SKIP rule has already been fully applied, because skipping reactives does not constitute work we want to parallelize. This simplifies the proof. SID-UP [69] implements an efficient way to not compute skips at all.

Lemma 5.16. For a propagation $p = ptrace(D_0, D_n)$ and device $D_i \in p$ with processed reactives $P \in D_i$ and store $\mu \in D_i$. If the SKIP rule is not applicable to D_i, then reevaluating a reactive $r \notin ready(P, \mu)$ violates one of our correctness guarantees.

Proof. By contradiction. Assume there exists a reactive $r \in dom(\mu)$ but $r \notin ready(P, \mu)$. It holds that $r \notin P$, otherwise r would be reevaluated twice (see Lemma 5.6). Inspecting the *ready* function shows that there must be an input $r_i \in inputs(r, \mu)$ which is not yet processed $r_i \notin P$. We use a similar argument as in Lemma 5.9 to show that either r_i or one of its predecessors must be ready. Due to complete propagation (Lemma 5.12) the reactive r_i will be reevaluated in the future because it has a ready predecessor. Thus, reevaluating r is not glitch-free (Lemma 5.8).

□

5.8 Eventual Consistency

Replicated reactives in $\mathscr{F}_r$ have an associative, commutative, and idempotent merge function for each replicated reactive $r \in dom(M)$, and all devices use the same merge function for the same replicated reactive r. Thus, each replicated reactive corresponds to a CRDT for which eventual consistency is established [110]. $\mathscr{F}_r$ combines eventual consistency and complete propagation (Lemma 5.12) to provide eventual consistency for the entire interactive application. In general, state derived from a replicated reactive r could become inconsistent between replicas, because both replicas observe a different number and order of activations of r. The key insight is that only the state of fold reactives depends on the number and order of activations. Thus, fold reactives derived from a replicated reactive must be distributed themselves to make the application eventually consistent. This property is captured by the following theorem.

Definition 5.17 (Consistent reactives). Given a reactive r and devices D_1 and D_2 with states $\mu_1 \in D_1$ and $\mu_2 \in D_2$, we say r is consistent if $r \notin dom(\mu_1) \cap dom(\mu_2)$ or $\mu_1(r).val = \mu_2(r).val$

Lemma 5.18 (Eventual consistency). Given two devices D_1 and D_2 and reactive $r \in dom(M)$. If there are no other changes to r and the devices eventually exchange values, then r is consistent.

Proof. Follows directly from commutativity and idempotence of $M(r)$ once both devices have received the remote value at least once with the SYNCHRONIZETX rule.

□

Theorem 5.19. A fully connected component of reactives R in the flow graph is eventually consistent, if all sources $R_s \subset R$ and folds $R_f \subset R$ are eventually consistent.

Proof. Once all R_s and R_f reach a consistent state, then because of complete propagation (Lemma 5.12), all derived reactives R become consistent.

□

5.9 Dataflow Causal Consistency

From a user's perspective, eventual consistency is a weak guarantee, because reactions may be perceived to happen out of order. Ideally, it is desirable to provide causal consistency, because the CALM theorem [103] states that we cannot do better without losing availability.

To reason about causal consistency we need a way to causally relate values. We say that all values in the store μ_t at the end of a transaction t are causally related. Causal consistency is violated if at any point in time (i.e., outside a transaction) one value v_1^t from a transaction t is visible on a device, but at least one other value v_2^t causally related to that value is not visible.

For our state-based CRDTs, a past value v_2 is visible if the current value v_1 is larger $v_1 \geq v_2$ than the past value according to the merge function of the reactive $v_1 \geq v_2 \iff merge(v_1, v_2) = v_1$. Past values are defined by the order of transactions. Transactions that happen on the same device are ordered in their natural order. Once synchronization happened, transactions on the sending device are ordered before transactions on the receiving device.

Definition 5.20 (Transaction order). Given a device D with propagations $p_{i,j}$ and transactions $t_{i,j} = transaction(p_{i,j})$ we say that $t_i < t_j$ if the t_i occurs first in the trace of the device $trace(D)$.

For two devices $D_{s,r}$ in a communication system $(M; D_1|...|D_s|...|D_r|...D_n)$ and transactions $t_{s,r}$ on the respective devices we say that $t_s < t_r$ if there was an application of the SYNCHRONIZETX involving D_s and D_r that happened in the communication trace after t_s and before t_r.

Definition 5.21 (Stores of transactions). Given a device D with propagation $p = ptrace(D, D')$ and transaction $t = transaction(p)$, we say that μ_t is the store at the end of transaction t.

Definition 5.22 (Causal consistency). A transaction store μ_t is causally consistent, if for each replicated reactive $r \in dom(M) \cap dom(\mu_t)$ the value in the store is larger than the value of all prior transactions $\mu_t(r).val \geq \mu_{t_0}(r).val$ for $t_0 < t$.

Single devices are trivially causally consistent because all CRDT operations only increase the internal state according to the merge function. We show that communication between exactly two devices leads to stores that are also causally consistency, by reasoning about synchronization. Finally, we show that any inconsistency on arbitrarily many devices would already exists with only two communicating devices.

Lemma 5.23 (Two device causal consistency). When there are exactly two devices D_1 and D_2 communicating, then all transactions on both devices are causally consistent.

Proof. We only look at the case of SYNCHRONIZETX transactions. All other transactions only affect the local device and thus are trivially causally consistent. We assume the stores of both devices before the transactions are causally consistent. The sending device D_s with store μ_s is not modified, thus remains consistent. The receiving device D_r starts a transaction t_r in store μ_r ending with store μ_r'. The receiving device D_r starts a transaction where the value of each replicated reactive r_i receives a new value $v_i = merge_i(\mu_s(r_i).val, \mu_r(r_i).val$. Due to complete propagation (Lemma 5.12) these are the final values of the replicated reactives in the receiving store μ_r'. Because merge is idempotent all values v_i are larger than the values of the sending and receiving devices. Because of the same reasoning as for local transaction, the remaining propagation does also only make updated reactives larger. Thus, SYNCHRONIZETX for two devices keeps causal consistency.

□

Lemma 5.24 (Causal consistency). All communicating devices are causally consistent.

Proof. Pairwise causal consistency (Lemma 5.23) directly generalizes to causal consistency if all replicated reactives are synchronized by each pair of devices. Because all reactives are synchronized and because of the monotonicity of the merge function any causal inconsistency on the receiving device would have already be present on a sending device. However, we assume that devices start in a causally consistent state and local transactions cannot introduce causal inconsistencies by definition.

□

5.10 Static Restoration

Because dicApps are non-deterministic in nature (there is always a user), the restored application behaves differently as soon as the first external input is processed by a transaction. However, due to the dynamic nature of the flow graph, the restoration process is never truly finished, as there could always be new reactives created that must be restored. Thus, we can only show that the store of a restored application is identical until the first transaction. Further restoration happens incrementally, and we show that all restored reactives individually behave as if the crash never happened, but the overall restored application may now have a different set of reactives.

We first define *proper* devices (devices with a snapshot that matches the store) and then show that restoring the application from a proper snapshot produces the same store.

A device is *proper*, if each value in the current snapshot matches the corresponding value in the store. The reduction relation $\rightarrow_D$ reduces configurations with proper devices to other proper devices, and propagations started at a proper device will result in a proper device.

Definition 5.25 (Proper device). A device $D = \Sigma \langle \mu \rangle t$ is proper if and only if all sources and folds in μ are included in Σ with $\forall r \in dom(\mu) : (inputs(r, \mu) = \emptyset \vee r \in inputs(r, \mu)) \rightarrow \mu(r).val = \Sigma(r)$.

Lemma 5.26 (Evaluation produces proper devices). If $D = \Sigma \langle \mu \rangle t$ is proper, then $D' = \Sigma' \langle \mu' \rangle t'$ with $D \rightarrow^* D'$ is also proper.

Proof. The SOURCE and FOLD rules ensure that created reactives are included in the snapshot, and at no other time is the store modified outside an ongoing propagation. The COMMIT rule updates the snapshot to include the values of all reactives which were active during propagation. □

For the remaining discussion, we individually look at two separate stages of executing an application: the first stage only creates new reactives, hence we call the creation prefix of the application, and the rest of the application, which does read or modify the restored values from the flow graph. For example, the chat application in Figure 2.2 only consists of a creation part. Languages such as FElm [65] only consist of such creation parts and modifications or observations of the flow graph are only executed by the runtime after the program has terminated. We show that for the creation prefix, the restoration produces an exact subset of the store before a crash, and that the rest of the program has a trace which produces the same observable behavior as if the crash had not happened.

Definition 5.27 (Creation prefix). The creation prefix of a device, $ctrace(D_0, D_n)$, is the longest prefix $D_0 \rightarrow_p^* D_n$ of $trace(D_0)$ where none of the steps in the trace was produced by the *TX or ACCESS rules.

Consider restoring just the creation prefix of an application. During restoration – because the application is deterministic – it reproduces the original trace. When the application has executed all creation reductions, its store is a subset of the store before the crash. All restored reactives have the same value they had before the crash. However, reactives created after the creation prefix have not yet been restored.

Lemma 5.28. For any proper device $D \in trace(D_0)$ with $D = \Sigma\langle\mu\rangle t$ and $D_0 = \Sigma_0 L$ the re-execution of the creation trace of the application with the snapshot $ctrace(\Sigma L, D')$ ending with D' has a store $\mu' \in D'$ which agrees with the original store $\mu' \subseteq \mu$.

Proof. Creation of reactives during restoration still happens in the same order, because there are no steps produced by the ACCESS rule in the creation prefix. That is, the execution of the process does not depend on the values in the snapshot. In particular, $dom(\mu') \subseteq dom(\mu)$. It remains to show that each reactive $r \in dom(\mu')$ matches the value before the crash $\mu(r).val = \mu(r).val$. Due to proper devices, it holds that $\forall r \in dom(\Sigma) \cap dom(\mu) : \mu(r).val = \Sigma(r)$. For the SOURCE and FOLD rules, the lookup function returns $\Sigma(r)$ as a new value, which is equal to $\mu(r).val$. For the MAP and FLATTEN rules, the new value is computed from the values of the dependencies, which is equal to $\mu(r).val$ because of complete propagation (Lemma 5.12). □

5.11 Incremental Restoration

The execution of applications after the creation prefix may observe values restored from the snapshot using the ACCESS rule. Thus, reactives may be created in different orders, and firing of new transactions may cause different reactions. We first show that the value of restored reactives is correct, independent of the order of restoration.

Lemma 5.29. Assume a restoration of $ctrace(D_0', D')$ as in Lemma 5.28. Creation of new reactives in the restored trace $D' \in trace(D_0')$ will produce a store μ' compatible with the store μ of the crashed device D: $\forall r \in dom(\mu) \cap dom(\mu') : \mu(r) = \mu'(r)$.

Proof. Lemma 5.28 does not depend on the order in which reactives are restored. Thus, the reactives are restored with the correct values, independent of the order of their restoration. □

Triggering new transactions after restoring the creation prefix of the device D'_0 also causes compatible changes. That is, the same transaction propagated in the original device D and the restored device D' will change the same reactives consistently.

Lemma 5.30. Assume a device D and the restored device D' as in Lemma 5.29, i.e., their stores $\mu \in D$ and $\mu' \in D'$ are compatible $\forall r \in dom(\mu) \cap dom(\mu') : \mu(r) = \mu'(r)$. Propagating the same transaction on both devices produces new devices $D \rightarrow^*_p D_p$ and $D' \rightarrow^*_p D'_p$ with stores $\mu_p \in D_p$ and $\mu'_p \in D'_p$ that are still compatible: $\forall r \in dom(\mu_p) \cap dom(\mu'_p) : \mu_p(r) = \mu'_p(r)$.

Proof. Follows from complete propagation for the same store and transaction (Lemma 5.12). The updated restored store is still a subset of the updated store before the crash $\mu'_p \subseteq \mu_p$, as the same values have been updated in both stores. □

When propagating the same transaction on the original device D and the restored device D', the propagation may update reactives of the original device, which have not been created on the restored device. In particular, if a fold reactive is not yet restored, any transaction changing that fold on the original device may cause the applications to diverge. However, this issue occurs independently of restoration, i.e., the application initialization was non-deterministic to begin with. Thus, restoration is also non-deterministic in such cases.

5.12 Restoration

In conclusion, Lemma 5.26 shows that snapshots contain enough information to restore after a crash and Lemma 5.28 shows that the restoration does produce the exact same store as before the crash. In addition, Lemma 5.29 and Lemma 5.30 show that new transactions may immediately change the store for a partially restored application without compromising correctness.

Theorem 5.31 (Restoration). The creation prefix of the execution of an application is restored exactly as before a crash, and after the creation prefix any transaction causes the same changes as before the crash.

Proof. Follows from Lemma 5.28, Lemma 5.29, and Lemma 5.30. □

5.13 Conclusion

The Cvtr programming paradigm is focused on providing developers with guarantees that allow them to implement applications that are intuitive to their users. The properties proven in this chapter reflect those intuitions.

Causal consistency aligns with the intuition of users that if a change becomes visible, then the cause of those changes should also be visible. Yet, causal consistency still allows the implementation enough flexibility to provide availability and offline usage.

Restoration balances the desire to provide a perfect replication of the crashed application with more practical concerns. Restoration behaves nearly identical to the normal start of an application, except that none of the important data of the user is lost. In particular, the restoration process does not require a complete restoration before the application is usable again, but instead allows immediate use. While this formally causes states that would not be possible without a crash the only divergence is limited to a different set of reactives that exists after the restoration. For the users this appears as a part of the application they are not currently using to be restored later.

Chapter 6

The REScala Project

In this chapter we describe how REScala extends $\mathscr{F}_r$ to provide a modular implementation of the Cvtr programming paradigm. This involves a discussion of the internal layout of REScala, how the abstractions of $\mathscr{F}_r$ are encoded in Scala, and how the project is modularized. Code examples in this chapter are taken directly from the implementation without modifications other than removing visibility modifiers of methods. But first, we look at the history behind REScala.

6.1 A Short History of REScala

REScala was created due to an independent desire to continue research on usability and comprehensibility of functional reactive programming into and integrating it into the object-oriented paradigm, inspired by projects such as FlapJax [146]. REScala technically was an extension of EScala [88] – an implementation of C# style events in Scala – with a new internal implementation to handle synchronous propagation. A focus of early version of REScala was a straightforward integration into typical object-oriented APIs [181], bridging the gap between code that could be used and understood by undergraduate students and the semantic benefits of functional reactive programming. Due to the success of this experiment the API of REScala remains overall similar to how it has been ever since 2014. However, besides the API, some tests, and some cases studies, nothing of the original implementation of REScala remains – it has been replaced part for part during the work on this thesis to support the features needed for the Cvtr paradigm.

From the very early days, the potential of REScala for the development of distributed systems was explored as part of the PACE ERC project. Joscha Drechsler, as part of his PhD project, designed an update algorithm for flow graphs which does not require global coordination named SID-UP [69]. SID-UP's implementation [1] uses node local source sets to track when an update may progress, instead of the global priority queue used by REScala. While the implementation of SID-UP is separate from REScala, the work was refined and the results were merged into the REScala code base as one of REScala's schedulers. Schedulers are explained in Section 6.5, while the integration is described in Joscha's dissertation [68]. To

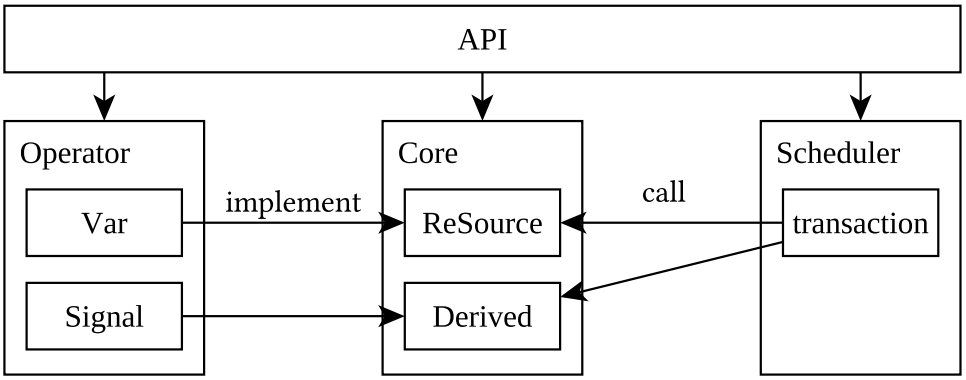

Figure 6.1: Relationship between REScala's packages. Core in Figure 6.2, operators in Figure 6.4, and scheduler in Figure 6.9.

enable the rapid development of different update algorithms with different guarantees and their comparison the internal design of REScala was refined multiple times until it reached both simplicity and elegance without sacrificing much flexibility or performance. The rest of this chapter explains this design and extends the concepts of the formal model towards a full implementation.

6.2 REScala's Internal Design

REScala is split into three main packages, the core, the operators, and the schedulers. The relationship between the packages is sketched in (Figure 6.1). The core serves as a common interface between the operators and schedulers. The operators implement the interfaces in the core and the schedulers call the methods provided by those interfaces. A fourth package, the API, provides developers a uniform interface to the functionality of the three internal packages.

The API of REScala corresponds to the surface syntax we have already seen – methods to create new reactives, start transactions, add observers, and interact with external systems. Technically, the API package often simply exports the functionality provided by the operators and schedulers, but bundled into a single package.

The operators of REScala define the behavior of reactives. This use of the term operator differs from the formalization where the operator only refers to the function used for computing new values. An operator is to a reactive like a class is to an object – the operator defines the behavior and the reactive is an instance of that behavior in the flow graph. Specifically, an operator defines the value type, the function for reevaluation, the activation condition, dynamic reconfiguration of input dependencies, and external effects. The implementations for operators are further subdivided into sources, events, signals, and observers.

Schedulers control the dependencies between reactives and thus are responsible for traversing the flow graph, in particular transactions and propagation. Schedulers implement the update algorithm that decides when reevaluation functions are

```
54 trait Struct {
55   type State[V, S <: Struct]
56   def canNotBeImplemented[A]: A
57 }
58
59 trait ReSource[S <: Struct] {
60   type Value
61   final type State = S#State[Value, S]
62   def state: State
63   def name: ReName
64   def commit(base: Value): Value
65 }
66
67 trait Derived[S <: Struct] extends ReSource[S] {
68   def reevaluate(ticket: ReevTicket[Value, S]): Result[Value, S]
69 }
```

Figure 6.2: Basic abstractions used in the core of REScala to interface between its operators and schedulers.

executed based on the knowledge encoded into the flow graph. In extension to $\mathscr{F}_r$, the schedulers also protect the values of the reactives from concurrent modifications, and provide distributed execution of the reevaluation functions.

The core of REScala defines the `ReSource` and `Derived` traits (traits are Scala's version of interfaces) to internally represent reactives as nodes in the flow graph. Their purpose is to provide the interface that glues the scheduler and the operators of REScala together. `ReSource` provides the declarations combine value parts of a reactive provided by the operator and the state parts provided by the scheduler. `Derived` is a subtype of `ReSource` and extends the latter with a method for reevaluation that allows the scheduler to call the method defined by the operator.

It is possible to provide new operators without changing schedulers, and to implement new schedulers that work with all operators. In the following sections, we look at the core, the operators, and schedulers of REScala in detail.

6.3 The Core

The core of REScala consists of `ReSource` and `Derived` – those two traits and all of their methods have to sole purpose of facilitating the communication between the schedulers and the operators. Figure 6.2 shows the code of the two abstractions.

ReSource

`ReSource` (Line 59) declares a state (Line 62) used by the scheduler and operator, a name (Line 63) for debugging and pretty printing, and a method to encode how

```
70 trait Result[T, S <: Struct] {
71   def activate: Boolean
72   def forValue(f: T => Unit): Unit
73   def forEffect(f: Observation => Unit): Unit
74   def inputs(): Option[Set[ReSource[S]]]
75 }
76
77 class StaticTicket[S <: Struct](creation: Initializer[S]) extends
       InnerTicket(creation) {
78   def collectStatic(reactive: ReSource[S]): reactive.Value
79   def dependStatic[A](reactive: Interp[A, S]): A =
         reactive.interpret(collectStatic(reactive))
80 }
```

Figure 6.3: Result trait communicating all the information we have already seen in $\mathscr{F}_r$ between the scheduler and operator of REScala and the ticket implementation used for reactives that allow dynamic access.

the value changes when the transaction commits (Line 64). As an example for committing: each event forgets its current value during commit and resets to an empty value.

The type of the state of a `ReSource` is defined `S <: Struct` – the structure type parameter. The `Struct` trait (Line 54) encodes a type-level function (Line 55) that computes the state type of a `ReSource` given the value type of the operator (and – for technical reasons – also the type of the structure itself). We use this type-level function to simulate a limited form of dependent classes [87], where the `ReSource` trait is implemented by an operator, but the inner state implementation is provided by the scheduler. This enables the scheduler to provide transactional guarantees on the state inside of operators without compromising type safety.

Most interfaces in REScala are parameterized on the `Struct` type. Most notably, signals, events, and schedulers carry this parameter. This enables the use of multiple different schedulers in the same program, while having the type checker ensure that reactives from different schedulers cannot be combined with each other. Unchecked combinations result in runtime violations, because different schedulers use completely different metadata for their reactives. It is of note, that the structure trait is a purely compile time construct, it never has any actual implementations (which is actually enforced by the construct in Line 56).

Note on Scala 3 Scala 3 has removed type projection (the `Struct#State` syntax) because it may lead to unsoundness when combined with certain other language features. For compatibility, we adapted the encoding of the state type in the implementation. However since this happened after the submission and assessment of this thesis the code and descriptions provided here remain as they were, but we do

provide a summary of the changes. Instead of projecting the state type from the type parameter provided to all core classes, these classes are now inner classes of a new outer scope. The state type is defined as a type member of the outer scope. This new encoding is technically more complex, but does not change the developer API nor any of the provided guarantees.

Derived

`Derived` (Line 67) extends a `ReSource` with the ability to reevaluate (Line 68). Reevaluation follows a protocol where a ticket for accessing the inputs of the reactives is provided by the scheduler and used by the operator. The reevaluation function returns a result, which encodes all the changes that need to be applied to the reactive, as well as information about how the propagation has to continue. Figure 6.3 shows the result trait and a ticket class. The result trait provides accessors for the activation, value, effect, and dynamic inputs that are caused by the reevaluation. The static ticket class is one of multiple ticket implementations, but all other implementations behave the same. The different tickets exist to improve performance for common cases, for example, when there are no dynamic dependencies. All tickets provide a `depend` method to access the value of a `ReSource` (Line 79).

Creation of Reactives and Transactions

The core module also facilitates the interaction between operators and the schedulers when creating reactives, and when starting new transactions. There are also abstractions in the core package for those interactions, but they are more of an implementation detail of REScala and less interesting for the concepts of Cvtr and the relation with $\mathscr{F}_r$. We will see the creation of reactives and transactions in more detail when discussing the scheduler and operator packages, but the short overview is as follows.

A reactive is created in three steps. The scheduler first creates the internal state. The state is then passed to the operator package and wrapped inside an operator implementation to form a complete reactive. Finally, the reactive is passed back to the scheduler to be added to the flow graph.

To start a new transaction, the user-defined transaction code collects an intended change from each source. These intended changes are then passed to the scheduler that applies the changes to all sources simultaneously, and then performs a propagation. The next section looks at these interactions from the operator's view to show how both the ticket is used, and the result is created.

6.4 The Operators

The operator of a reactive defines how that reactive behaves. We might say that a specific reactive is an instance of an operator similar to how we would say that a specific object is an instance of a class. For example, the map event operator

defines that its reactive has a single input, the value of the reactive is derived from the value of the input by applying a user-defined function. The concrete details – such as what the input or the user-defined functions are – are only specified when an instance of a reactive is created, they are not part of the operator. While there are many individual operators in REScala their behavior usually belongs to one of the following categories of operators. The categories consist of the two sources (Var and Evt) and the three derived reactives: signals, events, and observers. The similarities between operators in the same category makes the semantics of REScala more approachable for developers.

To understand the role of operators in the implementation of REScala, we will discuss how new operators are implemented. Our formalization, $\mathcal{F}_r$, does not distinguish between categories such as signals and events and the same is true for the internal implementation of REScala. Thus, new implementations of operators are also not limited to these categories, instead each new operator independently defines the following properties and behaviors of derived reactives and source reactives (the separation into sources and derived reactives is required by the core package).

- The initial value and type of value of the reactive. (For both sources and derived reactives.)
- The initial list of inputs.
- Which static and dynamic inputs are accessed during reevaluation.
- The activation function of the reactive.
- The new value computed during reevaluation.
- Any effects to be executed at the end of the transaction.
- A change to the value during the commit of a transaction.
- How external changes are applied to sources.

Custom Sources

Figure 6.4 shows the implementation of a new operator (`CustomDerivedString` in Line 81) for derived reactives and how to create a new reactive using that custom operator (`customDerived` in Line 99).

Instances of `CustomDerivedString` define all the reevaluation related functionality. The initial state (Line 82) and single initial input (Line 83) are the parameters to the operator and define how two reactives using this operator differ.

The reevaluation method (Line 91) of the custom operator accesses (Line 92) the value of the `inputSource` statically, and returns a transformed value. Using `withValue` (Line 93) to return a value also makes the reactive activate – signaling to the scheduler that the change must be propagated.

The implementation of the `commit` method (Line 89) states that the value after the transaction stays unchanged.

To create reactives we need the state implementation from the scheduler, because the scheduler controls when reactives are created and how. Technically, the

```
81  class CustomDerivedString(
82      initState: S#State[String, S],
83      inputSource: Interp[String, S]
84  ) extends Derived[S]
85      with Interp[String, S] {
86    type Value = String
87    def state: State               = initState
88    def name: ReName               = "I am a name"
89    def commit(base: Value): Value = base
90
91    def reevaluate(input: ReIn): Rout = {
92      val sourceVal = input.dependStatic(inputSource)
93      input.withValue(sourceVal + " :D")
94    }
95
96    def interpret(v: Value): String = v
97  }
98
99  val customDerived: Interp[String, S] =
100   Ticket.fromScheduler(scheduler)
101     .create(
102       Set(customSource),
103       "This is the initial value",
104       inite = false
105     ) { createdState =>
106       new CustomDerivedString(createdState, customSource)
107     }
```

Figure 6.4: Implementing a new type of operator for a derived reactive.

API to create new state is provided by another type of `Ticket` (Line 100), but the process is essentially handled by the scheduler. To create the state for the new reactive, the scheduler is informed of the input reactive (Line 102), the initial value (Line 103), and a boolean stating if the operator requires initial evaluation of the reactive (`inite` in Line 104). The scheduler will then provide a new `createdState` (Line 105), which contains the initial value, as well as scheduler specific information, and that state is passed to the operator to create a new reactive. Note that the `create` method discussed in the next section from the schedulers side has a slightly different API, because the ticket used by operators rearranges the parameters (adding the ticket itself) when forwarding the call to the scheduler.

The type of the returned reactives is `Interp` (Line 99). We have already seen `Interp` – it is the type used by the reevaluation ticket to access the value of a reactive. For example, the internal type of `Signal[Int]` is `ReSource[Pulse[Int]]` (the pulse

```
108 class CustomSource[T](initState: S#State[T, S]) extends
        ReSource[S] with Interp[T, S] {
109   outer =>
110
111   type Value = T
112   def state: State               = initState
113   def name: ReName               = "I am a source name"
114   def interpret(v: Value): T     = v
115   def commit(base: Value): Value = base
116
117   def makeChange(newValue: T) =
118     new InitialChange[S] {
119       val source = outer
120       def writeValue(base         : outer.Value,
121                      writeCallback: outer.Value => Unit
122                     ): Boolean = {
123       if (base != newValue) {
124         writeCallback(newValue)
125         true
126       } else false
127     }
128   }
129 }
130
131 val customSource: CustomSource[String] =
132   Ticket.fromScheduler(scheduler)
133     .createSource("Hi!") { createdState =>
134       new CustomSource[String](createdState)
135     }
136
137 transaction(customSource) {
        _.recordChange(customSource.makeChange("Hello!")) }
```

Figure 6.5: Implementing a new type of operator for a source reactive.

wrapper is used for internal bookkeeping of signal and event operators), and the `Interp[Int]` encapsulates the conversion from `Pulse[Int]` to just `Int`. Usually a returned reactive would have a more interesting type than `Interp`, such as `Signal` or `Event` – providing some methods to derive new reactives – but in this example we stick to the minimum.

Custom Derived Reactives

Creating custom sources is shown in Figure 6.5, and is similar to operators for derived reactives, with the exception that sources have no `reevaluate` method. Instead, the `makeChange` method (Line 117) provides an interface to convert an external value into an `InitialChange` object (Line 118). A transaction (Line 137) records one or more of these initial changes before the transaction starts and then applies them simultaneously to their respective sources.

Through the use of Scala's path dependent types, the signature of the `writeValue` method (Line 120) in the `InitialChange` object must be an inner class of the source operator to actually provide a new value to the source, because the concrete type of the `Value` type member is unknown outside the `CustomSource` implementation. Thus, we use the type system here to ensure that the operator controls what value is applied to a reactive even though the scheduler controls when the value is changed.

The `writeValue` method for our example checks that a new value is not identical to the current value (Line 123), if so it executes the `writeCallback` (Line 124) which instructs the scheduler to actually store the value. The return value of `writeValue` (Line 125) decides whether the changed source activates (i.e., propagates its value) during the transaction. All operators of REScala currently use this same logic for activation, where a value is propagated exactly if it is not equal to the current value.

The creation of sources is analogous to the creation of derived reactives using a ticket to ask the scheduler to create a new state for the source (Line 133). Setting a source uses the `makeChange` method inside a transaction (Line 137).

6.5 The Schedulers

Schedulers mostly use the core interface and the implementation provided by the schedulers. The role of the schedulers consists of defining the structure of the state of reactives, ensuring correct creation and initialization of reactives, and managing the transaction life cycle and propagation. REScala supports multiple schedulers to enable execution specific to the runtime environment, to provide different tradeoffs to applications, and to facilitate research. REScala has the following schedulers.

Level Based The level-based scheduler assigns an integer level to each reactive. Sources have a level of 0 and derived reactives have a level one larger than the maximum level of their inputs. Propagation happens by adding reactives that need evaluation into a priority queue. This scheduler uses a two version strategy for state where each reactive has a current and updated value to enable transactions to abort on error. This scheduler is the default for JavaScript-based execution environments.

ParRP ParRP [152] extends the level-based scheduler with a two phase locking scheme before transactions to allow multiple independent transactions to execute in parallel. ParRP is described in detail in Section 6.6. ParRP does not compile to JavaScript because of the lack of concurrency primitives in

JavaScript. ParRP is the default scheduler for the JVM, because it enables parallel transactions in independent modules to remain independent. However, ParRP has about twice the overhead to create new transactions compared to the level-based scheduler.

FullMV The FullMV [70] scheduler uses multi-version reactives to support pipelining of transactions. This enables parallelization on the granularity of individual reactives, but comes at a very significant overhead for individual transactions. Thus, FullMV is suited for applications with heavy computation within the flow graph. FullMV support neither JavaScript nor aborting transactions on errors.

Distributed FullMV The distributed FullMV [68] implementation uses a distributed serialization graph to provide a total order on transactions in a distributed system. Distributed FullMV thus provides distributed transactions, however, at the usual cost that the system may become unavailable when the network is unreliable. Note that replication in Cvtr uses transactions, but is an independent concept. Thus, it is possible to use distributed transactions from FullMV and combine them with replicated reactives.

Research Schedulers The F scheduler implements the algorithm used in $\mathscr{F}_r$ (see Chapter 5). The Simple scheduler is a minimal implementation of a scheduler in REScala used for educational and experimental purposes. The Test scheduler enables property-based testing of REScala applications. The Debugging scheduler integrates with the debugger (see Chapter 9).

While all schedulers have their differences, they all have to implement a minimal set of functionality. In the remainder of the section, we discuss how schedulers create new reactives, how transactions change the flow graph, and how schedulers handle the reevaluation of a reactive.

Initialization

To create a new reactive the scheduler must define the type, provide new instances, and initialize the state of a reactive.

We have discussed in Section 6.4 how the state of reactives is defined by the structure type parameter. Figure 6.6 shows an example definition structure of such a structure taken from the F scheduler. The `FStruct` (Line 138) is the type used for the `S` type parameter we have seen in other examples. Structure traits in REScala define the state type as a function of the value type (Line 139). See the discussion of the core package for the `Struct` parameter of the `State` type function. In this example, the state type is a `StoreValue[V, S]`, where `V` is the value type of the reactive. To mutable parts of `StoreValue` (Line 142) are the current `value: V`, and a set of inputs `ReSource[S]`.

The `create` method of the current scheduler generates a state implementation for new instances of reactives. In the current implementation, all schedulers share a single implementation of the create method shown in Figure 6.7. The implemented protocol is simple, first the internal state for the reactive is created (Line 151) –

```
138 trait FStruct extends Struct {
139   override type State[V, S <: Struct] = StoreValue[V, S]
140 }
141
142 class StoreValue[V, S <: Struct](var value: V) {
143   var inputs: Set[ReSource[S]] = Set.empty
144 }
```

Figure 6.6: Example of the structure for reactives corresponding to the calculus in Chapter 5.

```
146 def create[V, T <: Derived[S]](incoming: Set[ReSource[S]],
147                                 initv: V,
148                                 inite: Boolean,
149                                 creationTicket: CreationTicket[S])
150                                (instantiateReactive: S#State[V, S]
                                      => T): T = {
151   val state = makeDerivedStructState[V](initv)
152   val reactive = instantiateReactive(state)
153   ignite(reactive, incoming, inite)
154   reactive
155 }
```

Figure 6.7: The schedulers part of creating a new reactive.

the `makeDerivedStructState` just returns a new instance of the `State` type seen in Figure 6.6. Then, the state is passed to a callback provided by the operator implementation (Line 152) – we have seen an implementation of this callback when discussing custom operators (Figure 6.4 Line 105). The complete reactive returned by the callback then may be `ignited` (Line 153), which does a reevaluation of the new reactive. For example, all normal signals are initialized with an empty value and acquire their current value by executing an initial reevaluation executed as part of ignition.

Note that transactions in REScala may mix creation of reactives with propagation of changes – creation of reactives may even happen during the reevaluation of other reactives. To provide transactional guarantees the schedulers must ensure that created reactives are fully initialized before they are used. The schedulers in REScala generally do so, with a technical limitation that creation of reactives should not be performed by reactives with dynamic inputs. However, our case studies show no clear evidence that mixing creation of reactives with propagation in the same transaction is necessary, but it remains future work to show if the expressiveness would be reduced.

```
156 def forceNewTransaction[R](initialWrites: Set[ReSource[S]],
157                           admissionPhase: AdmissionTicket[S] =>
                                  R): R = {
158   val tx = makeTransaction(_currentInitializer.value)
159
160   val result =
161     try {
162       tx.preparationPhase(initialWrites)
163       val result = withDynamicInitializer(tx) {
164         val admissionTicket =
                tx.makeAdmissionPhaseTicket(initialWrites)
165         val admissionResult = admissionPhase(admissionTicket)
166         tx.initializationPhase(admissionTicket.initialChanges)
167         tx.propagationPhase()
168         if (admissionTicket.wrapUp != null)
              admissionTicket.wrapUp(tx.accessTicket())
169         admissionResult
170       }
171       tx.commitPhase()
172       result
173     } catch {
174       case e: Throwable =>
175         tx.rollbackPhase()
176         throw e
177     } finally {
178       tx.releasePhase()
179     }
180   tx.observerPhase()
181   result
182 }
```

Figure 6.8: Phases of a transaction in REScala's default scheduler.

Transactions

The main work of a scheduler is to manage the life cycle of transactions. Figure 6.8 shows the implementation of a transaction in the default scheduler in REScala. The `forceNewTransaction` method (Line 156) is the internal implementation of the `transaction` method used in Figure 6.5 (the signature is different because of an indirection in the API). The scheduler creates a new transaction (Line 158), which contains metadata such as the reactives that still require reevaluation. Each transaction of the default scheduler executes multiple phases in order.

Preparation phase (Line 162) This phase makes preparations for thread safety, such as locking the set of reactives that will be changed. The set of initial writes is already known, and other affected reactives are computed based on the graph structure. No changes are visible yet.

Admission phase (Line 165) Executes the transaction initialization code provided by the developer that defines the changes to source reactives (c.f., Figure 6.5). The admission ticket allows to read the value of reactives included in the `initialWrites` or their successors to make decisions on which changes to apply.

Initialization phase (Line 166) Writes the changes recorded in the admission phase into the corresponding source reactives and prepares the propagation.

Propagation phase (Line 167) Executes the reevaluation method of reactives whose inputs are propagating new values. This phase is central in ensuring that execution happens in topological order, to ensure synchronous semantics. The default scheduler of REScala assigns levels to each reactive and uses a priority queue to decide which reactive to execute next [181]. We will look at the details of reevaluation in the next paragraph.

Wrap up phase (Line 168) Executes transaction specific handlers before committing.

Commit phase (Line 171) Does cleanups on internal metadata of the transactions and makes new values permanent, although, not yet accessible to other transactions.

Rollback phase (Line 175) This phase does cleanup if an exception occurs during propagation. Note that not all schedulers support rollbacks, because it is not required for the core of the Cvtr paradigm. See Chapter 7 for a discussion of error handling using rollbacks.

Release phase (Line 178) Frees resources and locks held by the transaction. After this phase, all changes are visible to other transactions.

Observer phase (Line 180) Runs the registered observers. Observers read the values of reactives during the transaction, but the transaction ends before observers are executed. This enables observers to start new transactions (REScala does not support nested transactions). However, ending transactions makes a race condition possible, where the observers for two transactions are not executed in serialization order. We only keep the current behavior for backwards compatibility with our case studies.

```
183 def evaluate(
184     reactive: Derived[FStruct],
185     dynamicOk: ReSource[FStruct] => Boolean,
186     creationTicket: SimpleCreation
187 ): (Boolean, Boolean) = {
188   val dt = new ReevTicket[reactive.Value, FStruct](
189                 creationTicket, reactive.state.value) {...}
190   val reev = reactive.reevaluate(dt)
191
192   def finishReevaluation() = {
193     reev.forValue(reactive.state.value = _)
194     reev.forEffect(_.execute())
195     (true, reev.activate)
196   }
197
198   reev.inputs() match {
199     case None => // static reactive
200       finishReevaluation()
201     case Some(inputs) => //dynamic reactive
202       reactive.state.inputs = inputs
203       if (dynamicOk(reactive)) finishReevaluation()
204       else (false, reev.activate)
205
206   }
207 }
```

Figure 6.9: Reevaluation handling in the scheduler for `FStruct`.

Reevaluation

For the reevaluation we show the code from the F scheduler that imitates $\mathscr{F}_r$, because the implementation of the default scheduler is too concerned with performance to serve well for educational purposes. We have already seen the structure (`FStruct`) of reactives for that scheduler in Figure 6.6 – that is, the structure that has a mutable value and a mutable set of inputs. The reevaluation logic for reactives with the `FStruct` is shown in Figure 6.9. We have already shown how this interaction looks like from the side of the operator in Figure 6.4. The scheduler for the `FStruct` uses a simplified variant of the phases discussed in the previous paragraph, where observers are executed as soon as their value is known, and rollback is impossible. The implementation for the propagation phase ensures that each reactive is reevaluated in topological order. To evaluate a single reactive, the scheduler creates a reevaluation ticket (Line 188) and passes that to the reevaluation function implemented by the operator (Line 190). The result from the reevaluation contains a list

of new inputs for dynamic reactives, a new value, and side effects to be executed. The scheduler is responsible for writing the returned results to update the structure of the reactive (Line 192). In the case when new inputs are detected, it is possible that the evaluation did not happen in topological order (Line 201). This scheduler thus does not update reactive and instead continues the propagation. The skipped reactive is later reevaluated again after the new inputs have been reevaluated.

6.6 ParRP

ParRP enables concurrent transactions to execute in parallel, while preventing conflicts between overlapping transactions. In complex applications with many modules, the sets of reactives affected by two transactions may be in independent modules. Thus, the transactions should execute in parallel, to increase throughput and prevent independent modules from having to wait for each other. Since the original publication in my master thesis [152], the algorithm behind ParRP has not changed, but the implementation was refined and a simplified explanation is provided in this section to serve as a demonstration for the power and flexibility of REScala's design.

ParRP provides serializability for transactions and executes independent transactions in parallel. To do so, ParRP computes the write set – all reactives that could change in a transaction – and acquires exclusive locks before writing any changes. Note that any transaction A executing in parallel can change the graph, thus changing the write set for a transaction B. In many systems, such interactions require rollbacks to ensure correct results. However, because ParRP operates on a directed acyclic graph we can show that it suffices to extend the transaction B (the one that had its write set changed) to also propagate values to (parts of) the reactives in the write set of transaction A by "inheriting" the write set from one transaction to another. ParRP detects such situations and transaction A transfers its locks to transactions B without releasing them, enabling B to continue without rollback or deadlock. For the remaining discussion in this thesis, it suffices to understand that reactives in ParRP must support being locked and must support having multiple values to enable handling complex interactions between parallel transactions.

As another point of modularization, the concurrency control of ParRP does not depend on the used propagation algorithm and preserves the semantic properties of the used algorithm (e.g., glitch freedom). In the current version, ParRP uses the priority queue [137] for propagation. After locking is done the propagation algorithm can run unmodified, except that ParRP must be notified when new dataflow edges are added.

Figure 6.10 shows the structure for ParRP reactives. The structure provides the same interface as the structure in Figure 6.6, but has much more internal state for bookkeeping. The `ParRPStruct` (Line 208) is used to encode the type level function, with `ParRPState` (Line 212) being the implementation of state of reactives. `ParRPState` also has a `current` (Line 213) value, but also an `updated` (Line 218) value, which belongs to a `owner` transaction identified by a `Token` (Line 217). Finally, the sets of `incoming` (Line 240) and `outgoing` (Line 241) edges in the flow graph.

```
208 trait ParRPStruct extends Struct {
209   type State[P, S <: Struct] = ParRPState[P, S]
210 }
211
212 class ParRPState[V, S <: Struct](
213     var current: V,
214     val lock: ReLock[ParRPInterTurn])
215   extends Committable[V] {
216
217   var owner: Token = null
218   var update: V    = _
219
220   def write(value: V, token: Token): Boolean = {
221     assert(owner == null || owner == token, s"buffer owned by
          $owner written by $token")
222     update = value
223     val res = owner == null
224     owner = token
225     res
226   }
227   def base(token: Token): V = current
228   def get(token: Token): V = { if (token eq owner) update else
        current }
229
230   def commit(r: V => V): Unit = {
231     current = r(update)
232     release()
233   }
234   def release(): Unit = {
235     update = null.asInstanceOf[V]
236     owner = null
237   }
238
239   /* incoming and outgoing changes (some helpers not shown) */
240   var incoming: Set[ReSource[S]]                  = Set.empty
241   var outgoing: mutable.Map[Derived[S], None.type] =
        mutable.HashMap.empty
242 }
```

Figure 6.10: The structure for reactives when using ParRP.

For graph structures which allow parallel propagation and a multi-core processor, we can show an increase in performance over the synchronous implementation. See Chapter 13 for detailed performance experiments. How large the increase is depends on the size of the changed graph and the frequency of conflicts. Our benchmarks suggest a 1.4x throughput increase utilizing 2 cores up to a 5x increase utilizing 8 cores for common graphs. For transactions without conflicts, ParRP scales linearly with the number of CPU cores, but the added cost of allocating data structures for conflict management only allows to realize the full speedup in cases where each transaction executes expensive computation that profits from parallelization. However, to support concurrency ParRP has a setup cost for each transaction: our benchmarks indicate at least 700 CPU cycles (in addition to 1170 CPU cycles of the level-based scheduler) and increasing with the size of the transaction.

In our case studies (Chapter 11), we observe that a common pattern for dicApps is to have a number of mostly disjunct subgraphs, which are joined into a single dependent reactive (e.g., a user interface consisting of multiple parts rendered to the same window). Because of two phase locking this single dependent has to be locked to start propagation on any of the subgraphs, so we cannot parallelize such cases under our current approach. A possible solution is to use pipelining when propagating multiple updates. With pipelining instead of a transaction waiting on the completion of the prior update the waiting becomes more fine-grained and waiting happens for individual reactives. These optimizations are again completely transparent to the programmer showing the strength of the Cvtr model.

6.7 CRDTs in REScala

While Cvtr benefits from the synergies between state-based CRDTs and transactional reactive programming the implementation of both are independent with the network runtime making use of both. In REScala any signal with a value for which a lattice type class exists can be replicated. We will look at that type class, how CRDTs are used by developers, and at the network runtime. However, note that CRDTs in REScala are still ongoing research, so the details are likely to change.

Lattices

The lattice interface is shown in Figure 6.11. For any type `A` a `Signal[A]` can be replicated if there is a corresponding instance of `Lattice[A]`. All that is needed for the instance is a merge function which must be associative, commutative, and idempotent. For example, Figure 6.11 shows the implementations of the lattice instance for `Set[A]`. The union operation of sets has the correct properties to be used as a merge function, thus the instance only forwards the call. REScala provides instances for sets, maps, and options in addition to custom CRDT implementations such as counters, sequences, and observe-remove sets. External libraries that provide state-based CRDTs may be used as well by providing a lattice instance that maps to the external libraries merge implementation.

```
244 trait Lattice[A] {
245   /** By assumption: associative, commutative, idempotent. */
246   def merge(left: A, right: A): A
247 }
248
249 // Lattice instance for Set[A]
250 implicit def setInstance[A]: Lattice[Set[A]] =
251   new Lattice[Set[A]] {
252     override def merge(left: Set[A], right: Set[A]): Set[A] =
          left.union(right)
253   }
```

Figure 6.11: Lattice API used by REScala and its network runtime, and the implementation of Lattice for Scala Set.

CRDT API

From the developers point of view, a replicated signal can be constructed like any other signal. For example, a fold reactive that uses a set for its state may use any built-in operation on sets to compute its local state. REScala does not interfere and directly provides developers access to the API of the used data type. This provides developers with the flexibility to choose a CRDT implementation that suites the desired features and performance requirements of their application. It is also possible for developers to only learn how to use the CRDTs they require, similar to how collection libraries can be learned and understood in small pieces. The ability to mix and match implementations, experiments with new APIs, and even develop new CRDTs for particular implementations is core to the flexibility of the Cvtr programming paradigm.

The only caveat is that the replication semantics is always specified by the merge function. For example, the merge function for sets in Figure 6.11 will always keep all elements of both sets. So if elements are removed locally from a replica, those elements will be added again during the next synchronization. To support removal of elements, a CRDTs implementation that supports a merge function with removal must be used. It is crucial for developers to understand the semantics of the merge function of their chosen data type. However, merging is the *only* additional concept required to write distributed programs in REScala. This simplifies distributed programming, because once an application works correctly with the given restrictions, it works correctly in all possible failure conditions.

Network Runtime

The network runtime in REScala uses ScalaLoci [202] to abstract communication over channels such as TCP and Websockets. The network runtime makes use of lattices and the transactional semantics of schedulers. Figure 6.12 shows the signature

```
254 def replicate[A: Lattice, S <: Struct: Scheduler](
255     signal: Signal[A, S],
256     registry: Registry)
257    (binding: Binding[A => Unit]{
258       type RemoteCall = A => Future[Unit] })
259    : Unit
```

Figure 6.12: Method of the network runtime to replicate signals.

of the replicate method of the network runtime. This method instructs the network to also replicate the `signal` in the network defined by the `registry`. Registries are a ScalaLoci abstraction to group a set of connections to other devices and bindings of remote values. The `binding` is a type-safe identifier to detect that two reactives on two different devices are replicas of each other. It also provides the serialization strategy of the values of the reactive. The replicate method interacts with the scheduler to send the state of replicated reactives to other devices connected to the registry after each transaction. The network runtime also starts a new transaction whenever an update from the network is received. Values are serialized using common Scala libraries such as Circe and Jsoniter, which support type-safe serialization for most built-in immutable Scala data types. Custom serializers can be provided using type classes. The serializer for signals is special and causes the signal to be published as described in Section 3.3.

6.8 Snapshots

REScala's snapshot and restoration mechanism supports storing snapshots in arbitrary key-value stores. We have two implementations for in-memory stores: one that writes directly to disk (for JVM and Android) and another one that uses HTML5 localStorage [4] for web browsers.

Using fault-tolerant reactives is thread-safe, but care must be taken when reactives are created in a multi-threaded environment. Snapshots in REScala require unique IDs to identify reactives, and our current implementation uses thread-local counters to generate IDs, e.g., the IDs `UI-0` and `UI-1` are associated to the first and to the seconds reactive created in the context of the UI thread. This ID generation strategy is well suited for applications with a fixed set of threads, but it cannot cope with the case in which threads are created and used dynamically in a thread pool, because the IDs generated for those threads do not remain the same between restarts of the application. REScala requires the developer to explicitly handle the assignment of IDs to reactives if the automatic mechanism is insufficient. In practice, it is in most cases sufficient to ensure that dynamically scheduled tasks (e.g., those in thread pools) are assigned deterministic names to enable correct automatic generation of unique IDs.

Key Value Stores for Snapshots

Key-value stores are a good fit as the underlying storage for our snapshots, where the unique name of the reactive is the key, and the corresponding values of all changed reactives are stored to create a new snapshot. Many key-value stores are available in a wide range of application domains. We can take advantage of some storage features, if available. If the storage provides atomic all-or-nothing writes for a set of keys, we can store the set of changed reactives more efficiently and do not need to synchronize after an update. Fault-tolerant stores would make the whole restoration process robust against storage failures.

The consistency guarantees provided by the update propagation of the reactive runtime to ensure glitch freedom can be used to relax the requirements on the store. First, the reactive runtime ensures that there is no concurrent access to a single reactive, so the store does not have to synchronize writes. Second, reads of snapshot values only happen on recovery, so the store does not need to synchronize reads and writes of the same value. This also allows us to work with eventually consistent stores, as no more writes happen after a crash and recovery can wait until the store is consistent.

6.9 Conclusion

REScala shows that the declarative nature of the programming paradigm is also reflected in the implementation. The definition of operators is strictly separate from the semantics of transaction and the propagation of values through the flow graph. This enables not only to have multiple implementations of each for different use cases and combining them as needed, but also makes research on advanced features such as concurrency control, parallelization, and distribution much easier. We highly recommend REScala as a platform to experiment with the Cvtr programming paradigm and other research on languages for dicApps.

Part II

Extensions, Experience, Evaluation

Interlude

Thus far, we have introduced the core of the Cvtr programming paradigm and the implementation of REScala. We have given an intuition how the programming paradigm is used to implement applications. We have proven that applications exclusively using this paradigm tolerate partial failures of distributed architectures, provide causal consistency when connections are unreliable, and have a transactional behavior for user interactions matching developers and users expectations.

In the rest of this thesis we validate the wider applicability of the Cvtr programming paradigm beyond just the core language and the basic principles. We believe it is important that developers do not have to give up all of their existing tools, knowledge, and applications to benefit from Cvtr. All modern general-purpose programming languages support multiple programming paradigms and Cvtr intends to expand those choices not replace them. Thus, in this part we explore the practical implications of using Cvtr and REScala. We provide several extensions of the core concepts provided so far to demonstrate that Cvtr integrates with a wide range of other programming paradigms and concepts, including exceptions, CRDTs and monotonic programs, embedded devices with limited resources, and debuggers and live programming. We present case studies implemented in REScala that show a range of different applications as evidence that the programming paradigm is expressive enough and the implementation efficient enough for actual use. Based on the case studies, we then share our experience of using REScala and discuss typical caveats we observed over the years when developing with Cvtr. Finally, we discuss targeted performance experiments that evaluate individual parts of the implementation and that demonstrate empirically what REScala is capable of.

Chapter 7

Errors and Exceptions

The use of convergence and eventual consistency in Cvtr means that developers never have to deal with explicit error handling, but there are still cases where they may want to. First, errors also occur in contexts – such as operating with files – that are not covered by Cvtr. Second, sometimes an application wants to behave differently when connectivity could not be established, for example, informing the user with a status icon or using an approximation of the remote value. Especially when interfacing with remote systems not implemented in REScala both of the above apply and developers have application specific strategies how to handle such cases.

We first give an intuition of how error propagation works in REScala, then discuss how error handling looks like by modifying the shared calendar example from Chapter 2, and finally compare the syntactic overhead of built-in error propagation with encoding errors as values. Before we conclude, we discuss a generalization of error propagation to multiple different channels (values and errors). Our case studies in Chapter 11 suggest that our API for error handling aligns very well with the native mechanism of Scala and our performance experiments in Chapter 13 show that error handling has no additional runtime cost.

7.1 Intuition of Error Propagation

Many imperative, object-oriented languages such as Java [92] or C++ [122] have an exception handling mechanism [44]. To integrate with the existing language mechanisms the built-in error handling strategy of REScala is based on exceptions.

Exceptions usually propagate up the call stack until the closest matching handler. The rationale is that code locations up in the stack have more information to handle the exception and automatically propagating exceptions prevents error handling code from being spread in all intermediate stack frames. However, in REScala it is not the caller of a function that expects a result, but rather the derived reactives. The notion of the call stack is not natural for the flow graph and when used would lead to the transaction semantics of Cvtr being violated by an exception stopping the transaction without evaluating all derived reactives. To solve this issue REScala

```
260 val selectedEntries = Signal {
261   entrySet.value.toSet.filter { entry =>
262     try selectedWeek.value == Week.of(entry.date.value)
263     catch { case DisconnectedSignal => false }
264   }
265 }
266
267 selectedEntries.observe(
268   onValue = Ui.displayEntryList,
269   onError = Ui.displayError)
```

Figure 7.1: Excerpt of the updated source code for the shared calendar application.

internally treats exceptions as values that are propagated in the flow graph similar to any other value. We refer to such values as errors to disambiguate from the built-in exception mechanism. To support error handling, only the operators that define the internal semantics for reactives have to be individually adapted to define the operators' semantics when an input has an error value. Most existing operators forward any error in their inputs. In addition, the API makes errors available in user-defined functions using the normal language features (of Scala) and by providing several operators specifically for error handling. Treating errors as other values means that our core calculus $\mathcal{F}_r$ does not have to change to support errors – most importantly all properties still apply.

7.2 Internal Representation of Errors

The error propagation mechanism is integrated into REScala. We have discussed the implementation of operators in Section 6.4. Each operator has an internal value type – which for events and signals is a `Pulse[T]`, where `T` is the type visible to developers. The `Pulse` trait distinguishes between `Value[T]` and `NoChange`. To support error propagation, we introduce a third alternative, `Exceptional`, and update the case distinctions for all operators. The updated implementation wraps any exception that is not caught in user-defined functions into the `Exceptional` type and propagates that in the same way as other values. However, accessing the value of a signal or event with an error causes the error to be thrown (usually to be handled by a user-defined function). Overall, because the propagation of errors reuses most of the work done for the propagation of normal values, our implementation strategy for errors induces no performance overhead when no faults are present, as discussed in Section 13.4.

7.3 A Shared Calendar with Errors

Figure 7.1 shows an excerpt of the shared calendar application with error handling. The signal expression for `selectedEntries` (Line 260) uses Scala's built-in filter function on sets to select only calendar entries of the current week. The date of each calendar entry is a signal, thus the signal expression accesses many nested signals (Line 262). The set of entries is a replicated reactive, thus the entries themselves – including the nested signals – are replicated on the network. For this example we assume that, depending on the network model and protocol, the transfer of a single entry may fail or be delayed. While this is normally not possible because of causal consistency in Cvtr if the entries are accessed from a different service – such as an existing calendar server – such failures and delays are common.

In the case where the nested entry signal is not yet replicated accessing the `entry.date.value` causes an error, which is thrown as an exception. We handle the error using the standard Scala exception handling mechanisms. In this case, we handle the error right where it occurs by excluding the entry from the filter, thus not displaying it to the user. However, not all exceptional circumstances can be handled right where they occur. To address this, signals and events in REScala propagate all errors along the flow graph. When errors are propagated they may be handled by any signal expression accessing an input propagating the exception or by the `.recover` method that replaces errors with normal values. All errors in the flow graph must be handled. The latest place to handle an error is as part of an observer (Line 262). Otherwise, when an exception is observed but not handled, then the transaction aborts and the exception is thrown to the location where the transaction was created.

7.4 Error-aware Reactives

The new error-aware semantics of reactives is a superset of their original semantics, thus existing code carries over unchanged. We extend the API of REScala to enable integrations with errors from external APIs and to allow developers to customize how errors are processed. In particular, we provide an integration that allows imperative code to push errors into the flow graph, we add new operators that allow to recover errors into values, we extend observers to handle errors differently, we enable user-defined functions to handle errors using the normal language features, and we extend fold reactives such that they may recover from errors that have become part of their state.

Injecting Errors Into the Flow Graph

We extend the API of `Evt` and `Var` with methods to fire errors. `Evt.admit(error)` and `Var.admit(error)` behaves similar to their value propagating counterparts, but start the propagation with an error instead of a value. The main use of this API is to support the integration of existing frameworks, e.g., converting an exception of

a networking library to an error in the flow graph. Consider an existing networking library with a callback-based API. When a timeout occurs in the network, the imperative library callback is converted into a reactive propagation:

```
270 val fromNetwork = Evt[NetworkMessages]()
271 Network.onTimeout { error => fromNetwork.admit(error) }
```

Observe and Recover

We extend the observer's API to accept an additional handler parameter called `onError`, which is used to observe propagated errors. This handler has the same purpose as `catch` blocks, and, similar to the standard `observe` call, has the goal of producing a side effect, e.g., displaying an error message. The error handler on observers can be missing: any unhandled error terminates the program in the same way as traditional uncaught exceptions. In the calendar example in Figure 2.4, any error is displayed to the user by the error handler defined on the signal in Line 267 using the extended observer API:

```
272 selectedEntries.observe(
273   onValue = Ui.displayEntryList,
274   onError = Ui.displayError)
```

Instead of simply observing errors, the application developer can recover from the error inside the flow graph using the `recover` operator for signals and events, which is parameterized with a recovery function that converts an error to a normal value.

```
275 val input: Signal[String] = ...
276 val recovered: Signal[String] = input.recover{ error =>
      error.toString }
```

In the example, the `input` signal may contain an error instead of a string, but the `recovered` signal converts all errors into string values. The value is then propagated as the output of the `recover` operator. Any normal value that flows through the `recover` operator in the flow graph is propagated without change. The `recover` operator handles errors while they are propagated through the flow graph and before they reach an observer. Recovering from an error is most useful, when errors can be locally converted back into normal values. This case is relevant in several applications. For example, values can have local fallbacks, such as an unavailable location service that can be replaced by using more expensive or inaccurate local data. Another example is a signal holding a UI widget, where an error can be handled by displaying it to the user.

User-defined Computations

A user-defined computation such as those in a signal expression may access any number of dependencies. When any of the dependencies propagates an error, the error is raised as a Scala exception by the `.value` call performing the access. The

exception may be handled by the application using the default Scala exception handling mechanisms. Unhandled exceptions in a user-defined computation are propagated along the flow graph.

The use of Scala exceptions enables our error handling scheme to integrate well with most libraries in the JVM. For example, the shared calendar in Figure 2.4 filters the list of all calendar entries to only include entries of the current week in Line 260. All entries containing an error are removed using a Scala `try/catch` block:

```
277 val selectedEntries = Signal {
278   entrySet.value.filter { entry =>
279     try selectedWeek.value == Week.of(entry.date.value)
280     catch { case e: NetworkError => false }
281   }
282 }
```

When the `entry.date` (Line 279) contains an error, the error is thrown as a Scala exception and handled in the `catch` by returning false (Line 280), causing the filter to drop the entry (Line 278).

Folds

Recall that the `fold` operation supports converting events into signals. Given an event `e`, an initial value `init` and a function `f`, which are passed to it as parameters, `fold` returns a signal that is initialized with `init` und gets updated every time `e` fires by applying `f` to the current value of the signal. Thus, unlike other derived reactives, a `fold` signal accesses its own current value, i.e., the fold (indirectly) depends on the complete history of the input event. For example, the signal `allEntries` collects all new entries into a set.

```
283 allEntries.fold(Set.empty) { (entries, entry) => entries + entry}
```

The current state of the fold (`entries`) is treated like any ordinary dependency. If it is accessed and it holds an error, the error is thrown as a Scala exception. If the exception is not handled inside the user-defined computation (i.e., the function body of `fold`), then an error is propagated by the `fold` reactive to other reactives that depend on it. Alternatively, by handling the exception a developer can resume the computation of the `fold` reactive after an error. We present an example for fold with custom error handling next.

We illustrate custom error handling using fold with the implementation of the `count` operator.

```
284 def count( ) = fold(0) { (counter, occurrence) =>
285   occurrence // access the (unused) value to propagate errors
286   try counter + 1 // increase count in non-error case
287   // continue counting after errors
288   catch { case (value, error) => value + 1 }
289 }
```

The operator counts the number of non-error event occurrences. The `count` signal starts with its initial state initialized to zero (Line 284). The folding function takes the current `counter` of the fold and the incoming event, called `occurrence`, as parameters. When `occurrence` is accessed in Line 285, there are two possibilities: the access raises an error or returns a normal value.

In the case of an error an exception is thrown that aborts the execution of the user-defined computation. The exception is caught by the fold operator internally, stored for future processing, and propagated in the flow graph.

If the access of `occurrence` returns a normal value processing continues (the value is ignored, because we are only interested in the number of occurrences and not in their values). Next, `counter` is accessed in Line 286. If the current `counter` is a normal value, it is incremented, and the increased count is returned (Line 286).

If any prior evaluation of the fold results in an error, then accessing the `counter` throws that error as an exception. The exception is immediately caught by the user-defined function in Line 288. The pattern matching in the `catch` block binds the last non-error value stored in the fold and the current error. Our example handler ignores the error and continues by incrementing the last non-error state, thus implementing a counter that resumes counting when a new occurrence arrives after an error.

7.5 Errors and Aborting Transactions

The default schedulers in REScala (see Section 6.5) support aborting transactions. A transaction is aborted when errors are observed but not handled or when the developer explicitly uses the abort operator. In both cases, the encountered error is then thrown as an exception in the context of the call that created the transaction. The exception thrown by aborted transactions includes the information about the where the error originated and which observers did not handle the error. This information allows to reconstruct the path of the error through the flow graph.

The abort operator is useful for flow graphs, which are unable to handle errors or were designed without error handling in mind. The abort operator mirrors any value of the input reactive, but if the input reactive propagates an error then the transaction is guaranteed to abort. Thus, the `derived` signal in the following example can safely be used by code that is not aware of errors.

```
290 val input: Signal[A] = ...
291 val derived: Signal[A] = input.abortOnError()
```

In addition to explicitly using the abort operator we also abort transactions when an error reaches an observer that has no error handler. This aborting by default behavior enables existing code that does not specify error handlers to be used without changes. The code that does not deal with errors is still semantically broken if an error occurs, but aborting the transaction and providing error output improves the development experience compared to simply ignoring the existence of exceptions in the host language.

```
292 val changeTrack: Evt[Track] = Evt[Track]()
293 val currentTrack: Signal[Track] = changeTrack.latest()
294 val currentScreen = composeInfoScreen(currentTrack)
295
296 currentTrack.observe(playMusic)
297 currentScreen.observe(displayScreen)
298
299 def composeInfoScreen(track: Signal[Track]): Signal[Screen] = {
300   val metadata = track.map(loadMetadata)
301   val infoScreen = Signal {
302     renderInfoScreen(track.value, metadata.value)
303   }
304   infoScreen
305 }
306
307 def loadMetadata(t: Track): Data = {
308   ... fetch data, assume no faults occur ...
309 }
310
311 def renderInfoScreen(t: Track, md: Data): Screen
312 def playMusic(t: Track): Music
313 def displayScreen(s: Screen): Unit
```

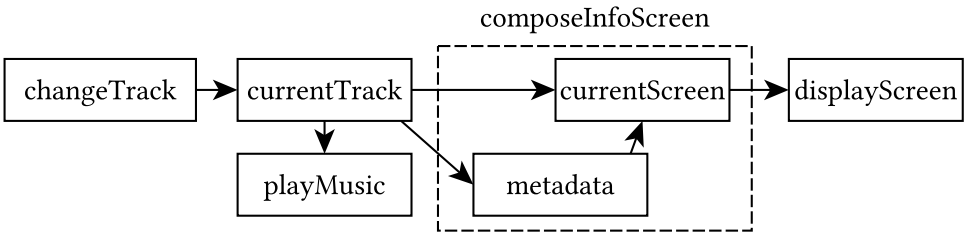

Figure 7.2: Music application without fault handling.

```
314 val changeTrack: Evt[Track] = Evt[Track]()
315 val currentTrack: Signal[Track] = changeTrack.latest()
316 val currentScreen = composeInfoScreen(currentTrack)
317
318 currentTrack.observe(playMusic)
319 try { currentScreen.observe(displayScreen) }
320 catch {
321   case SomeException => displayScreen(errorScreen)
322 }
323
324 def composeInfoScreen(track: Signal[Track]): Signal[Screen] = {
325   val metadata = track.map(loadMetadata)
326   val infoScreen = Signal {
327     renderInfoScreen(track.value, metadata.value)
328   }
329   infoScreen
330 }
331
332 def loadMetadata(t: Track): Data = {
333   ... throw SomeException ...
334 }
335
336 def renderInfoScreen(t: Track, md: Data): Screen
337 def playMusic(t: Track): Music
338 def displayScreen(s: Screen): Unit
```

Figure 7.3: Incorrect example: Exception not thrown in scope of handler.

7.6 Alternatives for Error Handling in Dataflow Languages

To discuss alternatives for error handling, we use the code example shown in Figure 7.2. The example represents the concept of a music player application that shows information about the currently playing track. The figure includes a visualization of the flow graph and the focus of our analysis will be the `composeInfoScreen` function, which creates two reactives. Important for error handling is that the metadata about the track is fetched from some resource that could potentially fail (Line 300). The code currently assumes that no failure occurs. The metadata is then passed through the application (Line 301) until it is finally observed (Line 297). There are different strategies to handle this failure which we discuss next.

Naive Exception Handling

In Figure 7.3 (an adapted version of Figure 7.2) we assume that `loadMetadata` fails and throws some exception as suggested in Line 333. In that case, we want to

```
339 val changeTrack: Evt[Track] = Evt[Track]()
340 val currentTrack: Signal[Track] = changeTrack.latest()
341 val currentScreen = composeInfoScreen(currentTrack)
342
343 currentTrack.observe(playMusic)
344 currentScreen.observe(displayScreen)
345
346 try {
347   changeTrack.fire(beethoven)
348 } catch {
349   case SomeException =>
350     // the code here should not have to know
351     // about loadMetadata
352 }
353
354 def composeInfoScreen(track: Signal[Track]): Signal[Screen] = {
355   val metadata = try {
356       track.map(loadMetadata)
357     } catch {
358       case SomeException =>
359         // cannot handle here
360     }
361   val infoScreen = Signal {
362     renderInfoScreen(track.value, metadata.value)
363   }
364   infoScreen
365 }
366
367 def loadMetadata(t: Track): Data = {
368   ... throw SomeException ...
369 }
370
371 def renderInfoScreen(t: Track, md: Data): Screen
372 def playMusic(t: Track): Music
373 def displayScreen(s: Screen): Unit
```

Figure 7.4: Handlers not at the correct position to handle error

display an error message instead of displaying the status information, but if we try to catch the error as suggested in Line 321 it will not work as desired. The exception is propagated up the stack, but the throw statement is not executed in the scope of the handler.

Figure 7.4 shows the two scopes where the `loadMetadata` function is executed thus these are the scopes where exception could be caught. First, when the metadata signal is created in Line 356 and, second, every time the track is changed by an imperative call to `changeTrack.fire` in Line 347. None of the two call sites, however, allow to notify the user about the error. The first call site when the reactive is created is concerned with the flow graph and not yet able to reason about values in the graph that will only be used later. The second call site when the source is changed is able to reason about which operation has failed (changing tracks) but it is not known why the failure happened or how to handle it.

Instead, the error should be handled as part of the info screen. The screen should show the user of the application why the metadata of the track could not be loaded. This is only possible by propagating the error in the flow graph.

Monadic Approach: Encoding Errors as Values

Functional languages often provide mechanisms to encode errors as part of the return value [200]. Developers can use this approach with REScala without changes. In the example, we can implement this behavior by encoding the error as part of the return value of `loadMetadata`. The result is wrapped in a `Try[Screen]` in Line 396 of Figure 7.5, the `Try` is a value that can either represent the correct result, or contain an error. The encoding in Figure 7.5 provides the desired semantics, as shown in Line 379, but requires all parts of the program which depend on the track data to be rewritten to propagate or handle the error, as in Line 387. Mixing error handling code into the normal control flow hides the main logic and makes the code harder to understand [44], and – especially when definitions of reactives are concise – unwrapping values twice is impractical. For example, compare the original expression and the one with manual error handling (using typical syntactic sugar):

```
404 Signal {
405   renderInfoScreen(track.value, metadata.value)
406 }
407
408 Signal {
409   metadata.value.map {unwrappedData =>
410     renderInfoScreen(track.value, unwrappedData)
411   }
412 }
```

Errors in the Flow Graph

The example from Figure 7.2 implemented with our solution is shown in Figure 7.6. The fault in the `loadMetadata` function triggers an exception in Line 431. In our

```
374 val changeTrack: Evt[Track] = Evt[Track]()
375 val currentTrack: Signal[Track] = changeTrack.latest()
376 val currentScreen = composeInfoScreen(currentTrack)
377
378 currentTrack.observe(playMusic)
379 currentScreen.observe {
380     case Success(screen) => displayScreen(screen)
381     case Failure(e) => displayScreen(makeErrorScreen(e))
382   }
383
384 def composeInfoScreen(track: Signal[Track]): Signal[Try[Screen]] =
        {
385   val metadata = track.map(loadMetadata)
386   val infoScreen = Signal {
387     metadata.value match {
388       case Successful(unwrappedData) =>
389         renderInfoScreen(track.value, unwrappedData)
390       case Failure(e) => Failure(e)
391     }
392   }
393   infoScreen
394 }
395
396 def loadMetadata(t: Track): Try[Data] =
397   Try {
398     ... throw SomeException ...
399   }
400
401 def renderInfoScreen(t: Track, md: Data): Screen
402 def playMusic(t: Track): Music
403 def displayScreen(s: Screen): Unit
```

Figure 7.5: Errors as values.

```
414 val changeTrack: Evt[Track] = Evt[Track]()
415 val currentTrack: Signal[Track] = changeTrack.latest()
416 val currentScreen = composeInfoScreen(currentTrack)
417
418 currentTrack.observe(playMusic)
419 currentScreen.recover(makeErrorPage)}
420   .observe(displayScreen)
421
422 def composeInfoScreen(track: Signal[Track]): Signal[Screen] = {
423   val metadata = track.map(loadMetadata)
424   val infoScreen = Signal {
425     renderInfoScreen(track.value, metadata.value)
426   }
427   infoScreen
428 }
429
430 def loadMetadata(t: Track): Data = {
431   ... throw SomeException ...
432 }
433
434 def renderInfoScreen(t: Track, md: Data): Screen
435 def playMusic(t: Track): Music
436 def displayScreen(s: Screen): Unit
```

Figure 7.6: Example reactive program with error handling.

example, the error could be handled as soon as at the `metadata` signal in Line 423, but it is not. When an error is not explicitly handled by a reactive, it is propagated further to the next reactive in the graph and can then be handled at any point, at the latest at an observer before it gets exposed outside the flow graph. In Figure 7.6, the error is handled at the `currentScreen` signal, before the value of the signal is observed by the `makeErrorPage` function (Line 419).

7.7 Multichannel Propagation

Errors propagation can be seen as a second channel of information using the normal flow graph. We can extend the concept of additional propagation channels to include more values modeling typical challenges in distributed systems, such as missing or late values.

Missing values occur when a part of the distributed application is started, but not yet connected and there is no useful initial value for created signals. Events have no initial state, so missing values do not apply to events. Missing values are

similar to exceptional values, i.e., a signal which depends on an empty signal is itself empty. Missing values are otherwise ignored by observers and conversions to events; only directly reading the value of an empty signal with an imperative call will result in that call to fail with a normal Scala `NoSuchElementException`.

Late values frequently happen in systems that make a dynamic trade-off between reliability and performance by using cached values. For example, to get correct results from a replicated database it is a common strategy to query multiple replicas. The value returned by the majority of the replicas is considered the correct result. However, to get the correct result the application needs to wait for all replicas, which can result in latency problems if one of the replicas is slow. As Guerraoui et al. [96] observe, it is often useful to already start speculative execution from the first result, as that result is likely to be correct, and we do not have to wait for slow replicas. Speculative execution can be handled transparently. When a not yet confirmed value arrives it is propagated similarly to a normal value, but new changes remain hidden to the external program. As soon as the value is confirmed it will be compared to the speculative value, if they are equal, any evaluations of signals can be skipped and only observers must be executed to finish the propagation. If the fixed value differs from the speculative execution, then a new update is started as usual, and the speculative results are dropped.

We believe that the full potential for integrating external concepts into the flow graph has not yet been realized. The inclusion of missing and speculative values already provides more special cases in addition to error propagation, but it remains future research to fully understand how these concepts generalize.

7.8 Conclusion

Displaying an exception to a user is often a good strategy for temporary or spurious errors such as network connectivity problems, allowing the user to take action if desired (e.g., fixing internet connectivity). In addition, observers are used to extend the behavior and invariants from the flow graph to external imperative libraries and errors/exceptions should be part of this interface. While not all languages have exception mechanisms, we believe that any embedding of Cvtr into a language should follow the conventions of the host language to provide a consistent development experience. In addition, extending the propagation of values to include metadata about the validity of the value seems to be a generally useful extension to Cvtr that has uses beyond simple error cases. Our evaluation (Section 13.4) shows that the overhead to support such additional data during propagation is negligible and the only cost are the added memory requirements to store the additional data.

Chapter 8

Compiling Cvtr for Embedded Devices

This chapter explores how to deploy the core of $\mathscr{F}_r$ to embedded devices with very limited resources. Typical dicApps we consider in the rest of this thesis are associated with devices with powerful processors and plenty of main memory (this includes smartphones). However, because of the distributed nature of dicApps it is useful to consider executing parts of the application on embedded devices. In this chapter, we consider the example of executing on a specialized processor embedded in all smartphones – the Wi-Fi chip. As a platform, the Wi-Fi chip has very limited capability of executing generic dicApps, but the Wi-Fi chip has efficient access to special resources – Wi-Fi packets and channel connection information. Accessing those special resources from the processor of the host device is often expensive, because the amount of data that can be transferred efficiently is limited and on mobile devices it is desirable to not activate the main CPU to conserve energy. Thus, there is considerable advantage of executing code directly on the Wi-Fi chip. For a similar reason, it is desirable to execute parts of a distributed application on other embedded devices such as networked sensing devices with limited bandwidth where aggregating and filtering data on the device saves costly communication.

In our prior work [192] we explored a purely reactive paradigm for Wi-Fi chips without distribution, interactivity, or connectivity. But ultimately, the goal is to develop a single application using our programming paradigm Cvtr that accesses special resources as if they were available to the local device. The compiler or runtime should automatically split the flow graph to produce the most efficient deployment. While doing this automatically is left for future work, in this chapter we show that the programming model of Cvtr is very well suited for the development of embedded hardware. We provide a compilation strategy for $\mathscr{F}_r$ programs that removes all dynamic overhead typically associated with the flow graph. We present a compiler that compiles a flow graph into sequential code that fits the execution model of sequential embedded processors. The only drawback for the programming model is that the ability to dynamically reconfigure the flow graph is lost – a property that is

Figure 8.1: Stages of compiling ReactiFi to Wi-Fi chips.

important to dicApps but not necessary for the parts of the application that should be offloaded to embedded devices.

8.1 Compilation Overview

To compile the flow graph it must be static. Static means that no dynamic edges between reactives are allowed, i.e., the `flatten` method cannot be supported. There may be no use of the `fire` and `value` methods during the creation of the flow graph, because it is impossible to interact with values during compilation. Thus, all integrations must happen using the sources and observers that are understood by the compiler. We call this reduced subset of the calculus ReactiFi – named after the first target platform of that language (the Wi-Fi chip). ReactiFi uses the C programming language for user-defined functions, but uses the actual calculus for the embedding language – in contrast, REScala uses Scala for both the embedding and for user-defined functions.

A ReactiFi program is processed in four steps (Figure 8.1): (i) an interpreter executes the ReactiFi program to construct the flow graph, (ii) the compiler compiles the flow graph into a C program that represents a sequential execution of the change propagation discussed in Chapter 4, (iii) the C source code is compiled into a binary, which (iv) is loaded to the Wi-Fi chip and linked into the firmware at runtime. Constructing the flow graph is very similar to the semantics described in Chapter 4. In this chapter, we are interested in the second step: compiling the flow graph to C code.

Once constructed, the flow graph is analyzed to extract the following information that is passed to the subsequent processing phases: a topological order of all reactives, a set of conditions guarding the activation of each reactive, and types and memory requirements of reactives in the flow graph. Given this information and the C code of the function bodies embedded into reactive definitions, the ReactiFi compiler generates a single sequential update function in C, which implements the reaction to external events.

To simplify the presentation of the compilation process we first discuss a simplified procedure where ReactiFi programs are compiled line-by-line into C programs, without any global transformations based on the information in the flow graph. This compilation procedure produces semantically correct code, but without global optimizations. We then discuss those optimizations separately in Section 8.3.

$$
\begin{aligned}
C[\![\text{val } x_0 = \overline{x}.\,\text{Map}\{f\}]\!] &= \text{if } (A[\![\overline{x}]\!])\{x_0 = C[\![f(\overline{x})]\!]\} \\
C[\![\text{val } x_0 = x.\,\text{Fold}\,(v)\{f\}]\!] &= \text{if } (A[\![x]\!])\{x_0 = C[\![f(x_0, x)]\!]\} \\
C[\![\text{val } x_0 = x.\,\text{Filter}\{f\}]\!] &= \text{if } (A[\![x]\!] \&\& C[\![f(x)]\!])\{x_{0f} = \mathit{true}; x_0 = C[\![x]\!]\} \\
C[\![\text{val } x_0 = \text{Source}(s)]\!] &= \text{if } (A[\![s]\!])\{x_0 = C[\![s]\!]\} \\
C[\![\text{val } x_0 = C[\![\overline{x}.\,\text{observe}(o)]\!]]\!] &= \text{if } (A[\![\overline{x}]\!])\{C[\![o(\overline{x})]\!]\} \\
C[\![\text{val } x_0 = (x_1 || x_2)]\!] &= \text{if } (A[\![x_1]\!])\{x_0 = C[\![x_1]\!]\} \\
&\quad\ \text{else if } (A[\![x_2]\!])\{x_0 = C[\![x_2]\!]\} \\
C[\![\text{val } x_0 = x_1.\,\text{Snapshot}(x_2)]\!] &= \text{if } (A[\![x_1]\!])\{x_0 = C[\![x_2]\!]\}
\end{aligned}
$$

Figure 8.2: Compiling individual ReactiFi reactives (left) to C code (right).

8.2 Compiling Individual Reactives

To simplify the presentation of the compiler we show how each individual reactive is compiled when in single static assignment form, i.e., the code is a sequence of definitions in the form val $x_0 = t$ where t is a term that creates a reactive, source, or observer. In addition, we assume the program code does not contain any function calls on the top level. Thus, the top level program consists only of the assignments of names to reactive definitions. The overall compilation process then applies the transformations of individual reactives to the sequence of statements in the program. This representation is exactly capable of representing all static and flow graphs. Figure 8.2 shows how each of the assignments for all the different reactive definitions are compiled into C-like statements.

Compilation from ReactiFi to C is written $C[\![\text{val } x_0 = r]\!]$. The result of compiling a reactive is generally in the form "if $(A[\![\overline{x}]\!])\{x_0 = C[\![f]\!]\}$" where $A[\![\overline{x}]\!]$ computes when the reactive activates. If it does activate, the state x_0 of that reactive is updated. Again, for ease of presentation each statement is compiled individually and the produced code is conceptually executed in the order of the ReactiFi program. The order of statements in the ReactiFi program corresponds to the topological order of the flow graph because the code does not allow forward references. However, in reality, the ReactiFi program is first executed to extract the flow graph (c.f., Figure 8.1 and the compiler then groups the execution of related statements for efficiency as discussed later.

For the line-by-line compiled code, each statement first checks if the activation condition (e.g., $A[\![\overline{x}]\!]$) for its inputs (e.g., $\overline{x}$) are fulfilled, and only then updates the current value of the reactive (x_0). In contrast to $\mathscr{F}_r$ only source reactives and filter reactives have their activation tracked at runtime, because we want to minimize the amount of used memory for propagation. All other reactives compute their activation based on their transitive dependencies. Each reactive activates exactly if one of their inputs activates, but the same holds true for the inputs. Thus, to know if a reactive r activates in a given propagation, we only need to compute if

any of the transitive sources or filters that r depends on activates. We discuss in Section 8.3 how we optimize these checks, so they do not need to occur for every single reactive.

Due to the memory constraints, ReactiFi primarily uses event semantics for its reactives, i.e., the state of reactives is not stored between propagations. The only exception are sources which have their state available directly in the firmware of the Wi-Fi chip and fold reactives which do store state. Compilation is shown in Figure 8.2. All reactives are directly compiled into assignments of new values to their storage location. The interesting exceptions are filter reactives, which in addition to assigning the value of their input without change also set their filter activation x_{0f} to true. Source reactives are underspecified and their activation and value is dependent on whatever the runtime of the Wi-Fi chip can support. There are no user-defined sources in ReactiFi. The same is true for observers, which may only execute pre-defined effects known to the compiler and runtime. Due to ReactiFi having only simplified event semantics except for folds, we show two more reactive methods that are not present in the core of $\mathscr{F}_r$. These are choice reactives $(x_1||x_2)$ and snapshot reactives. Choice reactives have two different activation conditions depending on either one or the other input, but not both, and their value is the value of whichever input activates (the first one if both activate). Choice is not a primitive operation for REScala, because the value of events is wrapped in an optional data type – thus it can be safely accessed and tested for presence. In contrast, in ReactiFi the value may only be accessed if the reactive activates otherwise the program would read uninitialized memory. The snapshot method is required to access the state of a fold reactive when another reactive activates. This access is safe because the state of folds is always available. The type system of ReactiFi distinguishes between fold reactives and other reactives and only allows folds to be used for the snapshot method. The compilation of snapshot checks one input for activation and uses the value of the second input.

Compiling constructs other than reactives works as expected. Compiling functions $C[\![f(\overline{x})]\!]$ result in a call to a fresh top-level function definition where the body of the function is kept as is – note that user-defined functions are already written in C. Compiling an *identifier* that binds a reactive $C[\![x]\!]$ produces code that accesses the value of that reactive.

8.3 Optimizations Using the Flow Graph

Line-by-line compilation produces a conceptually correct program, but using the flow graph to optimize the code that reacts to individual sources allows for efficient execution on the limited target platform. Especially, we are interested in minimizing the time spend in each single update propagation process, because any latency may cause dropped Wi-Fi packets. We are also interested in minimizing memory consumption because there is only very limited memory available.

Static Sequential Scheduling

The flow graph specifies a logically concurrent execution order of reactives in response to individual external events; moreover, only a subset of reactives is typically activated for each external event. Wi-Fi chips only support sequential execution and network applications are latency-sensitive. To address these constraints, the compiler sequentializes the order of updating reactives, produces optimized per source update logic, and generates a minimum number of conditional branches to select the updated reactives.

A topological traversal of the flow graph is used to compute the sequential execution order of reactives. However, while the relative execution order of reactives can be statically fixed according to the topological order, whether and when reactives activate depends on runtime conditions. In particular, for many source reacitves it is statically known which of them may occur at the same time and which will always occur at logically distinct times, e.g., packets are processed sequentially by the Wi-Fi chip even if physically occurring at the same time. Thus, the compiler generates one update function per set of sources that may occur together. In the typical case this results in each source having a single concise update function that quickly updates the relevant state. All update functions share the same global set of state stored in fold reactives.

When multiple sources may occur at the same time, or in flow graphs with complex filter conditions and multiple paths, we do not want to check the activation conditions for each individual reactive. Instead, reactives are grouped into uninterrupted pipelines based on shared filtering conditions. The compiled update function only checks conditions once per group. Sources and filter reactives define new conditions. For all other reactives, the conditions are derived from the conditions of their input reactives. The choice and fold all reactives use the disjunction of the conditions of all inputs. All other reactives use the conjunction of the conditions of all inputs. For illustration, consider the flow graph of a case study in Figure 8.3. The case study is detailed in prior work [192], but we are only interested in the structure of the flow graph. There is a single source: `Monitor` but multiple filter and choice reactives that change the activation condition. The blue boxes mark the groups of reactives with the same activation conditions. For instance, the rightmost group will execute only if the `Monitor` source fires, the condition for the `frames` filter holds, and either `fromSource` or `fromAP` filter functions are true. Thus, the generated update function for the `Monitor` source will only check the activation conditions four times, once for each group – except for the source group which does not need to be checked. When there are multiple sources, then also the source groups need to check if that source activates or not.

Optimized Memory Management

Wi-Fi chips have limited memory. For instance, the memory built into the Nexus 5 has 768kB RAM, most of which is used by the basic firmware, with only as little as 100kB RAM left for higher-level functionality; to put this into context, a single

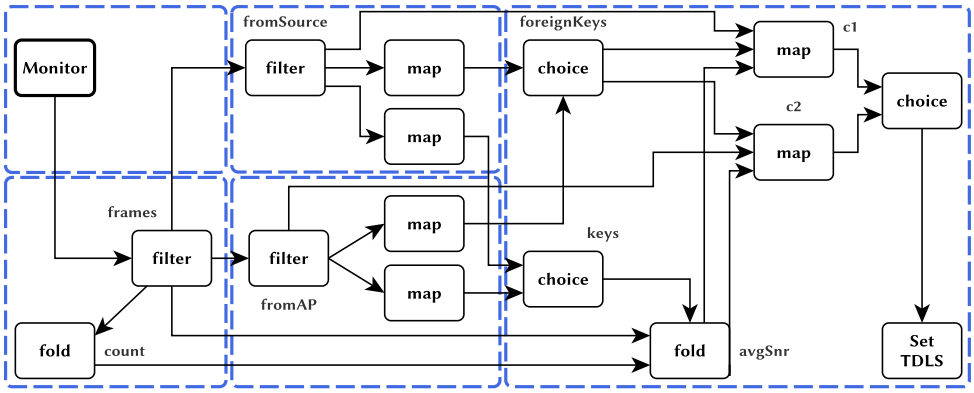

Figure 8.3: The flow graph of the adaptive file sharing case study.

IP packet is up to 2kB in size. Reactives are abstractions with zero runtime memory cost, i.e., the only memory requirements at runtime are those of the contained values and only when the values are in use. To facilitate compile-time estimation of the needed memory, ReactiFi allows only fixed size types T to be used in code. Memory for reactives is reclaimed at the earliest time possible. Technically, to find the reclamation point of a reactive r_1, the compiler traverses the sequential execution order from the back to find the last reactive r_2 that depends on r_1. The scope of r_1 extends from r_1 until after r_2. Unlike other reactives, the state of folds is stored between updates, thus never reclaimed. Overall, for each reactive r, the compiler knows how much memory is already allocated when a new value will be computed for r. This is the sum of the memory allocated to all folds in the program plus the sum of the memory allocated to all non-fold reactives in scope. This way, the compiler is able to maximize the memory available for executing the function bodies embedded in the reactives.

8.4 Exemplary Compilation

For illustration, consider the code below, defining a map reactive (`address`) with two inputs, `frame` and `subframe`; we assume that both frames are derived from a `Monitor` source (not shown for brevity).

```
437  val address = (frame, subframe).map((fr, sub) => { /*
         extractAddress */ } )
```

The generated C code is sketched in Figure 8.4. Since there are no folds, the program has only local state. The user-defined function is extracted to a top-level C function `extractAddress`. Within the `update` function, first, the variables for the source conditions (`monitor_condition`), the values computed for the input reactives (`frame_value`, `subframe_value`), and for the map reactive (`address_value`) are declared. Then, the activation conditions of sources are computed, followed

```
438 address_t extractAddress(frame_t fr, frame_t sub) { /*
        extractAddress */ }
439 // state of fold reactives would be above
440 void update() {
441     bool monitor_condition;
442     frame_t frame_value;
443     frame_t subframe_value;
444     address_t address_value;
445
446     monitor_condition = runtime_is_triggered(Monitor)
447     if (monitor_condition) {
448         frame_value = ...;
449         subframe_value = ...;
450         address_value = extractAddress(frame_value,
                subframe_value);
451         deallocate(frame_value);
452         deallocate(subframe_value);
453     }
454 }
```

Figure 8.4: Generated C code example.

by the guarded execution of reactives, when the sources are activated. The code illustrates the two compiler optimizations. First, the activation condition is only checked once for all reactives as opposed to once per reactive. Second, the values `frame` and `subframe` are deallocated immediately after they are used and no longer needed. We assume that `address_value` is used later in the program, otherwise the whole program would be optimized away.

8.5 Conclusion

This chapter shows how the Cvtr programming paradigm can be implemented by compilation to sequential code suitable for embedded devices. We believe this to be a significant demonstration of the utility of the programming paradigm, because embedded devices are at the opposing end of the spectrum of platforms that dicApps are typically executed on. REScala was designed for the JVM – with large heaps and fast JIT compilation. Thus, it is delightful that the same programming paradigm is usable on small embedded devices – even though we have to restrict the compilation to static flow graphs. Our hope for the future is, that we can use the same domain specific language to describe the flow graph for an application that is distributed on embedded devices and desktop devices and have communication be automated.

Chapter 9

Debugging and Live Tuning

The flow graph of Cvtr provides an interesting opportunity for developer tooling because it enables the development of interactive applications in a style that is well-suited for visualization, is flexible enough to be adapted at runtime by non-experts, and provides *data safety*, i.e., data is not lost or processed inconsistently even when modified by external factors.

Debuggers for reactive programming already allow inspection of an application through its flow graph during execution [178]. Based on those reactive debuggers, we pave the way towards modifying the behavior of applications at runtime – a process called live programming.

First, we extend an existing reactive debugger to allow for runtime modifications to the application. We provide some benefits of live programming to application developers while ensuring that modifications always stay within the confinements defined by the application code, specifically, guarantees given by the type system still hold, and data dependencies in the application are kept consistent.

Second, we investigate how applications may be inspected and modified even by their users. Our goal is to enable developers to create applications that can be tuned by domain experts. To this end, we propose patterns and idioms as well as new language abstractions for specifying tuning points of the applications for domain experts and thereby retaining basic safety and correctness guarantees of the application.

We assume three levels of expertise. First, the *language and tool authors* who design very general abstractions and mechanisms to implement arbitrary applications. This first group, however, has no knowledge of the domain or requirements of the final application, which is written by the second group – the *developers*. Developers use the abstractions and tools of the language to implement concrete application semantics. Developers determine concrete use cases for their application. They are responsible for ensuring that the application behaves as desired. We assume they are familiar with the abstractions provided by the language and are able to use them correctly. The third group, the *domain experts* of the application, have concrete tasks they want to solve with the application. Because tasks often vary

```
455 // input sensor for temperature
456 val temperature = Evt[Double]
457 // filter very high and low temperatures an outliers
458 val filtered =
459   temperature.filter(_ <= 100).filter(_  >= -40)
460 // build a history of up to 50 temperature values
461 val history: Signal[Seq[Double]] = filtered.last(50)
462 // an aggregation function that may change over time
463 val aggregation: Var[Seq[Double] => Double] = Var(average)
464 // apply the aggregation function to the temperatures
465 val aggregated = Signal {
466   val f: Seq[Double] => Double = aggregation.value
467   f(history.value)
468 }
469 // select which values to display in the dashboard
470 val onDisplay: Var[List[Signal[Seq[Double]]]] = Var(List(history,
      aggregated))
471 // assume a generic rendering function for values
472 def render = {
473   case d: Double => ...
474   case l: List[Double] => ...
475 }
476 // render selected signals on the dashboard
477 val dashboard = Signal {
478   for (item <- onDisplay.value)
479     yield render(item.value)
480 }
```

Figure 9.1: Dashboard application.

slightly and those variations are not always predictable by developers, domain experts may also need to vary the behavior of the application. However, because domain experts do not have prior knowledge of application development, it is only appropriate for them to change existing behavior in well-defined ways. We use the name *tuning* [119] to refer to this process in the rest of the paper.

9.1 The Chapter's Running Example

The code in Figure 9.1 implements an exemplary application that visualizes sensor data on a dashboard. `Evt` (Line 456) serves as an input from an external temperature sensor. The temperature event is processed by an operator pipeline (Lines 459, 461) filtering extreme temperatures (probably sensor faults) and aggregating the last 50

temperatures – a signal derived from an event. Events and signals may be derived from other events and/or signals as long as no cyclic dependencies are formed. The aggregation function (Line 463) can change over time, i.e., we model it as a signal, which holds the current aggregation function for every point in time. More specifically, we model it using a `Var`, which is an input "signal" that can be imperatively set. The signal expression (Line 465) applies the (time-changing) aggregation function to the (time-changing) history of temperature events. The dashboard (Line 477) then renders all values that should be on display (Line 470). Because the displayed list of values is a signal (a `Var` to be precise), the concrete displayed items may change, resulting in a dynamic flow graph.

9.2 Reactive Debugging

Debugging is the process of understanding the behavior of code to resolve bugs. Debuggers aid programmers at finding defects and help code comprehension by visualizing the application behavior during runtime. Traditional debugging tools focus on the analysis of control flow in sequential programs. They provide means to interrupt the application at certain points (breakpoints) and continue by stepping through each instruction of the program. However, debugging based on control flow is ill-suited for Cvtr, because it does not correspond to the flow graph. Therefore, developers often use "printf debugging" to trace the dataflow or develop their own ad-hoc visualizations of the flow graph [29].

Reactive debugging [178] is a debugging technique designed to fit the needs of reactive programming. Reactive Inspector, a tool that implements this technique, provides a set of features similar to those of traditional debuggers but adapts to a dataflow view, which is shown in Figure 9.2. Reactive Inspector directly visualizes the flow graph and enables navigation through the history of changes using history queries and a time slider. Reactive Inspector can also pause execution of the current application when selected nodes in the flow graph are modified, similar to breakpoints in sequential debuggers.

Visualization

Figure 9.2 shows Reactive Inspector – an extension of the Google Chrome debugger. Reactive Inspector displays the flow graph for the dashboard application code presented in Figure 9.1. The main view of the debugger shows the flow graph. Nodes in the flow graph are named after their corresponding variables in the source code. Data flows from left to right, beginning with the temperature event and ending at the dashboard signal. The view shows the flow graph directly after it has been created, so no temperatures have been recorded yet. The highlighted edge (orange) represents the latest modification to the graph – the connection between the aggregated value and the dashboard. When the graph changes dynamically during program execution, i.e., new nodes or new edges are added, those changes are reflected in the dataflow visualization.

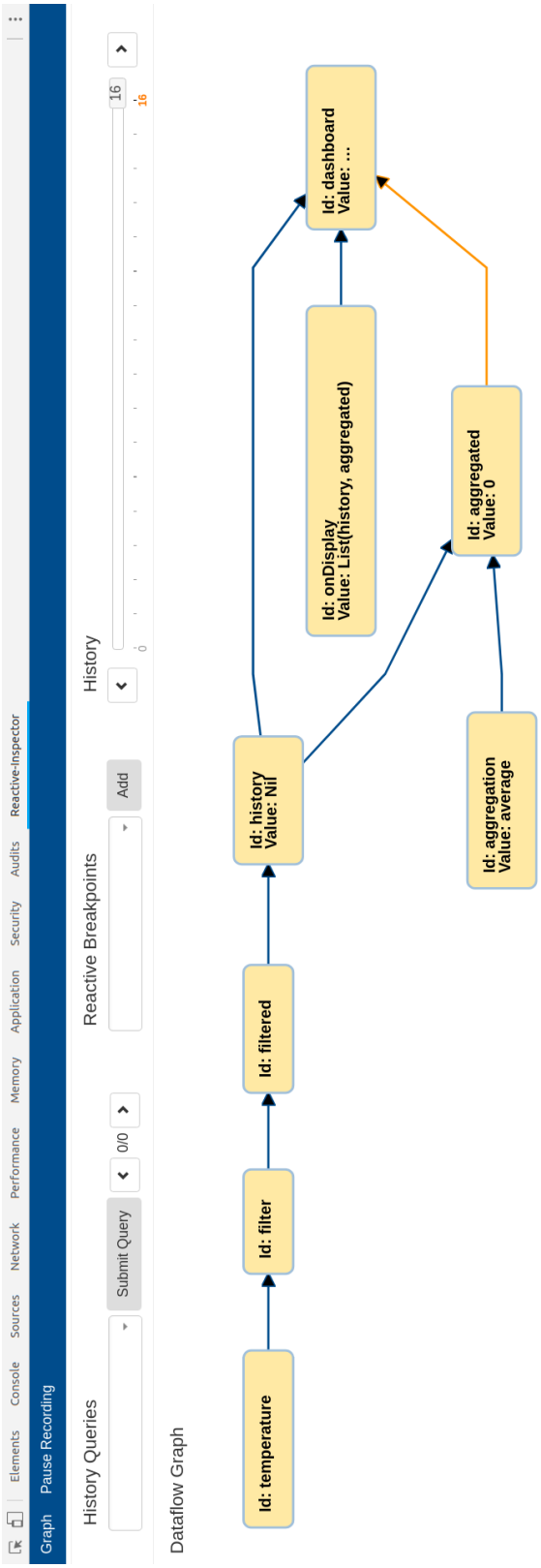

Figure 9.2: Reactive Inspector user interface.

History

History navigation is similar to stepping through a sequential program, but does not pause the execution of the application and works both forward and backwards. Applications only store the current state – it is normally impossible to inspect past states of the graph – hindering understanding of the past behavior of the application leading to the current state. Reactive Inspector solves this issue by storing every past state of the flow graph and providing users with a way to visualize the history of the graph. The history can be inspected by moving the history navigator slider shown in Figure 9.2, and points of special interest can be quickly found by issuing a history query. History interactions are only visualizations, the application is neither modified nor paused – new events are recorded in the future while inspecting the past.

Breakpoints

Similar to the breakpoints in traditional debugging tools, Reactive Inspector allows users to attach breakpoints to certain graph nodes which cause the program to halt when the corresponding node changes its value. Breakpoints may be conditional and only trigger on specific values passing through nodes. A breakpoint can also be set on the creation or deletion of nodes from the flow graph. When a breakpoint is triggered, Reactive Inspector delegates to the native Chrome debugger to step through the sequential code inside signal expressions.

9.3 From Inspecting to Modifying

Our paradigm is particularly promising for live programming because applications are already designed for interactivity, that is, reacting to external changes. For example, the developer might wonder what would happen if a very high (and thus filtered) temperature was measured when the history already contained 50 elements. *Would this temperature still trigger the update of the history and remove one old element, even though it was filtered?*

Yet, despite the flow graph is designed for processing dynamic and arbitrary input, and updating the application state, with the current tools "what if" questions are surprisingly hard to answer. Triggering an interesting (high temperature) event at the right point in time (history contains 50 elements) requires modifications to the application code to detect the desired event pattern and then trigger the event. The flow graph in Reactive Inspector allows one to easily answer such questions. We extend this tool to enable direct manipulation of the values in the flow graph.

Graph Modifications

Figure 9.3 shows the extended interface of Reactive Inspector. The interface allows changing arbitrary values in the flow graph, similar to how interactive debuggers allow changing values assigned to variables. However, changing arbitrary variables

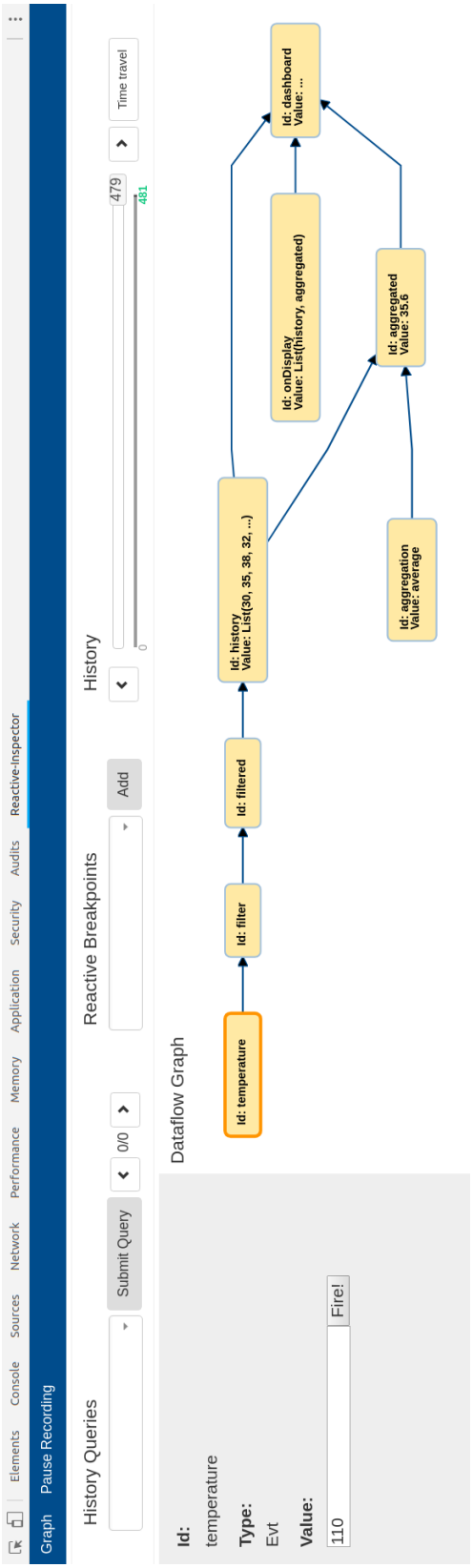

Figure 9.3: Reactive Inspector with extensions to modify the flow graph.

of an application may cause inconsistencies in the data model. Reactive Inspector, on the other hand, uses the same mechanism as if the application programmatically changed its input events (`Evt`) or signals (`Var`). Thus, using the flow graph prevents inconsistencies by ensuring that each change is fully processed by the application. For example, when the developer issues a new temperature event using the "Fire!" button in the left-hand pane of Figure 9.3, then the new event will be automatically added to the history of events. The developer can then easily answer the question above and conclude that the event did not change the history – it was filtered.

Exposing the graph through the debugger in such a principled manner allows developers to change the code without invalidating basic assumptions of the application. For more flexible debugging purposes, however, we also allow developers to modify arbitrary signals and events, not just inputs (`Var` and `Evt`). For example, the debugger allows one to directly set the average temperature value, even if the new value no longer reflects the current history of temperature events. Changes to intermediate values are propagated normally, e.g., the dashboard will be updated with the new average.

Time travel

A well-known technique of debuggers to modify the state of an application is time traveling [5], i.e., loading a past state of the flow graph. REScala has efficient support for snapshotting the flow graph [155] and the debugger can go back to the old state of the application by loading a snapshot. This approach enables users not only to inspect the history of an application, but also to have the actual application jump back and forth in time to enable user interaction with the application in a specific state. Time travel uses the history navigation to select the desired point in time. Time travel is especially useful for "what if" scenarios. The debugger is able to reset the application to a past state – allowing the developer to explore what led to the current state – and modify past values. Past modifications may change the current application state in the future. It is the developer's responsibility not to cause inconsistencies when time traveling.

9.4 Towards a Live Tuning Framework

Inspection and modifications of the flow graph is not only useful to developers, but also to enable customization – tuning – of the application by domain experts. In this section, we envision a live tuning framework built on top of Reactive Inspector.

Domain experts are assumed to have little or no knowledge of application development, but may need to comprehend and modify applications that they use. For this use case, we consider correctness and ease of tuning more important than full programmability. Domain experts should be able to understand, learn from, and adapt examples, but without risk of breaking the applications. Our goal is to provide limited tuning for most applications out of the box without additional developer effort. For additional flexibility, we provide dedicated language abstractions

to customize the tuning mechanisms. We describe the spectrum of our abstractions for supporting live tuning going from the most simple (ease of use) to the most flexible.

Tunable Values

The first step is to enable domain experts to modify input signals (`Var`) and trigger input events (`Evt`). Interacting with inputs is considered safe, as the flow graph never has control over external inputs, and has to deal with arbitrary values in any case. Reactive Inspector ensures that users may only enter values of the correct type and provides easy to use UI abstractions for custom values. For example, integer inputs are set via sliders, and string inputs with a text box.

However, tuning some inputs may require specifying Scala functions, such as the aggregation function of Figure 9.1. To save domain experts from writing Scala code we allow developers to provide predefined behaviors from which the domain experts select. The reactive debugger then displays a drop-down list of possible choices. In the example, the developer could provide an average and a sum function, as an enumeration of aggregation schemes:

```
481 enum Aggregate(val name: String,
482                val compute: Seq[Double] => Double) {
483   case Average extends Aggregate("average", ...)
484   case Sum extends Aggregate("sum", ...)
485 }
```

Tuning the Flow Graph

We make dynamic changes to the flow graph available via tuning. For example, the application developer may want to allow the domain experts to adapt the contents of a dashboard by adding or removing displayed elements. The dashboard example supports this use case by allowing the modification of the `onDisplay` signal. That signal contains a list of other signals and may be tuned to add new signals. It is only possible to add reactives with the correct type – this allows the developer to limit which reactives are selectable for dynamic reconfiguration.

Custom Tuning

At the end of the spectrum for application tuning are solutions defined by the developer. For example, developers may want to allow the domain experts to create new elements in the dashboard. New elements should be specified using a custom domain-specific language designed for data analysis.

The developer is free to specify how domain experts may interactively tune the application. However, it is then also the developer's responsibility to map such tuning mechanisms to the underlying flow graph modifications (e.g., changed nodes, changed code within nodes, changed edges). The runtime will still take care of

ensuring propagation of updates and keeping the application state consistent. We believe that Cvtr provides a strong basis for a framework for writing tunable applications, where the concrete runtime behavior is interactively developed, e.g., a data analysis application where domain experts can interactively build and combine queries.

9.5 Conclusion

We have shown that Cvtr facilitates writing of tunable live applications. The abstractions are suitable for both writing source code and representing code visually to interactively inspect, change, and evolve the behavior of an application. The presented approach to modify values in the graph is an initial step towards making the graph more tunable through live modifications. Augmenting Cvtr with abstractions specific for tuning allows for safe runtime modifications also by non-expert developers. In perspective, we want to support and to be able to mix both forms of traditional development (compiling and re-executing) and live development (changing code during runtime), so developers can freely choose to what degree they use which approach to solve specific problems during the development process. Refer to Section 14.7 for related approaches.

Especially in the area of distributed executions it is an exciting future extension to get an overview over a distributed flow graph. Such an overview is by necessity not always consistent with the actual state of the system, but using Cvtr allows an eventually consistent view that also would enable eventually consistent tuning options, thus allowing system operators to tune their distributed systems easily.

Chapter 10

CRDTs and Cvtr

When we introduced replication and causal consistency in Part I of this thesis we opted for a modular integration of CRDTs and reactives. Replicated reactives in $\mathscr{F}_r$ only assume the existence of an associative, commutative, and idempotent merge function. The implementation in REScala follows the same principles – the difference between a normal reactive and a replicated reactive is that the latter has an implementation of the merge function (c.f., Figure 6.11). The modular integration fits the fine-grained proofs in our operational semantics, because it only adds a minimal amount of cases to consider for the integration. For example, our proofs for optimal concurrency and transactional isolation are completely independent of the replication. Due to the same reason, the replication strategy could be extended or exchanged, as long as the result still adheres to the properties of the merge function. For example, it would be possible to exchange state-based CRDTs with the global sequence protocol [47]. The implementation also benefits from the same flexibility.

In this chapter we informally explore what a tighter integration of CRDTs and Cvtr entails. The discussions in this chapter explains the existing properties on a different level of abstraction, and shows that there is a logical equivalence between our synchronization algorithm used for causal consistency of the flow graph and the general concept of synchronization of CRDTs in the literature. We hope that the discussion provides a better understanding of the consequences of the Cvtr programming paradigm regarding replication and that it may serve other similar systems as guidance or inspiration. Furthermore, the new perspective allows us to use existing research on CRDTs and directly apply the concepts to find solutions for challenges in Cvtr.

We first discuss challenges for application design caused by the restrictions on operations of CRDTs. Then, we give an intuition of the benefits of Cvtr for the application design compared to a design using CRDTs without Cvtr. Specifically, this intuition states that the flow graph itself can be seen as a single complex CRDT, thus giving a high-level correspondence between these two parts of Cvtr. We also discuss a general strategy to represent all Cvtr applications as CRDTs independent

of the usual restrictions placed on CRDT operations. Finally, use this correspondence to discuss two extensions to Cvtr. The extensions include performance improvements based on delta replication for CRDTs, and the integration of strong consistency into Cvtr with an implementation of the Raft consensus algorithm.

10.1 CRDTs and their Challenges for Application Design

Every CRDT has a state, and a collection of operations – an API – that change that state. CRDTs can be roughly categorized as operation-based or state-based, depending on the properties a CRDT requires of its state and operations. The challenges on application design using CRDTs stems from these restrictions, which we discuss in the following.

While Cvtr generally makes use of state-based CRDTs – because they impose the least restrictions for our purposes – we will first discuss operation-based CRDTs. We then discuss the monotonicity requirement of operations on state-based CRDTs. Finally, we discuss the composition of multiple CRDTs, that is, how to support operations that change multiple CRDTs at the same time.

Operation-based CRDTs

Operation-based CRDTs [186] achieve consistency by synchronizing the set of applied operations. The current state is reconstructed by applying the operations. The network runtime would have to ensure that operations are applied exactly once and in the correct order to ensure consistency (the global sequence protocol [47] is based on that strategy). If the network runtime cannot guarantee order, operation-based CRDTs require that the order in which operations are applied does not matter, i.e., all pairs of operations are commutative. If the network cannot guarantee that operations are applied only once, operation-based CRDTs require that operations are idempotent. The only way to achieve causality is for the network to ensure that operations are applied in causal order.

The advantage of operation-based CRDTs compared to state-based ones is that – with commutative and idempotent operations – it is sufficient to individually send the executed operations. There is no need to send the complete state. Commutativity and idempotence of operations has implications for application design – it becomes harder to design new operations. For example, to support a new operation for an operation-based CRDT it must be ensured that the new operation is commutative with all other operations, thus it is no longer possible to design operations independently. In conclusion, we believe that the simpler conceptual design and higher flexibility of state-based CRDTs makes them a better fit when integrating CRDTs into the application logic. Thus, we only consider state-based CRDTs for Cvtr.

State-based CRDTs and Monotonicity

Cvtr makes use of state-based CRDTs – or, more precisely, the properties of algebraic lattices. A lattice is a partial order $\leq$ on states, where every two states σ_1 and σ_2 have a unique least upper bound $\sigma_{1,2} \leq \mathit{merge}(\sigma_1, \sigma_2)$. State-based CRDTs cannot support operations that decrease their state according to the lattice order. Consider when an operation is applied to state σ and the result σ_r is smaller $\sigma_r \leq \sigma$. Then, merging the two states is equal to the original state $\mathit{merge}(\sigma, \sigma_r) = \sigma$, that is, the intended effect of the operation is lost. Thus, each operation on a lattice must be "monotonic", i.e., must take effect by only increasing the state.

Monotonicity has an implication on the design of applications and their data structures. For example, to implement a counter it is no longer sufficient to increment or decrement an integer register. Instead, a state-based CRDT counter records an individual count for each replica – with the sum of all counts reflecting the original counter. The inherent problem here is that the lattice structure of the state of a CRDT must be designed to support its operations. Supporting many or generic operations causes the state to become more complex. Thus, it is often infeasible to support CRDTs that are as general as traditional data structures.

However, it is in practice evident that finding a monotonic implementation of a single operation by combining other monotonic operations is not any harder than traditional application development. Thus, it becomes very desirable to design CRDTs as part of the application and specifically to only support operations required by that application, which sidesteps the issue of having to design big APIs of many monotonic operations.

Composing State-based CRDTs

Composing two CRDTs is usually understood as having an operation that changes both CRDTs, while extending the guarantees of individual CRDTs to the composition. For example, consider two integer counters representing two bank accounts and a composed operation that transfers money by increasing one counter and decreasing the other counter. All devices should observe the two changes "at the same time" (as an atomic operation) to ensure that a transfer of money does not change the total balance – not even temporarily. Essentially, we need a form of "transactions" (different from Cvtr transactions) over CRDTs.

With traditional CRDTs, the way to achieve this is to combine all involved CRDTs into a single one during the design of the application. For state-based CRDTs this composition can be achieved systematically, by arranging two or more individual lattices into a single structure. Each operation and the merging of states then operates on the corresponding component of the structure. New operations affecting multiple components can then be added.

However, the application has to use the combined CRDT in all places instead of the individual smaller CRDTs. For example, an application can no longer operate on a single bank account, but must operate on the composed CRDT that represents all bank accounts that may have transactions between them. Since most applications

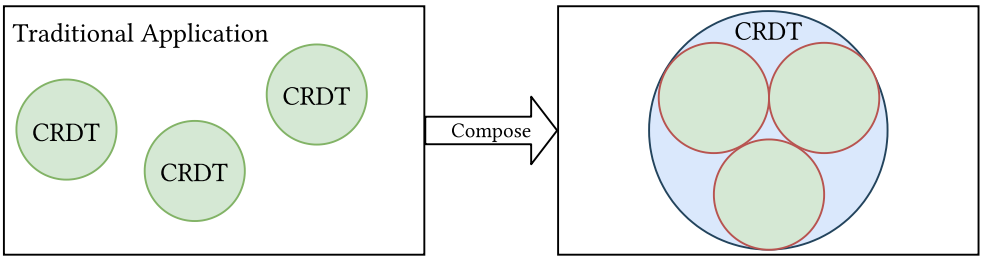

Figure 10.1: Composing multiple CRDTs in a traditional application results in a single monolithic CRDT, thus losing modularity.

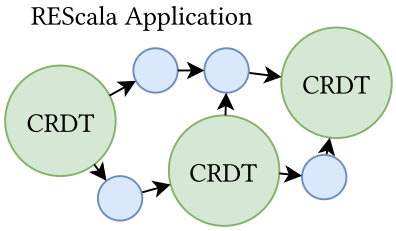

Figure 10.2: A flow graph provides modularity and transactions for multiple CRDTs.

have pairwise interactions between many of their components, the end result tends to be that all CRDTs of the application have to be composed into a single CRDT as indicated in Figure 10.1. Thus – because this large CRDT is then used by all parts of the application – modularity is lost [23].

10.2 How Cvtr Fosters Modular Composition

We have shown how Cvtr provides causal consistency for multiple CRDTs in Section 5.9. Causal consistency is ensured by synchronizing all replicated reacives when two devices communicate (as opposed to just the reactives present on both devices) to ensure that all causal relationsships are preserved even in a dynamic network with multiple hop communication. Thus, the ensured correctness guarantees and the required amount of synchronization of Cvtr are the same as if the application was using a single monolithic CRDT composed out of the state of all replicated reactives. This correspondence between Cvtr and a monolithic CRDT is not by chance. As we discuss next, there is a representation of the flow graph as a single CRDT. Because the application design is not affected by this representation, Cvtr provides transactions over CRDTs without sacrificing modularity.

In Chapter 4, we represent the state of an application as a store $\mu : r \mapsto v$ that maps reactives r to their values v (simplified view, the actual store contains more information that is not relevant here). Each replicated reactive has a merge function in a global map $M : r \mapsto \mathit{merge}$. Thus, the store – more precisely, the

part of it consisting of replicated reactives – maps each reactive to a state that is part of a lattice (i.e., the state has a merge function). From this it follows that the store μ itself is a lattice, because we can construct a merge function $merge(\mu_1, \mu_2) = \{M(r)(\mu_1(r), \mu_2(r))|r \in dom(M)\}$ that merges two stores by merging each reactive using its individual merge function. Because we have shown that the store forms a lattice, a transaction in Cvtr then corresponds to a single operation on the store CRDT that changes the state form one store to another (larger) store.

Figure 10.2 shows that CRDTs embedded into the flow graph remain independent, but are connected as part of reactives in the flow graph. Reactives may be published by modules and consumed in other modules, without losing independence between modules. Changes to reactives in different modules are orchestrated by transactions without the need to modify the design of the application. Thus, it is not necessary for developers to manually design new operations and fit them all into a single CRDT – the transactions provide those operations as needed. Furthermore, because each individual CRDT in Cvtr serves only a very specific purpose in the application (as the state of a single reactive), it is sufficient if each CRDT implements the minimal set of operations required for that reactive. Supporting only few operations allows the implementation of the CRDT to be as simple as possible.

Moreover, all Cvtr applications have a regular structure when automatically represented as a CRDT – while the structure of a manually designed monolithic CRDT depends on the application. Thus, the implementation can make use of the known structure to apply optimizations in certain network topologies (e.g., client-server model, or a peer-to-peer model). It is even possible to dynamically forfeit causal consistency for better performance without changing the application code.

One may argue that our solution already presumes the use of a different programming paradigm. However, this is not a limitation. On the contrary, previous studies have shown that designing applications in a dataflow style improves program comprehension thus reducing errors [182]. In addition, our case studies (Chapter 11) demonstrate that the programming model is suitable to implement the kind of applications we are interested in, which is still unclear for designing dicApps as CRDTs.

In conclusion, synchronization in Cvtr corresponds synchronization of a single CRDT. We believe that this correspondence will allow us carry over theoretical results of (state-based) CRDTs to Cvtr in future work. The rest of this chapter explores preliminary ideas we gain from this correspondence.

10.3 How Cvtr Fosters Monotonic Operations

Here, we discuss how any transactional system, including Cvtr, relates to a lattice with only monotonic operations. In other words, an alternative view of the relation between transactional systems and CRDTs.

To design a (state-based) CRDT, it is necessary to design monotonic operations, i.e., operations that only increase the "size" of the state according to the partial order of the underlying lattice. The state of transactional systems lends itself to a

natural monotonic representation: a log of transactions. As discussed, every local transaction on the flow graph can be seen as a single operation on a structure resembling a composed CRDT. However, because transactions in Cvtr are deterministic (c.f. Section 5.5) and executed in serial order (or at least in serializable order for real implementations), there is an alternative representation of the state of the flow graph where only the local transactions and their serial order are replicated. Due to the determinism of applying transactions to the flow graph a globally consistent state of the flow graph can be restored on each device by applying the transactions in the serialization order. Thus, no longer requiring the synchronization of state. In particular, no longer requiring operations on that state to be monotonic. The overall concept is similar to what many databases use for fault tolerance.

Using the transaction order for replications allows arbitrary – non-monotonic – operations instead of requiring CRDTs as the state of reactives. However, the issue with this approach is that a device that is working offline may create transactions very early in the global serialization order. As soon as that device synchronizes with other devices all the other devices must revert their state to the state of the earlier transaction and then re-execute all transactions that happened after. The global sequence protocol [47] shows how a system that reverts state in case of late messages can be made efficient when using a central server to ensure that only a limited amount of program history must be recomputed.

The advantage of using the transaction log for replication lies in the theoretical implication: causal consistency is possible for every Cvtr program. The only limitation are efficient implementations for important special cases. However, while this theoretically means that Cvtr allows using non-monotonic operations for replicated data, none of our case studies has ever shown a need to do so. The ability to compose simple CRDTs in multiple reactives proves to be flexible enough for program development. Derived reactives in Cvtr may use non-monotonic operations in any case, reducing the need for more complex CRDTs. Thus, using CRDTs as the state of the replicated reactives remains the more practical choice.

10.4 Profiting from the Correspondence Between CRDTs and Cvtr

In the following, we discuss two topic topics where we gain new insights based on an overlap of research between CRDTs and Cvtr. First, we discuss how research on delta CRDTs may improve performance for replication in Cvtr. Second, we implement the Raft consensus algorithm as a CRDT, thus providing applications a choice to use strong consistency if needed.

Addressing the Cost of Causality with Delta Replication

Figure 10.3 visualizes the following example of a distributed workflow that spans multiple users and devices. A user uploads a video, which is re-encoded by a server for multiple resolutions, simultaneously reviewed by a moderator, and published

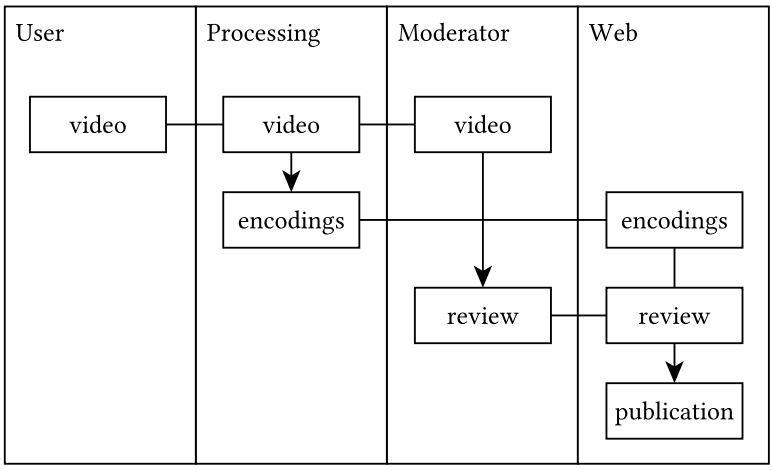

Figure 10.3: Visualization of a workflow with data at different devices.

on a website after review and encoding. In a workflow like this it is unreasonable to expect all devices to replicate all work, for example, encoding on a smartphone is a bad idea, while servers cannot execute processing steps that require human interactions (such as reviewing). In Figure 10.3, the work done on each device is represented by downward arrows and replication by horizontal lines. As we can see, parts of the workflow exist on different devices, yet the overall workflow represents a single application with state and consistency requirements distributed across devices.

When proving causal consistency for $\mathscr{F}_r$ in Section 5.9 we show that it is required to synchronize the state of all replicated reactives at the same time. In the workflow example in Figure 10.3, this would amount to all boxes to be present on all devices. Moreover, state-based CRDTs usually require to send their complete state for synchronization – thus performance of the workflow example would suffer from a lot of unnecessary replication. To address this problem, REScala only synchronizes a replicated reactive, if both devices include the reactive in their flow graph. For example, the re-encoded video is not replicated back to the user doing the upload.

However, doing so naively breaks causal consistency in certain network topologies (those where there are multiple communication paths between two devices). As an example, consider Figure 10.4 where the same four devices from our workflow example operate on two replicated reactives A and B. Only the devices next to each other can communicate. The User device creates a transaction T_1 that changes both A and B. Because the Processing device only knows reactive A and the Moderator device only reactive B, the connection between the changes is lost when replicating the individual state, thus creating separate transactions T_2 and T_3. Applying T_2 and T_3 on the Web device as separate transactions means that the original guarantees of T_1 are violated. It is possible to have causal consistency even when only partially

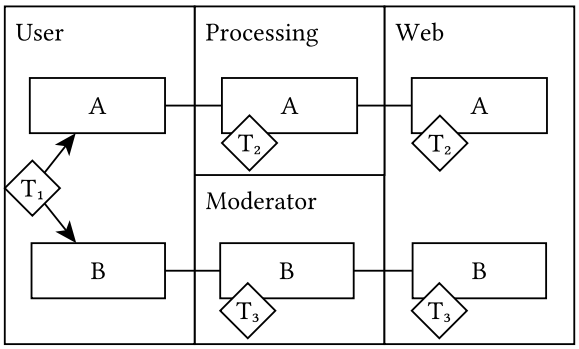

Figure 10.4: Loss of causality when transactions are split and later recombined.

replicating state by detecting such split transactions, but reasoning about such issues at the level of update propagations over individual replicated reactives turns out to be very unintuitive – the issues we care about are hidden between too many technical details. Instead of using our own solution, we believe that it should be possible to apply results about causally consistent delta replication for CRDTs [22].

A delta CRDT is a form of state-based CRDT, where each operation only returns its delta to the current state. A delta is a (small) state that, when merged with the current state of the CRDT, produces the full result of the operation. More precisely, for normal CRDTs, the sequence of states σ of a single replica of the CRDT is monotonically increasing $\sigma_1 \leq \sigma_2 \leq ... \leq \sigma_n$. Delta CRDTs, instead produce an ordered sequence of deltas $\delta_1, \delta_2, ..., \delta_n$ on each replica. The sum of the first i deltas is equal to σ_i, in other words, $\delta_1 = \sigma_1$; $merge(\delta_1, \delta_2) = \sigma_2$; ... or generally $merge(\delta_1, ..., \delta_i) = \sigma_i$. Note, all these operations represent the behavior of a CRDT at a single device, that is, all states and deltas above are implicitly scoped to one single device. Replication of delta CRDTs then only requires to transfer those deltas that are necessary and not yet available on another device, similar to our goal of only replicating the reactives a particular device is interested in.

The interesting part for us is that replication of delta CRDTs also suffers from loss of causality if used naively. State-based CRDTs normally guarantee causality because each state "includes" all smaller states. However, when deltas are merged and one of the deltas is missing (e.g., merging only $merge(\delta_1 + \delta_3) = \sigma_{13}$) then a state σ_{13} is computed that is not part of the total order of the original device. While consistency is restored when the missing delta is merged, the introduction of unordered states is exactly why causality is lost (e.g., σ_{13} may depend on results caused by changes in δ_2 without including that delta).

To guarantee causality, any remote device must merge deltas in the order they were produced by the origin device. This can be achieved in multiple ways, depending on the network conditions. The simplest solution is to use a transport protocol that ensures message ordering, such as TCP without disconnects. However, as a more interesting solution for our purposes, the metadata in CRDTs can be used to

detect state that can safely be merged [22]. The same mechanism can be used to ensure that two deltas that originate from the same transaction are applied as a single transaction on each receiving device, independently of the path of the deltas through the network.

While research on delta CRDTs may improve replication in Cvtr, our results should give an insight on how to compose delta CRDTs into larger CRDTs, which has not yet been sufficiently studied. Usually delta CRDTs store metadata about ordering of deltas for each CRDT without the possibility to combine those of multiple CRDTs. However, with Cvtr this metadata would be stored for each transaction, thus using the same metadata to apply delta messages of multiple CRDTs in a causally consistent order. This would also enable the use of delta CRDTs for individual reactives at no or little additional cost.

Paxos Made Eventually Consistent

The Cvtr paradigm is limited to causal consistency. However, dicApps can implement strong consistency as an application-level protocol. To demonstrate what that means, we have implemented the Raft algorithm as a CRDT in REScala. Raft [166] is a simplified version of the Paxos [132] algorithm (the simplified explanation of Paxos was still seen as too complicated). As an even more simplified explanation: Raft is a leader election algorithm, where a fixed set of participants agrees on a leader. The elected leader then propose values for sequentially increasing rounds of consensus. As long as a device is connected to a majority of the participants that device is guaranteed to see consistent results for each round.

So, how can a strongly consistent algorithm exist in a system that always provides availability? The hard part is only when someone wants to read a value, because that is when inconsistencies may arise. However, in Cvtr no values are ever read – the system only reacts to changes. This does not mean that we have finally made a distributed system that is both available and consistent. Availability is still lost if the progress of the application depends on the results of the consensus. To understand why, consider how consensus works. Each participant has a replica of the current state of the Raft algorithm, which includes information like votes for the leader, proposed values for the rounds of consensus, and votes for values. Each device may locally operate on the Raft CRDT independently, but for a proposed value to ever be accepted, the state needs to contain votes from at least half the participants. If those votes do not arrive at a device, that device will never see any vote succeeding.

This experiment shows an interesting insight: strong consistency can be implemented as a simple data type. The whole implementation is not at all concerned with low-level details, such as the existence of any network, messages, message orders, faults, none of that. The core algorithm is cleanly separated from the distributed nature of the runtime. Moreover, in the spirit of the rest of the chapter, this experiment proves that conceptually Cvtr does support applications that need strong consistency, however, at the usual cost of availability for the consistent parts.

10.5 Conclusion

The integration of CRDTs in Cvtr provides insights and advantages of both, while the limitation are of the two approaches are so similar that the cost does not have to be paid twice. Moreover, we do believe that Cvtr provides an insight into how the very useful operational properties of typical CRDTs – and ongoing research on CRDTs – can be applied beyond the classical way to express CRDTs. While REScala may or may not be the future of distributed programming, we believe the underlying approach of Cvtr is inevitable. There is a clear point in the design space of dicApps where causal consistency is the optimal choice (c.f. the CALM theorem [103]), but it is not yet clear what the optimal way to design such applications is. Cvtr shows that there is a lot of flexibility in the design space, without compromising the underlying guarantees.

Chapter 11

Case Studies

This chapter presents case studies to discuss the various aspects of REScala in the context of specific problems. Our case studies touch on a range of scenarios from traditional client-sever architectures to dynamic peer-to-peer networks. Our focus in this chapter is the impact of fault tolerance on the application design.

The TodoMVC (Section 11.1) and Conduit (Section 11.2) case studies are implementations of existing interactive example applications – a kind of common benchmark for UI frameworks. We chose these case studies, because they provide a fixed set of features that are required by interactive applications. Thus, implementing them shows that REScala provides all functionality that the communities around those case studies consider necessary for the development of interactive applications. We also use the first case study for a performance comparison with other implementations and show that REScala provides better or comparable performance for the same (or a stronger) guarantees.

The third case study (Section 11.3) also involves an interactive application similar to the first two, but the focus is on network communication. We dynamically switch between HTTP, WebSockets, or MQTT as the communication protocol depending on their availability. The application remains unaware of the underlying network protocol, thus demonstrating the flexibility of the Cvtr model regarding different runtime guarantees.

To understand the impact of fault tolerance on the design of applications, we look at REScala case studies that were designed prior to the research on Cvtr in Section 11.4. These case studies are written using reactive abstractions but without concern for distribution and faults. We study how the semantics of these applications would change in the worst-case scenario that all reactives are replicated, thus indicating how much additional effort is required to develop applications dealing with faults.

One of the existing case studies is a real-time game, which we discuss in more detail in Section 11.5. Games are a reasonable example for what users expectations are regarding real-time communication. Eventual consistency allows one player to make many moves without the other player getting a chance to react. Thus, the default semantics of REScala are not appropriate for this domain. Instead of

```
486 val taskData =
487   Events.foldAll(LastWriterWins(initial))(current => Seq(
488     doneEv >> { _ => current.map(_.toggle()) },
489     editStr >> { v => current.map(_.edit(v)) }
490   ))(uniqueId)
```

Figure 11.1: Implementation of the list of tasks.

eventual consistency, the use experience is improved by notifying the players about network issues and remove disconnected players from the game. We use the error propagation mechanism of REScala to add custom error handling to provide such functionality.

11.1 TodoMVC Case Study

This case study is an implementation of the TodoMVC [14] specification – extended with peer-to-peer data synchronization. We first discuss the benefits on the application code when fault tolerance is built into the programming model. We then compare the performance of our TodoMVC implementation with two other existing implementations that also add data synchronization between multiple devices.

TodoMVC is a to-do list application that is widely used to compare different languages and frameworks for programming interactive applications. The TodoMVC application shows a list of tasks, each representing one to-do item. Tasks can be added, changed, completed and removed. At the time of writing, the official website of TodoMVC presents 64 implementations in different languages and frameworks, and many more unofficial implementations (such as ours) exist. Synchronization of data across multiple clients is not part of the official feature set of TodoMVC, thus not all implementations support synchronization. However, we were able to find two implementations that do support synchronization and use them for comparison to REScala. One uses Twilio [127] to synchronize state with a centralized server, providing strong consistency, but prohibiting offline usage. The other uses Flask [189] to synchronize state with a central server, but does not provide any guarantees, i.e., changes may arbitrarily get lost. We discuss the three implementations (ours, Twilio, and Flask) individually before we compare them.

Our Implementation

To synchronize the full state of the application between devices, we implement both the list of tasks and each individual task as a replicated reactive.

Each task has its own “done” button and “edit” input box. The done and edit reactives are the inputs from which the task reactive in Figure 11.1 is derived. The done state and current text of the task are stored inside a single last-writer-wins CRDT [186] (Line 487). Tasks are short-lived and we do not expect many concurrent

```
491 val tasksRGA: Signal[RGA[Taskref]] =
492   Events.foldAll(RGA(initialTasks)) { tasks => Seq(
493     createTask >> {tasks.prepend},
494     removeAll.event >>> { dt => _ =>
495       tasks.filter(t => !dt.depend(t.contents).done)
496     },
497     tasks.value.map(_.removeClick) >> {
498       t => tasks.filter(_.id != t)
499     }
500   )}(implicitly, "tasklist")
```

Figure 11.2: Implementation of the list of tasks.

```
501 implicit val taskDecoder: Decoder[Taskref] =
502   Decoder.decodeTuple2[String, TaskData].map { case (s, td) =>
        maketask(td, s) }
503 implicit val taskEncoder: Encoder[Taskref] =
504   Encoder.encodeTuple2[String, TaskData].contramap[Taskref](tr =>
        (tr.id, tr.initial))
```

Figure 11.3: Encoder and decoder implementation for Tasks using Circe.

changes, thus we consider the last-writer-wins approach as sufficient even though only the last of any concurrent edits is kept. This type of fold reactive definition uses a list of handlers to handle each input individually. The task reactive has two handlers: one for the done input in Line 488 and one for the edit input in Line 489. Both handlers simply modify the state stored inside the last-writer-wins CRDT. The UI representing a single task is then derived from this task signal.

The list of tasks is a single replicated fold that reacts to adding tasks, removing tasks, and clearing all tasks. Figure 11.2 shows the code of the replicated fold reactive. The list of tasks is stored as a replicated growable array (RGA [186] in Line 491) that supports addition and removal of tasks at arbitrary positions and merges concurrent inserts and deletes. The fold reactive handles each of the three sources of changes for the list: creating tasks (Line 493), removing all tasks marked as done (Line 494), and removing each individual task (Line 497). This definition of the reactive contains declares both when and how the list of tasks changes. We want to stress that in the Cvtr paradigm the definition of all reactives always contains the exhaustive reasons of change for that reactive – it thus suffices to reason about single definitions of reactives at a time to understand their full behavior.

In addition to REScala this case study requires some other major components. The UI implementation is based on ScalaTags [135] for which REScala provides an integration library. The network transport layer is based on the work of Weisenburger et al. [202]. In general, the desired type of network communication varies

greatly between applications, from a web application using JSON messages over WebSockets to a server exchanging binary data on a TCP socket. Thus, REScala is independent of the implementation details of the concrete networking used.

An important part for synchronization is encoding of data structures to bytes. This part is handled by Circe [7], which derives codecs of simple data structures automatically. Developers only need to manually implement codecs for complex custom data structures, in particular for data structures with nested reactives. The list of tasks contains nested reactives (the tasks), thus we require a custom user-defined codec shown in Figure 11.3. The encoding is specific to Circe, the used encoding library. Circe requires an implicit encoder (Line 503) and decoder (Line 501). We define those by mapping from and to a pair of primitive values – the ID (`String`) and initial value of the task (`TaskData`). The pair can then be encoded automatically because it recursively only contains structured primitive data. The decoder (Line 501) is used both when restoring a task and also when receiving a new task over the network. The `maketask` function that is called during decoding first checks if a task with the same ID already exists locally if not simply proceeds with the normal application code that connects the new task with the UI.

Implementations Based on Twilio and Flask

Both implementations support recovery based on a central server. Thus, the application is unavailable when the device is offline. The two approaches use different approaches for state synchronization.

The Flask-based implementation replaces the local storage used by TodoMVC with a custom implementation that has a compatible interface. The result is that new or modified tasks are individually sent to a central server. The server orders all tasks based on arrival time. The application only requests state from the server during startup. That is, changes to the to-do list on one device become visible on other devices only when the application is restarted. Thus, no consistency is provided for the user.

The Twilio-based implementation has its own commercial implementation of a replicated list. The list has a primary replica on a central server that ensures consistency of the list. However, the replicated list requires local replicas to request confirmation for each change. Thus, Twilio ensures consistency at the cost of not supporting offline usage. Also, once the Twilio implementation displays more than 50 tasks at once, multiple devices become inconsistent when removing tasks, most likely due to an implementation bug in the application's UI.

Integration Effort for Fault Tolerance

We do not want to compare the design of the three implementations, because the other two make no claims about the quality of the application itself. However, all three approaches claim that their functionality is easy to integrate into the TodoMVC application. There are two types of changes required for fault tolerance.

The first are using data structures that are suitable for replication, and the second are additional code for network communication. We use lines of code as a proxy to measure the effort for these changes. While simple length is not a measure of complexity, all changes we find are not complex, but straightforward integrations between the network runtime and the application. We thus believe that lines of code is an acceptable metric. We also discuss if the additional code could be reused for other applications and what can be guaranteed.

Our implementation of TodoMVC consists of 276 lines of code. 73 of those lines are a custom UI and implementation we added to the case study that enables peer-to-peer connections and does the communication setup. This implementation could be shared between all web applications that want to use the same UI and network communication. The other two implementations do not have custom UI and use network communication that is built into the used library. Most of the remaining code expresses the application logic and is not specific to communication or fault tolerance. Out of the application logic only 5 lines of code accommodate the distribution of individual tasks by using CRDTs instead of plain data. Another 7 lines of code are required for the task list, including the custom codec implementation mapping each task to its ID and initial value (shown in Figure 11.3), the ID lookup, and the creation of new tasks during decoding. The rest of the application automatically reacts to changes from remote devices.

The Flask implementation has 268 lines of code. The implementation is based on one of the official versions of TodoMVC and only modifies 4 lines of code to exchange the local storage back end used by TodoMVC with the one used by Flask. However, the storage interface was not designed to support fault tolerance, which makes it impossible for the framework to provide synchronization features beyond remote storage in its alternative implementation. For example, the local storage API is not designed for values to be changed outside the application, thus the API does not provide any means to notify the application of remote changes. Even if notifications were available, handling them correctly would require restructuring the rest of the application.

The Twilio implementation has 818 lines of code. Out of these, 14 deal with authenticating with their central server, 32 implement routing for callback listeners when tasks are modified, and 70 implement the various interactions with the custom API of their distributed data structure. This boilerplate code is specific to this application, thus the functionality it supports cannot be easily moved to a library implementation.

In conclusion, the main issue we observe is the large amount of boilerplate code of the Twilio implementation. The underlying reason is that there are essentially two separate copies of all state in the application. One copy is used by the UI to display the current state of the user and the other copy is used by the Twilio library to replicate state to the central server. Most of the additional code in the Twilio implementation is then dedicate to ensure that changes applied to one copy are also applied to the other. In contrast, in REScala there is only one copy of the state used by both the application and for replication. This is possible because we separate the

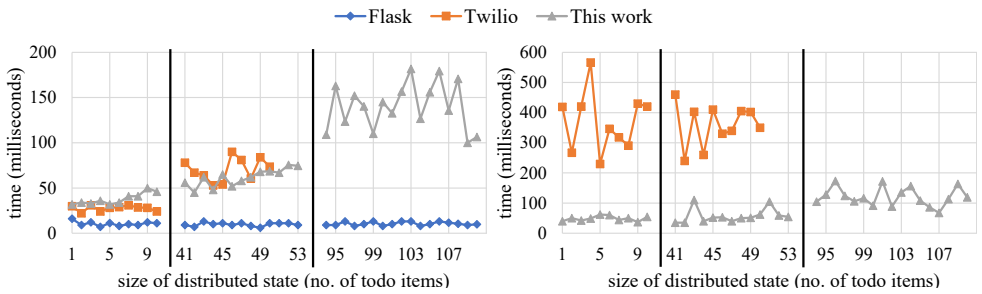

Figure 11.4: Time spent on computation (left) and network (right) when adding a new task.

logic for replication from the concrete implementation of the data structure, thus enabling the application to choose whatever data structure is right to represent its internal state.

Benchmark Setup

We use the Chromium developer tools (see browser setup in Section 13.1) to collect performance data and use reported script execution time as a measurement of computational overhead, and the idle time between events as a measure for network communication latency. All network connections happen in a gigabit LAN with less than 1 millisecond latency. The exception is the Twilio implementation, which uses its own external commercial service. We measured 1 to 2 ms latency to the Twilio cloud front servers, but we have no insight on how much network latency their API requests incur internally. This setup represents a typical execution environment for dicApps regarding computational power and platform. We minimize network latency as to not hide the overhead inherent to the approach, because both the Flask and the REScala implementations execute a single network message for each operation, thus any latency is simply added to the results of these measurements. While we cannot fully control the network conditions for the Twilio implementation, this accurately reflects a fact about using centralized services.

We study the performance of adding a single new task to task lists of different sizes. For REScala, we use a peer-to-peer connection between two devices running the same application. Even though REScala does not have to wait for remote confirmation to apply updates, we still measure the time until the local device receives a confirmation that the update has been applied remotely. Figure 11.4 shows the results of the experiment. The x-axis shows the size of the task list at which the operation happened, and the y-axis the time taken to add a single new task. The left graph shows the time spent on local computations. The right graph shows the round-trip time of sending tasks to the remote device, including processing time of the remote device.

Benchmark Results

The Flask-based implementation has the least computational overhead and the overhead is independent of the size of the list. Flask provides no fault tolerance and does not confirm that remote changes are applied, thus there is no network overhead to measure. The Twilio-based implementation is on the other extreme of the spectrum. When making changes, users must wait for both computation and communication to finish, because clients eventually receive a confirmation. This approach results in an average of 412 ms until any interaction produces a result. Note that the exact numbers heavily depend on a remote system outside our control.

In contrast to the Twilio-based implementation, REScala can display changes to user inputs without waiting for confirmation. Therefore, changes become visible after 109ms on average, which consists only of computational overhead. The computational overhead of REScala grows with the size of the task list because the list is stored in a replicated growable array (RGA), which has the most overhead of all CRDTs in our implementation. Concretely, our RGA implementation has an overhead of 357 bytes for added tasks, and 224 bytes for deleted tasks. However, the overhead is independent of the content of the tasks, i.e., of the actual size of application level objects, as those are synchronized separately. In terms of network communication, waiting for confirmation from the remote device requires a similar amount of time as a local change. This is expected, as the remote device mirrors the behavior of the local device before sending the confirmation.

Benchmark Summary

Our results show that both approaches providing some form of consistency have a negative impact on performance, but the performance of eventual consistency of REScala is comparable to strong consistency provided by Twilio. This is a very good result given that dealing with unreliable communication and crashes has a significant performance impact in the implementation of REScala. Moreover, performance in REScala does not depend on network conditions.

The performance of REScala is affected noticeably by the size of remotely shared data, compared to an approach that is independent of data sizes such as Flask. However, this is to a significant extent due the implementation of networking in REScala still being in its infancy, with two obvious major problems. First, currently encoding happens independently for snapshots and communication, which enables decoupling those features in the implementation, but causes duplicate computation. Detailed performance analysis shows that a list of size 100 is serialized in 23ms. We can address the problem by encoding each value only once and using it both for the snapshot and communication. The second issue stems from the growing state of the RGA that must be serialized for every small change to the list. Chapter 10 discusses how delta synchronization may help to drastically reduce this overhead.

11.2 Conduit Blogging Application

The main purpose of this case study is to show how REScala is used to implement a realistically sized collaborative application. Similar to TodoMVC, this case study is based on the RealWorld [9] set of example applications – a specification and a set of implementations for an online blogging application called Conduit.

The Conduit application simulates an online blogging platform where users can write articles, comment on these articles, read articles, and see various lists of articles (by user, by tag, by recency, etc.). Additionally, there are incidental features to allow registering new accounts, log into the system, and edit a user profile. Conduit is specified informally with examples and templates. The application is separated into the front end, the back end, and the API between those two.

We call our implementation of the case study Reconduit. It is a self-contained implementation of the front end and connects to existing back ends. Reconduit realizes the full specification of the client application – including navigation, multiple documents, and user accounts. The Conduit client specifies a custom protocol to interact with the server that our implementation has to interact with, thus exemplifying the ability of REScala to interact with existing application code. In addition, Reconduit supports offline use with a limited feature set (no downloads of new blog posts, no creation of accounts).

Reconduit has several views such as composing articles, reading articles, filtering by tags, or editing the profile. Each view corresponds to a single page of the web application and makes use of a subset of the shared content data such as articles, the currently used account, and comments. The original Conduit application fetches this data from the server on demand thus causing latency. In Reconduit the content data is mirrored locally and updated by background tasks whenever the user is online. Because views are derived from the content data reactives content is visible immediately when the application starts – even when offline – and us updated dynamically when a connection to a back end is available.

Web applications such as Conduit are typically developed in a mix of HTML and JavaScript. For REScala applications, we instead use our integrations with ScalaTags [135] to generate HTML and ScalaJS [67] to compile to JavaScript. Thus, the complete application is written as a single REScala application. There is no need to combine templates for HTML generation with any generated boilerplate to include the application logic. Compared to application frameworks that provide similar features to REScala – such as automatic updates, state restoration, and communication – REScala does not require learning new concepts specifically to web applications. Instead, we can reuse existing technologies for web development, because reactives are flexible enough to both express the application logic of the Conduit specification and integrate with external libraries.

Our language-based approach allows us to use the type and object system of Scala to structure the application. Applications such as Conduit have multiple views that a user may interact with at different times. For example, there is a list of blog post in summarized form with metadata and a full standalone page for each

post. Each view is represented as an independent module. REScala fills the gap when modules are composed to enable consistent transactions from one view to another and to represent the same data in multiple views consistently. Continuing the previous example, the state of the blog post is managed by REScala and the two views of the post are guaranteed to always display the same version.

Deriving each view from a single point of truth instead of individual pieces of state results in more polished experience for the user with less effort required from the developer. In the main Conduit implementation – which does not follow such a strategy – we could identify a case of inconsistency between views in the default implementation of Conduit with only manual testing. Of course, that Conduit implementation is only meant to demonstrate development methodology, thus only a moderate amount of effort is spent on finding bugs and usability, but that is why the programming paradigm should prevent such issues by automatically.

Another point is consistent restoration. The Conduit specification does not specify what the application state should be when the application is restored. The typical user expectation for a web application is that the view corresponding to the current URL is shown again when visited. For many other Conduit implementations there is only a very loose relation between the current URL and the current view. In contrast, the REScala implementation simply stores the application state representing the current view inside the URL as part of creating a snapshot. Thus, it is automatically ensured that the URL and the application state are always consistent. This, again, does not require developers to understand any specific framework and works by flexible use of existing REScala features.

Finally, the REScala implementation enhances the Conduit application with offline usage. In addition, REScala provides the infrastructure for users of the application to directly exchange blog posts. However, while peer-to-peer communication is technically feasible, it is unclear if it is desirable for such a website to allow spreading of content without the ability of (central) moderation. Integrating authentication and content filtering inside a CRDT is an interesting future direction for research, but outside the scope of this thesis.

In conclusion, the Reconduit implementation together with the TodoMVC implementation increases our confidence that REScala is widely applicable to typical problems in the area. We find that REScala integrates well with existing libraries for specific environment – such as HTML, storage back ends, and existing remote APIs. We never encountered issues with application logic that could not be expressed with REScala. Moreover, even if automatic guarantees are not required by the specification they enhance user visible parts of the application behavior such as offline usage and consistent behavior of different parts of the application.

11.3 Smart Street Light Citizen Application

The smart street light and the companion application for citizens are part of a demonstrator for the EmergenCity project. The functionality of the application includes browsing news and sharing pictures. Our citizen application is based on web

standards similar to the previous case studies, enabling a zero-installation distribution on all current citizen devices. Maximum compatibility and zero-installation are important requirements for our use case, to ensure that the application is available to every citizen in case of a crisis.

Our case study uses the following features of the smart street light: a Wi-Fi access point to connect to citizen devices and other nearby street lights, a wired connection to the internet, and various environmental sensors to enable autonomous detection of emergencies. Other features of the smart street light that are not directly relevant to our case study include illumination of the surrounding area.

The behavior of the companion application dynamically changes parts of its behavior depending on the available communication infrastructure and emergency status. The application has the goal of providing useful information over the internet to citizens during normal operations in "everyday mode" and to seamlessly switch to alternative communication channels in "emergency mode".

Everyday Mode

During everyday mode, the example application connects to a city-wide central server to acquire a feed of local information. We use city-wide news – similar to RSS or local Twitter feeds – as an example. Information can further include city-wide traffic information, public transport, availability of services such as bike-sharing, and other services of the digital city. Providing functionality for the everyday mode is necessary to give citizens a strong incentive for already using the application before a crisis, to ensure they are familiar with its usage, and to make it readily available. We use a client-server topology for the everyday mode to ensure low latency of updates, high quality of the provided information, and low communication overhead.

Emergency Mode

The application switches to emergency mode, when instructed to do so by the crisis detection mechanism of a local smart street light, or when the application itself encounters permanent connection issues attributed to an emergency situation. To synchronization replicated reactives during an emergency, the application on citizen devices connects to the local access point of nearby street light and directly to other citizen devices if possible. The exchange of information is no longer just between the central service and the end user devices, but the device start exchanging information with the smart street light in both directions. That is, both the citizen device obtains updates from the smart street light and the street light is updated by the citizen device if the device has newer information. The smart street light relays such information to other connected devices. The reason for bidirectional communication in a crisis situation is that only a few mobile phones may still enjoy connectivity to a cellular network to receive new information. In such a situation, the one connected device provides new information to the local streetlight and thus other nearby devices.

In addition to bidirectional communication with the street light individual devices opportunistically establish direct connections with other local devices to ensure that communication does not rely on the availability of the smart street light. From the application developers point of view all changes from the outside appear as changes to the respective replicated reactives. The uniformity alleviates application developers from the burden of developing, testing, and maintaining different applications for everyday and emergency mode.

The application may additionally provide functionality specific for emergency use cases. The emergency state itself is simply another source reactive that is updated by the network runtime developed for this emergency scenario. As an example for extended functionality, instead of only consuming information about the city and current situation, the application enables citizens to directly communicate with other citizens in the vicinity and with first responders during emergency situations. Citizens may locally disseminate information that is highly relevant for their current area (e.g., the location and the nature of emergency sites, or where help is needed or available) and which can no longer be made available through a central server due to the collapse of everyday communication infrastructure. To stop the network from overloading, user generated information is only disseminated from the source to directly connected devices, but those devices do not automatically propagate the information further. However, first responders have special permissions to share global information, which is distributed from application to application using any of the aforementioned communication channels.

Deployment and Network Runtime

The application is deployed via HTTPS using Web technology available on all common citizen devices. With a typical application size of less than 1 MB it is feasible to deploy the application over mobile networks. The application code can also be acquired directly from the smart street light. Technical advanced users are able to share the application directly from device to device using any ad-hoc communication channel still available during the emergency.

This case study uses a custom network runtime for communication. The runtime abstracts over the concrete used protocol – WebSockets for communication with the central server, WebRTC for communication between citizen devices, and MQTT for communication with the smart street light. Due to the flexibility of replicated reactives the only requirement of a network runtime is to deliver some messages – there is no need for the runtime to ensure delivery or order messages. Thus, it is easy to customize the network runtime to specific use cases, because most existing protocols can be directly used without requiring additional ordering or delivery guarantees.

Figure 11.5 shows an example message flow of the deployed application. First, communication happens via a central server where the city sends messages to the street light that are forwarded to the citizen application. When an emergency occurs and messages form the city no longer reach the street light, then citizen devices start communicating directly with each other.

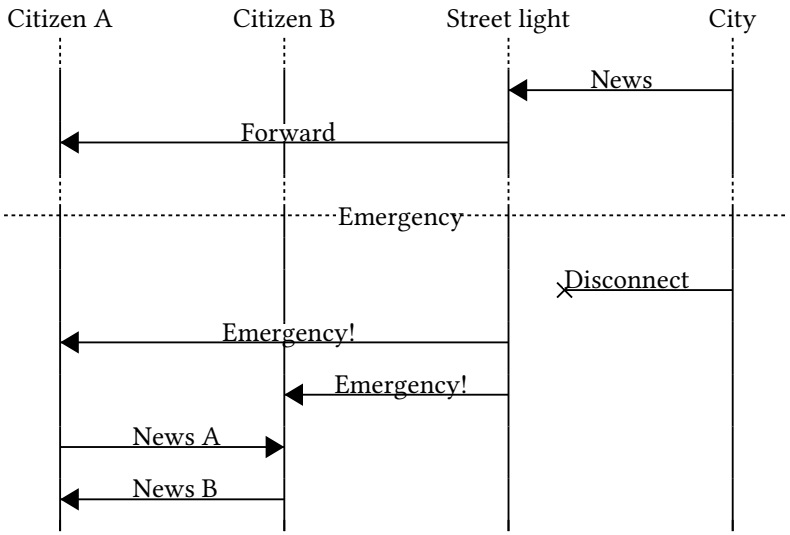

Figure 11.5: Communication in our network runtime.

We limit potential abuse for communication by only automatically forwarding messages from known participants, such as first responders. These known participants are determined by pre-shared public keys acquired during everyday operations. Other messages must be reviewed by the current user of the device to be forwarded to other devices.

In general, emergency mode requires additional resources as compared to everyday mode, such as storage and network from the devices of citizens, because those devices have to replicate some of the unavailable infrastructure. However, we only assume emergencies to last for a short time, a couple of days at most, after which normal operations resume, and additional resources are freed again.

Evaluation of Network Overhead

We evaluate our citizen application with respect to message sizes and message delays in Figure 11.6. We start at 10 news articles (number of messages) and continually add messages to the shared state of the application. We measured the time it takes for a single remote mobile device to acknowledge that a new article was received, processed, displayed to the user, and forwarded to other devices in the network. The combined time (round trip) of these operations stays under 100 ms during our measurements, and most of that time is spent on actually processing the messages. Due to the way synchronization of our infrastructure works and to ensure that all participants eventually have all state available, message sizes continually grow when more state is added. This size can be reduced by purging old messages, e.g., once they become irrelevant because they are out of date.

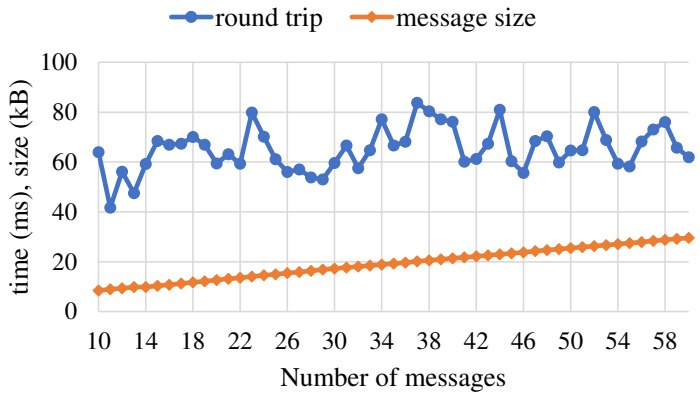

Figure 11.6: Round trip times and message sizes for increasingly more messages.

Conclusion

The requirements of the emergency scenario are unusual, because there is a conflict of interests between developers that may not want to spend effort testing and developing for emergency scenarios due to the high cost and citizens that depend on applications during an emergency. Thus, we do believe that programming paradigms with automatic fault tolerance such as Cvtr provide a workable compromise in such situations – developers have no additional cost while citizens do gain additional reliability. Furthermore, we as the designers of REScala only have to consider a new scenario such as the emergency scenario once and there is no additional cost for each application.

11.4 Influence of Fault Tolerance on Legacy Applications

REScala has a number of case studies that investigate how to improve the design of interactive applications compared to solutions using imperative paradigms – especially compared to the observer pattern [182]. However, none of these case studies were designed with consideration for faults. Using the assumption that those applications are well-designed for systems without faults, we analyze the impact of faults on the behavior of applications. To be clear, fault tolerance in REScala is fully automatic, but these applications may rely on the assumption that transactions are executed sequentially. This assumption is not true for replicated reactives. For this experiment, we analyze the case studies under the worst case assumption that all sources and fold reactives no longer execute transactions sequentially, but instead arbitrarily merge and reorder transactions. Our claim is that the changed assumption – no sequential execution of transactions – only has minimal impact on existing applications implemented with REScala. Thus showing that applications are automatically made fault tolerant. To validate this claim, we answer the following research questions:

Case study	`observe`		`fire`		`change`		**Total**		Description
	ok	nok	ok	nok	ok	nok	ok	nok	
CRDTs			9				9		CRDT integration
Datastructures			5				5		Reactive collections
Dividi			1		3		4		P2P distributed ledger
Editor			42		10	1	52	1	Swing text editor
Examples			39	2	9	19	48	21	Swing/console examples
Mill game			14		7	1	21	1	Turn based swing UI
Pong game	3		15	5	4	5	22	10	Multiplayer swing game
Reactive streams			1				1		Interface integration
Scalafx	3		1				4		JavaFX integration
Scalatags	2				1		3		HTML DOM integration
Swing			2			2	2	2	Swing integration
RSS			15		4		19		Swing RSS reader
Shapes	1		17		4	1	22	1	Swing drawing app
Todolist			11				11		TodoMVC app
Universe		1	8		2	9	10	10	Console simulation
Total	9	1	180	7	44	38	233	46	

Figure 11.7: Possibly problematic operators in case studies and extensions.

RQ1 To what extent do snapshots and restoration affect the application syntax and semantics?

RQ2 To what extent does the integration of replicated signals into the flow graph affect the application syntax and semantics?

To answer these questions, we analyze a set of case studies, consisting of ten applications (including games, simulations, and GUI applications) and five integrations with external libraries (e.g., an API to access the HTML DOM, bindings for JavaFX and for Java Flow), comprising a total of 13.000 LoC (counted with CLOC excluding comments and blank lines). The case studies are listed in Figure 11.7 and their code is publicly available from the REScala webpage.

(RQ1) Effects of State Snapshotting/Restoration on Application Semantics

Snapshotting is invisible to an application, since snapshots are automatically created at the end of a transaction. Restoration, on the other hand, is visible to the application, since restoration re-executes the application to restore the flow graph (cf. Section 3.2). The value of signals may differ between its first (normal) start and a restoration, causing different application behavior. Furthermore, certain inputs to the flow graph may be duplicated while restoring. For example, if a calendar application were to add a new calendar entry every time it starts, then this new entry would be duplicated when the application is restored. We refer to problems with different behavior during restoration as *restoration inconsistency*.

To quantify the extent of restoration inconsistencies, we inspect all input and output interactions of imperative code with the flow graph in our case studies.

These interactions are easy to localize, since they occur via a well-defined interface of the flow graph, consisting of the operations `fire` and `observe`. The columns for `fire` (input interactions) and `observe` (output interactions) of Figure 11.7 summarize our findings.

Firing events on some occurrence in the external world via the `fire` operations serves the purpose of entering new values into the flow graph, e.g., a user clicking a button, time passing, or receiving a network message. In our case studies, 180 out of 187 `fire` calls serve such a purpose and are not affected by state restoration. The 7 remaining calls that do exhibit the restoration inconsistency problem are instances of the same event usage anti-pattern: they incrementally build state during application startup. For example, the Pong game initializes the UI elements, and adds them one by one to a list of all UI elements, as shown below. As a result, this list would grow after each restoration.

```
505 val addElement = Evt[UIElement]
506 val allUIelements: Signal[List[UIElement]] = addElement.list()
507 addElement.fire(ball); addElement.fire(player1); ...
```

Firing of events must not be misused for initializing reactives. Manual inspection of usages of the `fire` method is required to find such misuses.

We analyzed if `observe` calls on signals cause inconsistencies during restoration. We found a total of 10 usages of signal observers in the case studies. Event observers are more common with 150 usages, but are never triggered during restoration, thus never cause inconsistencies. Out of those 10 signal observers, 9 are not affected by restoration inconsistencies. 7 of them are in bindings for external libraries and are used to set properties of UI toolkits, e.g., the window title as in `titleText.observe(UI.window.setTitle)`. Triggering these observers during restoration correctly causes the UI to display the restored state. Two observers execute cleanup code, which is not affected by restoration either. The only observer that is affected by restoration inconsistency is in a simulation application (`Universe` row in Figure 11.7). The simulation uses mutable state outside REScala, and if a fault occurs during a simulation step, this state is not restored.

We conclude that the state snapshotting/restoration feature of our approach operates mostly transparently. This means: (a) most of the potentially problematic interactions (181 out of 189, roughly 96%) are unproblematic in our fault-tolerant runtime, (b) the few problematic cases can be avoided, if application developers use the correct APIs of the flow graph, and ensure that mutable state outside REScala is also able to tolerate faults.

(RQ2) The Effect of Introducing Eventually Consistent Updates

Eventually consistent updates may affect the behavior of existing applications in two ways. First, they break the invariant that each occurrence of an input `Evt` is handled individually. Instead, after devices were disconnected for a while, all changes are replicated as a single large change to other devices. These combined

changes cause problems when the application expects each change individually, e.g., if our shared calendar were to display a notification each time an entry is added, the notification may be triggered for a group of entries, instead of each individual entry, and as a result, the notification system has to be able to handle multiple entries at once.

Second, they break assumptions that usages of the `change` operator on signals may make about its behavior. The `change` operator is used to reify and handle each individual change of a signal, and usages of `change` may assume that every intermediate change of the signal will occur individually. However, with eventual consistency intermediate changes may be grouped as described above, hence, assumptions on individual changes become invalid. For illustration, consider a simple clock implemented as below. The computation of `minute` relies on `second` change to 0. However, with eventually consistent propagation `second` could change from 59 to 2 skipping the intermediate step, because an aggregated update is received over the network, resulting in a missed minute.

```
508 val tick: Event[Unit] = ... // fires once per second
509 val second = Signal { tick.count() % 60 }
510 val minute = Signal { seconds.change.filter(_ == 0).count() % 60 }
```

To quantify to which extent the introduction of replicated signals affects the application semantics due to the existence of `change` operations on signals, we investigate whether the semantics of our case studies relies on each individual signal change being visible, as opposed to relying on a notification about its latest change. The results of this analysis are shown in the *change* column of Figure 11.7. Roughly 46% of `change` operators (38 out of 82 in 7 out of 15 case studies) have different behavior when individual changes are grouped or skipped due to eventual consistency. The results indicate that replicated signals with eventual consistent semantics cannot be introduced transparently, which, in fact, is not surprising. However, there are several mitigations depending on the needs of the application. The first is to keep computations that require strong consistency on a single device, and only distribute their results via replicated signals. Second, as discussed in Section 10.4, it is possible to use a CRDT implementation that does provide strong consistency on the application level. A custom replicated data type allows applications to decide their own requirements. Third, the application logic may be slightly rephrased to accommodate for unordered transactions. For example, the time management above could derive the current minute directly from the counted ticks (using `tick.count() % 3600`).

11.5 Error Propagation and the Pong Case Study

We use the Pong case study – a simple multiplayer game – to discuss the error propagation as discussed in Chapter 7 on a concrete example. The classic Pong game requires real time communication to provide an enjoyable experience, thus the application is not suited to the default strategy of eventual consistency in REScala.

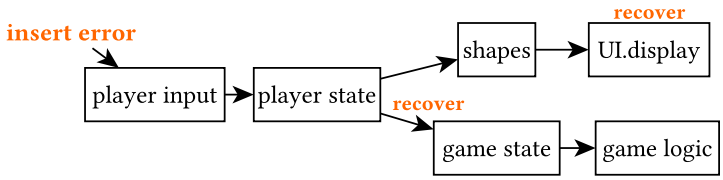

Figure 11.8: Recovering from errors in Pong.

Our version of Pong supports an arbitrary amount of players, and we want any disconnected players to be removed from the game until they reconnect.

The integration of error propagation into transactions allows propagating errors mostly transparently. Additional code is required only at specific places where the developer wants to handle errors. The key point is that intermediate reactives do not have to be updated to propagate the error, minimizing the total amount of application code that requires modification. Especially important to us is that error propagation does not "pollute" application code and instead only requires changes directly necessary to improve the user experience.

The case study consists of two application windows, one for each player. Without handling faults, if one player dropped, the game would get stuck or simply terminate. Figure 11.8 shows an abstract representation of the flow graph of the case study. Altogether, we update the game at three locations out of the 250 total LoCs. To evaluate error handling in REScala, we added functionality to allow players to leave and join the game. When a player disconnects, an error gets inserted into the position signal of the racket of that player:

```
511 UI.onClose{ Racket.pos.admit(PlayerDisconnected)  }
```

Following the dataflow of `Racket.pos` through the flow graph of the application, one can identify the places where the error needs to be handled. There are two such locations: when displaying the players on the screen and inside the game logic handling cleanup of data structures for disconnected players.

For handling the error when displaying the players, we reused an existing try/-catch block that handled missing game objects and added a handler for the `PlayerDisconnected` exception. Note that handling the disconnected event may require a different handler if the game should support different failure modes than just removing the player.

```
512 case _: NoSuchElementException |  _ : PlayerDisconnected =>
513 // remaining handler unchanged
```

As a final modification to the code, failed connections are observed and the corresponding player is removed from the game. To remove the player, a list of disconnected players is derived from the list of players, by filtering on the player connection:

```
514 val disconnectedPlayers = Signal{ players.value.filter { p =>
        Try(p.connection.value).isFailure} }
515 disconnectedPlayers.observe(Game.removePlayers)
```

If accessing the connection raises an error (checked with `Try(...).isFailure`), then the player is considered disconnected. The resulting list of failed players is observed and these players are removed from the game (closing the connection and updating the list of players).

Error propagation in Pong is particularly effective, because the application was already able to handle a dynamic amount of players, thus errors could reuse existing handling logic. Still, this shows that the error propagation enables errors to be handled at the specific place where there is enough information to do so. In addition, the case study clearly demonstrates the advantage of integrated error handling, when most of the application is not concerned with errors and thus does not need to be changed. Finally – while we believe causal consistency to be better suited for dicApps compared to explicit error handling – applications with some form of real-time requirement or other reasons that causal consistency is unsuited can take advantage of the option for manual error handling. Thus, error propagation does widen the applicability of REScala.

11.6 Conclusion

Our case studies repeatedly show that REScala is very well suited to implement dicApps. The provided abstractions are sufficient to model complex application logic, the resulting application code is easy to reason about, and provides automatic guarantees for fault tolerance.

Performance of REScala has never been an issue in any of our target applications, however, our results indicate that encoding application state into bytes for storage or synchronization will likely be the first bottleneck. Potential improvements include more efficient codecs [8] and ubiquitous use of delta encoding for all state (see Section 10.4). It is possible that future work can achieve better performance in these cases by integrating the data representation tightly into the programming paradigm, instead of delegating this to external implementations as is currently the case.

We have also shown that it is not necessary for developers to only use REScala to implement an application. Most case studies are implemented in other paradigms to some extent, enabling developers to choose whatever paradigms fits the application.

Chapter 12

Experience Using REScala

The reactive part of REScala is based on a long tradition of (functional) reactive programming (RP) and in this chapter we informally summarize our high-level experience using the paradigm. In general, RP is a paradigm that aims to expressing time-changing, interactive applications in purely functional languages. RP has been studied intensively [59, 65, 69, 75, 76, 108, 164, 168, 179, 182, 185, 196, 201] and has spread from the original purely functional setting into imperative and object-oriented languages [37, 61, 146, 181].

However, only little literature exists on developing applications in RP beyond the small case studies used to exemplify or evaluate each new RP language (similar to our contributions in Chapter 11) and an empirical study [182]. Thus, not much is known qualitatively about the programming experience with RP. This makes it hard to evaluate if a RP language fits a specific domain, to compare different RP language designs, or even to build knowledge about recurring problems when developing with an RP language.

Over the years, several people have used and contributed to REScala. We derive the insights discussed in this chapter from personal discussions with those contributors, inspecting contributed code, and our own experience from using REScala. A detailed list of contributors can be found in the REScala repository.

We first discuss common idioms and patterns and then discuss what influence our findings have on the design of RP languages. For idioms and patterns, we investigate usage experience with patterns emerged when using RP both in a purely functional setting and when including RP as part of imperative applications. For the language design, we report on our experience with REScala specifically, both as users and developers of the language, to provide insights on how choices in the RP language design influence its use in reactive applications. We believe that sharing our experience helps researchers to understand how RP is used effectively. It also guides developers to decide whether RP is applicable in their domain, and it supports designers of RP languages considering design trade-offs.

12.1 Idioms and Patterns

Certain idioms and patterns seem to reoccur in reactive programs, and we elaborate how they help with maintainability and testing of RP applications.

Idiomatic Code

As every programming paradigm, RP has a specific coding style that takes time to learn and refine. Developers with an imperative background perform side effects on shared state rather than opting for functional processing of event streams. For example, to count how often an event is fired, a mutable variable is updated inside a `map` expression, and the value of the variable is propagated:

```
516 var count = 0
517 val mapped: Event[Int] = event.map { _ =>
518   count += 1 // bad: mutation inside operator
519   count
520 }
```

However, mutations of `count` outside the user-defined function for the `map` operator are propagated. The idiomatic REScala solution is to use `fold`, an operator with internal state managed by the reactive language, that applies a user defined function to generate the new state from the old state:

```
521 val counted: Signal[Int] =
522   event.fold(0) { (count, _) => count + 1 }
523 // or just ‘event.count‘
```

In our experience, fold is adequate to model any stateful computation, but it requires programmers to understand purely functional state management (e.g., using the accumulator and return value to manage state). In some cases, it is necessary to handle multiple events in the same `fold` to make the state accessible to all event handlers. Examples are user interfaces, where a user modifies the same value by entering text, or dragging a slider. Our case studies include an example where an elevator computes the time it has spent waiting on the current floor, depending on a `reachedFloor` event, which resets the value, and a `tick` event, which increases the value:

```
524 val waitingTime: Signal[Int] =
525   reachedFloor.reset(0) { _ =>
526     tick.iterate(0) { acc =>
527       if (isWaiting()) acc + 1 else acc
528   }}
```

The code above requires a nested call of `iterate` inside a `reset` both of which have semantics derived from fold. Nesting complex operators is potentially hard to understand even for seasoned RP developers. Fortunately, there is a more idiomatic alternative when folding over multiple events. REScala supports an extended syntax

for folds, which takes a list of events and associated update functions, to compute the next state from the current one:

```
529 val waitingTime: Signal[Int] =
530   Events.fold(0)( acc => Seq(
531     reachedFloor >> {_ => 0},
532     tick         >> {_ => if (isWaiting()) acc + 1 else acc }
533   ))
```

The extended fold states the intention more clearly. Each event is paired with a handler function (Line 531) describing the behavior of the fold when the corresponding event occurs. The `reachedFloor` resets the state to one, and `tick` increments the state by one if the elevator is currently waiting. When multiple events occur at the same time, the handlers are executed from top to bottom.

While this is only an anecdotal observation derived from our experience with the case studies, the correct use of fold seems to be a common problem when developers with an imperative background write applications with RP. However, developers seem to know that mutable state is problematic and they try to use `fold`, but they often end up with complex and hard to understand `fold` expressions. We believe the issue to be a lack of available examples and references for writing readable complex fold expressions.

Maintenance

For maintainability, we want to highlight one crucial part of RP: statically known dependency relations with automatic update propagation. As a contrived example – for the sake of brevity – consider a door system that imperatively turns the lights on or off, when the doors open or close:

```
534 var light = On
535 object DoorSystem {
536   def onClose() = { light = Off }
537   def onOpen() = { light = On }
538 }
```

Other objects rely on the state of `light` and must be notified when it changes, but the API does not specify how notifications happen. With RP dependencies are made explicit, and notifications happen automatically:

```
539 object DoorSystem {
540   val closed: Signal[Boolean]
541 }
542 val light = Signal {
543   if (DoorSystem.closed()) Off else On
544 }
```

In the updated example, the `closed` state of the door is a proper part of the public API of the `DoorSystem` object, including the fact that the state changes over time.

The example generalizes to bigger projects. RP operators integrate seamlessly into existing APIs, and work with existing tools: the type system, IDE-based refactoring, linters, etc.

Testing with Reactive Programming

Our experience with testing in the cases studies is limited to unit tests. We observed no particular difficulties: Testing RP is not different from testing code in the object-oriented paradigm. RP operators are tested at a similar granularity as method calls, and tests exercise the operators and observe the resulting values, using imperative accessors. For events, it is often useful to use the `list` operator (Lines 549, 551) to make all occurrences of the event accessible as a list:

```
545 val input = Evt[String]
546 val greeting: Event[String] =
547   input.map(name => s"Hello $name")
548 val inputLog: Signal[List[String]] =
549   input.list()
550 val greetingLog: Event[List[String]] =
551   greeting.list()
552
553 assert(inputLog.now == Nil)
554 assert(greetingLog.now == Nil)
555
556 input.fire("world")
557
558 assert(inputLog.now == List("world"))
559 assert(greetingLog.now == List("Hello world"))
```

RP programs are written in a more functional style composing behavior based on small individual functions, such as the `map` in Line 547.

To test applications the language provides tools similar to mock based solutions in object-oriented languages. It is possible to simulate the value of individual events or signals to test derived operators on specific inputs. Assume that the value of some `input` signal is not under control of the tests, but a derived signal `testMe` is tested:

```
560 val input: Signal[String] = ...
561 val testMe = Signal { input().toLowerCase  }
562
563 reevaluate(testMe).assuming(input -> "TEST String")
564 assert(testMe.now == "test string")
```

The code in Line 563 forces a reevaluation of the signal, and simulates the values of the inputs of the signal. Using this technique allows very fine granular tests without modifying the code to be tested, and we believe the strategy supports testing arbitrarily complex applications, much bigger that the examples in our case studies.

12.2 Considerations on Language Design

As with any language RP evolves with its usage. In this section we discuss considerations when designing an RP language, and show how our case studies guide the design of the language.

API Size

We discuss multiple RP languages and the size of their APIs, and how API design relates to the intended use cases and concepts.

FElm [65] is designed for asynchronous composition of GUIs, and requires three operators for this purpose, combining signals, folding over past values, and making a computation asynchronous. Fran [75] is designed for animations, a similar use case to FElm. Fran does not include asynchronous computations, but has operators for transforming time, and dynamically recombining signals, required to change the speed and behavior of animations. In total, the Fran API lists around 10 to 20 operators in the API, including some minor variations of common operators.

In contrast, reactive extensions [136], includes operators specific to collections, asynchronous execution, and mathematical aggregation. The result is a library with over 450 operators [11], 80 of which are considered *core operators*. Core operators are those that exist not only because of multiple overloads, such as `averageInteger` and `averageDouble`.

REScala does not target a specific application domain, and includes operators for combining events, signals, and converting between the two, as well as operators which integrate with imperative code. On the other hand, REScala refrains from including operators for a specific domain, such as mathematical aggregations. In total REScala has an API size of around 40 operators. About half are convenience operators for events, which form a small event processing sub language. The inclusion of operators into the REScala API is driven by our case studies. Most of the REScala operators are generic and used in multiple case studies – their usage is not limited to specific domains. Other operators such as `list` (c.f., Section 12.1) are mainly used during testing, but are so useful there to warrant their inclusion. The case studies often derive more complex or specific operators from the basic ones, however none of them are common enough to warrant inclusion into REScala.

API Design

RP languages minimize the number of concepts necessary to use their API, exposing only events and signals abstractions to programmers. As we discuss in Section 12.2, RP languages offer a rich set of operators in their API, building on top of events and signals. The derived operators have a semantics that is expressible as (a combination of) basic concepts and such semantics in REScala directly corresponds to the operator implementation.

For example, consider the `||` operator in REScala, which is derived from the following event expression:

```
565 def || [A](a: Event[A], b: Event[A]): Event[A] =
566   Event { if (a.value.nonEmpty) a.value else b.value }
```

Accessing the value of an event inside an event expression, yields a Scala `Option` representing the value of the event. The event expression above defines, that the result of `a || b` is a new event `e` with the following behavior: If the reactive `a` activates with some value (`a` is `nonEmpty`), then `e` activates with the value of `a`, otherwise `e` activates with the value of `b` (if `b` is also empty, then `e` does not activate). Specifying operators using a small core set of concepts, makes the language much more accessible for novices and allows one to easily explain the semantics of complex programs. Note that in $\mathscr{F}_r$ (c.f., Chapter 4) we opted to provide one method for deriving a reactive for each unique feature in the calculus (map, fold, flatten, filter), but their only difference is how they initialize the reactive they create. The reactives are the single core concept used for semantics in both REScala and Cvtr.

Purity in (Functional) Reactive Programming

RP has been initially developed in the context of pure functional languages, and operators in RP are still free of side effects. However, imperative languages allow to imperatively change the graph during execution. Consider creating a button for a UI, deriving a label signal that states how often the button was clicked, and displaying the label text on the button itself:

```
567 val button = new Button
568 val label: Signal[String] = button.click.count
569   .map(c => s"This button was clicked $c times")
570 label.observe(txt => button.text = txt)
```

The last line sets the text of the button when the label changes – an imperative interaction not directly allowed in a pure language. In pure languages, the label text of the button has to be provided when the button is created, which leads to a cyclic definition (button depends on label, and label on button). A solution is shown in the following code example where the handler is registered as part of the button creation and no imperative change is required:

```
571 def pipeline(click: Event[Unit]): Signal[String] =
572   click.count
573   .map(c => s"This button was clicked $c times")
574 val button = new Button(pipeline)
```

We provide a function to the button, and the function describes the pipeline to process the button clicks. With this approach the created pipeline is private to the button. To share events and signals between multiple UI components (buttons, labels, etc.), they all have to be initialized together with a combined and predefined pipeline of operators. As a result, purely functional RP languages may exhibit non-modular design constraints and only support limited use cases, such as a single animation or a single UI window. Elm [65] applied functional reactive programming

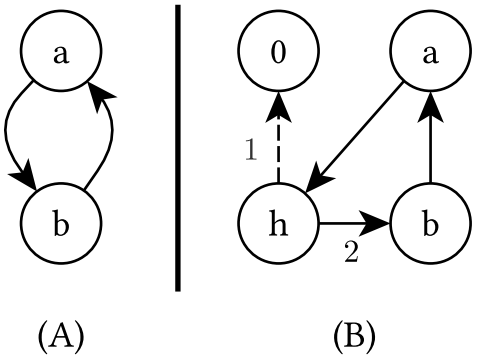

Figure 12.1: Creating a cycle

to web applications. However, to keep purity, the RP abstractions were removed from Elm, as they suffered from the problems we explained. In REScala we make the opposite choice and sacrificed purity to keep the abstractions of RP and integrate them with imperative applications. The integration allows programmers to add and reconnect events and signals as required.

Cycles

RP languages require that the flow graph is acyclic to ensure that propagation terminates, but most RP languages do not statically enforce the graph to be acyclic. However, problems with cycles only rarely occur in practice. Consider the following code to create a cycle between signals a and b:

```
575 val a = Signal { b.value + 1 }
576 val b = Signal { a.value - 1 }
```

Figure 12.1(A) shows the cycle the code is supposed to create. However, the code does not compile for REScala, because the embedding language prevents cyclic definitions of variables. To create a cyclic graph, imperative code is used:

```
577 val h = Var(Signal(0))
578 val a = Signal { h.value.value + 1 }
579 val b = Signal { a.value - 1 }
580 h.set(b)
```

Figure 12.1(B) illustrates this approach. The first line creates the source signal named `h` containing the constant signal with value 0. The signal `a` depends on `h` and the inner value of `h` – `h.value` accesses the value of `h` and `h.value.value` additionally accesses the inner value. The last line changes `h` to point to `b`.

Code that explicitly sets vars to other reactives is rare in practice. In our case studies, we find that only the universe simulation and the pong game have the potential for cycles. In both cases, the cycle is related to the update steps of the

simulation, i.e., the position of the pong ball depends on the speed, and the speed depends on the position (the speed changes when the ball bounces of a wall). In both cases, the creation of cycles is explicitly avoided.

12.3 Conclusion

Syntactically, Cvtr adds only replicated reactives and snapshots to RP. Thus, we believe that the remarks of this chapter still apply to Cvtr. Our general experience is that idiomatic RP code also is an improvement for dicApps and results in the guarantees of Cvtr to fully apply. Integrations with other libraries and the applicability of RP is only expanded by Cvtr to include collaboration in distributed systems. The overall design decisions of Cvtr align well with RP – we still strive to maintain an API with a minimal amount of different concepts to make code more readable for developers.

Chapter 13

Performance Experiments

This chapter contains detailed performance evaluation of individual parts and features of REScala. The evaluation often compares cost of large operations that are still only in the order of microseconds. We do believe that small improvements for fundamental technologies such as the implementation of a programming paradigm are worthwhile. However, we first want to discuss several rough estimations that should give an intuition of what kinds of dicApps the performance of REScala is suitable for.

First, it is important to note that REScala is typically not a performance critical part of dicApps as shown by our case studies (Chapter 11). The programming paradigm describes the outermost layer of an application that reacts to individual user inputs – transactions in the flow graph are normally not executed inside tight loops as is the case with these performance experiments. To give an intuition a 144Hz Monitor renders a frame every 7 milliseconds. A transaction changing 100 reactives (more than required for any of our interactive case studies) takes roughly 11 microseconds using the LaptopHigh configuration detailed below. That is, the application has 99.84% of its time budget remaining for something that is not REScala. See Figure 13.1 for a visualization to get a better impression of just how little time a transaction requires.

A different consideration compared to execution time is memory usage. The required memory of dicApps is often a binary problem: either a device has enough memory to run an application or not. The required memory for a single reactive in the flow graph is about 400 bytes on the JVM. We have successfully experimented with creating graphs containing up to 30 million reactives requiring about 12 GB to store the graph. A device then requires several more gigabytes of memory to

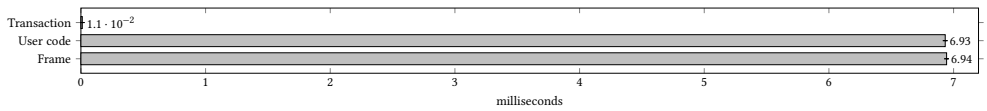

Figure 13.1: Frame time versus transaction time.

process transactions over all the reactives. The processing time of such large transactions depends on the graph layout. In the worst case layout, transactions of 10 million reactives are possible in the order of seconds on our LaptopHigh system (see below). Very wide graphs cause more problems to the propagation algorithm, because all reactives at the same depth in the graph are added at the same time into a priority queue resulting in a lot of computation time spend on adding and removing elements from that queue. We have no indication that any realistic case study will ever reach close to these numbers of reactives, thus we do not concern ourselves with the memory requirement of the flow graph.

Regarding memory requirement, the state of a single reactive – especially of CRDT reactives – may quickly reach dozens of kilobytes. Encoding of such large data structures into bytes (for snapshots and synchronization) may take more than the target frame time of 7 milliseconds – especially when executing on the Web platform. However, the focus of our performance experiments lies on the implementation of the core programming paradigm which encoding and decoding performance does not belong to. We have discussed in Chapter 5 how the programming paradigm can minimize the amount of state in a snapshot and the amount of state shared over the network. There are possible technical improvements available to applications such as the use of more efficient codecs [8]. We have also shown in Chapter 11 that performance is acceptable for realistic applications.

Now that we have given an outline of the extreme cases where performance may become an issue, it is still valuable to measure the precise performance impact of different implementations of REScala and its extensions. We take a closer look at snapshot performance, at the performance behavior of different schedulers, and of language integrated error propagation. In addition, these benchmarks exercise the basic performance behavior of reactives and transactions, thus providing a general overview of the performance of REScala. All benchmarks in this thesis, as well as many more specific experiments can be found in the *microbenchmarks* project of the REScala repository.

13.1 Setup and Threats to Validity

This thesis presents performance results acquired over the course of multiple years, multiple devices, and multiple runtimes. Performance does not necessarily increase over time, workarounds of CPU security bugs have caused slowdowns, the Scala compiler has changed the collection library for better maintainability, and REScala also sometimes favors simpler code over raw performance. Thus, we try to provide the exact context for each of the benchmarks with the note that your mileage may vary. The following systems have been used for experiments and case studies:

Cluster Intel Xeon E5-2670 CPUs with 16 cores at 2.6 - 3.3 GHz (scaling based on core usage); compiled with Scala 2.11.{7,8}; 64-Bit Oracle JRE 8u74 with 1 GB heap; CentOS Linux 7.2.

LaptopMid Intel(R) Core(TM) i5-5300U with 2 cores (laptop CPU); compiled with Scala 2.12.x; Ubuntu 18.10 (64-bit).

Browser Same as LaptopMid using Chromium 73.0.3683.75.

LaptopHigh Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz with 4 cores (8 threads) scaling up to 3.3GHz; compiled with Scala 2.13.4; OpenJDK version "14.0.2" 2020-07-14; Ubuntu 20.10 (64-bit).

Our JVM benchmarks are implemented using the OpenJDK benchmarking framework Java Microbenchmark Harness [2]. If not stated otherwise results are the average of at least 25 iterations of the measured benchmark. To reduce the influence of non-deterministic optimizations, we fork the JVM 5 times, each doing 5 iterations with proper warm-up. Each iteration runs for about 1 second.

There are both internal and external threats to the validity of our results. Internal threats include the selection of benchmarks and their implementation. There is no agreed upon benchmark suite for dicApps, thus our selection of benchmarks may be biased towards the strengths of REScala. We also use the same set of benchmarks for continued performance monitoring and improvements of performance over time that we use for evaluation, thus REScala is potentially optimized for these specific cases. Finally, we are experts in using REScala but not so for systems we compare against, thus it is possible that the implementations of the experiments for REScala are of higher quality. The external threat is that the benchmarks may be too small and not sufficiently diverse for the results to be generalizable to dicApps.

13.2 Snapshots and Restoration

Crash tolerance in REScala has a synergy with the transactional semantics thus their performance is tightly related. As a trend, restoration reduces throughput. In the following, we evaluate our claim that the performance impact of restoration on typical REScala applications is reasonable. The experiments in this section use the Cluster (c.f., Section 13.1) type of devices. Specifically, we answer the following questions:

RQ1 What is the performance overhead introduced by our snapshotting mechanism?

RQ2 What is the performance trade-off between restoring state from the snapshot versus recomputing the state?

RQ3 How does the performance of our recovery mechanism compare to the performance of the recovery mechanism of an industrial-strength data streaming system?

(RQ1) Overhead of Snapshots

Snapshots happen after every transaction on the flow graph and affect the overall application performance. Snapshot overhead consists of the internal overhead for determining all the updated state and of the overhead for encoding that state into bytes. The snapshots in these experiments are stored in an in-memory database, because we do not want to measure time spent writing to disk, since this overhead

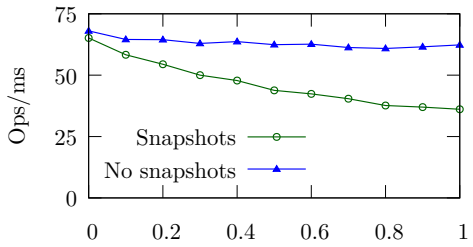

Figure 13.2: The cost of snapshots. X-axis is the percentage of fold reactives in the graph.

is not specific to our solution. We quantify the snapshot overhead as a function of the number of folds in an application, since only the state of fold reactives is included in a snapshot. For this purpose, we parameterize our benchmarks with the number of fold reactives in the application.

Figure 13.2 shows the throughput for a flow graph consisting of a single input event with 100 reactives derived from that input. On the x-axis is the percentage of fold reactives out of all the derived reactives. We selected this topology since it allows us to create a full snapshot of all fold reactives with a single input change. To factor out the influence of computations not involved in snapshotting, user-defined computations of both folds and stateless derived reactives only do simple integer arithmetic with negligible overhead. We executed the benchmark twice, with and without snapshots enabled. The relative throughput is on the y-axis of Figure 13.2 (higher is better).

We observe that the throughput of the benchmark with snapshotting is overall lower than without and further decreases when the number of fold reactives is higher. In the best case, i.e., there are no fold reactives, the overhead is minimal; our solution incurs performance overhead only when state is actually stored, i.e., there is no overhead for an active but unused feature. In the worst case, i.e., when every reactive is a fold, the throughput of the run with snapshots is still about 58% of that with no snapshot. For typical reactive programs, – which contain roughly 14% fold signals [181] – the relative throughput is 85%. This experiment is also rather extreme, because it is unusual that all reactives in an application are changed at the same time. Even in the worst case scenario REScala only requires 33 microseconds per transaction (each changing 101 reactives), which we still consider reasonable performance for real applications.

(RQ2) Restoring From Snapshots Versus Recomputing

We first quantify the overall cost that recovery adds when restarting the application. We also quantify the trade-off between taking minimal snapshots versus taking bigger snapshots.

Regarding the cost of recovery, Figure 13.3 (left) shows the results of measuring the cost of recovery for the graph from (RQ1). Each bar on the x-axis shows the

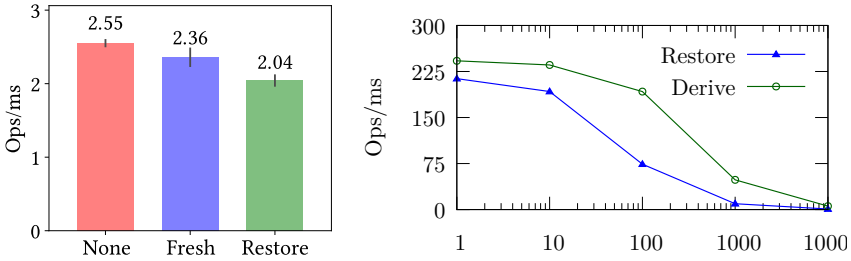

Figure 13.3: Left: cost of restoration. Right: restoring vs. recomputing lists of various sizes.

throughput of creating a graph (a) without any support for fault tolerance thus no overhead for restoration, (b) with support for fault tolerance but when restoring from an empty (fresh) snapshot, as is the case when an application is started for the first time, and (c) when restoring the graph from an existing fully populated snapshot. The overhead we observe in the last case is the result of creating the initial snapshot and restoring the (serialized) values from the snapshot. We conclude from Figure 13.3 (left) that while restoration has a certain overhead, the cost is comparable to normal application startup times, since REScala restores the graph of 100 reactives twice per millisecond, compared to starting the application, which is performed 2.5 times per millisecond.

Regarding the trade-off between minimal snapshot and recomputation, where our approach by default minimizes the amount of state that is stored in snapshots. That is, instead of restoring derived state we recompute it. Intuitively, one would assume that our restoration has higher overhead compared to one that starts from a maximal snapshot, as it has to recompute more. However, a small experiment indicates that this does not necessarily have to be the case. In the experiment, we run two versions (labeled Restore and Derive) of a benchmark with a reactive that stores a list containing integers 1 to N. In the Restore version the list is part of the snapshot, while in the Derive version the snapshot only contains the size of the list and the list itself is recomputed during restoration. The graphs in Figure 13.3 (right) show the results, with N in the x-axis and throughput in the y-axis. We observe that both restoring and recomputing derived state get linearly more expensive with the size of N. We also observe that recomputing the list given its size is faster than restoring from a complete snapshot. This indicates that the performance impact of deriving as much state as possible is highly dependent on the involved operations.

(RQ3) Comparison to an Industrial-strength Data Streaming System

Our objective in this experiment is to compare the performance of our implementation for snapshots and recovery to a functionally similar industrial-strength system. The objective is to measure an upper bound for the performance of our REScala. We compare against Flink [19], a state-of-the-art, big data processing engine for real-time analytics used, among the others, in the Alibaba real-time search ranking, in

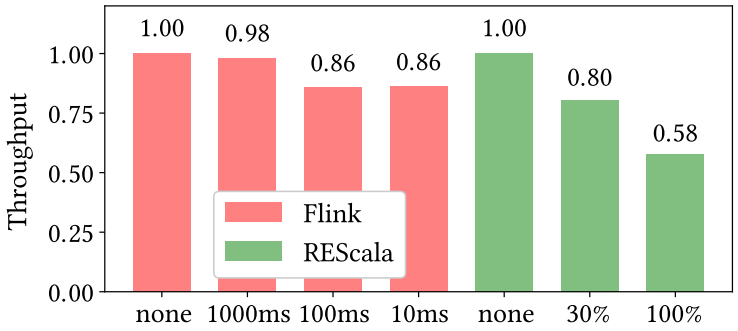

Figure 13.4: Flink vs. REScala snapshot performance.

Zalando's business process monitoring and in Netflix's complex event processing system [3]. Flink is suitable as a reference due to the following reasons: (a) it is functionally similar to reactive applications in that it also manages state inside a flow graph (a property it shares with other streaming systems), (b) it is implemented in Scala, hence the runtime environment is similar to ours, (c) it is well known for its focus on fault tolerance, (d) it is also possible to enable/disable snapshots, and (e) both Flink and REScala serialize snapshots to memory.

We implemented a similar graph structure as in (RQ1) for Flink. However, Flink and REScala target different usage scenarios. REScala immediately reacts to individual occurrences of input events, such as button clicks. Flink, on the other hand, processes and aggregates complete input streams of data. Hence, we do not compare the absolute performance of Flink and REScala, but only measure the relative overhead of creating snapshots.

In Figure 13.4, we show the relative throughput with and without snapshots (checkpoints in Flink terminology) within the same system. Snapshots in Flink are created periodically instead of after each update (we have created them every 10 ms, 100 ms, and 1000 ms, respectively), and always include the complete state of the system. While the overhead of REScala is higher when a full snapshot is created, in the case when only 30% of the flow graph is stored in the snapshot – which is the realistic case – the relative overheads of both systems are similar, with snapshots slowing down the systems to 86% and 80% of normal throughput respectively.

We conclude that the performance of our snapshot algorithm has reasonable overhead for our use cases. The use of time-based snapshots in Flink may be an interesting addition to REScala for certain use cases. However, we currently believe that – while our approach behaves worse in benchmarks – it is more useful to snapshot after every interaction, because our case studies show that typical dicApps may only have interactions every couple of seconds.

13.3 Schedulers and Transaction Performance

REScala has multiple schedulers as discussed in Chapter 6. The initial motivation of these schedulers was the support for parallel execution. Parallel execution became necessary when REScala transitioned to supporting distribution and collaboration, because transactions are no longer started only by user interactions but may happen at any time caused by network messages. Efficient parallelization may also allow use of REScala embedded into applications that require efficient use of multiple processors such as simulations or data processing applications.

We compare four different implementations of locking strategies for schedulers all using the same implementation for the actual propagation of changes to keep them comparable. See Section 6.5 for the detailed explanation of schedulers. The default scheduler of REScala on the JVM is ParRP. ParRP supports multiple transactions executing in parallel. The default scheduler for the Web platform where parallelism is unavailable is G-Lock. G-Lock uses a single global lock to synchronize changes when executed on the JVM. STM delegates scheduling to ScalaSTM [43], a former state-of-the-art Software Transactional Memory implementation. Handcrafted is not a single scheduler, but instead a class of fine-grained locking handcrafted for each application. Performance heavily depends on the features used by an application, thus the evaluation investigates how these approach perform in the presence of these features:

RQ1 Only static dependencies in the flow graph.
RQ2 With dynamic dependencies in the flow graph.
RQ3 With bottlenecks in the flow graph that cause contention between transactions.
RQ4 When there are read only transactions favoring the opportunistic approach of STM.
RQ5 With a varying set of workloads and topologies.

RQ1 and RQ2 quantify scalability and performance overhead. RQ3 covers the common topology of many inputs funneling into at few aggregation points, e.g., user interfaces displaying many values in one window or a server aggregating values of many clients. RQ4 covers a common workload for which STM and database benchmarks are optimized [116, 176, 177]. RQ5 explores the impact of different graph topologies on scalability. The experiments are executed using the Cluster (c.f., Section 13.1) type of devices.

RQ1, RQ2, and the Dining Philosophers

In these experiments, we use our interpretation of the dining philosophers problem as the topology for the flow graph. In this topology there are a fixed number of philosophers and the same number of forks arranged in a circle with alternating placement of philosophers and forks. Each philosopher has a source reactive that models the target state of the philosopher (eating or thinking). Each fork is derived

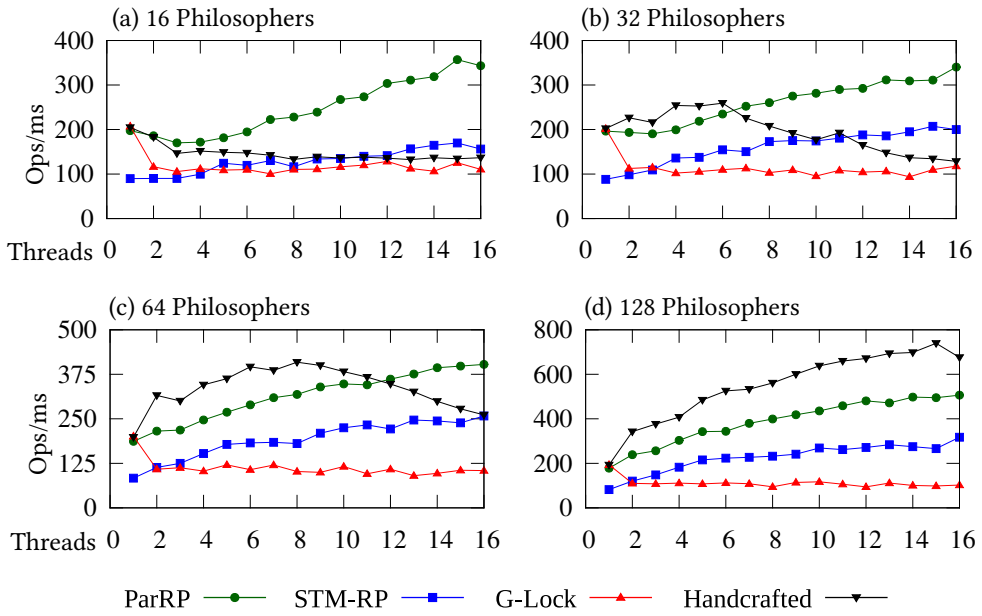

Figure 13.5: Throughput (ops/ms) per active threads, static graph.

from the two adjacent philosophers and its state describes which philosopher is using the fork. The actual state of the philosopher (called its sight) is derived from the two forks next to that philosopher and depends on whether that philosopher has acquired both forks. Each operation measured in the benchmark consists of two transactions: one philosopher switching form eating to thinking and back. Philosopher are assigned to threads round-robin, each thread randomly updates one assigned philosopher. E.g., with 16 threads and 64 philosophers, thread #0 randomly updates one of $\{p_0, p_{16}, p_{32}, p_{48}\}$ for a single measurement.

To answer RQ1, we measure throughput (steps per time) with an increasing number of active threads. Figure 13.5 shows throughput for 16, 32, 64 and 128 philosophers. More philosophers reduce the frequency of interactions between updates. In each case, we increase the number of threads from 1 to 16. Performance of G-Lock and Handcrafted degrades under high contention due to lack of contention management. Note that more recent versions of the JVM seem to solve the contention issue for G-Lock removing the steep decline of throughput when adding a second thread. ParRP and STM both gain performance from more threads, even under high contention. The overhead of STM over Handcrafted is comparable to prior benchmarks of STM [99, 101], finding, e.g., fine-grained locking 1.8x faster than STM [177].

For RQ2, Figure 13.6 shows results of the same experiments on philosophers with a dynamic flow graph. The relations between schedulers are similar to those from Figure 13.5. The absolute numbers are overall lower, indicating that the cost of edge changes is significant compared to scheduling overhead.

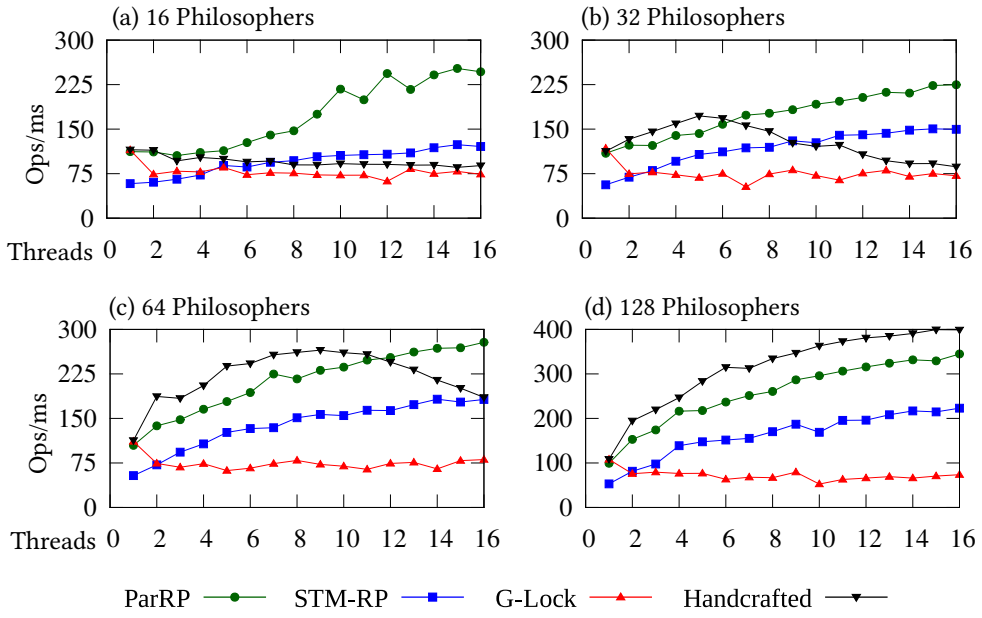

Figure 13.6: Throughput (ops/ms) per active threads, dynamic graph.

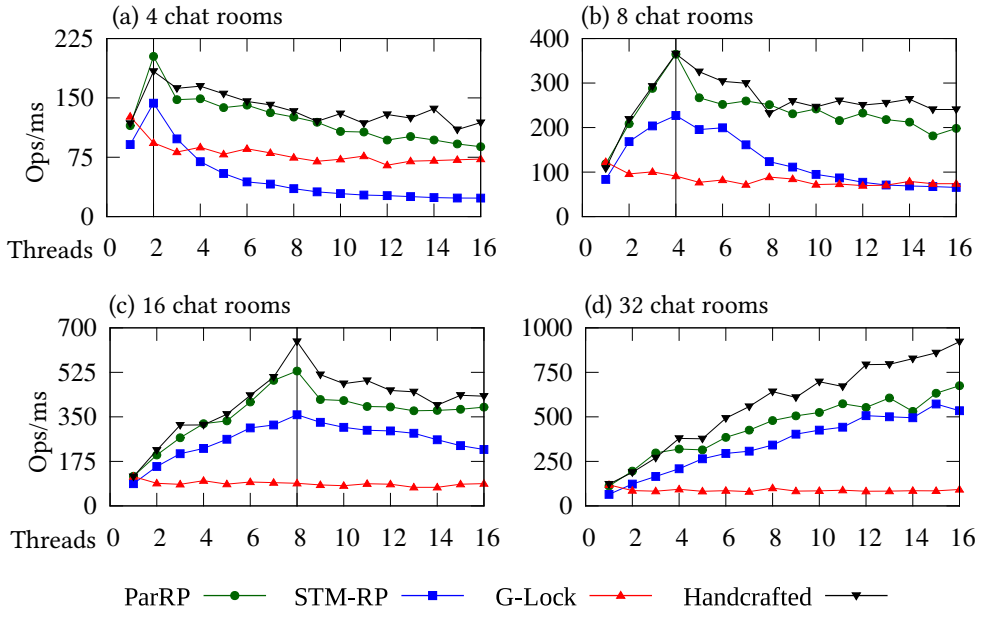

Figure 13.7: Throughput (ops/ms) for the chat server depending on the number of rooms.

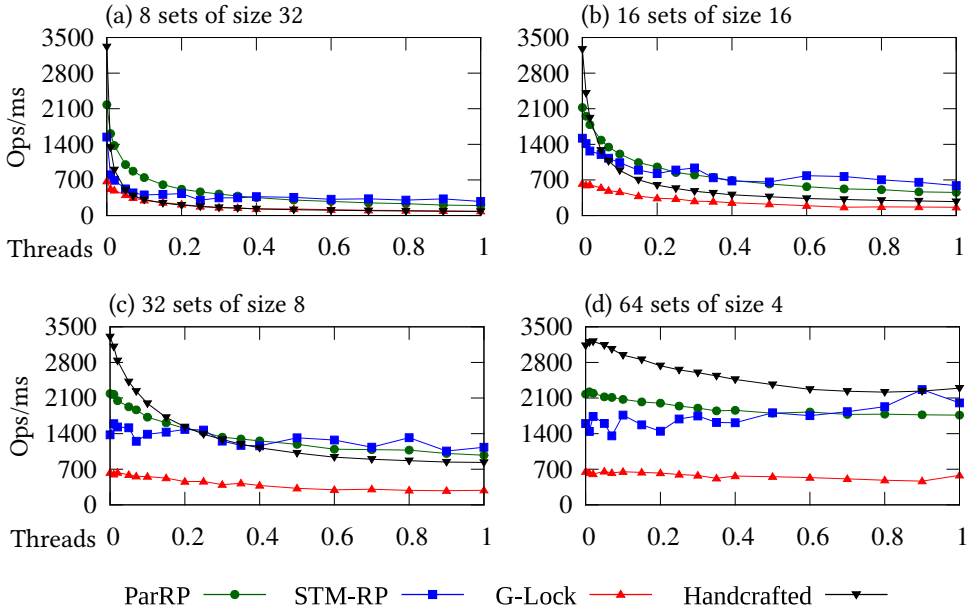

Figure 13.8: Throughput (ops/ms) per relative number of read operations depending on read windows sizes.

RQ3 and the Chat Server

In the chat server experiment each thread is a client joining exactly two chat rooms. Each step involves sending one message to both rooms. The order of messages in each room is consistent, thus, each chat room forms a bottleneck, because only a single client may write a message at the same time. Figure 13.7 shows the throughput depending on the number of clients (threads) and rooms. The vertical bars represent the point at which all rooms are occupied by a client and additional clients only increase contention. All schedulers (except G-Lock) display a near-linear increase in performance up to this bottleneck. For more threads, ParRP behaves similar to Handcrafted, which can handle contention well here due to the simple structure of one lock per room. STM struggles, as the underlying STM is not designed for high numbers of write conflicts.

RQ4 and the Bank Accounts

We address RQ4 using a bank account benchmark. There are 256 accounts and two types of transaction, T_W and T_R. T_W transfers money between two randomly chosen accounts, i.e., a write transaction. For T_R, accounts are divided into sets of 8, 16, 32 and 64. T_R picks one set at random and computes the total value of all contained accounts, i.e., a read-only transaction. The benchmark always runs 16 threads, but varies the percentage of T_R vs T_W, 0 meaning only T_W and 1 only T_R. Figure 13.8 shows the results. ParRP and Handcrafted both use exclusive locks for

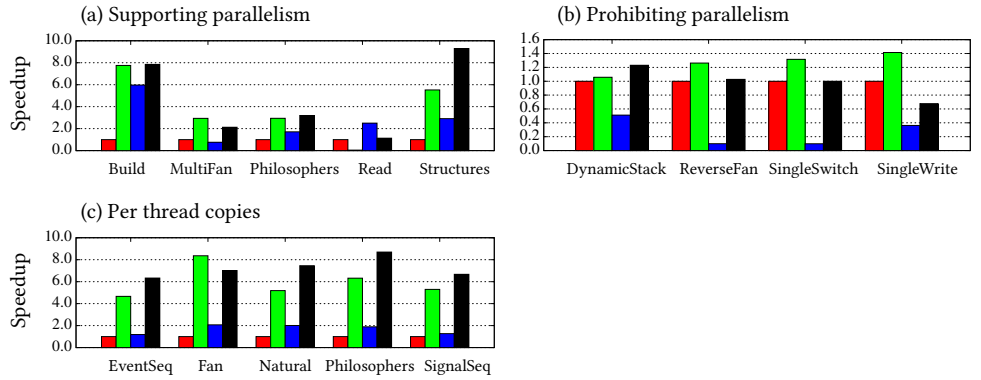

Figure 13.9: Throughput of different graph configurations relative to G-Lock performance using 8 threads.

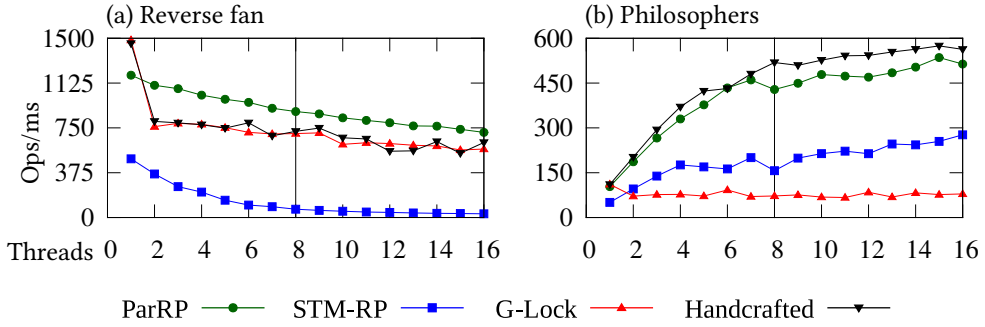

Figure 13.10: (a) ReverseFan detail (b) Philosopher detail.

reads, so each T_R needs exclusive access to the whole set of accounts. Thus, more T_R drastically decrease their throughput, because T_R causes more contention by accessing more accounts than T_W. STM does not exhibit significant improvements with an increased ratio of T_R, the higher concurrency is counter-acted by the bigger size of T_R. As interactions in dicApps mainly consist of reacting to external change, workloads of many large read-only transactions are rare. Given that and that the scenario is biased towards STM due to exclusive read locks in ParRP, we interpret the results positively, as ParRP shows limited performance degradation.

RQ5 and the Artificial Topology

For RQ5 we evaluate different workloads on artificially generated topologies to cover topologies that are more extreme than the experiments before above. Figure 13.9 displays how the schedulers perform in a variety of artificial topologies by comparing the throughput achieved by 8 threads relative to G-Lock. We group workloads depending on parallelization properties. Figure 13.9 (a) shows workloads that are naturally parallelizable. Speed-up depends on how much parallelism the topology allows.

Figure 13.9 (b) shows topologies that do not allow parallelism, hence Handcrafted degenerates to a manual implementation of G-Lock. These benchmarks test contention management. Figure 13.10 (a) shows the experiment results for 1 to 16 threads of the "ReverseFan" as an example for a topology that is not parallelizable. The data point of 8 threads shown in Figure 13.9 is marked with a bar. The other non-parallelizable exhibit similar behavior. ParRP is better than the JVM's built-in contention management used by G-Lock. This JVM issue has since been fixed in newer JVM versions. STM does not deal well with high contention in typical RP workloads (i.e., a significant ratio of writes).

Figure 13.9 (c) shows applications where updates are admitted such that they never interact. Figure 13.10 (b) shows the full benchmark (1 to 16 threads) for the "Philosopher" example in this group, the others are again similar. The performance gain per thread lessens when adding more threads. We speculate this is due to the processor slowing its clock speed under high load and the overhead due to the communication between the two processor sockets when more than 8 cores are used. Yet, at 8 threads we observe speed-ups of 6x.

Conclusion of the Parallelization Behavior of the Scheduler

Our concurrency control scales well with the number of threads and the overhead is always less than 100µs per transaction. Our scheduler outperforms the other generic scheduler (STM). Handcrafted is, as expected, faster than generic scheduling. Yet, besides being tedious to devise, it is also error-prone and fragile in presence of software evolution and composition [95]. Moreover, proving serializability of hand-crafted locking schemes is hard, and they are thus often excluded from evaluations [104]. ParRP is sound by design, including correctness preservation through composition.

While the parallelization of schedulers may currently not be important for most dicApps, we believe that these results show that REScala has potential to be applied in scenarios where efficient use of available processing power is important. This aligns with our overall goal to show that the Cvtr programming paradigm is applicable to many scenarios. It also provides an opportunity for future dicApps to make better use of such resources and integrate computation intensive tasks in the same programming paradigm.

13.4 Performance Effects of Language-Integrated Error Propagation

In Section 7, we motivated the need for language-integrated error propagation for the quality of application design. We empirically provided evidence that our approach to error propagation indeed barely pollutes the application code in Section 11.5. In this section, we ask: how does language-integrated error propagation affect application performance?

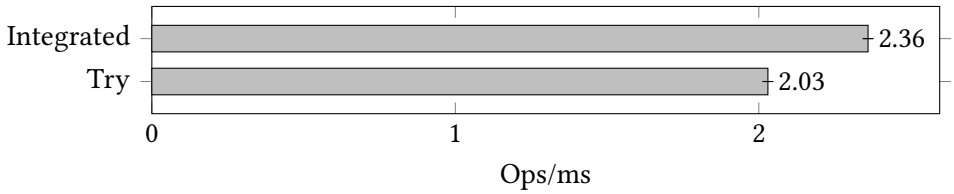

Figure 13.11: Integrated error propagation versus `Try`-based solution.

The experiments presented below analyze the potential performance effects of error handling. Specifically, we analyze the potential performance trade-offs of the language-integrated error propagation compared to programmatic error handling, and the overhead of the error propagation system in the absence of errors. The experiments are executed on the LaptopMid (c.f., Section 13.1) type of devices.

These experiments show that there is no additional cost for error propagation. As discussed in Section 7.2, this is due to the tight integration of errors into the existing runtime. Moreover, language-integrated error propagation exhibits better application performance compared to programmatic error handling.

To compare different approaches to error handling we implemented an experiment using Scala's `Try` to propagate errors handle errors, i.e., every `Signal[A]` is rewritten to be a `Signal[Try[A]]`. Using `Try` is the idiomatic way to represent errors as values in Scala, similar to the `Maybe` data type in Haskell. As shown in Figure 13.11, our solution outperforms the solution that uses `Try`-wrappers. This improvement is due to the fact that language integration integrates error propagation into the internal data structures of the language runtime, while `Try`-wrappers require an additional layer of indirection.

To measure the cost of support for error propagation even when no errors occur, we use an experiment setup called *natural graph*. This is an application that produces a flow graph with 25 reactives that are connected in a way to mimic the average application [181]. All user-defined computations only perform arithmetic additions to minimize the amount of work that is spent on actual computation and maximize the relative overhead of the error propagation. For this experiment, we did not measure any performance difference between a version of REScala with error propagation and a version without.

We believe that a general advantage of high-level programming paradigms such as Cvtr is that they often enable additional features at no additional cost. In this case, the synergy is due to the fact that REScala can automatically integrate different concerns into a single runtime abstraction, thus paying the cost only once.

13.5 Combined Effects of Errors and Snapshots

To understand the combined impact of snapshots and error propagation on REScala we run several experiments with and without support for error propagation and snapshots. We are especially interested in the overhead for graphs that do not

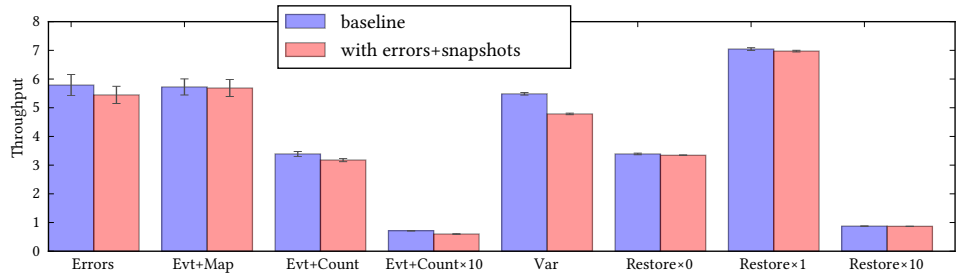

Figure 13.12: Cost of snapshots and errors. Throughput measured in: Errors per 100 ms; Restore×0 per 10 ms; Restore×1 and ×10 per 1 ms; others per 1 µs.

require these features. The experiments are executed on the LaptopMid (c.f., Section 13.1) type of devices Figure 13.12 shows the following benchmarks:

Errors A flow graph that is meant to have a representative structure of typical reactive programs.

Evt+Map A graph with an input event and a map operation, here no snapshots need to be stored because events are not included in snapshots.

Evt+Count Counts the number of the input event occurrences. This setup requires a single reactive constituting half of the flow graph to be stored.

Evt+Count×10 Same as before, but stores 10 count reactives derived from the same event.

Var A single var that is included in the snapshot.

Restore×0 Time required to restore the same graph as Evt+Map. The graph does not require restoration, so we measure pure overhead.

Restore×1 & Restore×10 Restore the graphs of Evt+Count and Evt+Count×10.

The results uniformly show that there seems to be a very slight overhead associated with errors and snapshots. However, the overhead is so negligible that we feel confident in always activating these features. Thus, support for snapshots and error propagation is always available in REScala applications.

13.6 Conclusion

We want REScala to be useful in as many cases as possible. The better the performance of the implementation the easier for application developers to choose REScala without worrying about trade-offs. Thus, our evaluation is focused on the differences between alternate implementations and the impact of optional features on the execution performance. We do believe performance of REScala to be excellent for all the use cases we envision. One of the reasons for the good performance is that REScala includes the discussed performance experiments as part of its development process. We believe that especially the performance of executing transactions and generating the flow graph is pretty well understood, and thus they also have

excellent performance. Similar arguments apply for both the error propagation and restoration.

However, the overhead of REScala is usually only a small part of the computational effort required by applications. The focus of future performance evaluation of REScala will shift towards more application specific integrations. There is a large impact of expensive user-defined functions executed as part of transactions. Some of these user-defined functions – such as integrations with the HTML DOM, and merge functions for CRTDs – could even be argued to be part of REScala itself. An interesting future extension to REScala is the ability to monitor performance of user-defined computations in individual reactives, to provide developers with feedback on performance bottlenecks within their application.

Chapter 14

Related Work

In this chapter we discuss related work that we either build upon or that provides an alternative to the solutions discussed in this thesis. To the best of our knowledge, there are no other solutions that cover the same design space as REScala and Cvtr, but there are many solutions that have an overlap for partial aspects.

First, we discuss various forms of reactive programming in Section 14.1 – which is where REScala draws the most influences from. Then, in Section 14.2, we discuss actors as an alternative programming paradigm that has seen success for distributed applications. Likewise, we discuss cluster and cloud systems as a good example for the convenience achieved by automatic fault tolerance in Section 14.3.

Solutions that provide consistency through the use of specific data types are covered in Section 14.4. Section 14.5 discusses formal approaches to distributed systems. Solutions that provide any form of collaboration framework with consistency guarantees for complete applications are discussed in Section 14.6. Related work on debugging and tuning of applications is discussed in Section 14.7.

14.1 The Reactive Programming Approaches

The term *reactive programming* has been used for many things in the literature, and many things that are similar use different terms. This section presents approaches that either are syntactically inspired by reactive programming or have semantics that are equivalent to a denotational description of application state over time.

Functional Reactive Programming

Elliott [74] states that the two principles of Functional Reactive Programming (FRP) are denotational design and continuous time. REScala does not support continuous time, and $\mathcal{F}_r$ is not a denotational semantic. This section explains why that is fine and why FRP is still relevant for this thesis.

Fran [75] (by Elliott) is one of the earliest libraries for FRP and is implemented in Haskell. In certain ways Fran is quite similar to REScala. Fran supports both events and signals (or behaviors, as signals are called in Fran), is implemented as

a library using combinators and is free of glitches. However, the target domain of Fran are animations and not general interactive applications. Instead of reacting to a variety of external changes the state of an animation is mostly defined by the current time. The representation of a signal in Fran reflects this domain and each signal conceptually corresponds to a function from the current time to the signal value. The actual implementation is more complicated than a simple function from time to value for technical reasons [73], resulting in an abstraction that superficially is similar to signals in REScala. Due to the discrete nature of computer displays the current state of signals is sampled at a regular interval to produce a series of discrete images. Note that the importance of continuous time is that the domain of animations is represented exactly until the last moment where it is discretized for display. As a rough analogy, functional reactive animations are to videos what vector graphics are to raster images.

Computer animations are much more interesting when they are interactive, thus even Fran has events to model discrete interactions with the system. However, representing an animation as a function from time to image data poses implementation challenges when dealing with interactive applications. A problem known as time and space leaks [126] refers to the fact that a naive implementation has to store all inputs (a space leak) and recompute aggregations over those inputs (a time leak) to be able to produce images at an arbitrary point in time. While being able to jump to any point of time is useful (see debugging and tuning in Chapter 9 for examples), for many interactive applications it is only necessary to produce the next state given the current state and the current inputs – and do so quickly.

Thus – to improve efficiency of the implementations – techniques such as push-based propagation [76] (same principle as propagation in REScala) were introduced to FRP. A discrete implementation of a continuous model is not perfect there are either still time and space leaks for edge cases or the model has to be restricted to disallow certain forms of dynamic graphs [125, 126] – especially those concerning nested reactives.

An alternative to improve efficiency is to change the programming model to enable more efficient implementations. One such approach is arrowized FRP [164] where functions to transform reactives are manipulated instead of the reactives themselves. The actual creation of the reactives is then delegated to the runtime to ensure that the application is not able to observe the internal mutability.

The use of FRP in one of these or similar forms has been investigated in many specific application domains beyond animations. The peculiarities of the specific application domains often lead to specific implementations and may also include influences outside pure FRP (see below). Examples include the simulation of autonomous vehicles [82], IoT [49] especially low powered embedded systems [184], robotics [108], network switches [84, 198], wireless sensor networks [162], scalable web servers [168], and reliable software for spacecraft [167].

On the theoretical side, Jeffrey [112, 113, 115] stated that a type system for FRP corresponds to linear time logic via Curry-Howard [107], i.e., an FRP implementation prooves certain LTL formulae. The implementation itself is based on lower level abstractions such as futures and observers [114].

Discrete Reactive Programming

The difficulty of efficient implementation in functional reactive programming stems from the desire to have a denotational semantics for a continuous domain, but then extend the use of that paradigm beyond just continuous animations. The authors of FrTime [59] observe that interactive applications deal with explicit mutability and discrete events, and that the efficient implementations of functional reactive programming are discrete and hide internal mutability. Their conclusion is to remove the gap between the implementation and the reality of the environments and instead directly enable the programming of the mutable reactive data structures – essentially the same concept as the flow graph of REScala albeit without explicit transactions. There are still events and signals and those still describe time-changing values, but any notion of abstract time is gone. We will call this approach Discrete Reactive Programming (DRP) for disambiguation even though the approach is still called functional reactive programming in most of the literature.

DRP has been widely successful in communities of programming languages that support mutability, because its operational nature allows application developers (as opposed to runtime developers) to integrate DRP with existing tools, frameworks, and APIs. However, there are also several implementations of DRP that integrate into various programming languages and have their own set of external integrations and features.

FrTime [59] uses Schemes macro system to provide automatic lifting of functions. Automatic lifting makes the use of reactives transparent to the programmer. Flapjax [146] implements DRP for JavaScript motivated by the frequency of callback-based applications in the domain of Web UIs. Flapjax provides a templating mechanism to embed signals as part of the displayed HTML. FELM [65] prohibits the use of dynamic reactives to allow pipelined execution of a push-based update propagation. The execution strategy of FELM allows the integration of non-blocking execution of long-running tasks into reactives. Scala.React [137] integrates with thread pools of external libraries, and also introduces a domain specific language to combine temporal sequences of event occurrences.

For further reading, Bainomugisha et al. [28] provide a comprehensive survey on reactive programming languages, with a focus on the implementation model and features supported by the individual implementations.

Synchronous Languages

As far as we can tell, synchronous languages such as Esterel [35, 83] predate functional reactive programming. Synchronous languages are in the literature treated as a completely separate topic to reactive programming, but there are several close relations beyond the fact that Esterel also operates on signals – in this case inspired by the concept of physical wires in the design of an electrical circuit. Esterel was designed for embedded reactive systems such as microcontrollers or drivers for hardware interacting with external systems. The goals of Esterel are to provide deterministic concurrency, that is, even though the program is written in a style that

looks like concurrent imperative code the resulting execution only has a single valid set of outputs for each input at a specific time. This is achieved by compiling Esterel programs into a dependency graph very similar in nature to the implementations of reactive programming languages. Due to the imperative syntax, it is possible to write syntacticlly correct Esterel programs that are not deterministic in nature or require circular logic which cannot be expressed as an acyclic graph, those programs are simply rejected by the compiler. Quite recently, Florence et al. [83] provided a calculus for Esterel that expresses the language in a style more similar to what functional reactive programmers would expect – although still consisting of explicit writes to signals instead of a declarative description.

What Esterel makes explicit, however, is the idea of synchronous execution. The idea of synchronous execution is that when a change in inputs occurs, then all derived values are conceptually computed at the same time. This is exactly the semantics of a REScala transaction and also how a functional reactive system behaves when a frame at a certain point in time is computed. However, the formulation using synchronous execution makes the relation to physical hardware and parallel execution much more apparent. That is, a static flow graph is related to a hardware design where all computations happen at the same time, but only the final stable value is visible.

There are other synchronous programming languages such as CoReA [38] and Céu [183] that focus on deterministic concurrency for embedded systems. Amadio et al. [24] discuss using synchronous programming for coordination of general multi-threaded systems. Differently from the imperative synchronous style pioneered by Esterel, there are two prominent synchronous dataflow languages, SIGNAL [89] and Lustre [54] that use a more declarative style do define their semantics.

There is clearly a lot of unexplored overlap between the synchronous and reactive programming communities. In Chapter 8 we have already discussed how the flow graph can be compiled for embedded devices something we started to investigate because of the relation between synchronous languages and the flow graph.

Functional and Reactive Programming

As we have discussed at length, interactive applications update their internal state and produce results in reaction to external events. This includes not only user interfaces, but a wide range of domains such as Web servers, and sensor applications [180]. How are languages called that use a (remotely) functional approach to react to any form of external inputs in any of those domains? Well, the term people commonly seem to choose is functional reactive programming. For the sake of disambiguation, we will use the term Functional and Reactive Programming (FaRP) to describe such systems that do not belong to FRP.

Traditionally, the main design issue when implementing reactive applications has been to decouple the code that detects or triggers the events from the code that handles the events (i.e., to implement the semantic model of an application independently of the user interface). The observer pattern [86] already achieves decoupling,

but simple observers lack desirable features like composability [75] so that complex reactions can be build from simpler ones. The syntax used for composition should also make the semantics and the control flow of the composition obvious and not hide them behind boilerplate code or scatter them around different places of the code – a problem caused by inversion of control [97].

A popular approach belonging to this category are reactive extensions [136] or ReactiveX. ReactiveX is an API for asynchronous streams. Being asynchronous in nature is what sets ReactiveX apart most from FRP. Syntactically, ReactiveX is inspired by the combinator libraries used by early FRP languages, but, as an amusing side note, the ReactiveX documentation now provides a disambiguation [10] from the term FRP and clearly distances itself from that term, due to all the confusion that is caused by people refering to ReactiveX as FRP. Semantically, ReactiveX is a set of operators that allow composition of observers (as in the observer pattern). ReactiveX provides a useful API compared to programming with observers manually, but it offers no semantic benefits or correctness guarantees on top of observers. There are implementations of the ReactiveX API in many programming languages (similar to how there is an API for arrays in many programming languages).

The Elm [6] programming language is the modern evolution of what once was the FELM [65] research language. FELM implemented pure FRP as discussed before. Elm abandoned the possibility to compose signals and events into complex applications, and instead focuses on a single event handler that produces incremental changes to a single global state. This focus enables Elm to provide good developer ergonomics and automatic guarantees for "Single Page Applications" while keeping the programming paradigm quite simple.

Facebook's React [79] is another well known library with a naming similar to FRP. React is a JavaScript library for creating web pages from "functional components". Components are functions that transform part of a (global) state into a piece of HTML (e.g. a widget that shows the currently active contacts on Facebook). React has similar use to Elm, but is suitable for more types of web pages, since it integrates into existing web pages instead of completely replacing them. However, React focuses on the UI part, not on application design or any advanced features.

There are many more libraries and frameworks that roughly fall in this category, see the project page of the TodoMVC [14] project to get an overview of some of those systems. In general, most libraries seem to focus on the API side of providing useful combinators, but do not implement synchronous semantics, not to mention any form of continuous time.

Distributed Reactive Programming

A recent focus of FRP and DRP (and even FaRP) is distribution [52, 64, 69, 138, 139, 174]. Research on Distributed Reactive Programming (distributed RP) includes new consistency models, such as eventual consistency via CRDTs for replicated signals [159] and notions of relaxed glitch freedom that account for an error margin [171]. Also, in the area of distributed RP, researchers proposed mechanisms

to achieve fault tolerance. Leased signals [160] enable reacting to a partial failure when a remote host does not produce a value after a timeout.

In general, all forms of reactive programming seem to be a natural fit for distributed applications, with events representing messages from the network or user input. However, many functional reactive languages and frameworks do not provide support for unreliable networks. Typically, reactive languages [65, 137, 146] simply delegate the responsibility for error handling to the host language, and ultimately to the programmer. In distributed reactive programming [63, 69, 138], reactives on different devices are connected to each other and update messages are sent over the network whenever a remote dependency changes. In the presence of faulty devices and unreliable connections, such update messages may get lost causing several problems, such as glitches, changes that are visible on one host but not on another host, or application unresponsiveness when new changes cannot be processed while messages are being resent to a device that failed and is restored.

Unreliability has been partially investigated in the context of some FRP derivatives. Timeouts have been introduced to a distributed runtime and dataflow [160]. ReactiveX [136] integrates and propagates errors into as part of the dataflow similar to REScala. However, to the best of our knowledge, no solution exists to automatically restore and reconnect a flow graph after a crash. DREAM [138, 139] is a middleware for distributed reactive programming, which lets the programmer choose among different levels of consistency guarantees in distributed reactive systems, including FIFO consistency, causal consistency, glitch freedom and atomic consistency. Ur/Web [57] is a multitier programming language that uses reactive programming to update the client UI. However, to the best of our knowledge, there is no integration of network errors and the reactive part, hindering application-wide reasoning and lacking common abstractions for distribution and reactivity. AmbientTalk/R [53] implements distributed RP for mobile peer-to-peer networks. It is focused on providing some communication in loosely coupled networks instead of any consistency guarantees. The SID-UP algorithm [69] implements distributed RP with glitch-freedom, but uses global locking. However, none of the approaches provides the full feature set of REScala with the same level of automation of fault tolerance.

Parallel Synchronous Reactive Programming

As far as we are aware, REScala is still the only reactive programming language with transactions that have synchronous semantics and can execute in parallel while preserving serializability. The formal model of Elm [65] supports pipelined execution, but the model was never implemented. Esterel [83] supports parallel execution in hardware, but not for arbitrary transactions in an application.

Related approaches such as ReactiveX [10] do provide parallel execution for their observables (an observable corresponds to a reactive), but the behavior is only defined for each individual observable – there are no guarantees for the overall behavior of the application regarding concurrency – thus developers have to manually consider race conditions. Only a transactional programming paradigm – such

as Cvtr – provides the necessary framework to define what conflicts are and thus provide correctness automatically.

Formal Models of (Functional) Reactive Programming

$\mathscr{F}_r$ is not the only calculus that exists in the space of reactive programming, but the only one that also includes automatic fault tolerance.

FELM [65] provides a calculus for the functional surface language that defines the reactive application. Yet, the actual behavior of the application at runtime – i.e., how values flow through the graph – is captured by a translation to ConcurrentML, hindering formal reasoning about the behavior of the complete system. An additional limitation of FELM is that the flow graph cannot be modified at runtime.

Kamina and Aotani [117] provide a formalization which aims to simplify the abstractions used in RP. Thus, their formalization is intentionally minimalistic. In particular, their formalization uses standard operational semantics for the evaluation of the surface language but uses denotational semantics (functions on sets) for propagation of changes through the flow graph. While the separation simplifies the presentation of the calculus, we believe it complicates the reasoning about the details of transactions – a necessity when reasoning about faults.

DREAM [138, 139] also formalizes the propagation of events in a distributed system, however with a different focus. The goal is to provide a formal foundation for analyzing the consistency level provided by different propagations algorithms (e.g., fifo versus glitch freedom). However, DREAM does not deal with unreliable message delivery or restoration, and does not integrate distributed state with local operations.

Building Blocks for Reactive Programming Implementations

In the area of the implementation of RP languages, Ramson and Hirschfeld [172] proposed Active Expressions as a fundamental primitive for different RP implementations.

Publish-subscribe systems are the distributed counterpart of observers. Publish-subscribe systems aim to support loose coupling among event publishers and observers. By contrast, complex event processing (CEP) [62] is about correlating events. Both types of systems usually rely on callbacks with all their problems [78, 141, 169].

Self-adjusting computations [98, 100] are a class of techniques to make existing batch programs incremental, i.e., when only a small part of the input changes then only the dependent part of the output must be recomputed. The strategy essentially works by automatically decomposing the program into something conceptually similar to the flow graph. However, the point of REScala and similar languages is to improve application design by directly programming the flow graph instead of first using an inferior representation of the program logic.

14.2 Actors

The actor programming paradigm [118] is similar to object-oriented programming with strong encapsulation of internal state. The difference is that execution of messages (or methods) happens asynchronously in the actor paradigm. Asynchronous execution and encapsulation of state are what makes actors suited for concurrent and distributed systems [148] – a message in the programming paradigm has a direct representation as a message in the physical network and encapsulated state stays on the device hosting the actor. However, this also means that the actor paradigm does not protect developers from any of the issues of distributed systems, such as message reordering, lost messages, and partial crashes. In addition, the loose application structure implied by the actor paradigm makes automatic reasoning about the overall system consistency hard. Furthermore, we consider message-passing to be rather an implementation mechanism for enabling communication, but not a proper substitute for providing first-class composable and programmable abstractions as part of the programming paradigm. Despite the shortcomings of actors as a programming paradigm many implementations of actor languages provide extensions for fault tolerance.

Crash Tolerance

Actor supervision mechanisms as provided by Akka [17] and Erlang [25] allow specifying user-defined handlers when an actor crashes (usually due to a programming error). The typical strategy to handle a single crashed actor is to replace it with a fresh instance of the actor and losing all state in the process. The scheme can be extended to also provide tolerance against device crashes by replacing all actors on a crashed device with new instances on another device. Restarting of actors works well for applications that require continued operation, such as telephony networks – for which actors were first designed. In a telephony network, it is cheap to reconnect a dropped call, and a missing part of the conversation can be tolerated, but it should always be possible to establish new calls. It is possible to use crash handlers to restore the state of an actor after restart. The Mnesia [140] database provides a managed runtime for this purpose, but the details of state saving and restoration is completely left to the application logic.

Orleans [33] provides the actor paradigm but with fully automatic deployment and life cycle management of actors. This allows Orleans to automatically restore the state of each actor after a crash. However, the state of each actor is stored independently without consistency guarantees between multiple actors, thus – in the presence of faults – it is difficult to reason about application properties that span more than a single actor.

The cluster extension for Akka [16] provides similar restoration capabilities for individual actor state in cluster environments, still without overall system consistency. In addition, Akka cluster requires developers to modify the code of each actor that needs fault-tolerant state, making it impossible to reuse existing actors that are developed without support for fault tolerance.

Reliable Communication

The cluster extension for Akka [16] also provides a library of CRDTs for data synchronization, but with barely any integration into the language. Thus, requiring applications to manually manage all communication and individually apply received changes to the local replica of a CRDT.

AmbientTalk [64] is an actor language specifically designed for mobile ad hoc networks. Direst [159] builds on top of AmbientTalk and adds reactive abstractions and automatic eventually consistent state replication. However, Direst uses a centralized replica to provide eventual consistency, hindering any communication between devices, when the centralized replica is not available. Furthermore, applications in Direst must statically declare their dependencies, because the network runtime relies on global knowledge of the flow of data between devices.

Orleans has an extension called reactive caching [45] which adds a layer for push-based state for client devices. Essentially, whenever the state of an actor in the cluster changes, that state is preemptively send to interested devices outside the cluster (such as user interfaces running on a user's device). This allows limited tolerance of intermittent networks, because local devices always only access their local replica. However, the device outside the cluster may not change any of the remote data.

14.3 Automatic Fault Tolerance for Clouds and Clusters

Clusters of devices – such as The Cloud – usually have fast and reliable data storage and many more devices than are required for each individual task, but very unpredictable workloads of tasks. Each task typically spans many devices and may run for long periods of time. Because tasks have different priority it is often desirable to stop – or scale down – less important tasks as soon as utilization of the cluster is high. This – in addition to the use of commodity devices that are individually not very reliable – results in the typical computing environment where tasks (i.e., applications) must be able to tolerate partial "crashes" of some or most devices and still be able to continue the task or at least resume it later. This environment has produced a series of fault-tolerant programming paradigms.

MapReduce

MapReduce [66, 131] tackles the reliability problem in data centers by simplifying and restricting the programming model down to just two operations – map and reduce. Or more precisely a map and reduce on keyed subsets of data. MapReduce enables parallel execution of tasks on multiple devices. Failure recovery is automated with a central coordinator responsible for rescheduling operations of failed devices. The coordinator is often implemented as a distributed consensus that ensures reliability at negligible cost, because of fast networks and the possibility to assign reliable coordinator devices.

MapReduce excels in automation and fault tolerance, but lacks high-level programming abstractions. This has fostered research on adding abstractions on top of MapReduce. Dryad [109], PigLatin [165], and FlumeJava [55] all provide such new abstractions on top of MapReduce. However, these solutions still only support the same data-parallel type of computations also supported by MapReduce. These systems abstract away the distribution – this makes them unsuited beyond mostly reliable clusters, where failed devices can always be replaced.

Formal methods have been used for parts of data center computations. Cloud Calculus [111] accounts for security regarding firewall configurations, and Lämmel [131] studied MapReduce as a functional programming paradigm.

Generalized Batch and Stream Processing Paradigms

Frameworks for general batch and stream processing of data, such as Spark [206] and Flink [19], handle crashes of worker machines to minimize lost work when machines fail. They have syntax inspired by functional reactive programming that uses operators to build a dataflow graph. Because they execute in cluster environments with full control of communication and distribution of work among machines, Spark and Flink can offer abstractions for distribution and fault tolerance with automatic correctness guarantees [50]. To provide these guarantees, applications are written in domain specific languages, but, in contrast to Cvtr, the execution runtime is not connected to the embedding application. This solution is well-suited for data parallel applications, where computations are broken down into independent tasks. It is, however, unsuitable for the interactivity and collaboration requirements of dicApps and also unsuitable for other kinds of distributed environments without fast network and replaceable devices. Restarting parts of the application on a different machine is not an option for dicApps – the email client of a user cannot simply be restarted on another user's device.

MBrace [71] extends F# with expressions for cloud computations. The use of immutable global references allows the distributed runtime to automatically re-execute tasks on failed devices without causing inconsistencies. Errors that are raised during the evaluation of cloud expressions are transparently propagated along the dataflow path of the expression. In contrast to Cvtr, the propagation of values extends across the distribution boundaries, thus allowing non-localized error handling. However, since the distributed state is immutable, the abstractions are not well suited to interactive or collaborative applications.

Viering et al. [197] develop a typing discipline for a core model that captures the above approaches to ensure correct execution of such systems in the presence of faults. This static approach explicitly assumes the existence of central coordination and replacements for failed devices, singling them out as the central solutions for automatic fault tolerance in a cluster environment. We generally have neither coordination nor replacements available for dicApps.

14.4 Consistency and Data Types

The notion of consistency of data types is intimately linked to the notion of monotonicity. We regard consistency as a state in which certain information in a system no longer changes. For example, a consensus algorithm fixes a decision and will never provide a new answer for an old decision. On the other hand, new information can always be added to a consistent state – a monotonic increase of the available information. Hellerstein and Alvaro [103] show that all eventually consistent systems without central coordination must always be equivalent to a logically monotonic system. However, there are many approaches to consistency with their own trade-offs. We have limited ourselves to the use of state-based CRDTs with REScala, but, as our formalization shows, we inherit the consistency guarantees of the data type used for replication. In this section, we discuss potential alternatives that may provide different advantages and are candidates to address the challenges of dicApps.

The Global Sequence Protocol

The Global Sequence Protocol (GSP) [47] provides a semantic foundation for eventual consistency using a central server connected to clients over an unreliable network. Clients have two separate states, a last known consistent state and their own local working state. Clients may always execute transactions on their own local state and immediately observe the effect. When the clients are connected to the central server, they send their local transactions to that server. The server decides a global order for transactions and replicates that order to all other clients. If the global order of transactions received from the server matches the local order of the client, then the local working state becomes the new consistent state. Otherwise, the local working state is discarded and the new consistent state is computed from the received transaction order.

Global Operator Sequencing (GOS) [93] generalizes the approach of the GSP to all systems that use an eventually consistent sequence of operations. We have discussed the idea of replicating the order of transactions instead of the state in Section 10.3. The generalization in GOS allows to prove the equivalence of GSP and certain weak memory models of CPUs, a direction that we have not yet explored in the context of Cvtr, but may be interesting when compiling to embedded devices.

Bernstein et al. [34] use the GSP for synchronization of data between multiple Orleans clusters. This allows for geo-replication of data centers even if connections between them may sometimes be unreliable. GSP works well in this case, because connections are mostly fast and reliable, and GSP has little overhead in terms of the size of state for such cases. However, they did not extend their approach to client devices.

Reactive Caching [45] introduces an eventually consistent layer for push-based state for clients connected to Orleans [33] clusters. The work includes a formalization with proofs of correctness. However, a severe limitation is that clients may only observe the state and never modify it locally.

CloudTypes [46] target client devices and are based on a predecessor of the GSP. They are designed as a drop in replacement for primitive data types that allow eventually consistent replication of primitive using a central server, thus their goal is also the implementation of dicApps. However, CloudTypes are limited to implementation of the basic types that are provided, thus they are only suited for applications that do not require the synchronization of complex state.

Overall, the GSP and related approaches have a diverse set of concrete implementations. We believe this is due to very different performance characteristics of the specific environments targeted at the time. CloudTypes was for mobile applications, while the integration of GSP into Orleans was about leveraging and improving performance of the existing cluster environments. Ultimately, GSP and related approaches may not be worthwhile for the use on end-user devices, because the requirement of a central coordinator means that offline usage is limited and use of peer-to-peer communication is unfeasible. State-based CRDTs do not share those limitations, and we believe that similar optimizations are also applicable to CRDTs in specific execution environments.

Monotonic Synchronization

Hellerstein and Alvaro [103] discuss why monotonic data structures work well in distributed systems, because they can represent exactly the class of applications that can be available and eventually consistent [102]. This is reflected in several programming paradigm approaches in addition to Cvtr.

Bloom [58] is an experimental language that explores a programming paradigm where all operations are monotonic. Due to composability of monotonicity all program that are implemented using only monotonic operations are also monotonic. The experiments with Bloom show that programming using monotonic operations is not that much different from programming using other paradigm – although it is necessary to develop new algorithms for many common tasks (e.g., sorting).

The Lasp [142, 143] language tackles the problem of deriving one CRDT from another, while keeping track of which change on the source CRDT caused a change on the derived CRDT. Thus, Lasp prevents multiple derivations on different hosts to create duplicate entries. To achieve this, however, Lasp only supports a limited set of derivation operations.

LVars [128] use monotonicity for deterministic concurrency, by only allowing threshold reads, which make reading of a lattice-based data structure deterministic in the presence of multiple parallel writes. Kuper and Newton [129] discuss the combined use of lattices for deterministic parallelism and distribution.

CRDTs and Databases

Recent advances include CRDTs directly into distributed databases [187]. Client applications connect to an instance of the database. The database takes care of all communication and replication, allowing clients to be written in a traditional

client-server style. There are two layers to consider, the API for communication between the application and the database, and the communication between multiple instances of the database.

The communication between the application and the database is how individual operations are replicated for operation-based CRDTs. Applications send commutative CRDT operations to the database. The instance of the database applies those operations locally. Replication to other instances of the database happens asynchronously. Communication between database instances is similar to how replication works in Cvtr.

Using an operation-based API to communicate with the database has all the problems of operation-based CRDTs. Requiring support for CRDTs in the database also results in a loss of flexibility for the application. New CRDTs can no longer be added by the application, but must be part of the database instead. To the best of our knowledge there is no database using CRDTs that allows an application to extend or program the used CRDTs.

Goldstein et al. [91] argue for using database techniques to provide durability for general purpose applications, by logging all incoming actions for the application and replaying the log in case of failures. This approach allows applying many common database operations, but it also requires applications to be rewritten in a way to be deterministic, based on the received inputs. This has some similarities to the global sequence protocol and is related to the discussion of transactions as monotonic operations in Section 10.3.

Operational Transformation and Text Synchronization

Text editing was one of the original problems of digital collaboration and still remains a challenge. Grishchenko and Patrakeev [94] provide a survey of the history for collaborative text editing. The main strategy used today is operational transformation (OT) where individual operations are sent to remote devices and applied there. To prevent conflicts, the applied operations are transformed to match the current state of the device, e.g., the insertion position of a new character is changed to accommodate for other characters having been inserted previously. The issue with OT is the complexity of the algorithms to ensure correct transformations in a decentralized setting. To address this challenge in practical deployments of such systems, most implementations of OT employ a centralized server to ensure correct ordering and transformation of operations. The alternative to OT are specialized CRDTs. While CRDTs are simpler than OT and work in a peer-to-peer setting, they often have a high overhead for each individual atomic element, i.e., overhead for each single character in a text editing setting. This usually results in high metadata overhead for CRDT-based approaches, although this issue is under active research.

Mixed Consistency

Myter et al. [161] investigate enabling switching between strong consistency and eventual consistency for communication in a single language. They have an eventually consistent data type similar to replicated reactives, but show that strongly consistent data types may co-exist in the same level of abstraction, as long as the application is tolerant to unavailability in case of connection problems. Meiklejohn [144] argues for the need of languages to support both forms of synchronization, based on earlier arguments by Landin [133]. We have explored this direction a little by implementing a consensus algorithm as a replicated reactive in REScala in Section 10.4. This implementation shows that our model does indeed allow multiple types of consistency at the application level, even if our programming paradigm only offers a single form of consistency.

Zhang et al. [207] argue that many traditional applications become distributed, so a flexible model where distribution is specified as library function is needed. Their solution allows customizable behavior for remote communication for each group of objects. However, they make use of centralized services with unclear behavior in case of faults.

ConSysT [123] extends the object-oriented paradigm with a type system that tracks the consistency level of objects. Similar to an information flow type system, ConSysT ensures that there are no dependencies where an object with strong consistency derives its value from an object with weak consistency.

Language-based Concurrency Abstractions

Small-scale concurrency tools, e.g., atomic compare-and-swap operations or locks for mutual exclusion, are ubiquitous, but hard to compose. Language abstractions such as futures and async/await [193] allow composing asynchronous tasks. Software Transactional Memory [188] schedules arbitrary, composable [163] read/write transactions. Reagents [195] are abstractions for implementing data structures that can be safely composed into concurrent algorithms.

14.5 Calculi for Concurrent and Distributed Systems

Several formal calculi have been developed for concurrent and distributed systems. In addition, there are formal approaches that also argue for correctness properties based on high-level programming paradigms, because those are easier to understand by developers. Finally, there is research on type systems that ensure guarantees of the communication between multiple devices.

Concurrent Calculi

There is a wide variety of calculi [51, 85, 106, 149, 150] covering traditional distributed and concurrent systems. These calculi usually base their communication on message passing and use parallel composition of processes. The focus of these

calculi in general is on smaller distributed systems, such as a single processor. Many processes are created and completed during execution, thus the communication structure between processes is always changing. These calculi usually have no inherent concept of unreliable communication over the Internet and instead they specify how the system behaves when messages are reliably sent and received.

Caires [48] adds linear types to the π-calculus, to model protocols including unreliable message delivery. However, even with unreliability addressed at the message level, these calculi do not understand how dataflow of the processes is connected, thus making automatic fault tolerance based on the application semantics hard to achieve.

Lago et al. [130] investigate theoretical foundations for typed runtime errors in the π-calculus. Cloud calculus [111] applies the above to a distributed system, but focuses on upholding security policies when migrating processes between virtual machines, which considers low level equivalence between implementations. More recently, Atkey [26] adds external communication to classical processes with a focus on the external behavior of processes and less on internal details.

The work by von Gleissenthall et al. [199] uses the fact that typical applications have limited communication structures and extracts synchronous specifications from programs with only minimal annotations. Properties of the asynchronous communication behavior of the application are derived and proven based on these annotations. Our calculus $\mathscr{F}_r$ has synchronous behavior which should make such verification techniques applicable.

CPL [42] is a calculus to combine multiple services in a type safe manner. While CPL does not directly provide fault tolerance automatically, it might be used to combine $\mathscr{F}_r$ applications with other services and checking that application level assumptions are not violated.

PL View on Lower-level Synchronization

In the area of low-level consistency and synchronization there is research focusing on the language view of properties and guarantees. Similar to our work, they to provide guarantees matching the expectations of programmers.

Concurrent Objects [81] provides objects with serializability and linearizability by guaranteeing that concurrent implementations behave observational equivalent to sequential implementations, i.e., matches how programmers expect objects to behave. They also want to make concurrency more widely usable by common applications, however they do not consider distribution nor fault tolerance.

Attiya [27] argues that transactional memory must integrate both the systems side of view and the language side. A complex but efficient implementation should provide same behavior as a simpler implementation of the same high-level API – behavior equivalence should not only exist on the level of low level memory cells. This approach is very similar to how $\mathscr{F}_r$ enforces language level guarantees, by managing the lower level runtime systems implementing those guarantees, but $\mathscr{F}_r$ considers distributed systems instead of only local state.

Typed Communication Systems

Chandra and Toueg [56] describe how failed devices are reliably detected in an unreliable network. While such a solution does not provide automatic correctness guarantees, it could be used to detect permanent disconnects in our system. Verdi [204] is a system, which uses an implementation together with a specification to automatically generate reliable message transfer in case of message loss or reordering. However, Verdi does not support use of applications while offline, and it does not handle crashes of application. Both are limitations stemming from reducing the application model to the sent and received messages without considering their complete execution. Viering et al. [197] develop a typing discipline to ensure that correctly typed programs, will continue correct execution even in the presence of faults, but they require a central coordinator – devices disconnected from the central system are no longer considered part of the execution.

Function passing [147] is a style of distributed programming that defines a graph of immutable values and operations over these values. The result is a graph similar to Spark RDDs, but using arbitrary Scala functions instead RDD transformations. However, fault tolerance and reactivity are not part of the paradigm.

The work on ScalaLoci [202] addresses formal reasoning about the correctness of distributed systems with decentralized state. This work is orthogonal to ours in two ways. First, it addresses correctly typed usage of data across the nodes in a distributed dataflow system as an orthogonal correctness aspect. Second, it does so with an orthogonal formal reasoning approach – a static type system. Given this orthogonality, it makes sense to combine these two approaches as we did in our case studies.

14.6 Consistency as a Service

There are several designs that provide a more complete "out of the box" solution to develop dicApps. The issue with these solutions in general – especially the commercial solutions – is their inflexibility which makes it harder to develop new kinds of dicApps.

Cloud Synchronization Frameworks

There are many vendors that offer "a collaboration back end as a service". Including Microsoft, Google, and Twilio. We have discussed Twilio in the context of our TodoMVC case study in Section 11.1. These systems provide a consistent back end as a library and service to the application. Using such a service provides applications with a lot of functionality out of the box – if the application fits the particular structure of these frameworks. However, in most cases there is no way to develop applications with new kinds of features except hoping that the provider of the service will implement the required functionality.

Serverless Becoming Stateful

Serverless computing is a programming environment, where the service provider is responsible for managing all parts of the application stack up to and including the application runtime, for example providing the JVM to run Java applications. Serverless computing typically includes a standardized way to send requests to the application running inside the provided runtime. Serverless provides the computing power that is needed by many data processing applications, but without a way to store intermediate data in a processing pipeline.

There are newer approaches such as Cloudburst [190] that add state to serverless functions. State is desirable to make representing pipelines simpler, and to represent stateful (fold) operations. The goal of these frameworks is presumably to make it easier to develop dicApps by expanding the scope of serverless computing to stateful applications. Most of these approaches seem to work towards a second generation of "map reduce improved" – they want to integrate the runtime into the cloud providers service. The gains of this are actually unclear; it seems like it would be much better to enable runtimes such as Flink to make flexible and fast scaling decisions in the cloud, thus providing the same scaling of serverless approaches but with the strong guarantees of an existing data processing framework.

Local-first Software

The work on local-first software [121] uses eventual consistency to enable offline usage, data privacy, and collaboration. Their overall research has a strong focus on usability of a complete application package. Their approach combines functional and reactive programming (see Section 14.1) with their own CRDT library for JSON [120]. Thus, their software stack provides a complete environment to develop what they call local-first software – a type of dicApps that requires to provide usable functionality offline and store all date with the user. However, their main research question is whether the kind of applications we imagine can be user-friendly at all. Their results are very promising and, because the targeted applications have many similarities, confirm our belief in the Cvtr programming paradigm.

14.7 Debugging and Tuning Approaches

We discussed our extensions for live tuning and debugging in Chapter 9. Other debugging approaches more fitting to different high-level approaches have been proposed. In object-centric debugging [60, 173], breakpoints are set on access or change of a specific objects fields, not only on all instances of a certain class, similar to how Reactive Inspector sets breakpoints on specific reactives in the flow graph.

Programming paradigms with lazy evaluation also have non-sequential execution order, which cause similar problems for debugging. Addressing non-sequential evaluation led to specialized debugging techniques such as oracle debugging [40], where the user interacts with the debugger in a dialog style using questions and

answers. Another approach is to record evaluations in a lazy program and then use a strict debugger to inspect the records [39, 77], similar to our use of the chrome debugger for sequential code in the flow graph.

A common feature of advanced debugging is to the ability to record the execution and inspect the log afterwards. Clematis [21] is such a tool to record all input events (user actions, incoming server messages, timeouts) and outputs (DOM changes, network request) in a web application. Sahand [20] is an extension to Clematis and additionally records all server inputs (incoming client messages, etc.). These events are correlated and visualized as stories – a graphical representation of the execution not unlike our flow graph. However, these stories are sequential in nature and thus not as easy to follow as the flow graph, and also do not lend themselves to modifications. Barr et al. [30, 31] record execution logs of imperative applications. They use those logs to enable forward and backward navigation in a visualizing debugger similar to our history navigation. By taking advantage of information in the garbage collector about the running program, they reduce the amount of memory needed to record. However, in contrast to the flow graph, the state of the imperative application cannot be changed after recording, prohibiting modifications. Rxfiddle [29] allows one to visualize the marble diagram and the dataflow of a reactive application. They have not considered modifications.

There are some tools that support live modifications of applications: The Elm debugger [5] allows one to go back in the past, but modifications have to be done on the source code. Kato and Goto [119] have presented the idea of live tuning, where users are presented with a simplified interface – only the tuning sliders – of a full-fledged live programming IDE. We use this idea to provide live tuning based on an extended debugger instead of a full IDE. ZenSheet [18] pushes spreadsheets towards general purpose programming, whereas the reactive part of Cvtr can be described as spreadsheet semantics for general purpose programming languages. Our approach has the advantage that existing tools, libraries, and infrastructure can be reused, however at the cost of less flexible changes possible at runtime. We believe that both approaches provide their own value, and it will be interesting to see how far research on each approach can push the boundaries in the corresponding directions.

Chapter 15

Conclusion and Future Work

Fault-tolerant dicApps are easy to achieve when a suitable programming paradigm – such as Cvtr – is used. It is well known, that it is impossible to achieve the guarantees of a single reliable device in a distributed system. However, the Cvtr programming paradigm does not need strong consistency to provide interactivity and collaboration. Users of dicApps have requirements that align well with the realities of distributed systems.

To address the remaining challenges of application development, we provide developers with automatic fault tolerance for dicApps. We have formalized the core model of the programming paradigm and proven the correctness guarantees that it provides. We have proven causal consistency and crash restoration together with transactional guarantees. REScala implements the programming paradigm, keeping all of its guarantees, while adding modular integration with other paradigms. We have seen that REScala is suitable to implement case studies that – combined – face all the challenges of dicApps. Integration with exceptions, compilation to embedded devices, and a protocol for debugging and live modifications show that the programming paradigm and its implementation has a broad potential for extensions without sacrificing its core guarantees.

Looking at the larger picture, our reason to design a programming paradigm with automatic fault tolerance was to democratize the development of dicApps. We do believe that REScala is a significant step in enabling many developers to create correct and useful dicApps quickly. However, there are interesting directions for future improvements.

Cvtr chooses causal consistency as the only form of communication between devices. But, we also discussed the use of consensus as part of a replicated reactive in Section 10.4, and Drechsler [68] implements serializable distributed transactions for REScala. Thus, developers already have a choice of potential consistency guarantees available. However, developers want to reason about the behavior of their application – the application's invariants – not choose a combination of consistency guarantees and worry about if those fulfill their requirements. Whittaker and Hellerstein [203] show one solution to automatically combine multiple forms of consistency based on invariants given for an application. These results align well

with the goal of Cvtr to let developers focus on the applications semantics and automate fault tolerance. We believe it is possible to enhance the existing knowledge about the applications structure of Cvtr with invariants to provide a correct mix of different consistency levels automatically.

Consider deployment of dicApps on many devices. We have discussed in Chapter 8 that the Cvtr programming paradigm is suited to be compiled to embedded devices and our case studies have shown that REScala can run everywhere from smartphones to laptops and servers. Currently, developers have to manually decide on which device a reactive is executed. This fits the classic execution model where the responsibilities of classes of devices (such as "client" and "server") can be predicted during development. However, consider a computation that requires specialized resources such as a GPU. It is conceivable that future extensions to Cvtr could automatically place such a computation on a fitting device at runtime, because the structure of the flow graph enables reasoning about how the involved data must be moved. The problem becomes more relevant if many classes of devices exist in the network. Networks with many device classes seem to be common in IoT deployments, with devices ranging from small sensors, to routers, laptops, and even home servers. With capabilities such as compiling for Wi-Fi chips, the need for automatic deployments even occurs on a single device using different types of processors. A device that has a programmable Wi-Fi chip (or GPU, etc.) may want to offload computation to that specialized hardware. Because it may not be clear during development of applications what device capabilities are available, it is desirable to have the runtime automatically place computations based on some optimization criteria.

Also related to deployment on many devices – but many similar devices in this case – are clusters and "the cloud". We have discussed that distributed data processing platforms already provide a suitable model for effectively using computation resources in a cluster. However, it is easy to conceive applications that are a mix out of dicApps and data processing systems. For example, video editing, computer aided design, 3D rendering, etc., are interactive and collaborative applications, but they also need to execute typical data processing tasks. The programming paradigms underlying systems such as Flink (c.f., Section 14.3) are similar to Cvtr – they also describe a form of flow graph. Currently, Cvtr has no concept of executing the same computation on different data in parallel, but investigating a combination of these programming paradigms could eventually lead to a unified model for dicApps and data processing systems in the future. Drechsler [68] already shows how to pipeline processing of values in the flow graph and a similar strategy could be applied to process events in parallel.

Versatile event correlation [41] provides data processing systems with much more flexible specifications for computations. Cvtr currently treats user-defined computations as black boxes, and this black box could contain an event correlation system. However, versatile event correlation has the potential to allow reasoning about the order and time of transactions, and also provides declarative specifications of complex interactions of multiple reactives. Thus, versatile event correlation

interacts with core parts of Cvtr and an integration of the two paradigms could potentially lead to better guarantees – especially if the correlation engine could be used to reason about eventually consistent reactives.

Finally, there are other programming paradigms that build something akin to the flow graph. One example is TensorFlow [12], which composes operations on multidimensional matrices (tensors) into a graph to compute the derivate of the composed operations. Considering the graph-based nature of such programming paradigm raises the question of what their relation to Cvtr is and if there is an elegant unification of such paradigms. Maybe there is a future where the prevalent programming paradigms are those that use declarative connections between their individual building blocks.

Bibliography

[1] SID-UP repository. https://github.com/stg-tud/SID-UP, 2015.

[2] Java microbenchmark harness. http://openjdk.java.net/projects/code-tools/jmh/, 2017.

[3] Flink success stories. https://cwiki.apache.org/confluence/display/FLINK/Powered+by+Flink, 2017.

[4] HTML5 localStorage. https://www.w3schools.com/html/html5_webstorage.asp, 2017.

[5] Elm's time traveling debugger. http://debug.elm-lang.org/, 2018.

[6] Elm. http://elm-lang.org/, 2018.

[7] A JSON library for Scala powered by Cats. https://circe.github.io/circe/, 2020.

[8] Jsoniter benchmarks. https://plokhotnyuk.github.io/jsoniter-scala/, 2020.

[9] RealWorld Conduit. https://github.com/gothinkster/realworld#frontends, 2020.

[10] ReactiveX introduction. http://reactivex.io/intro.html, 2020. Accessed 2021-01-10.

[11] ReactiveX operators. http://reactivex.io/documentation/operators.html, 2020. Accessed 2021-01-10.

[12] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems, 2016. URL http://arxiv.org/abs/1603.04467.

[13] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. ISBN 0-201-53771-0. URL http://webdam.inria.fr/Alice/.

[14] Addy, Sindre, Pascal, Stephen, Colin, Arthur, and Sam. TodoMVC: Helping you select an MV* framework. http://todomvc.com/, 2020.

[15] Gul A. Agha. *ACTORS - a model of concurrent computation in distributed systems*. MIT Press series in artificial intelligence. MIT Press, 1990. ISBN 978-0-262-01092-4.

[16] Akka. Distributed data clusters. https://doc.akka.io/docs/akka/current/distributed-data.html, 2019.

[17] Akka. Documentation. http://akka.io/docs, 2019.

[18] Enzo Alda and Monica Figuera. ZenSheet: A live programming environment for reactive computing. In *Workshop on Live Programming Systems (LIVE)*, 2017.

[19] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. The stratosphere platform for big data analytics. *VLDB J.*, 23(6): 939–964, 2014. doi: 10.1007/s00778-014-0357-y. URL https://doi.org/10.1007/s00778-014-0357-y.

[20] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. Understanding asynchronous interactions in full-stack javascript. In Laura K. Dillon, Willem Visser, and Laurie Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 1169–1180. ACM, 2016. doi: 10.1145/2884781.2884864. URL https://doi.org/10.1145/2884781.2884864.

[21] Saba Alimadadi, Sheldon Sequeira, Ali Mesbah, and Karthik Pattabiraman. Understanding javascript event-based interactions with clematis. *ACM Trans. Softw. Eng. Methodol.*, 25(2):12:1–12:38, 2016. doi: 10.1145/2876441. URL https://doi.org/10.1145/2876441.

[22] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Delta state replicated data types. *J. Parallel Distributed Comput.*, 111:162–173, 2018. doi: 10.1016/j.jpdc.2017.08.003. URL https://doi.org/10.1016/j.jpdc.2017.08.003.

[23] Peter Alvaro, Peter Bailis, Neil Conway, and Joseph M. Hellerstein. Consistency without borders. In Guy M. Lohman, editor, *ACM Symposium on Cloud Computing, SOCC '13, Santa Clara, CA, USA, October 1-3, 2013*, pages 23:1–23:10. ACM, 2013. doi: 10.1145/2523616.2523632. URL https://doi.org/10.1145/2523616.2523632.

[24] Roberto M. Amadio, Gérard Boudol, Frédéric Boussinot, and Ilaria Castellani. Reactive concurrent programming revisited. *Electron. Notes Theor. Comput. Sci.*, 162:49–60, 2006. doi: 10.1016/j.entcs.2005.12.104. URL https://doi.org/10.1016/j.entcs.2005.12.104.

[25] Joe Armstrong. Erlang. *Commun. ACM*, 53(9):68–75, 2010. doi: 10.1145/1810891.1810910. URL https://doi.org/10.1145/1810891.1810910.

[26] Robert Atkey. Observed communication semantics for classical processes. In Hongseok Yang, editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala,*

Sweden, April 22-29, 2017, Proceedings, volume 10201 of *Lecture Notes in Computer Science*, pages 56–82. Springer, 2017. doi: 10.1007/978-3-662-54434-1_3. URL https://doi.org/10.1007/978-3-662-54434-1_3.

[27] Hagit Attiya, Alexey Gotsman, Sandeep Hans, and Noam Rinetzky. A programming language perspective on transactional memory consistency. In Panagiota Fatourou and Gadi Taubenfeld, editors, *ACM Symposium on Principles of Distributed Computing, PODC '13, Montreal, QC, Canada, July 22-24, 2013*, pages 309–318. ACM, 2013. doi: 10.1145/2484239.2484267. URL https://doi.org/10.1145/2484239.2484267.

[28] Engineer Bainomugisha, Andoni Lombide Carreton, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. A survey on reactive programming. *ACM Comput. Surv.*, 45(4):52:1–52:34, 2013. doi: 10.1145/2501654.2501666. URL https://doi.org/10.1145/2501654.2501666.

[29] Herman Banken, Erik Meijer, and Georgios Gousios. Debugging data flows in reactive programs. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 752–763. ACM, 2018. doi: 10.1145/3180155.3180156. URL https://doi.org/10.1145/3180155.3180156.

[30] Earl T. Barr and Mark Marron. Tardis: Affordable time-travel debugging in managed runtimes. In Andrew P. Black and Todd D. Millstein, editors, *Proceedings of the 2014 ACM International Conference on Object-Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 67–82. ACM, 2014. doi: 10.1145/2660193.2660209. URL https://doi.org/10.1145/2660193.2660209.

[31] Earl T. Barr, Mark Marron, Ed Maurer, Dan Moseley, and Gaurav Seth. Time-travel debugging for javascript/node.js. In Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su, editors, *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 1003–1007. ACM, 2016. doi: 10.1145/2950290.2983933. URL https://doi.org/10.1145/2950290.2983933.

[32] Lars Baumgärtner, Jonas Höchst, Patrick Lampe, Ragnar Mogk, Artur Sterz, Pascal Weisenburger, Mira Mezini, and Bernd Freisleben. Smart street lights and mobile citizen apps for resilient communication in a digital city. In *IEEE Global Humanitarian Technology Conference, GHTC 2019, Seattle, WA, USA, October 17-20, 2019*, pages 1–8. IEEE, 2019. doi: 10.1109/GHTC46095.2019.9033134. URL https://doi.org/10.1109/GHTC46095.2019.9033134.

[33] P. Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. Orleans: Distributed virtual actors for programmability and scalability. Technical report, (MSR-TR-2014-41, 24), 2014. URL http://aka.ms/Ykyqft.

[34] Philip A. Bernstein, Sebastian Burckhardt, Sergey Bykov, Natacha Crooks, Jose M. Faleiro, Gabriel Kliot, Alok Kumbhare, Muntasir Raihan Rahman, Vivek Shah, Adriana Szekeres, and Jorgen Thelin. Geo-distribution of actor-based services. *Proc. ACM Program. Lang.*, 1(OOPSLA):107:1–107:26, 2017. doi: 10.1145/3133931. URL https://doi.org/10.1145/3133931.

[35] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.*, 19(2): 87–152, 1992. doi: 10.1016/0167-6423(92)90005-V. URL https://doi.org/10.1016/0167-6423(92)90005-V.

[36] BMWi. GAIA-X: Eine vernetzte datenstruktur für ein europäisches digitales Ökosystem. https://www.bmwi.de/Redaktion/DE/Dossier/gaia-x.html, 2021.

[37] Elisa Gonzalez Boix, Kevin Pinte, and Wolfgang De Meuter. Object-oriented reactive programming is not reactive object-oriented programming. 2013. URL http://soft.vub.ac.be/Publications/2013/vub-soft-tr-13-16.pdf.

[38] Frédéric Boniol and Martin Adelantado. Programming distributed reactive systems: a strong and weak synchronous coupling. In André Schiper, editor, *Distributed Algorithms, 7th International Workshop, WDAG '93, Lausanne, Switzerland, September 27-29, 1993, Proceedings*, volume 725 of *Lecture Notes in Computer Science*, pages 294–308. Springer, 1993. doi: 10.1007/3-540-57271-6_43. URL https://doi.org/10.1007/3-540-57271-6_43.

[39] Bernd Braßel. A technique to build debugging tools for lazy functional logic languages. *Electron. Notes Theor. Comput. Sci.*, 246:39–53, 2009. doi: 10.1016/j.entcs.2009.07.014. URL https://doi.org/10.1016/j.entcs.2009.07.014.

[40] Bernd Braßel and Holger Siegel. Debugging lazy functional programs by asking the oracle. In Olaf Chitil, Zoltán Horváth, and Viktória Zsók, editors, *Implementation and Application of Functional Languages, 19th International Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007. Revised Selected Papers*, volume 5083 of *Lecture Notes in Computer Science*, pages 183–200. Springer, 2007. doi: 10.1007/978-3-540-85373-2_11. URL https://doi.org/10.1007/978-3-540-85373-2_11.

[41] Oliver Bračevac. *Event Correlation with Algebraic Effects - Theory, Design and Implementation*. PhD thesis, Technische Universität Damstadt, Germany, 2019. URL https://tuprints.ulb.tu-darmstadt.de/9258/.

[42] Oliver Bračevac, Sebastian Erdweg, Guido Salvaneschi, and Mira Mezini. CPL: a core language for cloud computing. In Lidia Fuentes, Don S. Batory, and Krzysztof Czarnecki, editors, *Proceedings of the 15th International Conference on Modularity, MODULARITY 2016, Málaga, Spain, March 14 - 18, 2016*, pages 94–105. ACM, 2016. doi: 10.1145/2889443.2889452. URL https://doi.org/10.1145/2889443.2889452.

[43] Nathan G. Bronson, Hassan Chafi, and Kunle Olukotun. CCSTM: A library-based STM for Scala. In *The First Annual Scala Workshop at Scala Days*, 2010.

[44] Peter A. Buhr and W. Y. Russell Mok. Advanced exception handling mechanisms. *IEEE Trans. Software Eng.*, 26(9):820–836, 2000. doi: 10.1109/32.877844. URL https://doi.org/10.1109/32.877844.

[45] Sebastian Burckhardt and Tim Coppieters. Reactive caching for composed services: Polling at the speed of push. *Proc. ACM Program. Lang.*, 2(OOPSLA): 152:1–152:28, 2018. doi: 10.1145/3276522. URL https://doi.org/10.1145/3276522.

[46] Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P. Wood. Cloud types for eventual consistency. In James Noble, editor, *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*, volume 7313 of *Lecture Notes in Computer Science*, pages 283–307. Springer, 2012. doi: 10.1007/978-3-642-31057-7_14. URL https://doi.org/10.1007/978-3-642-31057-7_14.

[47] Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich. Global sequence protocol: A robust abstraction for replicated shared state. In John Tang Boyland, editor, *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, volume 37 of *LIPIcs*, pages 568–590. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015. doi: 10.4230/LIPIcs.ECOOP.2015.568. URL https://doi.org/10.4230/LIPIcs.ECOOP.2015.568.

[48] Luís Caires and Jorge A. Pérez. Linearity, control effects, and behavioral types. In Hongseok Yang, editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10201 of *Lecture Notes in Computer Science*, pages 229–259. Springer, 2017. doi: 10.1007/978-3-662-54434-1_9. URL https://doi.org/10.1007/978-3-662-54434-1_9.

[49] Ben Calus, Bob Reynders, Dominique Devriese, Job Noorman, and Frank Piessens. FRP IoT modules as a Scala DSL. In Guido Salvaneschi, Wolfgang De Meuter, Patrick Eugster, and Lukasz Ziarek, editors, *Proceedings of the 4th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems, Vancouver, BC, Canada, October 23, 2017*, pages 15–20. ACM, 2017. doi: 10.1145/3141858.3141861. URL https://doi.org/10.1145/3141858.3141861.

[50] Paris Carbone, Gyula Fóra, Stephan Ewen, Seif Haridi, and Kostas Tzoumas. Lightweight asynchronous snapshots for distributed dataflows. *CoRR*, abs/1506.08603, 2015. URL http://arxiv.org/abs/1506.08603.

[51] Luca Cardelli and Andrew D. Gordon. Types for mobile ambients. In Andrew W. Appel and Alex Aiken, editors, *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, pages 79–92. ACM, 1999. doi: 10.1145/292540.292550. URL https://doi.org/10.1145/292540.292550.

[52] Andoni Lombide Carreton, Stijn Mostinckx, Tom Van Cutsem, and Wolfgang De Meuter. Loosely-coupled distributed reactive programming in mobile ad hoc networks. In Jan Vitek, editor, *Objects, Models, Components, Patterns, 48th International Conference, TOOLS 2010, Málaga, Spain, June 28 - July 2, 2010. Proceedings*, volume 6141 of *Lecture Notes in Computer Science*, pages 41–60. Springer, 2010. doi: 10.1007/978-3-642-13953-6_3. URL https://doi.org/10.1007/978-3-642-13953-6_3.

[53] Andoni Lombide Carreton, Stijn Mostinckx, Tom Van Cutsem, and Wolfgang De Meuter. Loosely-coupled distributed reactive programming in mobile ad hoc networks. In Jan Vitek, editor, *Objects, Models, Components,*

Patterns, 48th International Conference, TOOLS 2010, Málaga, Spain, June 28 - July 2, 2010. Proceedings, volume 6141 of *Lecture Notes in Computer Science*, pages 41–60. Springer, 2010. doi: 10.1007/978-3-642-13953-6_3. URL https://doi.org/10.1007/978-3-642-13953-6_3.

[54] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre: A declarative language for programming synchronous systems. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*, pages 178–188. ACM Press, 1987. doi: 10.1145/41625.41641. URL https://doi.org/10.1145/41625.41641.

[55] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. Flumejava: Easy, efficient data-parallel pipelines. In Benjamin G. Zorn and Alexander Aiken, editors, *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 363–375. ACM, 2010. doi: 10.1145/1806596.1806638. URL https://doi.org/10.1145/1806596.1806638.

[56] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996. doi: 10.1145/226643.226647. URL https://doi.org/10.1145/226643.226647.

[57] Adam Chlipala. Ur/web: A simple model for programming the web. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 153–165. ACM, 2015. doi: 10.1145/2676726.2677004. URL https://doi.org/10.1145/2676726.2677004.

[58] Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. Logic and lattices for distributed programming. In Michael J. Carey and Steven Hand, editors, *ACM Symposium on Cloud Computing, SOCC '12, San Jose, CA, USA, October 14-17, 2012*, page 1. ACM, 2012. doi: 10.1145/2391229.2391230. URL https://doi.org/10.1145/2391229.2391230.

[59] Gregory H. Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In Peter Sestoft, editor, *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings*, volume 3924 of *Lecture Notes in Computer Science*, pages 294–308. Springer, 2006. doi: 10.1007/11693024_20. URL https://doi.org/10.1007/11693024_20.

[60] Claudio Corrodi. Towards efficient object-centric debugging with declarative breakpoints. In Mircea Lungu, Anya Helene Bagge, and Haidar Osman, editors, *Post-proceedings of the 9th Seminar on Advanced Techniques and Tools for Software Evolution, Bergen, Norway, July 11-13, 2016*, volume 1791 of *CEUR Workshop Proceedings*, pages 32–39. CEUR-WS.org, 2016. URL http://ceur-ws.org/Vol-1791/paper-04.pdf.

[61] Antony Courtney. Frappé: Functional reactive programming in java. In I. V. Ramakrishnan, editor, *Practical Aspects of Declarative Languages, Third International Symposium, PADL 2001, Las Vegas, Nevada, USA, March 11-12, 2001, Proceedings*, volume 1990 of *Lecture Notes in Computer Science*, pages 29–44. Springer, 2001. doi: 10.1007/3-540-45241-9_3. URL https://doi.org/10.1007/3-540-45241-9_3.

[62] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3): 15:1–15:62, 2012. doi: 10.1145/2187671.2187677. URL https://doi.org/10.1145/2187671.2187677.

[63] Tom Van Cutsem, Stijn Mostinckx, Elisa Gonzalez Boix, Jessie Dedecker, and Wolfgang De Meuter. Ambienttalk: Object-oriented event-driven programming in mobile ad hoc networks. In *XXVI International Conference of the Chilean Computer Science Society (SCCC 2007), 8-9 November 2007, Iquique, Chile*, pages 3–12. IEEE Computer Society, 2007. doi: 10.1109/SCCC.2007.4. URL https://doi.org/10.1109/SCCC.2007.4.

[64] Tom Van Cutsem, Elisa Gonzalez Boix, Christophe Scholliers, Andoni Lombide Carreton, Dries Harnie, Kevin Pinte, and Wolfgang De Meuter. Ambienttalk: Programming responsive mobile peer-to-peer applications with actors. *Comput. Lang. Syst. Struct.*, 40(3-4):112–136, 2014. doi: 10.1016/j.cl.2014.05.002. URL https://doi.org/10.1016/j.cl.2014.05.002.

[65] Evan Czaplicki and Stephen Chong. Asynchronous functional reactive programming for guis. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 411–422. ACM, 2013. doi: 10.1145/2491956.2462161. URL https://doi.org/10.1145/2491956.2462161.

[66] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008. doi: 10.1145/1327452.1327492. URL http://doi.acm.org/10.1145/1327452.1327492.

[67] Sébastien Doeraene. Scala.js: Type-directed interoperability with dynamically typed languages. Technical report, EPFL, 2013.

[68] Joscha Drechsler. *Concurrency and Distribution in Reactive Programming*. PhD thesis, Darmstadt University of Technology, Germany, 2019. URL http://tuprints.ulb.tu-darmstadt.de/5228/.

[69] Joscha Drechsler, Guido Salvaneschi, Ragnar Mogk, and Mira Mezini. Distributed REScala: An update algorithm for distributed reactive programming. In Andrew P. Black and Todd D. Millstein, editors, *Proceedings of the 2014 ACM International Conference on Object-Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 361–376. ACM, 2014. doi: 10.1145/2660193.2660240. URL https://doi.org/10.1145/2660193.2660240.

[70] Joscha Drechsler, Ragnar Mogk, Guido Salvaneschi, and Mira Mezini. Thread-safe reactive programming. *Proc. ACM Program. Lang.*, 2(OOPSLA):107:1–107:30, 2018. doi: 10.1145/3276477. URL https://doi.org/10.1145/3276477.

[71] Jan Dzik, Nick Palladinos, Konstantinos Rontogiannis, Eirik Tsarpalis, and Nikolaos Vathis. Mbrace: Cloud computing with monads. In Tim Harris and Anil Madhavapeddy, editors, *Proceedings of the Seventh Workshop on Programming Languages and Operating Systems, PLOS 2013, Farmington, Pennsylvania, USA, November 3-6, 2013*, pages 7:1–7:6. ACM, 2013. doi: 10.1145/2525528.2525531. URL https://doi.org/10.1145/2525528.2525531.

[72] Jonathan Edwards. Coherent reaction. In Shail Arora and Gary T. Leavens, editors, *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, pages 925–932. ACM, 2009. doi: 10.1145/1639950.1640058. URL https://doi.org/10.1145/1639950.1640058.

[73] Conal Elliott. Functional implementations of continuos modeled animation. In Catuscia Palamidessi, Hugh Glaser, and Karl Meinke, editors, *Principles of Declarative Programming, 10th International Symposium, PLILP'98 Held Jointly with the 7th International Conference, ALP'98, Pisa, Italy, September 16-18, 1998, Proceedings*, volume 1490 of *Lecture Notes in Computer Science*, pages 284–299. Springer, 1998. doi: 10.1007/BFb0056621. URL https://doi.org/10.1007/BFb0056621.

[74] Conal Elliott. Why program with continuous time? http://conal.net/blog/posts/why-program-with-continuous-time, 2010.

[75] Conal Elliott and Paul Hudak. Functional reactive animation. In Simon L. Peyton Jones, Mads Tofte, and A. Michael Berman, editors, *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9-11, 1997*, pages 263–273. ACM, 1997. doi: 10.1145/258948.258973. URL https://doi.org/10.1145/258948.258973.

[76] Conal M. Elliott. Push-pull functional reactive programming. In Stephanie Weirich, editor, *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell, Haskell 2009, Edinburgh, Scotland, UK, 3 September 2009*, pages 25–36. ACM, 2009. doi: 10.1145/1596638.1596643. URL https://doi.org/10.1145/1596638.1596643.

[77] Robert Ennals and Simon L. Peyton Jones. Hsdebug: Debugging lazy programs by not being lazy. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2003, Uppsala, Sweden, August 28, 2003*, pages 84–87. ACM, 2003. doi: 10.1145/871895.871904. URL https://doi.org/10.1145/871895.871904.

[78] Patrick Th. Eugster, Pascal Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2): 114–131, 2003. doi: 10.1145/857076.857078. URL https://doi.org/10.1145/857076.857078.

[79] Facebook. React: A JavaScript library for building user interfaces. https://reactjs.org/, 2020. Accessed 2021-01-01.

[80] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, 103(2):235–271, 1992. doi: 10.1016/0304-3975(92)90014-7. URL https://doi.org/10.1016/0304-3975(92)90014-7.

[81] Ivana Filipovic, Peter W. O'Hearn, Noam Rinetzky, and Hongseok Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 411(51-52):4379–4398, 2010. doi: 10.1016/j.tcs.2010.09.021. URL https://doi.org/10.1016/j.tcs.2010.09.021.

[82] Bernd Finkbeiner, Felix Klein, Ruzica Piskac, and Mark Santolucito. Vehicle platooning simulations with functional reactive programming. In *Proceedings of the 1st International Workshop on Safe Control of Connected and Autonomous Vehicles, SCAV@CPSWeek 2017, Pittsburgh, PA, USA, April 21, 2017*, pages 43–47. ACM, 2017. doi: 10.1145/3055378.3055385. URL https://doi.org/10.1145/3055378.3055385.

[83] Spencer P. Florence, Shu-Hung You, Jesse A. Tov, and Robert Bruce Findler. A calculus for esterel: If can, can. if no can, no can. *Proc. ACM Program. Lang.*, 3(POPL):61:1–61:29, 2019. doi: 10.1145/3290374. URL https://doi.org/10.1145/3290374.

[84] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 279–291. ACM, 2011. doi: 10.1145/2034773.2034812. URL https://doi.org/10.1145/2034773.2034812.

[85] Cédric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In Hans-Juergen Boehm and Guy L. Steele Jr., editors, *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, pages 372–385. ACM Press, 1996. doi: 10.1145/237721.237805. URL https://doi.org/10.1145/237721.237805.

[86] Gamma, Helm, Johnson, and Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, 2000. ISBN 0-201-63361-2.

[87] Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. Dependent classes. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 133–152. ACM, 2007. doi: 10.1145/1297027.1297038. URL https://doi.org/10.1145/1297027.1297038.

[88] Vaidas Gasiunas, Lucas Satabin, Mira Mezini, Angel Núñez, and Jacques Noyé. EScala: Modular event-driven object interactions in Scala. In Paulo Borba and Shigeru Chiba, editors, *Proceedings of the 10th International Conference on Aspect-Oriented Software Development, AOSD 2011, Porto de Galinhas, Brazil, March 21-25, 2011*, pages 227–240. ACM, 2011. doi: 10.1145/1960275.1960303. URL https://doi.org/10.1145/1960275.1960303.

[89] Thierry Gautier and Paul Le Guernic. SIGNAL: A declarative language for synchronous programming of real-time systems. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture, Portland, Oregon, USA, September 14-16, 1987, Proceedings*, volume 274 of *Lecture Notes in Computer Science*, pages 257–277. Springer, 1987. doi: 10.1007/3-540-18317-5_15. URL https://doi.org/10.1007/3-540-18317-5_15.

[90] Seth Gilbert and Nancy A. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2): 51–59, 2002. doi: 10.1145/564585.564601. URL https://doi.org/10.1145/564585.564601.

[91] Jonathan Goldstein, Ahmed S. Abdelhamid, Mike Barnett, Sebastian Burckhardt, Badrish Chandramouli, Darren Gehring, Niel Lebeck, Christopher Meiklejohn, Umar Farooq Minhas, Ryan Newton, Rahee Peshawaria, Tal Zaccai, and Irene Zhang. A.M.B.R.O.S.I.A: providing performant virtual resiliency for distributed applications. *Proc. VLDB Endow.*, 13(5):588–601, 2020. URL http://www.vldb.org/pvldb/vol13/p588-goldstein.pdf.

[92] James Gosling, William N. Joy, and Guy L. Steele Jr. *The Java Language Specification*. Addison-Wesley, 1996. ISBN 0-201-63451-1.

[93] Alexey Gotsman and Sebastian Burckhardt. Consistency models with global operation sequencing and their composition. In Andréa W. Richa, editor, *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, volume 91 of *LIPIcs*, pages 23:1–23:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi: 10.4230/LIPIcs.DISC.2017.23. URL https://doi.org/10.4230/LIPIcs.DISC.2017.23.

[94] Victor Grishchenko and Mikhail Patrakeev. Chronofold: a data structure for versioned text. In Alan Fekete and Martin Kleppmann, editors, *7th Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC@EuroSys 2020, Heraklion, Greece, April 27, 2020*, pages 7:1–7:9. ACM, 2020. doi: 10.1145/3380787.3393680. URL https://doi.org/10.1145/3380787.3393680.

[95] Rui Gu, Guoliang Jin, Linhai Song, Linjie Zhu, and Shan Lu. What change history tells us about thread synchronization. In Elisabetta Di Nitto, Mark Harman, and Patrick Heymans, editors, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 426–438. ACM, 2015. doi: 10.1145/2786805.2786815. URL https://doi.org/10.1145/2786805.2786815.

[96] Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. Incremental consistency guarantees for replicated objects. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 169–184. USENIX Association, 2016. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/guerraoui.

[97] Philipp Haller and Martin Odersky. Event-based programming without inversion of control. In David E. Lightfoot and Clemens A. Szyperski, editors, *Modular Programming Languages, 7th Joint Modular Languages Conference, JMLC 2006, Oxford, UK, September 13-15, 2006, Proceedings*, volume

4228 of *Lecture Notes in Computer Science*, pages 4–22. Springer, 2006. doi: 10.1007/11860990_2. URL https://doi.org/10.1007/11860990_2.

[98] Matthew A. Hammer, Umut A. Acar, Mohan Rajagopalan, and Anwar M. Ghuloum. A proposal for parallel self-adjusting computation. In Neal Glew and Guy E. Blelloch, editors, *Proceedings of the POPL 2007 Workshop on Declarative Aspects of Multicore Programming, DAMP 2007, Nice, France, January 16, 2007*, pages 3–9. ACM, 2007. doi: 10.1145/1248648.1248651. URL https://doi.org/10.1145/1248648.1248651.

[99] Derin Harmanci, Vincent Gramoli, and Pascal Felber. Atomic boxes: Coordinated exception handling with transactional memory. In Mira Mezini, editor, *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings*, volume 6813 of *Lecture Notes in Computer Science*, pages 634–657. Springer, 2011. doi: 10.1007/978-3-642-22655-7_29. URL https://doi.org/10.1007/978-3-642-22655-7_29.

[100] Robert Harper. Self-adjusting computation. In Josep Díaz, Juhani Karhumäki, Arto Lepistö, and Donald Sannella, editors, *Automata, Languages and Programming: 31st International Colloquium, ICALP 2004, Turku, Finland, July 12-16, 2004. Proceedings*, volume 3142 of *Lecture Notes in Computer Science*, pages 1–2. Springer, 2004. doi: 10.1007/978-3-540-27836-8_1. URL https://doi.org/10.1007/978-3-540-27836-8_1.

[101] Tim Harris and Keir Fraser. Language support for lightweight transactions. In Ron Crocker and Guy L. Steele Jr., editors, *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003, October 26-30, 2003, Anaheim, CA, USA*, pages 388–402. ACM, 2003. doi: 10.1145/949305.949340. URL https://doi.org/10.1145/949305.949340.

[102] Joseph M. Hellerstein. The declarative imperative: Experiences and conjectures in distributed logic. *SIGMOD Rec.*, 39(1):5–19, 2010. doi: 10.1145/1860702.1860704. URL https://doi.org/10.1145/1860702.1860704.

[103] Joseph M. Hellerstein and Peter Alvaro. Keeping CALM: when distributed consistency is easy. *CoRR*, abs/1901.01930, 2019. URL http://arxiv.org/abs/1901.01930.

[104] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software transactional memory for dynamic-sized data structures. In Elizabeth Borowsky and Sergio Rajsbaum, editors, *Proceedings of the Twenty-Second ACM Symposium on Principles of Distributed Computing, PODC 2003, Boston, Massachusetts, USA, July 13-16, 2003*, pages 92–101. ACM, 2003. doi: 10.1145/872035.872048. URL https://doi.org/10.1145/872035.872048.

[105] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. doi: 10.1145/363235.363259. URL https://doi.org/10.1145/363235.363259.

[106] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978. doi: 10.1145/359576.359585. URL https://doi.org/10.1145/359576.359585.

[107] William A Howard. The formulae-as-types notion of construction, 1980.

[108] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In Johan Jeuring and Simon L. Peyton Jones, editors, *Advanced Functional Programming, 4th International School, AFP 2002, Oxford, UK, August 19-24, 2002, Revised Lectures*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer, 2002. doi: 10.1007/978-3-540-44833-4_6. URL https://doi.org/10.1007/978-3-540-44833-4_6.

[109] Michael Isard and Yuan Yu. Distributed data-parallel computing using a high-level programming language. In Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 987–994. ACM, 2009. doi: 10.1145/1559845.1559962. URL https://doi.org/10.1145/1559845.1559962.

[110] Radha Jagadeesan and James Riely. Eventual consistency for crdts. In Amal Ahmed, editor, *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10801 of *Lecture Notes in Computer Science*, pages 968–995. Springer, 2018. doi: 10.1007/978-3-319-89884-1_34. URL https://doi.org/10.1007/978-3-319-89884-1_34.

[111] Yosr Jarraya, Arash Eghtesadi, Mourad Debbabi, Ying Zhang, and Makan Pourzandi. Cloud calculus: Security verification in elastic cloud computing platform. In Waleed W. Smari and Geoffrey Charles Fox, editors, *2012 International Conference on Collaboration Technologies and Systems, CTS 2012, Denver, CO, USA, May 21-25, 2012*, pages 447–454. IEEE, 2012. doi: 10.1109/CTS.2012.6261089. URL https://doi.org/10.1109/CTS.2012.6261089.

[112] Alan Jeffrey. LTL types FRP: linear-time temporal logic propositions as types, proofs as functional reactive programs. In Koen Claessen and Nikhil Swamy, editors, *Proceedings of the sixth workshop on Programming Languages meets Program Verification, PLPV 2012, Philadelphia, PA, USA, January 24, 2012*, pages 49–60. ACM, 2012. doi: 10.1145/2103776.2103783. URL https://doi.org/10.1145/2103776.2103783.

[113] Alan Jeffrey. Causality for free!: Parametricity implies causality for functional reactive programs. In Matthew Might, David Van Horn, Andreas Abel, and Tim Sheard, editors, *Proceedings of the 7th Workshop on Programming languages meets program verification, PLPV 2013, Rome, Italy, January 22, 2013*, pages 57–68. ACM, 2013. doi: 10.1145/2428116.2428127. URL https://doi.org/10.1145/2428116.2428127.

[114] Alan Jeffrey. Functional reactive programming with liveness guarantees. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 233–244. ACM, 2013. doi: 10.1145/2500365.2500584. URL https://doi.org/10.1145/2500365.2500584.

[115] Alan Jeffrey. Functional reactive types. In Thomas A. Henzinger and Dale Miller, editors, *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, pages 54:1–54:9. ACM, 2014. doi: 10.1145/2603088.2603106. URL https://doi.org/10.1145/2603088.2603106.

[116] Hyungsoo Jung, Hyuck Han, Alan Fekete, and Uwe Röhm. Serializable snapshot isolation for replicated databases in high-update scenarios. *Proc. VLDB Endow.*, 4(11):783–794, 2011. URL http://www.vldb.org/pvldb/vol4/p783-jung.pdf.

[117] Tetsuo Kamina and Tomoyuki Aotani. Harmonizing signals and events with a lightweight extension to java. *The Art, Science, and Engineering of Programming*, 2(3):5, 2018. doi: 10.22152/programming-journal.org/2018/2/5. URL https://doi.org/10.22152/programming-journal.org/2018/2/5.

[118] Rajesh K. Karmani and Gul Agha. Actors. In David A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1–11. Springer, 2011. doi: 10.1007/978-0-387-09766-4_125. URL https://doi.org/10.1007/978-0-387-09766-4_125.

[119] Jun Kato and Masataka Goto. Live tuning: Expanding live programming benefits to non-programmers. In *Workshop on Live Programming Systems*, LIVE, 2016.

[120] Martin Kleppmann and Contributors. Automerge. https://github.com/automerge/automerge, 2020. Accessed 2021-01-01.

[121] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. Local-first software: You own your data, in spite of the cloud. In Hidehiko Masuhara and Tomas Petricek, editors, *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2019, Athens, Greece, October 23-24, 2019*, pages 154–178. ACM, 2019. doi: 10.1145/3359591.3359737. URL https://doi.org/10.1145/3359591.3359737.

[122] Andrew Koenig and Bjarne Stroustrup. Exception handling for C++. In *Proceedings of the C++ Conference. San Francisco, CA, USA, April 1990*, pages 149–176. USENIX Association, 1990.

[123] Mirko Köhler, Nafise Eskandani, Pascal Weisenburger, Alessandro Margara, and Guido Salvaneschi. Rethinking safe consistency in distributed object-oriented programming. *Proc. ACM Program. Lang.*, 4(OOPSLA):188:1–188:30, 2020. doi: 10.1145/3428256. URL https://doi.org/10.1145/3428256.

[124] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: A distributed messaging system for log processing. In *Proceedings of 6th International Workshop on Networking Meets Databases (NetDB), Athens, Greece*, 2011.

[125] Neelakantan R. Krishnaswami. Higher-order functional reactive programming without spacetime leaks. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 221–232. ACM, 2013. doi: 10.1145/2500365.2500588. URL https://doi.org/10.1145/2500365.2500588.

[126] Neelakantan R. Krishnaswami, Nick Benton, and Jan Hoffmann. Higher-order functional reactive programming in bounded space. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 45–58. ACM, 2012. doi: 10.1145/2103656.2103665. URL https://doi.org/10.1145/2103656.2103665.

[127] Dominik Kundel. Building a TodoMVC with Twilio sync and JavaScript. https://www.twilio.com/blog/2017/09/building-a-todomvc-with-twilio-sync.html, 2017. Accessed 2021-01-01.

[128] Lindsey Kuper and Ryan R. Newton. Lvars: Lattice-based data structures for deterministic parallelism. In Clemens Grelck, Fritz Henglein, Umut A. Acar, and Jost Berthold, editors, *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing, Boston, MA, USA, FHPC@ICFP 2013, September 25-27, 2013*, pages 71–84. ACM, 2013. doi: 10.1145/2502323.2502326. URL https://doi.org/10.1145/2502323.2502326.

[129] Lindsey Kuper and Ryan R. Newton. Joining forces: Toward a unified account of LVars and convergent replicated data types. *WoDet*, 2014.

[130] Ugo Dal Lago, Marc de Visme, Damiano Mazza, and Akira Yoshimizu. Intersection types and runtime errors in the pi-calculus. *Proc. ACM Program. Lang.*, 3(POPL):7:1–7:29, 2019. doi: 10.1145/3290320. URL https://doi.org/10.1145/3290320.

[131] Ralf Lämmel. Google's mapreduce programming model - revisited. *Sci. Comput. Program.*, 70(1):1–30, 2008. doi: 10.1016/j.scico.2007.07.001. URL https://doi.org/10.1016/j.scico.2007.07.001.

[132] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, December 2001.

[133] Peter J. Landin. The next 700 programming languages. *Commun. ACM*, 9(3): 157–166, 1966. doi: 10.1145/365230.365257. URL https://doi.org/10.1145/365230.365257.

[134] Jean-Claude Laprie. Dependable computing: Concepts, challenges, directions. In *28th International Computer Software and Applications Conference (COMPSAC 2004), Design and Assessment of Trustworthy Software-Based Systems, 27-30 September 2004, Hong Kong, China, Proceedings*, page 242. IEEE Computer Society, 2004. doi: 10.1109/CMPSAC.2004.1342838. URL https://doi.org/10.1109/CMPSAC.2004.1342838.

[135] Haoyi Li. ScalaTags. http://www.lihaoyi.com/scalatags/, 2012. Accessed 2016-09-10.

[136] Jesse Liberty and Paul Betts. *Programming Reactive Extensions and LINQ*. Apress, Berkeley, CA, 2011. ISBN 9781430237471 9781430237488. doi: 10.1007/978-1-4302-3748-8. URL http://link.springer.com/10.1007/978-1-4302-3748-8.

[137] Ingo Maier and Martin Odersky. Deprecating the observer pattern with Scala.React. Technical report, 2012.

[138] Alessandro Margara and Guido Salvaneschi. We have a DREAM: distributed reactive programming with consistency guarantees. In Umesh Bellur and Ravi Kothari, editors, *The 8th ACM International Conference on Distributed Event-Based Systems, DEBS '14, Mumbai, India, May 26-29, 2014*, pages 142–153. ACM, 2014. doi: 10.1145/2611286.2611290. URL https://doi.org/10.1145/2611286.2611290.

[139] Alessandro Margara and Guido Salvaneschi. On the semantics of distributed reactive programming: The cost of consistency. *IEEE Trans. Software Eng.*, 44 (7):689–711, 2018. doi: 10.1109/TSE.2018.2833109. URL https://doi.org/10.1109/TSE.2018.2833109.

[140] Håkan Mattsson, Hans Nilsson, and Claes Wikström. Mnesia - A distributed robust DBMS for telecommunications applications. In Gopal Gupta, editor, *Practical Aspects of Declarative Languages, First International Workshop, PADL '99, San Antonio, Texas, USA, January 18-19, 1999, Proceedings*, volume 1551 of *Lecture Notes in Computer Science*, pages 152–163. Springer, 1999. doi: 10.1007/3-540-49201-1_11. URL https://doi.org/10.1007/3-540-49201-1_11.

[141] René Meier and Vinny Cahill. Taxonomy of distributed event-based programming systems. In *22nd International Conference on Distributed Computing Systems, Workshops (ICDCSW '02) July 2-5, 2002, Vienna, Austria, Proceedings*, pages 585–588. IEEE Computer Society, 2002. doi: 10.1109/ICDCSW.2002.1030833. URL https://doi.org/10.1109/ICDCSW.2002.1030833.

[142] Christopher Meiklejohn and Peter Van Roy. The implementation and use of a generic dataflow behaviour in erlang. In Hans Svensson and Melinda Tóth, editors, *Proceedings of the 14th ACM SIGPLAN Workshop on Erlang 2015, Vancouver, BC, Canada, September 4, 2015*, pages 39–45. ACM, 2015. doi: 10.1145/2804295.2804300. URL https://doi.org/10.1145/2804295.2804300.

[143] Christopher Meiklejohn and Peter Van Roy. Lasp: A language for distributed, coordination-free programming. In Moreno Falaschi and Elvira Albert, editors, *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming, Siena, Italy, July 14-16, 2015*, pages 184–195. ACM, 2015. doi: 10.1145/2790449.2790525. URL https://doi.org/10.1145/2790449.2790525.

[144] Christopher S. Meiklejohn. On the design of distributed programming models. *CoRR*, abs/1701.07615, 2017. URL http://arxiv.org/abs/1701.07615.

[145] Christian Meurisch, The An Binh Nguyen, Martin Kromm, Andrea Ortiz, Ragnar Mogk, and Max Mühlhäuser. Disvis 2.0: Decision support for rescue missions using predictive disaster simulations with human-centric models. In *26th International Conference on Computer Communication and Networks, ICCCN 2017, Vancouver, BC, Canada, July 31 - Aug. 3, 2017*, pages 1–2. IEEE, 2017. doi: 10.1109/ICCCN.2017.8038474. URL https://doi.org/10.1109/ICCCN.2017.8038474.

[146] Leo A. Meyerovich, Arjun Guha, Jacob P. Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: A programming language for ajax applications. In Shail Arora and

Gary T. Leavens, editors, *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, pages 1–20. ACM, 2009. doi: 10.1145/1640089.1640091. URL https://doi.org/10.1145/1640089.1640091.

[147] Heather Miller, Philipp Haller, Normen Müller, and Jocelyn Boullier. Function passing: A model for typed, distributed functional programming. In Eelco Visser, Emerson R. Murphy-Hill, and Crista Lopes, editors, *2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2016, Amsterdam, The Netherlands, November 2-4, 2016*, pages 82–97. ACM, 2016. doi: 10.1145/2986012.2986014. URL https://doi.org/10.1145/2986012.2986014.

[148] Mark S. Miller, Eric Dean Tribble, and Jonathan S. Shapiro. Concurrency among strangers. In Rocco De Nicola and Davide Sangiorgi, editors, *Trustworthy Global Computing, International Symposium, TGC 2005, Edinburgh, UK, April 7-9, 2005, Revised Selected Papers*, volume 3705 of *Lecture Notes in Computer Science*, pages 195–229. Springer, 2005. doi: 10.1007/11580850_12. URL https://doi.org/10.1007/11580850_12.

[149] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980. ISBN 3-540-10235-3. doi: 10.1007/3-540-10235-3. URL https://doi.org/10.1007/3-540-10235-3.

[150] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, 1992. doi: 10.1016/0890-5401(92)90008-4. URL https://doi.org/10.1016/0890-5401(92)90008-4.

[151] Ragnar Mogk. Concurrency control for multithreaded reactive programming. In Jonathan Aldrich and Patrick Eugster, editors, *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity, SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 77–78. ACM, 2015. doi: 10.1145/2814189.2815374. URL https://doi.org/10.1145/2814189.2815374.

[152] Ragnar Mogk. REScala in concurrent applications: Correctness and performance. Master's thesis, Technische Universität Darmsatdt, 2015.

[153] Ragnar Mogk. Reactive interfaces: Combining events and expressing signals. In *Workshop on Reactive and Event-based Languages & Systems (REBLS)*, 2015. URL https://2015.splashcon.org/track/rebls2015#event-overview.

[154] Ragnar Mogk and Joscha Drechsler. REScala – principled distributed reactive programming. In *Poster <Programming>*, 2017. URL http://tubiblio.ulb.tu-darmstadt.de/101440/.

[155] Ragnar Mogk, Lars Baumgärtner, Guido Salvaneschi, Bernd Freisleben, and Mira Mezini. Fault-tolerant distributed reactive programming. In Todd D. Millstein, editor, *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*, volume 109 of *LIPIcs*, pages 1:1–1:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi: 10.4230/LIPIcs.ECOOP.2018.1. URL https://doi.org/10.4230/LIPIcs.ECOOP.2018.1.

[156] Ragnar Mogk, Guido Salvaneschi, and Mira Mezini. Reactive programming experience with REScala. In Stefan Marr and Jennifer B. Sartor, editors, *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming, Nice, France, April 09-12, 2018*, pages 105–112. ACM, 2018. doi: 10.1145/3191697.3214337. URL https://doi.org/10.1145/3191697.3214337.

[157] Ragnar Mogk, Joscha Drechsler, Guido Salvaneschi, and Mira Mezini. A fault-tolerant programming model for distributed interactive applications. *Proc. ACM Program. Lang.*, 3(OOPSLA):144:1–144:29, 2019. doi: 10.1145/3360570. URL https://doi.org/10.1145/3360570.

[158] Ragnar Mogk, Pascal Weisenburger, Julian Haas, David Richter, Guido Salvaneschi, and Mira Mezini. From debugging towards live tuning of reactive applications. In *Workshop on Live Programming Systems*, LIVE, 2019.

[159] Florian Myter, Tim Coppieters, Christophe Scholliers, and Wolfgang De Meuter. I now pronounce you reactive and consistent: Handling distributed and replicated state in reactive programming. In Guido Salvaneschi, Wolfgang De Meuter, Patrick Eugster, and Lukasz Ziarek, editors, *Proceedings of the 3rd International Workshop on Reactive and Event-Based Languages and Systems, Amsterdam, Netherlands, November 1, 2016*, pages 1–8. ACM, 2016. doi: 10.1145/3001929.3001930. URL https://doi.org/10.1145/3001929.3001930.

[160] Florian Myter, Christophe Scholliers, and Wolfgang De Meuter. Handling partial failures in distributed reactive programming. In Guido Salvaneschi, Wolfgang De Meuter, Patrick Eugster, and Lukasz Ziarek, editors, *Proceedings of the 4th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems, Vancouver, BC, Canada, October 23, 2017*, pages 1–7. ACM, 2017. doi: 10.1145/3141858.3141859. URL https://doi.org/10.1145/3141858.3141859.

[161] Florian Myter, Christophe Scholliers, and Wolfgang De Meuter. A capable distributed programming model. In Elisa Gonzalez Boix and Richard P. Gabriel, editors, *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2018, Boston, MA, USA, November 7-8, 2018*, pages 88–98. ACM, 2018. doi: 10.1145/3276954.3276957. URL https://doi.org/10.1145/3276954.3276957.

[162] Ryan Newton, Greg Morrisett, and Matt Welsh. The regiment macroprogramming system. In Tarek F. Abdelzaher, Leonidas J. Guibas, and Matt Welsh, editors, *Proceedings of the 6th International Conference on Information Processing in Sensor Networks, IPSN 2007, Cambridge, Massachusetts, USA, April 25-27, 2007*, pages 489–498. ACM, 2007. doi: 10.1145/1236360.1236422. URL https://doi.org/10.1145/1236360.1236422.

[163] Yang Ni, Vijay Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In Katherine A. Yelick and John M.

Mellor-Crummey, editors, *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2007, San Jose, California, USA, March 14-17, 2007*, pages 68–78. ACM, 2007. doi: 10.1145/1229428.1229442. URL https://doi.org/10.1145/1229428.1229442.

[164] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, pages 51–64, New York, NY, USA, 2002. ACM. ISBN 1-58113-605-6. doi: 10.1145/581690.581695. URL http://doi.acm.org/10.1145/581690.581695.

[165] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: A not-so-foreign language for data processing. In Jason Tsong-Li Wang, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 1099–1110. ACM, 2008. doi: 10.1145/1376616.1376726. URL https://doi.org/10.1145/1376616.1376726.

[166] Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In Garth Gibson and Nickolai Zeldovich, editors, *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, pages 305–319. USENIX Association, 2014. URL https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

[167] Ivan Perez. Fault tolerant functional reactive programming (functional pearl). *Proc. ACM Program. Lang.*, 2(ICFP):96:1–96:30, 2018. doi: 10.1145/3236791. URL https://doi.org/10.1145/3236791.

[168] John Peterson, Valery Trifonov, and Andrei Serjantov. Parallel functional reactive programming. In Enrico Pontelli and Vítor Santos Costa, editors, *Practical Aspects of Declarative Languages, Second International Workshop, PADL 2000, Boston, MA, USA, January 2000, Proceedings*, volume 1753 of *Lecture Notes in Computer Science*, pages 16–31. Springer, 2000. doi: 10.1007/3-540-46584-7_2. URL https://doi.org/10.1007/3-540-46584-7_2.

[169] Peter R. Pietzuch and Jean Bacon. Hermes: A distributed event-based middleware architecture. In *22nd International Conference on Distributed Computing Systems, Workshops (ICDCSW '02) July 2-5, 2002, Vienna, Austria, Proceedings*, pages 611–618. IEEE Computer Society, 2002. doi: 10.1109/ICDCSW.2002.1030837. URL https://doi.org/10.1109/ICDCSW.2002.1030837.

[170] Jan Ploski and Wilhelm Hasselbring. Exception handling in an event-driven system. In *Proceedings of The Second International Conference on Availability, Reliability and Security, ARES 2007, The International Dependability Conference - Bridging Theory and Practice, April 10-13 2007, Vienna, Austria*, pages 1085–1092. IEEE Computer Society, 2007. doi: 10.1109/ARES.2007.85. URL https://doi.org/10.1109/ARES.2007.85.

[171] José Proença and Carlos Baquero. Quality-aware reactive programming for the internet of things. In Mehdi Dastani and Marjan Sirjani, editors, *Fundamentals of Software Engineering - 7th International Conference, FSEN*

2017, Tehran, Iran, April 26-28, 2017, Revised Selected Papers, volume 10522 of *Lecture Notes in Computer Science*, pages 180–195. Springer, 2017. doi: 10.1007/978-3-319-68972-2_12. URL https://doi.org/10.1007/978-3-319-68972-2_12.

[172] Stefan Ramson and Robert Hirschfeld. Active expressions: Basic building blocks for reactive programming. *The Art, Science, and Engineering of Programming*, 1(2):12, 2017. doi: 10.22152/programming-journal.org/2017/1/12. URL https://doi.org/10.22152/programming-journal.org/2017/1/12.

[173] Jorge Ressia, Alexandre Bergel, and Oscar Nierstrasz. Object-centric debugging. In Martin Glinz, Gail C. Murphy, and Mauro Pezzè, editors, *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 485–495. IEEE Computer Society, 2012. doi: 10.1109/ICSE.2012.6227167. URL https://doi.org/10.1109/ICSE.2012.6227167.

[174] Bob Reynders, Dominique Devriese, and Frank Piessens. Multi-tier functional reactive programming for the web. In Andrew P. Black, Shriram Krishnamurthi, Bernd Bruegge, and Joseph N. Ruskiewicz, editors, *Onward! 2014, Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, part of SPLASH '14, Portland, OR, USA, October 20-24, 2014*, pages 55–68. ACM, 2014. doi: 10.1145/2661136.2661140. URL https://doi.org/10.1145/2661136.2661140.

[175] David Richter and Ragnar Mogk. Turning unobservable into unreachable: Dynamic reactive programming without leaks. In *Workshop on Reactive and Event-based Languages & Systems (REBLS)*, 2019. URL https://2019.splashcon.org/details/rebls-2019-papers/3/Turning-Unobservable-into-Unreachable-Dynamic-Reactive-Programming-without-Leaks.

[176] Mohamed M. Saad and Binoy Ravindran. Hyflow: A high performance distributed software transactional memory framework. In Arthur B. Maccabe and Douglas Thain, editors, *Proceedings of the 20th ACM International Symposium on High Performance Distributed Computing, HPDC 2011, San Jose, CA, USA, June 8-11, 2011*, pages 265–266. ACM, 2011. doi: 10.1145/1996130.1996167. URL https://doi.org/10.1145/1996130.1996167.

[177] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Ben Hertzberg. Mcrt-stm: A high performance software transactional memory system for a multi-core runtime. In Josep Torrellas and Siddhartha Chatterjee, editors, *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2006, New York, New York, USA, March 29-31, 2006*, pages 187–197. ACM, 2006. doi: 10.1145/1122971.1123001. URL https://doi.org/10.1145/1122971.1123001.

[178] Guido Salvaneschi and Mira Mezini. Debugging for reactive programming. In Laura K. Dillon, Willem Visser, and Laurie Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 796–807. ACM, 2016. doi: 10.1145/2884781.2884815. URL https://doi.org/10.1145/2884781.2884815.

[179] Guido Salvaneschi, Sven Amann, Sebastian Proksch, and Mira Mezini. An empirical study on program comprehension with reactive programming. In Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey, editors, *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 564–575. ACM, 2014. doi: 10.1145/2635868.2635895. URL https://doi.org/10.1145/2635868.2635895.

[180] Guido Salvaneschi, Patrick Eugster, and Mira Mezini. Programming with implicit flows. *IEEE Softw.*, 31(5):52–59, 2014. doi: 10.1109/MS.2014.101. URL https://doi.org/10.1109/MS.2014.101.

[181] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. REScala: Bridging between object-oriented and functional style in reactive applications. In Walter Binder, Erik Ernst, Achille Peternier, and Robert Hirschfeld, editors, *13th International Conference on Modularity, MODULARITY '14, Lugano, Switzerland, April 22-26, 2014*, pages 25–36. ACM, 2014. doi: 10.1145/2577080.2577083. URL https://doi.org/10.1145/2577080.2577083.

[182] Guido Salvaneschi, Sebastian Proksch, Sven Amann, Sarah Nadi, and Mira Mezini. On the positive effect of reactive programming on software comprehension: An empirical study. *IEEE Trans. Software Eng.*, 43(12):1125–1143, 2017. doi: 10.1109/TSE.2017.2655524. URL https://doi.org/10.1109/TSE.2017.2655524.

[183] Francisco Sant'Anna, Noemi de La Rocque Rodriguez, Roberto Ierusalimschy, Olaf Landsiedel, and Philippas Tsigas. Safe system-level concurrency on resource-constrained nodes. In Chiara Petrioli, Landon P. Cox, and Kamin Whitehouse, editors, *The 11th ACM Conference on Embedded Network Sensor Systems, SenSys '13, Roma, Italy, November 11-15, 2013*, pages 11:1–11:14. ACM, 2013. doi: 10.1145/2517351.2517360. URL https://doi.org/10.1145/2517351.2517360.

[184] Kensuke Sawada and Takuo Watanabe. Emfrp: A functional reactive programming language for small-scale embedded systems. In Lidia Fuentes, Don S. Batory, and Krzysztof Czarnecki, editors, *Companion Proceedings of the 15th International Conference on Modularity, Málaga, Spain, March 14 - 18, 2016*, pages 36–44. ACM, 2016. doi: 10.1145/2892664.2892670. URL https://doi.org/10.1145/2892664.2892670.

[185] Neil Sculthorpe and Henrik Nilsson. Safe functional reactive programming through dependent types. In Graham Hutton and Andrew P. Tolmach, editors, *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*, pages 23–34. ACM, 2009. doi: 10.1145/1596550.1596558. URL https://doi.org/10.1145/1596550.1596558.

[186] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of convergent and commutative replicated data types. Research Report RR-7506, Inria – Centre Paris-Rocquencourt ; INRIA, January 2011. URL https://hal.inria.fr/inria-00555588.

[187] Marc Shapiro, Annette Bieniusa, Nuno M. Preguiça, Valter Balegas, and Christopher Meiklejohn. Just-right consistency: Reconciling availability and safety. *CoRR*, abs/1801.06340, 2018. URL http://arxiv.org/abs/1801.06340.

[188] Nir Shavit and Dan Touitou. Software transactional memory. In James H. Anderson, editor, *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, Ottawa, Ontario, Canada, August 20-23, 1995*, pages 204–213. ACM, 1995. doi: 10.1145/224964.224987. URL https://doi.org/10.1145/224964.224987.

[189] Simplectic. Flask TodoMVC 2: Backbone sync. https://simplectic.com/blog/2014/flask-todomvc-backbone-sync/, 2014. Accessed 2021-01-01.

[190] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. *Proc. VLDB Endow.*, 13(11):2438–2452, 2020. URL http://www.vldb.org/pvldb/vol13/p2438-sreekanti.pdf.

[191] Artur Sterz, Lars Baumgärtner, Ragnar Mogk, Mira Mezini, and Bernd Freisleben. DTN-RPC: remote procedure calls for disruption-tolerant networking. In *2017 IFIP Networking Conference, IFIP Networking 2017 and Workshops, Stockholm, Sweden, June 12-16, 2017*, pages 1–9. IEEE Computer Society, 2017. doi: 10.23919/IFIPNetworking.2017.8264848. URL https://doi.org/10.23919/IFIPNetworking.2017.8264848.

[192] Artur Sterz, Matthias Eichholz, Ragnar Mogk, Lars Baumgärtner, Pablo Graubner, Matthias Hollick, Mira Mezini, and Bernd Freisleben. ReactiFi: Reactive programming of Wi-Fi firmware on mobile devices. *The Art, Science, and Engineering of Programming*, 5(2):4, 2020. doi: 10.22152/programming-journal.org/2021/5/4. URL https://doi.org/10.22152/programming-journal.org/2021/5/4.

[193] Sagnak Tasirlar and Vivek Sarkar. Data-driven tasks and their implementation. In Guang R. Gao and Yu-Chee Tseng, editors, *International Conference on Parallel Processing, ICPP 2011, Taipei, Taiwan, September 13-16, 2011*, pages 652–661. IEEE Computer Society, 2011. doi: 10.1109/ICPP.2011.87. URL https://doi.org/10.1109/ICPP.2011.87.

[194] Jonas Traub, Jorge-Arnulfo Quiané-Ruiz, Zoi Kaoudi, and Volker Markl. Agora: Towards an open ecosystem for democratizing data science & artificial intelligence. *CoRR*, abs/1909.03026, 2019. URL http://arxiv.org/abs/1909.03026.

[195] Aaron Turon. Reagents: Expressing and composing fine-grained concurrency. In Jan Vitek, Haibo Lin, and Frank Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 157–168. ACM, 2012. doi: 10.1145/2254064.2254084. URL https://doi.org/10.1145/2254064.2254084.

[196] Atze van der Ploeg and Koen Claessen. Practical principled FRP: forget the past, change the future, frpnow! In Kathleen Fisher and John H. Reppy, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on*

Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015, pages 302–314. ACM, 2015. doi: 10.1145/2784731.2784752. URL https://doi.org/10.1145/2784731.2784752.

[197] Malte Viering, Tzu-Chun Chen, Patrick Eugster, Raymond Hu, and Lukasz Ziarek. A typing discipline for statically verified crash failure handling in distributed systems. In Amal Ahmed, editor, *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10801 of *Lecture Notes in Computer Science*, pages 799–826. Springer, 2018. doi: 10.1007/978-3-319-89884-1_28. URL https://doi.org/10.1007/978-3-319-89884-1_28.

[198] Andreas Voellmy, Hyojoon Kim, and Nick Feamster. Procera: A language for high-level reactive network control. In Nick Feamster and Jennifer Rexford, editors, *Proceedings of the first workshop on Hot topics in software defined networks, HotSDN@SIGCOMM 2012, Helsinki, Finland, August 13, 2012*, pages 43–48. ACM, 2012. doi: 10.1145/2342441.2342451. URL https://doi.org/10.1145/2342441.2342451.

[199] Klaus von Gleissenthall, Rami Gökhan Kici, Alexander Bakst, Deian Stefan, and Ranjit Jhala. Pretend synchrony: synchronous verification of asynchronous distributed programs. *Proc. ACM Program. Lang.*, 3(POPL):59:1–59:30, 2019. doi: 10.1145/3290372. URL https://doi.org/10.1145/3290372.

[200] Philip Wadler. Monads for functional programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer, 1995. doi: 10.1007/3-540-59451-5_2. URL https://doi.org/10.1007/3-540-59451-5_2.

[201] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In Monica S. Lam, editor, *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*, pages 242–252. ACM, 2000. doi: 10.1145/349299.349331. URL https://doi.org/10.1145/349299.349331.

[202] Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. Distributed system development with scalaloci. *Proc. ACM Program. Lang.*, 2(OOPSLA):129:1–129:30, 2018. doi: 10.1145/3276499. URL https://doi.org/10.1145/3276499.

[203] Michael Whittaker and Joseph M. Hellerstein. Interactive checks for coordination avoidance. *Proc. VLDB Endow.*, 12(1):14–27, 2018. doi: 10.14778/3275536.3275538. URL http://www.vldb.org/pvldb/vol12/p14-whittaker.pdf.

[204] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In David Grove and

Steve Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 357–368. ACM, 2015. doi: 10.1145/2737924.2737958. URL https://doi.org/10.1145/2737924.2737958.

[205] Wei Yu, Fan Liang, Xiaofei He, William G. Hatcher, Chao Lu, Jie Lin, and Xinyu Yang. A survey on the edge computing for the internet of things. *IEEE Access*, 6:6900–6919, 2018. doi: 10.1109/ACCESS.2017.2778504. URL https://doi.org/10.1109/ACCESS.2017.2778504.

[206] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In Steven D. Gribble and Dina Katabi, editors, *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 15–28. USENIX Association, 2012. URL https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia.

[207] Irene Zhang, Adriana Szekeres, Dana Van Aken, Isaac Ackerman, Steven D. Gribble, Arvind Krishnamurthy, and Henry M. Levy. Customizable and extensible deployment for mobile/cloud applications. In Jason Flinn and Hank Levy, editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 97–112. USENIX Association, 2014. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/zhang.