In the 90's and early 2000's users interacted with computers synchronously: the user clicks a button, the computer takes a while to calculate the result, and the user patiently waits until this result is presented to them by the computer. Modern software is different. Users expect the software that they interact with to respond in real-time to their actions, and meanwhile the software also interacts with other devices which are connected via a network, e.g., smartphones, smart watches, smart home devices, etc. The software has become "reactive". To build reactive software, academia and industry have come up with techniques that are collectively referred to as "reactive programming", which make reactive systems faster and easier to develop without bugs.

This thesis observes that existing literature on reactive programming is only focussed on the *internal part* of a reactive system, which denotes the programming abstractions and techniques that were invented to build reactive software. However, the internal part is always coupled to an *external part* which is still programmed using ordinary imperative (i.e., non-reactive) programming techniques. Up until now this coupling between reactive and imperative is left as an afterthought when designing a reactive programming language or framework. Consequently, this thesis identifies a number of issues that occur when the coupling of reactive code with imperative code is not carefully designed. These problems are present to various degrees in all reactive languages and frameworks considered by the thesis.

A solution to the identified problems is proposed by this thesis, namely a new programming model called "The Actor-Reactor Model" which is demonstrated via a new prototype programming language called Stella. Stella is designed to develop distributed reactive programs for *open networks,* which is a type of network where devices can spontaneously come and go (e.g., wireless networks). This type of application is inherently full of code that is both imperative and reactive.

Stella runs on all platforms that support JavaScript (i.e., web browsers and Node.js). It was used to simulate an open network application using the "Villo!" (Brussels' bike sharing programme) open data on a cluster computer that consists of 160 Raspberry Pi single-board computers.

The Actor-Reactor Model and Stella lead to a better understanding of how reactive code and imperative code can coexist within a reactive program, and they demonstrate a novel approach to program reactive software for open networks.

ISBN 978-94-6444-328-8

On the Coexistence of Reactive Code and Imperative Code in Distributed Applications

Sam Van den Vonder

On the Coexistence of Reactive Code and Imperative Code in Distributed Applications

A Language Design Approach

Sam Van den Vonder

Faculty of Sciences and Bioengineering Sciences, Vrije Universiteit Brussel

On the Coexistence of Reactive Code and Imperative Code in Distributed Applications

A Language Design Approach

Sam Van den Vonder

Dissertation submitted in fulfilment of the
requirement for the degree of Doctor of Sciences

June 21st, 2022

Promotor:
Prof. Dr. Wolfgang De Meuter, Vrije Universiteit Brussel, Belgium

Jury:
Prof. Dr. Viviane Jonckers, Vrije Universiteit Brussel, Belgium (chair)
Dr. Ir. Roxana Rădulescu, Vrije Universiteit Brussel, Belgium (secretary)
Prof. Dr. Ir. Kris Steenhaut, Vrije Universiteit Brussel, Belgium
Prof. Dr. Coen De Roover, Vrije Universiteit Brussel, Belgium
Prof. Dr.-Ing. Mira Mezini, Technical University of Darmstadt, Germany
Prof. Dr. Tijs van der Storm, Centrum Wiskunde & Informatica and the University of Groningen, The Netherlands

Software Languages Lab
Department of Computer Science
Faculty of Sciences and Bioengineering Sciences
Vrije Universiteit Brussel

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

ISBN 978-94-6444-328-8
NUR 989 THEMA: UMA

Abstract

The continued increase in computing power and network resources has changed the landscape of software development. Nowadays, software is not confined to a single computer, and instead runs on many types of devices that interact on an *open network*, which is a network that various types of devices such as smartphones, smartwatches, and Internet of Things devices can spontaneously join and leave. Moreover, their software is continuously interacting with each other, e.g., a smartphone smart home application is updated in real-time based on the connected smart home sensors. In other words, the software that powers those devices has become *reactive*.

Traditionally reactive programs are written using callbacks and the observer pattern. Nowadays their drawbacks are well known, as they make code difficult to understand, debug, and maintain. About a decade ago, *reactive programming* has been proposed as a solution to these problems and has gained widespread adoption.

Existing reactive programming techniques exhibit 3 issues that make them difficult to apply to software for open networks. The first issue is related to *responsiveness*, where data received via a network can (accidentally) make the program *no longer reactive*. Second, we describe a *paradigmatic mismatch* between reactive code and imperative code, both of which are present in every reactive application. The final issue relates to the *open network*, where existing reactive programming techniques have little or no support to *discover devices* on the open network, and to manage their presence in the reactive program *correctly and efficiently*.

We propose a novel computational model – called the "Actor-Reactor Model" – that avoids the issues by using so-called "actors" and "reactors" to build distributed applications. First, reactive code is encapsulated by reactors which guarantee responsiveness. Second, imperative code and reactive code is strictly separated in actors and reactors respectively, thereby avoiding a paradigmatic mismatch. Finally, we design a mechanism to discover actors and reactors on an open network, and to maintain their presence correctly and efficiently throughout a distributed reactive program.

The Actor-Reactor Model was implemented in a new programming language called Stella, which serves as a linguistic vehicle to demonstrate the ideas of our approach. Compared to other reactive programming languages and frameworks,

we found that actors and reactors are highly suitable to develop distributed applications. Furthermore, they demonstrate a correct and efficient approach to define distributed computations that involve multiple discovered devices (e.g., sensors). All Stella code presented in this dissertation is executable and included in an artefact.

The Actor-Reactor Model leads to a better understanding of the coexistence of reactive code and imperative code in Stella and existing reactive programming languages and frameworks.

Samenvatting

De voortdurende toename in rekenkracht en netwerkmiddelen heeft het landschap van de softwareontwikkeling veranderd. Software is tegenwoordig niet meer beperkt tot één computer, maar draait op vele soorten apparaten die met elkaar communiceren op een *open netwerk*. Dit is een netwerk waar verschillende soorten apparaten zoals smartphones, smartwatches en Internet of Things apparaten spontaan mee kunnen verbinden en weer verdwijnen. Bovendien is er een voortdurende wisselwerking tussen hun software, bv. een smartphone-smarthometoepassing wordt in real-time bijgewerkt op basis van de aangesloten smart-homesensoren. Met andere woorden, de software die deze apparaten aanstuurt is *reactief* geworden.

Traditioneel worden reactieve programma's geschreven met behulp van callbacks en het observer patroon. Hun nadelen zijn tegenwoordig gekend omdat ze ervoor zorgen dat code moeilijk te begrijpen, debuggen, en onderhouden is. Reactief programmeren werd ongeveer een decennium geleden voorgesteld als een oplossing voor deze problemen, en deze techniek wordt vandaag massaal gebruikt.

Bestaande reactieve programmeertechnieken vertonen 3 problemen die het moeilijk maken om ze toe te passen op software voor open netwerken. Het eerste probleem heeft te maken met *responsiviteit*, waarbij gegevens die via een netwerk worden ontvangen er (per ongeluk) voor zorgen dat het programma *niet meer reactief* is. Ten tweede beschrijven we een paradigmatische mismatch tussen reactieve code en imperatieve code, dewelke beide aanwezig zijn in elk reactief programma. Het laatste probleem heeft te maken met het *open netwerk*, waar bestaande reactieve programmeertechnieken weinig of geen ondersteuning bieden om *apparaten te ontdekken* op het open netwerk, en om hun aanwezigheid in het reactieve programma *correct en efficiënt* te beheren.

We stellen een nieuw computationeel model voor – dat we het "Actor-Reactor Model" noemen – dat de problemen omzeilt door zogenaamde "actors" en "reactors" te gebruiken om gedistribueerde applicaties te bouwen. Ten eerste, reactors encapsuleren reactieve code en garanderen responsiviteit. Ten tweede, imperatieve code en reactieve code wordt strikt gescheiden door actors en reactors respectievelijk, waardoor ze de paradigmatische mismatch omzeilen. Ten laatste ontwerpen we een mechanisme voor actors en reactors om elkaar te ontdekken

op een open netwerk, en om hun aanwezigheid correct en efficient te beheren in een gedistribueerd reactief programma.

Het Actor-Reactor Model werd geïmplementeerd in een nieuwe programmeertaal genaamd Stella, welke dient als een linguïstisch vehikel om de ideeën van onze aanpak te demonstreren. Vergeleken met andere reactieve programmeertalen en frameworks, vonden wij dat actors en reactors zeer geschikt zijn om gedistribueerde applicaties te ontwikkelen. Voorts demonstreren zij een correcte en efficiënte aanpak om gedistribueerde programma's te schrijven waarbij meerdere ontdekte apparaten (bv. sensors) betrokken zijn. Alle Stella code die in dit proefschrift wordt gepresenteerd is uitvoerbaar.

Het Actor-Reactor Model leidt tot een beter begrip van de samenhang tussen reactieve code en imperatieve code in Stella en in bestaande reactieve programmeertalen en frameworks.

Acknowledgments

My gratitude goes to everyone who contributed scientifically or otherwise to this dissertation. These acknowledgements will highlight only some them. If you, the reader, do not see yourself included in the list, it may be an error on my part. A blank space will be left at the end to rectify the mistake.

In no particular order, thank you to:

- Wolfgang De Meuter, my promotor, whose role in shaping the content of the thesis was more "hands on" in the early years, but who allowed me to take over this role as I became more confident in my knowledge of the domain.
- The members of the jury, Viviane Jonckers, Roxana Rădulescu, Kris Steenhaut, Coen De Roover, Mira Mezini, and Tijs van der Storm, whose great efforts in deciphering this thesis cannot be overstated.
- Thierry Renaux for the scientific and non-scientific discussions, and for providing intelligent feedback on the first iteration of this thesis, as well as multiple of my publications.
- Fernando Suarez Groen for designing the cover of the book.
- Colleagues, friends and family for intangible support.
- _____, without whom none of this would have been possible.

This work was partially funded by Flanders Innovation & Entrepreneurship (VLAIO) with the FLAMENCO project under grant No. 150044, and the Research Foundation - Flanders (FWO) under grant No. 1S95318N.

Supporting Publications and Technical Contributions

Supporting Publications

Journal publications, international peer-reviewed

- Sam Van den Vonder, Thierry Renaux and Wolfgang De Meuter. 'Topology-level Reactivity in Distributed Reactive Programs: Reactive Acquaintance Management using Flocks'. In: *The Art, Science, and Engineering of Programming* 6.3 (2022), 14:1–14:37. ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2022/6/14
- Sam Van den Vonder, Thierry Renaux, Bjarno Oeyen, Joeri De Koster and Wolfgang De Meuter. 'Tackling the Awkward Squad for Reactive Programming: The Actor-Reactor Model'. In: *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference)*. Ed. by Robert Hirschfeld and Tobias Pape. Vol. 166. LIPIcs Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 19:1–19:29. DOI: 10.4230/LIPIcs.ECOOP.2020.19

Conference publications, international peer-reviewed

- Sam Van den Vonder, Joeri De Koster and Wolfgang De Meuter. 'Composable Actor Behaviour'. In: *Distributed Applications and Interoperable Systems - 19th IFIP WG 6.1 International Conference, DAIS 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17-21, 2019, Proceedings*. Ed. by José Pereira and Laura Ricci. Vol. 11534. Lecture Notes in Computer Science. Cham, Switzerland: Springer Nature Switzerland AG, 2019, pp. 57–73. DOI: 10.1007/978-3-030-22496-7_4

Workshop publications, international peer-reviewed

- Bjarno Oeyen, Sam Van den Vonder and Wolfgang De Meuter. 'Reactive Sorting Networks'. In: *Proceedings of the 7th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*. REBLS 2020. Virtual, USA:

Association for Computing Machinery, 2020, pp. 38–50. ISBN: 978-1-4503-8188-8. DOI: 10.1145/3427763.3428316

- Bjarno Oeyen, Humberto Rodríguez-Avila, Sam Van den Vonder and Wolfgang De Meuter. 'Composable higher-order reactors as the basis for a live reactive programming environment'. In: *Proceedings of the 5th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems, REBLS@SPLASH 2018, Boston, MA, USA, November 4, 2018*. Ed. by Guido Salvaneschi, Wolfgang De Meuter, Patrick Eugster, Lukasz Ziarek and Francisco Sant'Anna. New York, NY, USA: Association for Computing Machinery, 2018, pp. 51–60. DOI: 10.1145/3281278.3281284
- Bjarno Oeyen, Sam Van den Vonder and Wolfgang De Meuter. 'Trampoline Variables: A General Method for State Accumulation in Reactive Programming'. In: *Proceedings of the 8th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*. Ed. by Louis Mandel. REBLS 2021. Chicago, IL, USA: Association for Computing Machinery, 2021, pp. 27–40. ISBN: 978-1-4503-9108-5. DOI: 10.1145/3486605.3486787
- Cloé Descheemaeker, Sam Van den Vonder, Thierry Renaux and Wolfgang De Meuter. 'Poker: Visual Instrumentation of Reactive Programs with Programmable Probes'. In: *REBLS 2021: Proceedings of the 8th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems, Chicago, IL, USA, 18 October 2021*. Ed. by Louis Mandel. New York, NY, USA: Association for Computing Machinery, 2021, pp. 14–26. ISBN: 978-1-4503-9108-5. DOI: 10.1145/3486605.3486785
- Sam Van den Vonder, Joeri De Koster, Florian Myter and Wolfgang De Meuter. 'Tackling the awkward squad for reactive programming: the actor-reactor model'. In: *Proceedings of the 4th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems, Vancouver, BC, Canada, October 23, 2017*. Ed. by Guido Salvaneschi, Wolfgang De Meuter, Patrick Eugster and Lukasz Ziarek. New York, NY, USA: Association for Computing Machinery, 2017, pp. 27–33. DOI: 10.1145/3141858.3141863
- Sam Van den Vonder, Florian Myter, Joeri De Koster and Wolfgang De Meuter. 'Enriching the Internet By Acting and Reacting'. In: *Companion to the first International Conference on the Art, Science and Engineering of Programming, Programming 2017, Brussels, Belgium, April 3-6, 2017*. Ed. by Jennifer B. Sartor, Theo D'Hondt and Wolfgang De Meuter. New York, NY, USA: Association for Computing Machinery, 2017, 24:1–24:6. DOI: 10.1145/3079368.3079407

Technical Contribution

The main purpose of Stella as a language is to serve as a linguistic vehicle to disseminate the ideas presented in this dissertation. We have a full implementation

of Stella, and all Stella code snippets in this dissertation are extracted from real Stella applications. Additionally, a version of Stella presented at the European Conference on Object-Oriented Programming (ECOOP) 2020 was evaluated as part of ECOOP's artefact evaluation on consistency, completeness, reusability, and documentation quality.

- Sam Van den Vonder, Thierry Renaux, Bjarno Oeyen, Joeri De Koster and Wolfgang De Meuter. 'Tackling the Awkward Squad for Reactive Programming: The Actor-Reactor Model (Artifact)'. In: *Dagstuhl Artifacts Ser.* 6.2 (2020), 07:1–07:4. DOI: 10.4230/DARTS.6.2.7

Contents

List of Tables

List of Figures

Listings

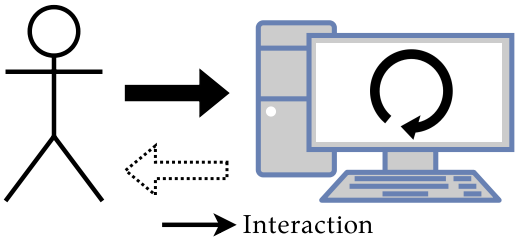
1. Introduction

The continued increase in computing power and network resources has changed the landscape of software development. The traditional (outdated) approach to software development is that an application runs locally on a desktop computer, and that it remains idle until the user triggers an action by clicking on buttons or menu options. This situation is depicted in Figure 1.1a. Whenever the user triggers an action, the software may enter a long calculation until it eventually presents an answer to the user.

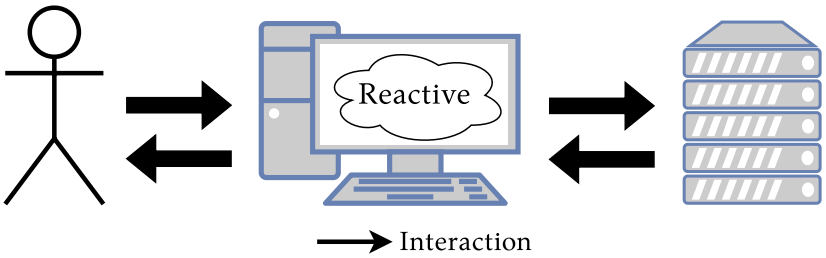
In the past decade the traditional computing model is no longer accepted by end users. Instead, users expect the interaction with their devices such as laptops, smartphones and smartwatches to be seamless, and that the information that is presented to them is updated in real-time [6, 23]. For example, in Figure 1.1b the software continuously interacts with the user and a server-based application, continuously updating as new events occur. The software that powers those devices has become *reactive*. We call this *level 1* reactivity, where the application is able to react to events caused by the user and external devices. Furthermore, the software running on the devices will consist of *prosumers* of data, a general term that we use to denote a software component that both consumes and produces data. Prosumers interact with each other via conceptual streams of incoming and outgoing data such as sensor measurements.

Due to the abundance of high-performance SoCs (system on a chip[1]) there are more devices interacting with each other than ever before. Nowadays they do so on an *open network*. This is a type of (wireless) network where devices can spontaneously appear and disappear based on their location, the quality of the connection, device power management, etc. Typical devices include laptops, smartphones, smartwatches, home automation devices (e.g., sensors), fitness trackers, and vehicle infotainment systems. Users still expect the software that powers these devices to be reactive, i.e., continuously updated based on the latest available info. Additionally, we observe that a new level of reactivity occurs for applications that operate on an open network. Not only does the software react to events such as sensors measurements (level 1 reactivity), but they also need to react to the constellation of collaborating devices that is continuously changing as devices spontaneously join and leave the network. We refer to this

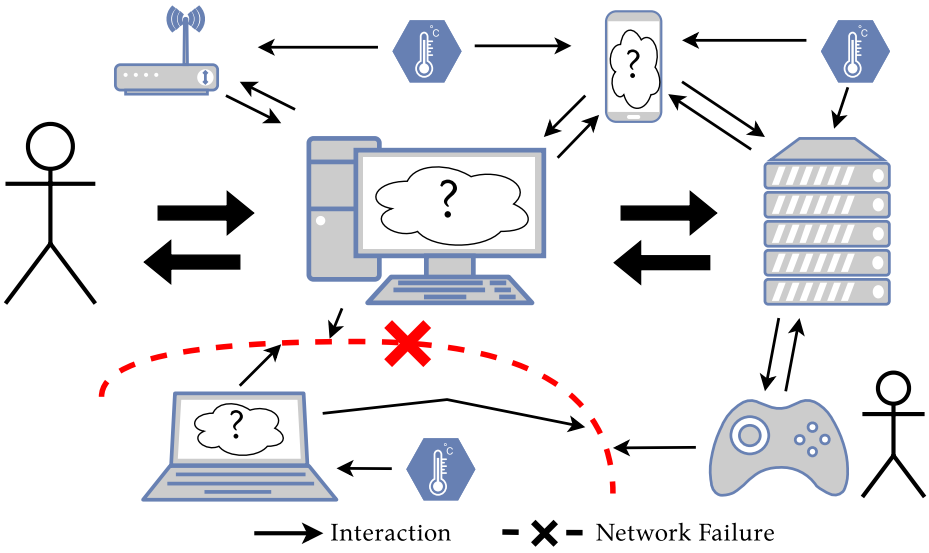
[1] A system on a chip combines most components of a computer, e.g., the CPU, memory, I/O, and various forms of connectivity such as Bluetooth and Wi-Fi.



(a) Traditional software development.



(b) Reactive software development, level 1 reactivity.



(c) Reactive software development for open networks, level 2 reactivity.

Figure 1.1.: Traditional vs. Reactive Software Development.

as *level 2* reactivity. For example, Figure 1.1c depicts the interaction within such a constellation of devices, where the network connection between some of them has (temporarily) dropped.

Reactive applications are difficult to program using conventional programming techniques such as imperative (sequential) code since the arrival of externally triggered events is impossible to predict or control. The traditional approaches to handle events revolve around asynchronous *callbacks* (event handlers) and design patterns such as the Observer pattern [53]. Nowadays their drawbacks are widely documented. Most notably they lead to *inversion of control* where the control flow of the program is dictated by the arrival of outside events rather than the code itself, and *callback hell* where the program logic is spread over many callbacks that coordinate via side-effects. In a nutshell, the traditional approaches have a detrimental effect on software development because the code is difficult to understand, can exhibit tricky and difficult to reproduce bugs, and can become unmaintainable [41, 68, 79, 89]. A presentation in 2007 revealed that a third of the code in Adobe's desktop applications was devoted to event handling, and that this code contained half of all bugs [101].

The *reactive programming paradigm* is an emerging approach to implement level 1 reactive applications without the issues such as callback hell and inversion of control [10]. In this dissertation we will investigate the problems that occur when reactive programming is used to support level 1 reactivity, and we propose new mechanisms to also support level 2 reactivity.

1.1. Research Context: The Reactive Programming Paradigm

Reactive programming requires a different code style and programmer mindset compared to other programming paradigms. For example, in a traditional imperative program a sequence of program statements is executed from top to bottom. The imperative code specifies *what* should happen (execute the program statements) and also *when* it should happen (in the order specified by the program text). Reactive code is different, since the code abstracts over the notion of events which arrive at various points in time. The programmer declaratively specifies *what* should happen, and the reactive run-time figures out by itself *when* it should happen.

There are currently two main variants of reactive programming, namely *Functional Reactive Programming* and *Reactive Streams*, which we briefly introduce in Section 1.1.1 and Section 1.1.2. Afterwards, in Section 1.1.3, we briefly discuss how to use them to build reactive programs that are distributed over a network.

1.1.1. Functional Reactive Programming

Functional Reactive Programming (FRP) can be easily explained using the programming model of spreadsheets. In a spreadsheet, when a cell `C1` contains the expression `=A1+B1`, then the value of `C1` is automatically recomputed every time the contents of cell `A1` or `B1` changes. FRP operates on the same principle, but elevated to the level of programming languages and frameworks. It has been used for a wide variety of systems such as modelling and simulation [95], robotics [63], networking [48, 143], GUIs [36, 65, 88], mobile ad hoc networks [27], Wi-Fi chips [134], real-time systems [149], audio processing and sound synthesis [64], computer vision [104], stage lighting [133], distributed systems [83, 90, 107, 117, 132], etc. Furthermore, results from an empirical study on program comprehension suggest that FRP code is easier to write and understand compared to the Observer pattern in object-oriented programming [124].

1.1.2. Reactive Streams

Reactive Streams is a term that encompasses a range of technologies based on asynchronous data streams, where a stream represents a (possibly infinite) sequence of values. Streams can be combined and processed using specially designed operators such as `map` and `filter`. Reactive streams are popular in industry via frameworks such as Akka Streams [119] and ReactiveX, a specification which has been implemented in 18 languages [113] some of which are developed or maintained by companies such as Microsoft and Netflix. An implementation of a "Reactive Streams" specification has been included in Java since version 9 [32, 111].

1.1.3. Distributed Reactive Programming

The problems of developing event-based applications also arise when developing distributed applications, where application updates include events that originate from other devices on the network (e.g., sensor updates). When using reactive programming to implement distributed reactive applications (e.g., using ReactiveX [113], Akka Streams [119], REScala [123]), every device on the network represents (part of) a reactive program. These programs consist of prosumers that interact with each other via data streams over a network.

In most cases the number of input and output streams of a reactive program is limited and known beforehand (i.e., a distributed variant of level 1 reactivity) because the number of prosumers is fixed. This is different when developing distributed applications for *open networks*, where programmers are faced with level 2 reactivity. Here, there can be any number of prosumers that provide input to the reactive program which cannot be known beforehand.

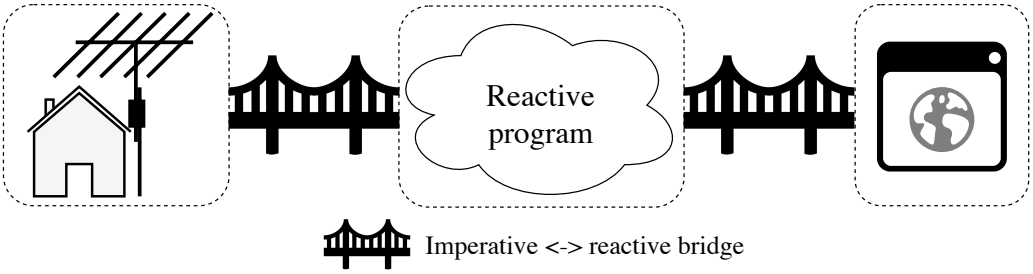


Figure 1.2.: Every reactive program needs to bridge the gap between the imperative input and output parts of the program.

1.2. Problem Statement

Reactive programming languages and frameworks that implement level 1 reactivity have been an active research topic for the past 2 decades. Existing reactive programming languages and frameworks focus on new application domains for reactive programming, as well as ever more powerful concepts to implement said level 1 reactivity. More specifically, they focus on the language features and abstractions whereby programmers can express reactions to events as easily and as declaratively as possible. However, we observe that they focus only on the *internal part* of a reactive program, i.e., the part which is programmed using reactive programming techniques (FRP or reactive streams). Reactive programs also have 2 other parts: An *input part* provides input to the reactive program by triggering events, and another (possibly different) *output part* processes the output of the reactive program in order to "do something". For example, Figure 1.2 depicts a reactive program where input is provided by a weather station that communicates via a network, and the output of the reactive program is displayed to the user in a Graphical User Interface (GUI). These input and output parts are not written using *reactive* code, but they are still written using *active* (imperative) code that interacts with the real world via side-effects, in this case networking and the GUI. Programmers have to bridge the gap between these imperative and reactive parts of programs. We have identified 2 issues that arise in existing literature because of this mixing of 2 paradigms, namely the *Reactive Thread Hijacking Problem* (Section 1.2.1) and the *Reactive-Imperative Impedance Mismatch* (Section 1.2.2)

Level 2 reactivity is a new addition to the field of reactive programming proposed by this dissertation which existing reactive programming languages and frameworks do not support. Moreover, when a programmer tries to implement a computation that uses level 2 reactivity by using the features of the language or framework, they run into the *Acquaintance Maintenance Problem* (Section 1.2.3).

1.2.1. Reactive Thread Hijacking Problem

The word *reactive* in "reactive programming" denotes both the programming style (code) as well as the run-time behaviour of reactive programs. In this case we consider the latter, where programs that are called *reactive* are expected to *react* in real-time to any changes that occur. This kind of reactivity, also called *responsiveness*, is one of the 4 key properties in the so-called *"Reactive Manifesto"* [23, 72], which says: *"Responsive systems focus on providing rapid and consistent response times, establishing reliable upper bounds so they deliver a consistent quality of service."* [23]

Unfortunately, existing reactive programming languages and frameworks often impose little to no restrictions on the types of expressions that can be used within the reactive program. When using them, it is often easy to write code that accidentally makes the program no longer reactive. We call this problem the **Reactive Thread Hijacking Problem**, where some data that enters the program can (accidentally) cause the reactive program to become unresponsive. This can be problematic when data is produced by distributed prosumers that are discovered on the open network, e.g., prosumers that are not controlled by the same program.

1.2.2. Reactive-Imperative Impedance Mismatch

The second problem concerns the development of distributed reactive programs, where it is inevitable that some parts of the reactive program will be responsible for handling traditional concerns of distributed programming. In the worst case scenario programmers resort to using low-level sockets, in the best case scenario they have access to higher-level abstractions to perform network requests or pass messages. However, in all cases the act of sending data over a network is a side-effect executed via paradigmatically imperative code rather than reactive code. After all, IO is an effectful aspect of programs.

In analogy with the *Object-Relational Impedance Mismatch* for object-oriented programming [66], we identify the **Reactive-Imperative Impedance Mismatch** as the set of problems that occur when combining reactive and imperative code, which is inevitable when feeding data into, and extracting data from reactive programs. We study their interaction in two directions, namely the embedding of reactive code within an application that is otherwise written using imperative code, and the embedding of imperative code within reactive code. Both directions of the embedding exhibit semantic issues.

In a nutshell, expressions that contain side-effects (e.g., sockets, message passing, or simple assignments) can cause issues when they are embedded in subexpressions of a reactive program. They can cause subtle and tricky bugs since the order of their execution cannot be determined beforehand, they are very difficult to coordinate, and have a detrimental effect on program composition. In the other

direction, there is currently no well-defined and sufficiently general mechanism to embed reactive code within imperative code *without* (accidentally) allowing imperative code to be embedded within reactive code.

1.2.3. Acquaintance Maintenance Problem

The final problem concerns interacting with varying numbers of prosumers on an open network. Here, at every point in time, a prosumer that takes part in a distributed application must know its set of *acquaintances* that can be reached over the network (other prosumers). These acquaintances vary throughout the lifetime of the application as devices join and become unreachable. Hence, a central aspect of distributed reactive programs to support level 2 reactivity is *acquaintance management*, which we describe as the combination of 2 mechanisms:

1. An *acquaintance discovery* mechanism to find acquaintances on the open network. Since prosumers are not necessarily offering a service, we have replaced the traditional term *service discovery* by a more general term *acquaintance discovery* which comprises one prosumer discovering the existence of other prosumers.
2. An *acquaintance maintenance* mechanism to subscribe to discovered streams in order to react as they appear, and to gracefully close the streams as they disappear.

The **Acquaintance Maintenance Problem** denotes the issues that occur within the reactive program to maintain the program state as acquaintances spontaneously appear and disappear. We will make the distinction between 2 different kinds of reactions that can occur: *application-level reactivity* and *topology-level reactivity*, which correspond to level 1 reactivity and level 2 reactivity respectively. In other words, application-level reactivity denotes the application-level values that flow through the reactive program (e.g., sensor measurements), and topology-level reactivity denotes the changes that occur to the internal structure of the reactive program whenever acquaintances appear or disappear. The reactive program needs to ensure that the program state is correctly and efficiently updated whenever a change occurs on the application-level *or* the topology-level. To this end, existing reactive programming languages and frameworks are either inefficient, or require a complex and error-prone mix of code when performing acquaintance maintenance.

1.3. Approach

Our analysis of the overarching problem concludes that **it is inevitable that imperative and reactive code both exist within a reactive program, and that combining them in a single unified language is exceedingly difficult**. The key

to their coexistence is thus to separate them both in terms of their code and their execution at run-time. **We have designed a programming model called the *Actor-Reactor Model* that epitomises such a separation**.

In the Actor-Reactor Model, imperative code can only execute inside *actors*. Actors behave as in the traditional actor concurrency model [70]. They support the full power of a Turing-complete language with side-effects and (infinite) loops. Reactive code is executed in so-called *reactors*. Actors and reactors communicate exclusively via message passing, which makes them highly suitable for developing distributed applications. Each actor and reactor represents a prosumer in a distributed reactive program which can both produce and consume data from other (re)actors.

The Actor-Reactor Model is implemented in a new programming language called Stella, which we use as a linguistic vehicle to demonstrate the core concepts and ideas. A new programming language allows us to communicate these ideas clearly and unambiguously without the technical limitations that would be encountered when writing a library for an existing language. Stella's implementation of the Actor-Reactor Model tackles the problems in the following ways:

Reactive Thread Hijacking Problem To ensure that incoming data cannot (accidentally) block the reactive program from being able to process any subsequent data, Stella ensures that any computation within a reactive program must *eventually* terminate. While stricter guarantees exist and are used in the field of reactive programming (e.g., for embedded systems), Stella's application domain of distributed reactive systems does not require such strict constraints. We believe Stella's so-called *eventual reactivity* is a reasonable trade-off between preventing that a reactive program can get stuck indefinitely and correctly shaping a programmer's thoughts to write reactive code that also behaves reactively.

Reactive-Imperative Impedance Mismatch One of the key features of the Actor-Reactor Model is to completely avoid the Reactive-Imperative Impedance Mismatch by separating imperative and reactive code using actors and reactors. Stella demonstrates a practical way to achieve this separation by defining the language in two tiers. An object-oriented base language is used to implement abstract data types, and the concurrent level of actors and reactors can share those data types between each other.

Acquaintance Maintenance Problem Stella offers acquaintance discovery via a so-called *flock*, a software component that automatically gathers prosumers (actors and reactors) that can be found on the network. Stella's acquaintance maintenance mechanism is conceived as a new operator for reactive programming languages which we call `deploy-*` and which integrates with flocks. The operator can correctly and efficiently update the result of reactive computations as the number of connected prosumers fluctuates.

1.4. Contributions

The contributions of this dissertation are as follows.

First, we present a **taxonomy of reactive programming languages and frameworks** according to the mechanisms used in related work to support distributed reactive programming, as well as a taxonomy of related work which indicates the degree to which they are subject to the discussed problems.

Second, we perform a thorough **problem analysis** that identifies the aforementioned problems, and which introduces new terminology and concepts to describe concepts from existing reactive programming languages and frameworks. Among others we introduce the terms *weak reactivity*, *eventual reactivity* and *strong reactivity* to describe the degree to which a reactive program guarantees that it will be able to react to new data, and we introduce *application-level reactivity* and *topology-level reactivity* to more precisely describe level 1 and level 2 reactivity which occur within a reactive program.

Third, we propose the **Actor-Reactor Model** as a programming model to describe and implement reactive programs. We conjecture that the model is already present in some way in related work, either intentionally or unintentionally, much like a "natural phenomenon", i.e., a structure that arises naturally when implementing a reactive language, framework, or application.

Fourth, we design **flocks** as a new programming abstraction to discover prosumers on an open network, and we introduce a novel operator for reactive programs called `deploy-*` which complements flocks, and which is used to implement correct and efficient computations based on fluctuating numbers of prosumers.

Finally, we practice an **artefact-based research method**. Besides a scientific solution to the described problems, we implemented and tested the proposed solutions to verify that they work in practice as well as in theory. To this end, the Stella language is the main technical contribution. Besides implementing the Actor-Reactor Model, Stella enforces eventual reactivity, which means that any reactive program's reaction must eventually terminate. Furthermore, to foster reproducibility of the research, we present a meta-implementation of a crucial part of Stella, namely that of the reactor, which is implemented in Stella itself.

1.5. Outline of the Dissertation

The dissertation is structured as follows.

Chapter 2: State of the Art: Reactive Programming We introduce the reactive programming paradigm and its 2 main variants, namely *Functional Reactive Programming* and *reactive streams*. A basic understanding of reactive programming is essential for the rest of the dissertation.

Chapter 3: State of the Art: Distributed Reactive Programming We discuss the techniques used by existing reactive programming languages and frameworks to build distributed reactive systems.

Chapter 4: Problem Statement We analyse the problems that occur when writing distributed reactive programs for open networks, namely the Reactive Thread Hijacking Problem, the Reactive-Imperative Impedance Mismatch, and the Acquaintance Maintenance Problem.

Chapter 5: Reactive Programming in Stella The solution to the identified problems is the Actor-Reactor Model which was implemented in Stella. This chapter introduces the Stella language and tackles the Reactive Thread Hijacking Problem and the Reactive-Imperative Impedance Mismatch.

Chapter 6: Distributed Reactive Programming in Stella This chapter introduces Stella's features that are tailored to distributed reactive programming in order to tackle the Acquaintance Maintenance Problem. More specifically, we introduce the *flock* to discover distributed prosumers on an open network, and the reactive operator called `deploy-*` to implement correct and efficient computations based on the discovered devices.

Chapter 7: Qualitative Evaluation: Comparison to the State of the Art We evaluate the Actor-Reactor Model and Stella qualitatively via an extensive taxonomy of related work. Firstly, we categorise existing reactive programming languages and frameworks according to the problems introduced in Chapter 4. We will show that many other reactive programming languages and frameworks are subject to the discussed problems in some way. Furthermore, we argue that some existing languages and frameworks already implement some form of Actor-Reactor Model (either intentionally or not).

Chapter 8: Mira: A Meta Specification of Reactors in Stella To foster reproducibility we provide a meta-implementation of Stella's reactors in Stella itself.

Chapter 9: Conclusion We conclude the dissertation and highlight avenues for future work.

2. State of the Art: Reactive Programming

The reactive programming paradigm aims to offer a solution to the problems that occur when writing highly interactive, event-driven applications. It offers abstractions to express programs as reactions to events that arrive over time. A language or framework will automatically manage dependencies between events, and it automatically takes care of updating the program state whenever events occur.

In this chapter we introduce reactive programming. We will distinguish between 2 main ways to develop reactive systems, namely *Functional Reactive Programming* (FRP) and *reactive streams*, which we introduce in Section 2.1 and Section 2.2 respectively. Additionally, we will briefly discuss *synchronous programming languages* in Section 2.3.

This chapter also provides a taxonomy of related work on (non-distributed) reactive programming in Table 2.1 on page 12. The 2nd column lists the host language in which the framework is embedded, or a – (hyphen) when the subject is a dedicated programming language. The 3rd column lists the language or framework family, namely the aforementioned FRP or reactive streams. Finally, the 4th column lists the type of evaluation model employed by the subject, namely a push or pull-based model (or support for both).

Lang./Lib.	Host language	Family	Evaluation Model
Reactive programming for interactive applications or GUIs			
Dunai [102]	Haskell	FRP	Push-pull
Elm [36]	-	FRP	Push
Flapjax [88]	JavaScript	FRP	Push
Fran [43]	Haskell	FRP	Pull
Frappé [33]	Java	FRP	Push
FRPNow [106]	Haskell	FRP	Pull
FrTime [31, 65]	Racket	FRP	Push
Haai [97, 98]	-	FRP	Push
Hokko [116]	Scala	FRP	Push
NewFran [44]	Haskell	FRP	Push-pull
RxJS [121]	JavaScript	Streams	Push
Scala.React [79]	Scala	FRP	Push
Yampa [63]	Haskell	FRP	Pull
Reactive programming for embedded systems (e.g., microcontrollers, networks)			
Frenetic [48]	Python	FRP	Push
Nettle [143]	Haskell	FRP	Pull
CFRP [136]	-	FRP	Push
EmFRP [127]	-	FRP	Push
Hae [150]	Haskell	FRP	Push
ReactiFi [134]	Scala	FRP	Push
RT-FRP [149]	-	FRP	Pull
Other			
Coherence [41, 42]	-	FRP	Push
Lively RaTT [9]	Operational semantics	FRP	-

Table 2.1.: Taxonomy of the state of the art in reactive programming. Note that we exclude reactive programming languages and frameworks for distributed reactive programming, which will be considered in Chapter 3 (Table 3.1 on page 32).

2.1. Introduction to Functional Reactive Programming

FRP languages and frameworks combine the ideas of reactive programming with the basic building blocks of functional programming. It was first formulated in 1997 in Fran (an abbreviation of "Functional Reactive Animation") [43], a library for Haskell [105] designed for graphics and animation. In general, the fundamental mechanism of functional reactive programming can be easily explained using the programming model of spreadsheets. When a cell `C1` contains the expression `A1 + B1`, then the value of `C1` is automatically recomputed every time the contents of cell `A1` or `B1` changes. FRP languages and frameworks operate on the same principle, but elevated to the level of programming languages.

The ideas introduced by Fran have been adopted by a wide range of languages and frameworks. Many of them are listed in Table 2.1 in the FRP family.

2.1.1. Signals, Behaviours, and Events

FRP languages and frameworks often offer two different abstractions to represent values that can change over time:

Signal or behaviour A *signal*, also referred to as a *behaviour*, is a time-varying value, i.e., an abstraction whose concrete value changes over time. In the original literature of Fran it is said to be continuous, meaning that its value is defined at every point in time (at infinitely small granularity). For example, time itself is often modelled as a signal. In the aforementioned example of a spreadsheet program, every cell in the program can be modelled as a signal.

Event (streams) An *event* is an abstraction for a value that occurred at a fixed point in time, for example, keyboard strokes and mouse clicks. Events are often used in combination with special operators that trigger when one or more events occur. Because events in FRP languages are often similar to reactive streams, events will be the focus in Section 2.2.

There is a duality between signals and event streams since either one can be used to implement the other [30]. FRP languages that contain both abstractions (e.g., Fran [43], FrTime [31], Flapjax [88] and REScala [123]) also include operations to convert one to the other. The difference between them is how the programmer approaches the solution to a particular problem, i.e., the code style. Whereas the continuous nature of signals lends itself to using them in combination with functions, event streams are used in combination with stream operators such as `map` and `filter`.

2.1.2. Lifting

Most reactive languages and frameworks use ordinary (i.e., non-reactive) functions to perform operations on signals. Most often these functions are defined in some

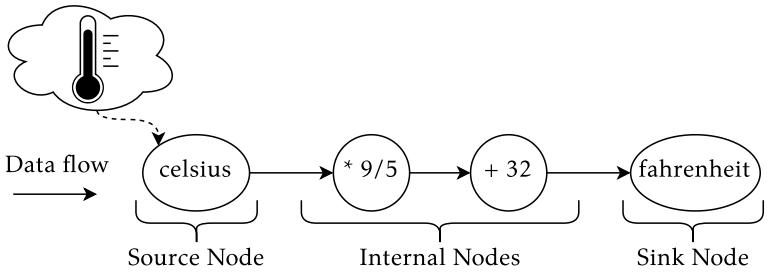


Figure 2.1.: Compiled structure of a reactive program to a DAG, which converts temperatures in Celsius to Fahrenheit. Data flow is usually depicted from top to bottom, but depicted here from left to right.

.

base language (e.g., Haskell, Scala, …). These ordinary functions can be applied to signals by *lifting* them to the level of signals. Often the lifting is done automatically when applying a function to a signal. For example, consider a signal `celsius` that is "driven by" some real thermometer, whose value at every point in time is the current temperature in Celsius. The following code snippet (written in FrTime [31], pronounced "Father time") converts each temperature measurement to `fahrenheit`.

```
(define fahrenheit (+ (* celsius 9/5) 32))
```

The arithmetic functions `+` and `*` from the base language (in this case Scheme) are automatically lifted by FrTime when they are applied to a signal such as `celsius`. Applying a lifted function to signals returns a new signal, which in this case stored in `fahrenheit`. Whenever the value of `celsius` is changed, then FrTime ensures that the value of `fahrenheit` will change as well by recomputing the value of `(* celsius 9/5)` and the subsequent invocation of the `+` function (just like a spreadsheet).

2.1.3. A Reactive Program's Directed Acyclic Graph (DAG)

Applying a function to signals such as `celsius` creates a new signal. This creates a chain of dependencies among signals which determines which signals should change whenever other signals change. We often think of a reactive program as a Directed Acyclic Graph (DAG). For example, the aforementioned Celsius to Fahrenheit converter can be represented by the DAG drawn in Figure 2.1.

We use the following terminology to refer to specific nodes of the DAG.

Source nodes correspond to the "input" signals of the reactive program. Their values are typically provided by code that is external to the reactive program, e.g., the code powering a thermometer.

Internal nodes are composed signals that constitute the reactive program logic, e.g., the arithmetic that converts a temperature in Celsius to Fahrenheit.

Sink nodes correspond to the "output" signals of the reactive program that constitute the program output. This output is usually processed by code that is external to the reactive program, e.g., to modify some GUI or to send values over a network.

When the value of a signal such as `celsius` changes, then the DAG shows which dependent signals such as `fahrenheit` should be updated as well. Thus we will often say that values are "propagating through the DAG" from the sources to the sinks, updating the value of signals along the way. While in this case the used example is simple (a linear DAG), in Chapter 5 we will use DAGs that have multiple connected sources, sinks, and internal nodes.

In general there are two strategies used by reactive programming languages and frameworks to propagate updates, which can be *pull-based* or *push-based*.

2.1.4. Pull-based Evaluation Model

A pull-based evaluation dictates that the updating of a signal is performed "bottom up", i.e., from the sinks of the DAG to the sources. Continuing the example of a Celsius-to-Fahrenheit temperature converter, in languages and frameworks that have a pull-based evaluation model, the `fahrenheit` node "pulls" a new value from the signals it depends on to update itself, as depicted in Figure 2.2. No computation occurs when the values of `fahrenheit` are not needed in the rest of the program, despite the value of `celsius` possibly changing at a high rate.

In Table 2.1 (page 12) we categorise related work according to their evaluation model. The reactive programming frameworks that are written in Haskell, e.g. Fran and Yampa are usually pull-based. This is because a pull-based evaluation model is a natural fit with the lazy evaluation of Haskell, where the evaluation of expressions is deferred until their results are needed by other computations.

Deferring the updating of signals can potentially cause issues because input values cannot be skipped. Thus, a history of changes to the input signals such as `celsius` is accumulated until eventually `fahrenheit` pulls new values. In literature this is called a *time leak* [71], because the reactive program then has to "catch up" until it has processed the entire history of values up until the most recent one, which can take a very long time (hence called a time leak). A pull-based model can also lead to *space leaks*, which refers to a memory leak that arise from (accidentally) capturing too much of the history of a signal [71], which often occurs when signals are first-class values (i.e., they can be the values of other signals).

Various techniques have been proposed to avoid spacetime leaks, each with varying degrees of functionality and complexity. For example, NewFran supports

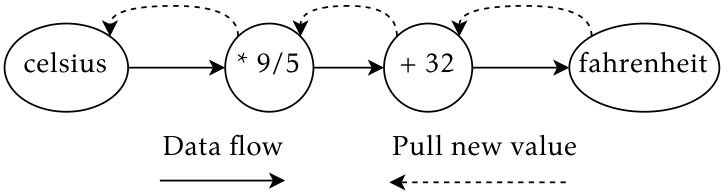


Figure 2.2.: Adaptation of the DAG in Figure 2.1 to depict a pull-based model.

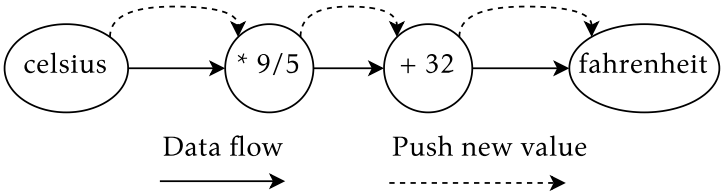


Figure 2.3.: Adaptation of the DAG in Figure 2.1 to depict a push-based model.

both a push and pull-based evaluation model, *Arrowized FRP* [63], and *Monadic Stream Functions* [103]. In this dissertation we will use a push-based evaluation model, which does not suffer from spacetime leaks, and which we consider to be more appropriate in the domain of distributed reactive applications.

2.1.5. Push-based Evaluation Model

A push-based evaluation model eagerly pushes values through the reactive program. Continuing the example of a Celsius-to-Fahrenheit converter, data will be pushed down the DAG whenever the value of the `celsius` signal is updated, as graphically depicted in Figure 2.3. Most languages and frameworks such as FrTime [31], Flapjax [88], and REScala [123] that are embedded in languages without lazy evaluation are usually push-based (cf. Table 2.1). In general we consider a push-based evaluation model to be more suitable for event-driven applications that are expected to react immediately to events, which is the case in the domain of distributed reactive applications. However, the cost is additional complexity for the developer of the language or framework because of *glitches*.

Glitches All languages and frameworks that implement a push-based model require an efficient mechanism to avoid *glitches*, which are computations of incorrect results [31]. Speaking in terms of the DAG, a glitch occurs when an update to a signal propagates through multiple paths of the DAG, causing a signal "downstream" to be updated multiple times. For example, consider the following expression that checks whether the Unix time in seconds is smaller than the same time but incremented.

(a) The two distinct propagation paths are highlighted in blue and red.

(b) The priority of nodes in the DAG is determined by their height in the DAG.

Figure 2.4.: DAG of an expression that can potentially cause a glitch.

```
(< seconds (+ seconds 1))
```

A programmer expects this statement to be perpetually `true` as the value of the `seconds` signal is updated for every tick of the Unix time. The problem occurs when using a naive approach to updating the DAG: When the value of a signal is updated, then its dependents should immediately be reevaluated as well. This means that an update to `seconds` causes the `<` expression and `+1` expressions to be reevaluated. In this scenario the `<` expression is updated twice, namely once after the value of the `seconds` signal changes, and once again after the value of `(+ seconds 1)` changes. The corresponding DAG representation given in Figure 2.4a highlights these 2 propagation paths in red and blue. Assuming that dependent signals are updated from left to right, then the first update of `<` along the red path yields `false` because `(+ seconds 1)` has not been reevaluated yet. This result semantically wrong, and in general it wastes computing resources. Hence, glitches should be avoided.

Glitch Prevention: The Reactive Engine Every push-based FRP language or framework has what we call a *reactive engine* that is responsible for propagating signal updates through the DAG in a sensible way to avoid glitches. The idea is to prevent a signal such as `<` from being updated until all of its dependencies have been updated to the latest value. We will briefly explain the algorithm proposed by FrTime, which serves as the basis for the implementation of other FRP languages and frameworks such as Flapjax, REScala, Scala.React, Haai, AmbientTalk/R, Frenetic, and Stella.

The main idea of FrTime's update algorithm is that the nodes of the DAG are topologically sorted, and reevaluated in that topological order. In other words, each signal can only be reevaluated *after* all of its predecessors have been reevaluated. In the example of Figure 2.4a, both the `seconds` and `(+ seconds 1)` nodes

are updated before the < node is updated, thus preventing the glitch. To this end, every node in the DAG is labelled with its height in the DAG. The DAG heights for the running example are depicted in Figure 2.4b. Source nodes have the lowest height 1, and each dependent node increases maximum height of its predecessors by 1.

Once every node has an assigned height, at run-time a priority queue can be used to schedule nodes for reevaluation. Each node's height determines the priority in the queue, where nodes with a lower height have a higher priority, and are thus executed first. For example, using the heights of Figure 2.4b, when node 1 changes and nodes 2 and 3 are scheduled, then node 2 will update before 3 thus preventing the glitch.

2.2. Introduction to Reactive Streams

Reactive streams is the collective term for frameworks based on asynchronous data streams. The first library based on reactive streams seems to originate at Microsoft[1], where Erik Meijer and his Cloud Programmability Team developed Rx, also called ReactiveX and Reactive Extensions. The first version of Rx for .NET developers was released to the public in 2009 [39] and open-sourced in 2012 [86]. Frameworks such as Rx are labelled in Table 2.1 (page 12) as belonging to the family of streams. Note that, despite our taxonomy only listing 3 streaming frameworks, since the original formulation of Rx it has been modernised and ported to over 18 languages [112] and is widely adopted by both hobbyist programmers and enterprises. The other implementations of reactive streams that we considered are Akka Streams [119] (Scala, Java) and Creek (Elixir). While other streaming frameworks exist such as Monix [91] and FS2 [51] which we did not include in Table 2.1, in our experience the core concepts they offer are very similar to those that we have already included.

The central abstraction in every streaming framework is the *stream*, a (possibly infinite) sequence of values. A rich library of built-in *stream operators* are used to functionally build, transform and compose streams. We will exemplify streams and their operators in RxJS (Section 2.2.1) and Akka Streams (Section 2.2.2), which we consider to be representative for the state of the art.

2.2.1. Reactive Streams in Rx

A stream in Rx is often called an *observable*, named after the `Observable` interface that they implement [121]. While `Observable` and the methods it implements are

[1] We only considered the modern incarnation of reactive streams sparked by Rx which is currently widely used to develop reactive programs, rather than stream programming in general, e.g., pLucid [8].

```
1 const numbers = from([1, 2, 3, 4, 5]);
2 numbers.subscribe((x) => console.log(x));
```

Listing 2.1: Building a stream using `from` in RxJS

```
1 numbers.subscribe(x => console.log(x));
```

Listing 2.2: Subscribing to a stream in RxJS.

well documented, rather than focussing on its implementation, we will show how to use the interface to build, transform, and combine streams. We will exemplify these aspects using RxJS, a modern and widely used implementation of Rx in JavaScript and TypeScript.

Building Streams: Hot vs. Cold Observables

Streams are often created using operators designed to transform ordinary data to a stream. For example, RxJS includes a `from` operator that turns (among others) an array into a stream, as shown in Listing 2.1. The stream referenced by `numbers` is a so-called *cold observable* (an instance of the `Observable` interface), which means that the stream lies dormant until a subscriber appears. The numbers 1 to 5 are emitted (in order) to every subscriber individually once they appear. For example, the assuming the definition of `numbers` is in scope, then Listing 2.2 adds a single subscriber to the stream by invoking `subscribe` on the observable. The `numbers` observable now "wakes up" and starts streaming the numbers in the array to the subscriber. In this case, the given JavaScript lambda with 1 argument is invoked for each element on the stream, after which the stream is stopped. Each additional subscriber will again receive the contents of the array, independent of any other streams that receive values from the same `numbers` stream.

Cold observables are useful for transforming fixed-size data structures to streams where each subscriber receives the entire contents of the data structure. A different kind of observable is called a *hot observables*, which are more suitable for representing (infinite) streams of values such as sensor measurements. Hot observables do not wait for subscribers to appear before emitting a value. Instead, subscribers can spontaneously show up while values are already being emitted. In this case they will not receive the entire history of emitted values, but instead they only receive the values that are emitted after they subscribed.

One way to create a hot observable is via a built-on operator called a `Subject`, which is a type of observable to which the programmer can manually push new values via a `next` method. The use of a `Subject` is demonstrated in Listing 2.3. Here, we define a `numbers` stream similar to the previous example, but the values

```
1 const numbers = new Subject();
2 numbers.next(1);
3 numbers.next(2);
4 numbers.subscribe(x => console.log(x));
5 numbers.next(3); // console logs 3
6 numbers.next(4); // console logs 4
7 numbers.next(5); // console logs 5
```

Listing 2.3: Manually pushing values to a stream in RxJS

```
1 const numbers = from([1, 2, 3, 4, 5]);
2 const odds = numbers.pipe(
3   map((x) => x + 10),
4   filter((x) => x % 2));
5 odds.subscribe((x) => console.log(x)); // console prints 11 13 15
```

Listing 2.4: Transforming a stream using `map` and `filter` in RxJS

1 to 5 are emitted by invoking `next`, rather than being supplied by an array. A new subscribe appears on line 4 after numbers 1 and 2 have already been emitted. These numbers are not received by the subscriber, and thus only the subsequent numbers 3-5 are printed to the console when they are emitted.

Unary Operators on Streams

A multitude of operators exist to transform the data that is emitted to streams. Many of them inspired by list-based operators in functional programming. In general, applying an operator to a stream returns a new stream that exhibits the characteristics of the applied operator. For example, applying a `map` operator to a stream will return a new stream where a function is applied to each emitted value, and a `filter` operator defines a new stream that only emits the values of a given input stream whenever a given predicate returns `true`.

In RxJS, operators are applied to streams by *piping* the values of streams through a sequence of operators. For example, Listing 2.4 defines a stream `odds` that pipes the values from a `numbers` stream through a `map` and `filter` operator. The `map` adds 10 to every value, and `filter` emits only the values that are odd. Note that the input stream `numbers` is an implicit argument of `map` and `filter`, which are arguments of the `pipe` function invoked on `numbers`.

N-ary Operators on Streams

Stream are not just unary transformations of values using maps and filters. Many operators exist to combine multiple input streams to 1 output stream. For example,

```
1 val numbers = Source(1 to 5)
2 val printlnSink = Sink.foreach(println)
3 numbers.runWith(printlnSink)
```

Listing 2.5: Building a stream in Akka Streams

operators such as `zipWith` take n streams as input and combine their values into an array: Given two streams that emit the natural numbers, `zipWith` produces a new stream that emits the arrays `[0, 0]`, `[1, 1]`, etc. In general, many different types of operators are built-in to cover many scenarios that a programmer may need. For instance, a drawback of `zipWith` is that it waits for a new element to be emitted by *every* input stream. This is problematic when streams produce values at different rates, e.g., a fast stream and a slow stream. Hence, a frequently used variant of `zipWith` is called `combineLatestWith`, which emits a combined value whenever *one* of the input streams emits a new value by using the latest (old) value for the other input streams.

2.2.2. Reactive Streams in Akka Streams

Akka Streams is a framework in Scala and Java that adds reactive streams to Akka, a popular actor library. The main novelty of Akka Streams is its combination with actors. Similar to RxJS, we will show how to build, transform and combine streams.

Building Streams: Materialization on Actors

In Section 2.1 we introduced FRP, and we showed the correspondence between the program text of a reactive program and its graphical representation as a DAG. In Akka Streams the DAG is explicit. For example, Listing 2.5 defines the same `numbers` stream from before by using Akka Stream's `Source` operator on line 1. However, unlike in RxJS, the value stored in variable `numbers` is not a stream object (e.g., an observable in RxJS). Instead it represents a part of a static DAG, more specifically an object of type `Source[Int, NotUsed]`[2]. This object directly corresponds to what we called a DAG's source node in Section 2.1.3.

Akka Streams requires a DAG to be complete before it can be run, which means that it should consist of at least a source node and a sink node. We define such a sink on line 2, which will invoke the given `println` function for each value that it receives. Finally, the invocation of `runWith` on line 3 connects `numbers` to the sink and runs an instance of the stream.

[2]While the exact type signature is not important, `Int` denotes the type of the values emitted by the source, and Akka Streams' `NotUsed` is the type of (optional) auxiliary information provided by the source, i.e. none in this example.

```
1 val numbers = Source(1 to 5)
2 val printlnSink = Sink.foreach(println)
3 val multiplyAndFilter = Flow[Int].map(x => x * 10)
4                                  .filter(x => (x % 2) == 0)
5 numbers.via(multiplyAndFilter)
6        .runWith(println Sink)
```

Listing 2.6: Transforming a stream using `map` and `filter` in Akka Streams

In RxJS streams are (usually) executed in the main JavaScript event loop, i.e., the main program thread. In Akka Streams, Akka actors are responsible for executing steams, which they call *materializing* a stream. Invoking `runWith` connects a partial DAG to a sink, spawns a new actor[3], and runs an instance of the complete DAG on that actor.

Unary Operators on Streams

A rich set of built-in stream operators can be used to build (parts of) a DAG. In Section 2.2.1 we used RxJS to implement a stream that multiplies each number by 10 and filters out the odd numbers. We implement the same example in Listing 2.6. Lines 1 to 2 defines the same DAG source and sink as before, and lines 3 to 4 defines a part of a DAG via Akka Streams' `Flow` class, which transforms values of type `Int` by mapping and filtering them. The source, internal nodes and the sinks of the DAG are composed and materialized on lines 5 to 6, which dictates that values from `numbers` should first pass via `multiplyAndFilter`.

N-ary Operators on Streams

Similar to RxJS, Akka Streams includes a wide range of built-in operators that help developers build the DAGs that they desire. Unlike RxJS, Akka Streams features a domain specific language that they call the *GraphDSL* to compose more complex DAGs with multiple inputs and outputs [78]. While we will not delve into the details of the GraphDSL, in Listing 2.7 we construct the same DAG as Listing 2.6 but using the GraphDSL. If one ignores the boilerplate code, lines 3 to 6 implement the same DAG components (source, internal nodes and sink). They are composed in the GraphDSL using the squiggly arrow syntax on line 7. These arrows directly correspond to edges in the DAG, but in code rather than a graphical depiction. Using the GraphDSL, a programmer can create much more complicated DAGs including operators that have multiple inputs or outputs, which we will not show for brevity.

[3] Akka Streams offers the programmer control over where the actor is spawned, but we have omitted this for brevity.

```
1 val g = RunnableGraph.fromGraph(GraphDSL.create() { implicit builder:
      GraphDSL.Builder[NotUsed] =>
2   import GraphDSL.Implicits._
3   val numbers = Source(1 to 5)
4   val printlnSink = Sink.foreach(println)
5   val multiplyAndFilter = Flow[Int].map(x => x * 10)
6                                    .filter(x => (x % 2) == 0)
7   numbers ~> multiplyAndFilter ~> printlnSink
8   ClosedShape
9  })
```

Listing 2.7: Combining Streams using the GraphDSL of Akka Streams

2.3. Synchronous Programming Languages

Synchronous programming languages are often included in the term "reactive programming". We did not include them in the taxonomy of Table 2.1 (page 12) and Table 3.1 (page 32), but we will briefly discuss them here.

Similar to FRP and reactive streams, synchronous languages are built for applications that continuously react to their environment. However, they were not designed to avoid the issues of writing event-driven programs using callbacks or observers, and are used to program an entirely different type of application. More specifically, they are used to program real-time systems and embedded systems with strict timing requirements, and where safety and reliability is critical [13, 61]. Languages such as Esterel [18], Lustre [59] and SIGNAL [73] originated in the early 1980's and remain actively used today, and more recent languages include Céu [126] and HipHop.js [20].

The programming model of synchronous languages is conceptually simple. A synchronous reactive system continuously reacts to a sequence of input events by producing a sequence of output events. Reactions to a set of events which occur at one instant of a logical clock are considered to be *instantaneous* as if executed by an infinitely fast machine. In other words, the program is always able to react fast enough to input events, and is able to produce all output events before processing the next set of events at the next instant of a logical clock. This property is called the *synchrony hypothesis* [18], a graphical depiction of which is given in Figure 2.5. Upon receiving a set of input events, all modules of the application (i.e., its logical components) are instantaneously activated at all levels of the hierarchy.

We will briefly introduce 2 forms of synchronous languages, namely imperative languages (Section 2.3.1) and dataflow languages (Section 2.3.2), and then we will discuss their applicability for developing distributed reactive programs for open networks (Section 2.3.3).

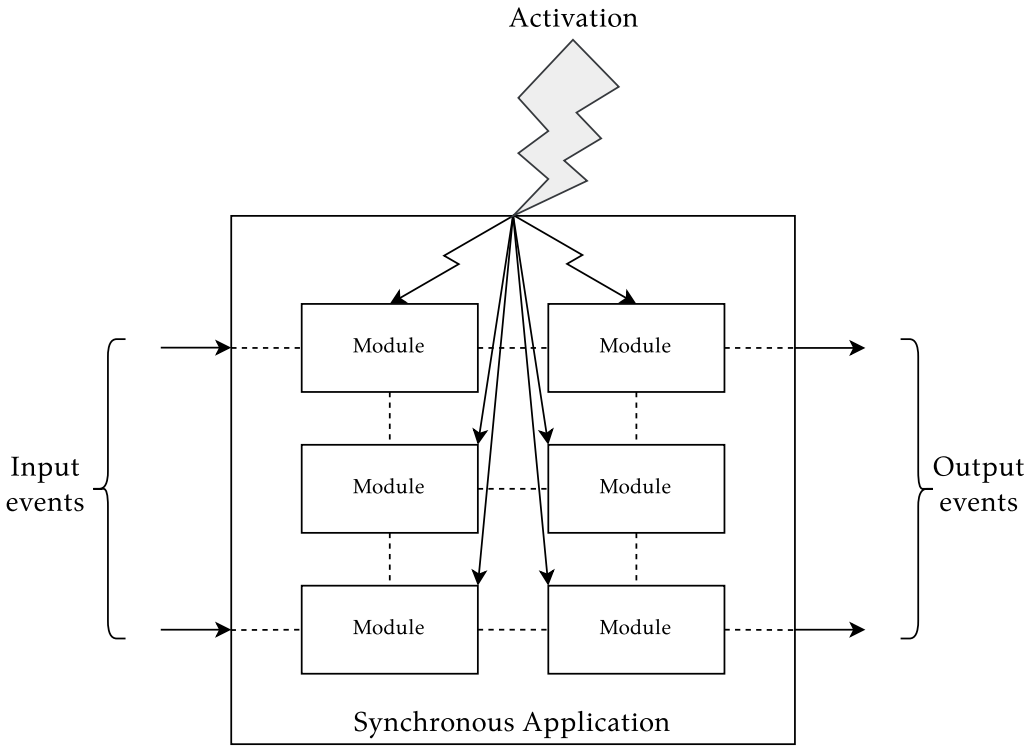


Figure 2.5.: Graphical depiction of a synchronous reactive system.

```
module M:
input names;
output names;
  statement
end module
```

Listing 2.8: Generic structure of a module in Esterel.

2.3.1. Imperative Synchronous Languages

Imperative synchronous languages are highly suited for applications that focus on control handling, e.g., alarm systems and machine interfaces. The first such language was Esterel [18], which continues to influence recently proposed languages such as Céu [125] and HipHop.js [20].

A pure Esterel program is conceived as a *module* as shown in Listing 2.8: a module has a name (e.g., `M`), a set of input events, a set of output events, and a body statement[4]. An event is either present or absent in a given *instant*, which is a single activation of the application, i.e., the complete (instantaneous) processing of a set of input events that generate a set of output events. The logic of an Esterel program is written in terms of the presence or absence of events, which are processed using language statements. There are 11 such core language statements that form the basis of Esterel [16] and a rich library of derived language statements.

The "Hello World!" of synchronous languages is called ABO: if the events `A` and `B` occur, then trigger output `O`. This example is implemented as a module in Listing 2.9. The module has 2 input events `A` and `B` and one output event `O`. The body of the module is defined as a single statement [5] that contains a subexpression of the form `p||q` (the `||` statement is pronounced "parallel"), which itself has 2 subexpressions, namely `await A`, `await B`. When the `||` statement is executed it will immediately (i.e., within the same logical clock cycle) execute both constituent statements in parallel, and the program will proceed past the `||` statement whenever both constituent statements terminate. In this case, the constituent `await` statements will only proceed whenever an event `A` and `B` respectively is present. When this is the case, the program proceeds and emits an `O` event as output.

[4] In the literature events are referred to as *signals*. We did not use this term to avoid confusion with the already established terminology of signals in FRP languages.

[5] Language statements may be arbitrarily grouped within brackets "`[`" and "`]`", and a statement `p;q` sequences 2 statements to be executed in the given order, i.e., first `p` then `q`.

```
module ABO:
input A, B;
output O;
  [ await A || await B ];
  emit O
end module
```

Listing 2.9: ABO in Esterel.

2.3.2. Synchronous Dataflow Languages

Synchronous dataflow languages such as Lustre [59] and SIGNAL [73] are highly suitable for applications such as electrical signal processing and monitoring. We will briefly introduce the concepts used in Lustre to demonstrate the difference in programming style compared to imperative synchronous languages such as Esterel.

Any variable and expression in Lustre denotes a *flow*, which is a pair that consists of:

1. A (possibly infinite) sequence of values of a given type.
2. A logical clock.

In other words, the current value of a flow is determined by the current value of a flow's clock. Basic Lustre expressions include:

`X = E;` The variable `X` is semantically identical to the expression `E`. I.e., `X` denotes exactly the same sequence of values with the same clock as `E`.

`pre(e)` If the current clock of the flow `e` is the value n, then the `pre` operator retrieves the value of flow `e` at its clock $n - 1$, i.e., its previous value (initially the null-value `nil`).

`E -> F` Given two flows `E` and `F` with the same clock, which are represented by their sequence of values $(e_1, e_2, \ldots, e_n)$ and $(f_1, f_2, \ldots, f_n)$, then the operator `->` (pronounced "followed by") defines a new flow whose sequence is $(e_1, f_2, \ldots, f_n)$. The new sequence is equal to the sequence of `F` except for the value at the first clock which is given by `E`. For example, the following expression implements a counter that counts the number of clock cycles in a program.

```
n = 0 -> pre(n) + 1;
```

Consider an electric signal that is either on or off (e.g., 1.4 volt signifies that the signal is on, and 0 volt means that it is off), then a rising edge is the moment at which the signal switches from 0V to 1.4V, and a falling edge is the moment at which the signal switches from 1.4V to 0V. Lustre can be used to detect such edges. For example, Listing 2.10 implements a *node*, which is Lustre's basic unit of composition, that detects a falling or rising edge. A node has a name and a

```
1 node EDGE(A: bool) returns (B: bool);
2 let
3   B = false -> A and not pre(A);
4 tel
```

Listing 2.10: The definition of an EDGE node in Lustre.

```
1 node ABO (A,B: bool) returns (O: bool);
2   var seenA, seenB : bool;
3 let
4   seenA = false -> A or pre(seenA);
5   seenB = false -> B or pre(seenB);
6   O = EDGE(seenA and seenB);
7 tel
```

Listing 2.11: ABO in Lustre.

number of inputs and outputs. In this case the node is called `EDGE`, which has one input flow called `A` of type `bool`, and one output flow `B` of type `bool`. The body which is defined between `let` and `tel` equates output `B` to be a boolean flow that captures `A`'s rising and falling edges. More specifically, the flow `B` holds `true` whenever the value of flow A transitions from `false` to `true` (rising edge), and it also holds `true` when `A`'s value transitions from `true` to `false` (falling edge).

Nodes can be reused to define larger applications. For example, Listing 2.11 implements the same ABO example that we used to demonstrate Esterel (Listing 2.9 on page 26). It has 2 input boolean flows `A` and `B`, and an output boolean flow `O` that should hold `true` when the flows `A` and `B` both contain `true`, i.e., if `A` and `B` occur, then output `O`. The implementation of ABO on line 2 declares 2 local variables `seenA` and `seenB` that are initialised on lines 4 to 5 whose value starts as `false`, and will remain `true` forever as soon as `A` or `B`'s value respectively is `true`. The output signal `O` uses the aforementioned `EDGE` node to detect the exact moment when `seenA` and `seenB` have both transitioned to `true`.

2.3.3. Discussion: Synchronous Languages for Open Network Distribution

Synchronous languages have many advantages for programming reactive systems where determinism, parallelism, strict timing constraints, and reliability (e.g., through formal verification) is important, e.g., aircraft flight control systems and nuclear power plant control systems [60, 122]. However, we do not consider them to be suitable in the domain of distributed reactive programs for open networks because they use a distribution model that is not comparable to distribution in

general-purpose applications, e.g., web-based applications and open network applications.

Distribution is common in the application domain of synchronous languages such as Esterel, Lustre, and SIGNAL, e.g., for fault tolerance, performance, and due to the distributed location of sensor systems [28]. When a synchronous program is distributed over a network, the main idea is that the distributed program provides the same guarantees as an equivalent non-distributed program. E.g., the synchrony hypothesis is upheld, the program is deterministic, can be formally verified, and is reliable. An approach to do so is via automated tools that, given a centralised synchronous program and a distributed memory architecture, generate distributed code with automatically inserted communication [52, 55].

More generally, a synchronous program can be transformed to run on an asynchronous architecture (i.e., a network) *if* that architecture satisfies the following assumption: "*The architecture obeys the model of a network of synchronous modules interconnected by point-to-point wires, one per each communicated signal; each individual wire is lossless and preserves the ordering of messages, but the different wires are not mutually synchronized.*" [14]. In other words, if network communication is reliable, then it can be proven that the behaviour of the asynchronous (distributed) program is consistent with that of the synchronous (non-distributed) program. This means that all local guarantees regarding determinism and formal verification remain valid. In contrast, when developing distributed reactive programs for open networks there is no such synchronisation, the programs at either end of the network are not necessarily part of the same codebase, and the network is assumed to be unreliable.

2.4. Summary

Reactive programming encompasses various techniques to write event-driven software (more) declaratively without introducing the problems with callbacks and the Observer pattern. In particular, we distinguished 2 ways based on Functional Reactive Programming and reactive streams.

Functional Reactive Programming introduces the notion of a time-varying value in a functional language. These values are often called signals or behaviours. Applying a function to a signal returns a new signal which applies the function to every value carried by the signal. This can be thought of as a conventional spreadsheet program: every cell is represented by a signal, and cells are glued together using spreadsheet operators (i.e., functions) such as arithmetic. Whenever the value of a signal changes, this value will propagate through the reactive program by reapplying the used functions using the signal's updated values.

Reactive streams are a variation on reactive programming where changes of values over time are made explicit via a streams. Here, a reactive program consists

of streams that are connected via stream operators such as `map` and `filter`, which essentially form a pipeline of stream processing steps. Pushing new data though the stream means that all operators in the pipeline process the data from start to completion.

Finally, reactive programs are intrinsically connected to their representation as a Directed Acyclic Graph (DAG). In Functional Reactive Programming the DAG arises naturally from the program text when considering the dependencies between signals via function applications. When using reactive streams, the DAG is created explicitly by wiring stream operators, or in some cases a special DSL is provided to create complex (non-linear) DAGs.

3. State of the Art: Distributed Reactive Programming

This chapter dives deeper into the application of Functional Reactive Programming and reactive streams to develop *distributed* reactive systems.

In general, distributed reactive programming means that the Directed Acyclic Graph (DAG) of a reactive program becomes distributed over a network. To do so, distributed reactive programming languages and frameworks implement various mechanisms to "publish" signals or streams to the network, and to allow other computers to discover these signals or streams, and to react to their changes. For example, a Celsius thermometer may publish its measurements as a signal to the network which is continuously updated as the measured temperature fluctuates. A Celsius-to-Fahrenheit converter that is running on a different device can then immediately react to the signal, rather than manually dealing with low-level networking abstractions (e.g., sockets).

We present a taxonomy of distributed reactive programming languages and frameworks in Table 3.1 (page 32). It has the same structure and classification as our taxonomy of Chapter 2 (Table 2.1 on page 12), but we added a 5th column for distributed reactive programming. Concretely we identified 6 mechanisms in the state of the art to distribute signals or streams over a network. This chapter is structured into 6 sections to discuss them. They are:

- Global signal registry (Section 3.1)
- Conflict-free Replicated Data Types (Section 3.2)
- Actors (Section 3.3)
- Ambient-oriented programming (Section 3.4)
- Multitier reactive programming (Section 3.5)
- Wireless sensor Networks (Section 3.6)

Lang./Lib.	Host lang.	Family	Eval. Model	Distr.
Reactive programming for embedded systems (e.g., microcontrollers, networks)				
Flask [81]	Haskell/Red	FRP	Pull	Sensor networks
Distributed reactive programming				
Akka Streams [77, 119]	Scala	Streams	Push	Actors
AmbientTalk/R [27]	AmbientTalk	FRP	Push	Ambient
Creek [138, 139]	Elixir	Streams	Push-pull	Actors
DREAM [83, 84]	Java	FRP	Push	Global registry
Gavial [118]	Scala	FRP	Push	Multitier
REScala [40, 90, 123]	Scala	FRP	Push	CRDT
ScalaLoci [152]	Scala	FRP	Push	Multitier
Stella [141, 147]	-	FRP	Push	Actors
XFRP [132, 151]	-	FRP	Push	Actors
Other				
ActiveSheets [142]	MS Excel	FRP	Push	Global registry

Table 3.1.: Taxonomy of the state of the art in distributed reactive programming. This taxonomy complements the taxonomy in Chapter 2 (page 32) of non-distributed reactive programming languages and frameworks.

3.1. Global Signal Registry

The first category that we discuss from Table 3.1 (page 32) is a global signal registry. A global signal registry is a registry at some known location (e.g., addressable by hostname or URL) where programs can register their local signals or streams to be discovered by other programs. Other programs can retrieve signals or streams from the registry and use them as if they were defined locally. ActiveSheets and DREAM support such a global signal registry. Due to their very different nature, we will discuss them separately.

3.1.1. ActiveSheets

ActiveSheets is a programming language and tool embedded in Microsoft Excel [142]. It extends Excel with 3 features that enable reactive and distributed programming.

1. Streams of live data can be imported into spreadsheet cells[1]. The value of the cell automatically updates whenever the connected stream emits a new value. Just like a normal spreadsheet, any computations that are derived from such a cell will automatically update when the value of the cell changes.

[1] The original work makes a distinction between a stream and a *feed*. They call a stream an infinite sequence of attribute/value pairs, whereas a feed is an infinite sequence containing only one of those attributes. In our discussion this distinction is not important.

2. In addition to capturing the latest value of a stream, "projections" of streams can be defined, e.g., to capture a sliding window of values emitted to a stream. The size of the window is determined by the user, e.g. to capture the 10 most recent values of a stream in cells A1 to A10.
3. Cells contain additional operations to capture state that can evolve over time.

ActiveSheets extends Microsoft Excel's GUI with widgets that enable ActiveSheets to operate as a distributed reactive program. The first step to do so is to connect the spreadsheet to an external server, which acts as a global signal registry. Users can import streams that are offered by the server, and they can export local data to the server which creates additional streams (importable by other ActiveSheets clients).

In principle, an exported stream terminates as soon as the spreadsheet that exports that stream is closed (e.g., when closing the local application). This means that other clients that use the stream will no longer receive new data. To prevent the stream from being closed, a user can export an entire spreadsheet to the server via a GUI widget, which then perpetually remains active on the server rather than the client.

3.1.2. DREAM

DREAM is a Functional Reactive Programming framework in Java used to investigate consistency guarantees[2] when designing distributed reactive programs. Similar to ActiveSheets, it employs a global signal registry to exchange signals in a distributed application. We will demonstrate its distribution mechanism via a Celsius to Fahrenheit converter.

Listing 3.1 implements a Celsius to Fahrenheit converter. A new signal is defined using the `Var` class, which is used on line 1 to implement a reactive value `celsius` that holds values of type `Integer`. The first argument of `Var` is a (unique) name for the signal, in this case `thermometer`, and the second argument is an initial value for the signal, such as Java's object representation of the primitive number 20.

A new signal called `fahrenheit` is derived from `celsius` by instantiating the `Signal` class. The first argument is a unique name, and the second argument is a Java lambda that implements the computation itself. The `celsius.get()` expression in the body retrieves the current value of the signal. The third argument is used to register which signal gives rise to a reevaluation of the lambda whenever the value of the signal changes, which in this case is only the `celsius` signal.

[2] In the context of distributed reactive programming, *consistency* denotes the degree to which computed values in a DAG that spans different machines are in a consistent state with each other. For example, to prevent glitches in a distributed reactive program.

```
1 Var<Integer> celsius = new Var<>("thermometer", Integer.valueOf(20));
2 Signal<Integer> fahrenheit =
3   new Signal<>(() -> (celsius.get() * 9/5) + 32, celsius);
4 celsius.set(Integer.valueOf(21));
```

Listing 3.1: Converting a Celsius signal to Fahrenheit in DREAM

```
1 RemoteVar<Integer> temp = new RemoteVar<Integer>("Host1", "thermometer");
```

Listing 3.2: Looking up signals in DREAM's global signal registry.

Finally on line 4 the value of the `celsius` signal is modified to be 21, which will propagate through the reactive program.

Provided that the program of Listing 3.1 has registered a hostname such as `Host1` in a configuration file (not shown), then its signals are made available to other DREAM applications on the network via Java's remote object registry [100] (commonly referred to as `rmiregistry`). Other DREAM programs can obtain a reference to those signals via the global registry by looking up signals by name. For example, the code in Listing 3.2 which is running on a different client can address the signal called `thermometer` that is exported by the client with hostname `Host1`. The local object stored in variable `temp` is a proxy for the remote signal whose value will be updated whenever the value of the remote signal is updated.

3.2. Distributed Reactive Programming With CRDTs

A second category in Table 3.1 is the Conflict-free Replicated Data Type (CRDT). One framework, namely REScala, supports distributed reactive programming via CRDTs. We briefly introduce CRDTs and REScala before showing how CRDTs are used to develop distributed REScala programs.

3.2.1. Conflict-free Replicated Data Types

A *Conflict-free Replicated Data Type* (CRDT) is a data structure that can be replicated across multiple programs running on different computers in a network [131]. Every replica can be used and updated by a program without synchronising or coordinating with the other replicas. In other words, a local application can continue to use and update any local state stored in a CRDT even when the network disconnects.

The underlying run-time guarantees *eventual consistency*: If no new updates are made by the programs to their version of the CRDT (i.e., their "replica"), then the state of each replica will *eventually* converge to a single value. In other

```
1 val newEntry = Evt[Entry]()
2 val automaticEntries: Event[Entry] = App.nationalHolidays()
3 val allEntries = newEntry || automaticEntries
4
5 newEntry.fire(Entry("my event title", Date.today))
```

Listing 3.3: Adding calendar entries in REScala with streams. Example from [90].

words, whenever inconsistencies occur between the state of each replica, e.g., after a temporary network failure, then it is mathematically possible to resolve the conflict between different versions of the data.

Not every data structure can be used as a CRDT due to the required mathematical guarantees. Those that are well understood include counters, registers and sets [130].

3.2.2. REScala

REScala is a Functional Reactive Programming library in Scala. It supports both signals and what REScala calls "events". Using our terminology, REScala's "events" abstraction equates to a stream, i.e., a possibly infinite sequence of discrete values. We introduce REScala by using the example from the authors [90], which we also use in Chapter 4.

The example used in [90] is that of a shared calendar where users can add calendar events and select a week to display the calendar events from. The part of the example to add calendar events is shown in Listing 3.3. All calendar events are collected on a stream called `newEntry` (line 1), and the application has a stream of built-in calendar events called `automaticEntries` (line 2). The events from both streams are merged using the `||` stream operator which returns a single output stream called `allEntries` that contains the events from both input streams. An example of adding a calendar event is given on line 5, where a new `Event` (implementation not shown) is pushed onto the `newEntry` stream.

Signals are used to represent a user's currently selected calendar date and week. They are defined in Listing 3.4. A signal `selectedDay` is created by instantiating REScala's `Var` with an initial value of `Date.today`. Then, a signal `selectedWeek` is derived via REScala's `Signal` expression. The definition of the signal between curly braces can make use of other signals such as `selectedDay` which are defined in scope. Here, using `selectedDay.value` will ensure that whenever `selectedDay`'s value changes, then the `Signal` expression is recomputed as well. Finally we show how to modify the value of a source signal on line 4, which will immediately propagate through the DAG and cause the `selectedWeek` signal to update as well.

```
1 val selectedDay: Var[Date] = Var(Date.today)
2 val selectedWeek = Signal { Week.of(selectedDay.value) }
3
4 selectedDay.set(Date.tomorrow)
```

Listing 3.4: Selecting calendar dates in REScala with signals. Example from [90].

```
1  val entrySet: Signal[Set[Entry]] =
2    if (distribute)
3      ReplicatedSet("SharedEntries").collect(allEntries)
4    else
5      allEntries.fold(Set.empty) { (entries, entry) => entries + entry}
6
7  val selectedEntries = Signal {
8    entrySet.value.filter { entry =>
9      try selectedWeek.value == Week.of(entry.date.value)
10     catch { case DisconnectedSignal => false }
11   }
12 }
```

Listing 3.5: Using CRDTs in REScala to implement a shared calendar. Example from [90].

3.2.3. Distributed Reactive Programming in REScala

CRDTs are used by REScala to distribute a reactive program's DAG over a network [90]. The main idea is that CRDTs are used behind the scenes to replicate signals over the network. RESCala takes care of the mechanism required for signals to discover each other, and CRDTs are used to ensure that a signal's value is correctly propagated over a network.

The code responsible for sharing calendar events is shown in Listing 3.5. It defines 2 signals `entrySet` which contains a `Set` of all calendar events, and `selectedEntries` which contains a `Set` of only the calendar events that fall within the user selected week. As a visual aid, we draw the DAG of the corresponding program in Figure 3.1.

The `entrySet` signal is defined on lines 1 to 5. If a `distribute` flag is set, then a new `ReplicatedSet` CRDT is created which is identified (on the network) via its name `SharedEntries`. All local events from the `allEntries` stream are collected by the CRDT and shared over the network. In Figure 3.1 we draw 3 such replicas of the CRDT (for brevity we did not draw any DAGs across the network boundary). If the `distribute` flag is not set, then all local events given by `allEntries` are aggregated into a regular (non-replicated) `Set`. The result stored in `entrySet` is a signal that contains sets: A newly updated `Set` is propagated to dependent signals every time the calendar entries change, e.g., when a new calendar entry is added.

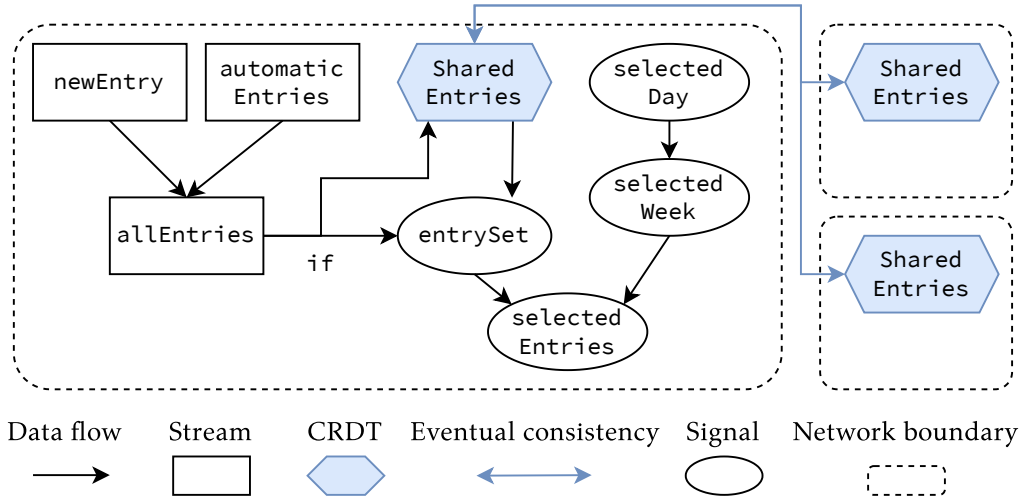


Figure 3.1.: Diagram of the shared calendar application DAG of Listing 3.5.

The `selectedEntries` signal on lines 7 to 12 filters all calendar entries for those that fall within the week selected by the user. The `filter` expression can be considered as standard Scala code. However, note the use of 2 signals in the body, namely `selectedWeek.value` and `entry.date.value`. Hence the filter is automatically recomputed whenever the selected week changes, whenever the date of a calendar changes, or when a calendar entry is added or removed from the `Set` (e.g., when CRDTs synchronise).

3.3. Actors

A third category listed in Table 3.1 (page 32) is the actor model or actors. The actor model is a computational model where *actors* are the basic unit of concurrency [70]. The properties of actors, namely their isolated state and message passing, make them very desirable to build distributed programs. In practice actors have been widely adopted via languages such as Erlang [7], Elixir [67] and JavaScript [57] (Web Workers), and various frameworks such as Akka [119] (Scala, Java), Orleans [15] (C#/.NET), and many more.

In recent years the actor model has been adopted to build distributed reactive programs (cf. Table 3.1), namely by Akka Streams, Creek, XFRP and Stella. We will briefly explain how Akka Streams, Creek and XFRP are used to build distributed reactive programs.

3.3.1. Akka Streams

Akka Streams is a stream-based reactive programming framework for Scala and Java [77, 119]. We previously introduced Akka Streams in Section 2.2.2 on page 21. We discussed the comprehensive API to build a reactive program's DAG, and that such a DAG has to be *materialized* (instantiated) which executes the DAG on an actor.

An Akka Streams program DAG consisted of source nodes, internal nodes and sink nodes. The examples previously shown in Section 2.2.2 used a source that emits numbers (e.g., the expression `Source(1 to 5)`), and the sink of the reactive program printed the result to the console. Materializing this DAG means that a single actor is spawned on which the DAG is running. In the previous examples data was generated locally on the spawned actor, and we did not show any actor-to-actor communication. Since actors are the key enabler of distributed programming in Akka Streams, we will briefly show how to construct Akka Streams programs where multiple (possibly distributed) actors interact with each other.

There are many types of source nodes to feed an Akka Streams program with data, e.g., collections, Java 8 streams, timers, etc. One of these sources is an `actorRef` source [76], which allows the reactive program to react to messages sent by other actors. A use case of `actorRef` is given in Listing 3.6, which implements a Celsius to Fahrenheit converter as an Akka Streams program. The DAG of the program is defined in 3 parts. Lines 1 to 5 define the source node of the DAG, line 7 defines the temperature converter, and line 8 defines a sink that prints the result to the console. For the sake of this discussion, the constructor arguments of `Source.actorRef` can be ignored. They indicate the various strategies to be used by Akka Streams whenever a message is received that cannot be processed, and what happens when messages are sent faster than they can be processed (i.e., they are buffered).

Materializing the DAG on line 10 yields a reference to the actor that runs the stream. This object can be freely passed to other (distributed) actors that are created using standard Akka code (the actor framework that underpins Akka Streams). Other actors can send messages to the stream just like any other actor, i.e. using the `!` (send) operation as demonstrated on lines 11 to 12.

Akka Streams leverages the underlying Akka actor library for distributed actor discovery. It offers abstractions such as a `Receptionist` [74] which is essentially a distributed publish-subscribe system where actors can be registered and retrieved via a unique name.

```
1  val celsiusSource = Source.actorRef(
2    completionMatcher = { case Done => CompletionStrategy.immediately },
3    failureMatcher = PartialFunction.empty,
4    bufferSize = 100,
5    overflowStrategy = OverflowStrategy.dropHead)
6
7  val CToF = Flow[Int].map(c => (c * 9/5) + 32)
8  val sink = Sink.foreach(println)
9
10 val actorRef = celsiusSource.via(CToF).to(sink).run()
11 actorRef ! 22
12 actorRef ! 21
```

Listing 3.6: A Celsius to Fahrenheit converter in Akka Streams which accepts messages from other actors.

```
1 remoteThermometer
2   .map(fn c -> (c * 9/5) + 32 end)
3   .runWith(foreach(f => {
4     printf("Current temp: %fF\n", f))})
```

Listing 3.7: A Celsius to Fahrenheit converter in Creek.

3.3.2. Creek

Creek is a streaming framework for Elixir designed for distributed reactive programming and metaprogramming[3]. Similar to Akka Streams, a Creek program defines a DAG which describes how data is processed, and "instances" of those DAGs are run on actors.

Given a remote thermometer, Listing 3.7 implements a Celsius to Fahrenheit converter as a stream. In this case `remoteThermometer` is assumed to be an actor that implements the correct protocol to communicate with other actors that will run the Celsius to Fahrenheit DAG.

Creek offers an actor discovery mechanism that is tailored towards the Internet of Things, where many devices such as sensors ought to be discovered automatically on the network. Concretely, it offers a global `network` stream that announces meta-messages which indicate that actors join or leave. For example, we can extend the example of Listing 3.7 with additional code to discover a remote thermometer on the network. The original code of Listing 3.7 remains present in Listing 3.8 on lines 4 to 7. We use Creek's `network` stream that will contain device join and leave events. The stream is filtered on line 2 to only keep the tuples where the event is the symbol `:join`, and the type of the device is `:thermometer`.

[3] Metaprogramming in Creek is described in [139]. An early version of Creek that describes its distributed capabilities was called Potato, and is described in [138].

```
network()
  .filter(fn { event, n } -> n.type == :thermometer && event == :join end)
  .map(fn {_ , remoteThermometer } ->
         remoteThermometer
           .map(fn c -> (c * 9/5) + 32 end)
           .runWith(foreach(f => {
           printf("Current temp: %fF\n", f))})
         end)
```

Listing 3.8: Discovering remote actors in Creek.

```
1 module CelsiusToFahrenheit % module name
2 in { client1@hostname } celsius : Float % temperature sensor
3 out fahrenheit : Float
4
5 { client2@hostname } node fahrenheit = (celsius * 1.8) + 32
```

Listing 3.9: A distributed Celsius to Fahrenheit converter in XFRP.

The subsequent `map` operator on line 3 runs the Celsius to Fahrenheit converter for each thermometer that passes the filter.

3.3.3. XFRP

XFRP is an actor-based reactive programming language that compiles to Erlang, and which integrates with Erlang to support distributed programming.

Every XFRP program consists of `module` declarations that implement a functional reactive program. For example, a distributed Celsius to Fahrenheit converter is implemented in Listing 3.9. The module called `CelsiusToFahrenheit` declares a single source node called `celsius` which expects values of type `Float` (denoted by the keyword `in`), and it declares a single sink node `fahrenheit` also containing values of type `Float` (denoted by the keyword `out`). The body of the module implements a single node called `fahrenheit` which was designated as the output of the module, and which implements the temperature conversion in the style of Functional Reactive Programming.

All source, internal and sink nodes in an XFRP program are compiled to Erlang actors. To build a distributed program, in Listing 3.9 we used the optional node placement types before a node's declaration, e.g., `{ client1@hostname }`. They are Erlang host specifiers that denote on which machine (hostname) and Erlang instance (e.g., `client1`) the compiled actors should be spawned. The XFRP runtime takes care of the propagation of values over the network.

```
1  def Thermometer := object: {
2    def temp := 0;
3
4    def temperature() {
5      temp;
6    };
7
8    def @Mutator update(newTemp) {
9      temp := newTemp;
10   };
11 };
```

Listing 3.10: Digital twin of a thermometer in AmbientTalk.

3.4. Ambient-oriented Reactive Programming

A fourth category in Table 3.1 (page 32) is ambient-oriented programming. *Ambient-oriented programming* is a programming paradigm for developing distributed applications [37]. The main difference with traditional paradigms is the base assumption that wireless networks and mobile devices are volatile (e.g., when devices go out of range), and can thus appear and disappear spontaneously. The programming language should incorporate network failures throughout all parts of the application, rather than treating them as an exceptional circumstance (e.g., via exception handling). The main ambient-oriented programming language is AmbientTalk [34]. We briefly introduce AmbientTalk's syntax and object-oriented base language before showing how it is used to build distributed reactive programs.

3.4.1. AmbientTalk

AmbientTalk is a prototype-based object-oriented programming language. For example, we can use a prototype to represent a thermometer's digital twin (i.e., a virtual representation of a physical thermometer). Listing 3.10 defines such a prototype object via the `object: { }` syntax. The prototype has one local field called `temp` which is initialised to 0, a getter method called `temperature`, and a setter method called `update` that assigns to the local `temp` field. The prototype object is assigned to a variable called `Thermometer`. Note that the `@Mutator` annotation in the definition of `update` is not standard AmbientTalk, but is a part of AmbientTalk/R.

```
1 def CelsiusToFahrenheit := object: {
2   def convert(celsius) {
3     (celsius * 9/5) + 32;
4   };
5 };
6
7 def sensor := makeReactive(Thermometer.new());
8 def result := CelsiusToFahrenheit.convert(sensor.temperature());
```

Listing 3.11: A Celsius to Fahrenheit converter in AmbientTalk/R.

```
1 deftype ThermometerT;
2 def sensor := makeReactive(Thermometer.new());
3 exportBehavior: sensor as: ThermometerT;
4
5 def allNearbySensors := ambientBehavior: ThermometerT @All;
```

Listing 3.12: Sharing objects over the network with AmbientTalk/R.

3.4.2. AmbientTalk/R

AmbientTalk/R is a reactive extension of AmbientTalk that combines the ambient-oriented features of AmbientTalk with functional reactive programming [27].

Methods and fields of AmbientTalk objects are used to create reactive dependencies between objects. For example, Listing 3.11 implements a reactive Celsius to Fahrenheit converter as a prototype object with a `convert` method. However, objects such as `Thermometer` and `CelsiusToFahrenheit` are not automatically reactive. AmbientTalk/R provides a `makeReactive` primitive that turns a non-reactive object into a reactive object. For example, line 7 clones the `Thermometer` object by invoking the `new` method which every object has, and turns the clone into a reactive object. Now, accessing the `temperature` getter on line 8 creates a dependency (just like in Functional Reactive Programming) between the `convert` method and `sensor`. Thus, whenever the sensor's `update` method is called, then the change to its temperature will automatically change the value of `result`.

AmbientTalk/R shares the capabilities of AmbientTalk to distribute objects over a network. In general, objects are published to the network via type tags, and other AmbientTalk/R applications can discover the objects via the same type. For example, Listing 3.12 defines a `ThermometerT` type on line 1, and shares the `sensor` object using `exportBehavior: as:`. The application can discover all objects shared using the `ThermometerT` type via `ambientBehavior:`, as demonstrated on line 5. The returned value that is stored in `allNearbySensors` is a collection of discovered sensors which can be used by the reactive program.

```
1 @peer type Client <: { type Tie <: Single[Server] }
2 @peer type Server <: { type Tie <: Single[Client] }
```

Listing 3.13: Defining a client-server architecture in ScalaLoci where each client is tied to a single server and vice-versa.

3.5. Multitier Reactive Programming

A fifth category listed in Table 3.1 (page 32) is multitier programming. *Multitier programming* is a programming paradigm where the multiple tiers of a distributed systems architecture (e.g., a 2-tier client-server architecture, 3-tier architecture, peer-to-peer, etc.) can be programmed using a single codebase in one programming language. Multitier programming was pioneered by languages such as Hop and HipHop [19, 129], and later adopted by the Functional Reactive Programming language ScalaLoci [152] and the Gavial [118] framework. We briefly discuss how ScalaLoci is used to develop distributed reactive programs, which we consider to be representative for the state of the art. In general, the concepts from ScalaLoci translate to Gavial as well.

3.5.1. ScalaLoci

ScalaLoci is a multitier and reactive domain-specific language in Scala. The main novelty compared to traditional multitier programming is that ScalaLoci can be used to define signals within specific tiers, and which can be used within other tiers as well. When the different application tiers are split and run on different hardware, then ScalaLoci automatically takes care of the communication between signals defined on one tier to the other tiers.

The architecture of a distributed program is explicitly defined in the program using *peers* and *ties*. A peer represents a single tier in the distributed system, e.g., a client or server tier. A tie specifies the relationship between tiers. For example, Listing 3.13 defines a client-server architecture where each client is tied to a *single* server, and each server is tied to a *single* client.

A variation of the client-server architecture is defined in Listing 3.14 where each client is still tied to a single server, but each server is tied to *multiple* clients. In general, ScalaLoci can be used to implement various architectures including a peer-to-peer architecture. For brevity we will limit our examples to using `Single` ties.

Consider the implementation of a Celsius to Fahrenheit converter in Listing 3.15. Lines 2 to 4 define the used tiers, in this case a `Thermometer` tier which will represent a thermometer, and a `Converter` tier which will convert the thermometer's measurements to Fahrenheit. Since the various tiers in the application share

```
1 @peer type Client <: { type Tie <: Single[Server] }
2 @peer type Server <: { type Tie <: Multiple[Client] }
```

Listing 3.14: Defining a client-server architecture in ScalaLoci where a server is tied to multiple clients.

a single codebase, both code and data are annotated with *placement types* that indicate which tier they are a part of. For example, the variable `celsius` on line 6 implements the current temperature of a thermometer as a reactive signal. Signals are created in ScalaLoci using `Var` that is given an initial value, in this case 0. This code has the placement type `on[Thermometer]` which indicates that the specified code block (between curly brackets) that contains the signal definition is executed on the `Thermometer` tier.

Similarly to `celsius`, the `fahrenheit` computation on line 7 is annotated with the placement type `on[Converter]`, which means that the provided code block is executed on the `Converter` tier. In this case it contains the definition of a new signal that converts the value of the `celsius` signal to Fahrenheit. Note the call to `celsius.asLocal()`: ScalaLoci makes the distribution explicit in the code, indicating that data will be moved from the `celsius` signal on the `Thermometer` tier to a local representation of the same signal on the `Converter` tier.

The `main` function on lines 11 to 15 is called when the program starts, and further highlights how code is split into different tiers. It consists of two parts which are executed on the `Thermometer` and `Converter` tier respectively. The thermometer will change the value of its `celsius` signal to a random number between 0 and 25, and the converter will observe any changes to its `fahrenheit` signal by printing those values to the `Converter` tier's console.

3.6. Reactive Wireless Sensor Networks

The final category listed in Table 3.1 (page 32) are (reactive) Wireless Sensor networks. *Wireless Sensor Networks* are networks that consist of low cost, low-power devices that collect data, often about the physical world (e.g., environmental conditions, volcano activity, etc). The devices can be deployed in large numbers such that they form a dense wireless network. Functional Reactive Programming has been applied to sensor networks via frameworks such as the Haskell-based FRP framework Flask [81]. We briefly explain how Flask is used to build distributed reactive programs for Wireless Sensor Networks.

One of the main contributions of Flask is how to reconcile existing work on FRP with the extremely limited resources of small devices (e.g., with only 10K of RAM). Essentially, Haskell is used as a meta-language to generate low-level object

```
@multitier object ThermometerService {
  @peer type Peer
  @peer type Thermometer <: Peer { type Tie <: Single[Converter] }
  @peer type Converter <: Peer { type Tie <: Single[Thermometer] }

  val celsius = on[Thermometer] { Var(0) }
  val fahrenheit = on[Converter] {
    Signal { (celsius.asLocal() * 9/5) + 32 }
  }

  def main(): Unit on Peer =
    (on[Thermometer] {
        celsius.set(scala.util.Random.nextInt(25))
      }) and
    (on[Converter] { fahrenheit.changed observe println })
}
```

Listing 3.15: A multitier Celsius to Fahrenheit converter in ScalaLoci.

code that can be executed on the sensors[4]. In this case the low-level code can be either nesC [54] (an extension of the C language for TinyOS) or Red, a language whose syntax is compatible with Haskell but with additional constraints that are suitable for embedded systems (e.g., no recursion or allocation of closures).

A Flask program consists of Haskell code that constructs a reactive program's DAG. The Haskell type system is used to ensure that the computations in the DAG are either nesC or Red, meaning that the reactive program DAG that is generated by the Haskell code can be executed on sensors.

While we will not go into the details, the type signatures of some of the main FRP functions used to construct a DAG are shown in Listing 3.16. The `map` function applies an object-level function to a signal of values. The first argument is the function to apply to the signal. The used type constructor `N` denotes the type of the object-level code, in this case a function that transforms values of type `a` into a value of type `b`. The 2nd argument of type `S a` denotes a signal that carries values of type `a`, and the return value of `map` is a new signal that carries values of type `b`. Similarly, the `filter` function accepts an object-level predicate and returns a new signal that only contains the values for which the predicate returns `True`. The `&&&` ("parallel") and `>>>` ("and then") operators are commonly found in other Haskell FRP frameworks such as Yampa [63]. In a nutshell, the `&&&` combinator combines the values of 2 signals in a signal that contains a tuple with both values, and the `>>>` combinator chains 2 functions that operate on signals.

[4]Note that Haskell code is not transpiled to low-level object code, but rather the Haskell code will generate low-level object code by carefully using the correct Flask library calls.

```
1 map :: N (a -> b) -> S a -> S b
2 filter :: N (a -> Bool) -> S a -> S a
3 &&& :: S a -> S b -> S (a, b)
4 >>> :: (S a -> S b) -> (S b -> S c) -> (S a -> S c)
```

Listing 3.16: Type signatures of Flask signal operators.

```
1 send :: FlowChannel -> S a -> S ()
2 recv :: FlowChannel -> S a
```

Listing 3.17: Type signatures of Flask's distribution operators.

Distributed programming is supported by Flask by allowing values to be broadcast wirelessly to the network via radio channels. It offers 2 built-in functions `send` and `recv` to send and receive data on a radio channel. The type signature of these functions are shown in Listing 3.17. The `send` function requires a channel and a signal, and will publish the signal's values to the network. In the other direction, the `recv` function turns a channel into a new signal to which the program can react.

3.7. Summary of Distributed Reactive Programming Approaches

In this chapter we outlined the state of the art reactive programming languages and frameworks to develop distributed reactive systems. As we have shown, the commonality between all of them is that they allow the reactive program's DAG to become distributed, such that data between 2 parts of the DAG can flow over a network. The discussed related work was categorised in Table 3.1 (page 32) according to the used techniques, which we briefly summarise.

Global Signal Registry A global signal registry is used by ActiveSheets and DREAM to export local signals to the network. Other programs connected to the same registry can lookup signals in the registry by name, and use them in their own programs as if they were defined locally.

Conflict-free Replicated Data Types To ensure that a distributed reactive program is resilient against network failures, RESCala uses CRDTs to handle the propagation of values over the network. Essentially a signal is exported to the network with a unique name, and other programs can lookup the signal using that name. Behind the scenes a CRDT is created on each side of the network. The properties of CRDTs ensure that when the CRDT on one side of the network is updated, then eventually the same CRDT on the other side of the network will be updated as well.

Actors We demonstrated Akka Streams, Creek and XFRP which build a reactive programming framework on top of actors. Since actors are already an often used programming model for distributed systems, these frameworks keep using actors as the basic unit of discoverability in the application. For Akka Streams this means relying on existing features to exchange actor references, e.g., using publish-subscribe. On the other hand, Creek directly integrated actor discovery with reactive streams by reifying the discovery and disconnection of actors as a data stream in the application, which can be mapped and filtered to obtain the desired actors (e.g., to use only thermometer actors). Finally, a different approach is offered by the functional reactive programming language XFRP. Here, the reactive program is specified in terms of nodes which perform a computation, and each node is compiled to an actor. Distribution is achieved via placement types which indicate on which (remote) host the compiled actors should run.

Ambient-oriented Programming Ambient-oriented programming is specifically designed to program applications for wireless ad-hoc networks. To build distributed reactive programs for such networks, the AmbientTalk/R reactive programming language provides a mechanism where a program's signals can be published to the network by using a type tag. It also provides primitives to discover the signals that are available on the network under a given type tag, which results in a collection of signals to which the reactive program can react.

Multitier Reactive Programming Multitier languages such as ScalaLoci and Gavial provide programmers with the ability to program multitier applications in a single language, using a single codebase. The language or framework takes care of splitting the code into the various tiers, which can be executed on different machines. In this case signals that are running on one tier can refer to the signals of another tier because they share the same codebase, and the language or framework takes care of managing distribution when the tiers are split.

Reactive Wireless Sensor Networks Functional Reactive Programming has been used by Flask to program sensors in a Wireless Sensor Network. These devices have extremely limited capabilities, and thus their networking is often limited to a simple radio. The signal discovery features of Flask reflect the limited capabilities of sensors, and manifest themselves as a simple `send` and `receive` where the values of a signal can be broadcast on a radio channel. Vice-versa, each radio channel can be reified as a signal in the reactive program to receive broadcasts from other devices.

Of the identified techniques, 4 are not suitable for developing distributed reactive programs for open networks. Namely, a global signal registry and CRDTs are unsuitable because the number of signals that are published to the network is fixed by the program code (i.e., not *open*). The broadcasting mechanism used by reactive Wireless Sensor Networks is too low level, and it requires a programmer

to build additional abstractions, e.g., at least to distinguish between multiple signals from similar devices (e.g., a multiplexer). Multitier reactive programming is not suited because it requires all tiers in a distributed system to be programmed using a single codebase, which is not the case for open networks.

Actors and ambient-oriented programming are potential candidates, since their discovery mechanisms can be used for open networks. However, signal discovery is only one aspect of programming open networks. Another aspect, which will be discussed in Chapter 4, is how the reactive program can correctly and efficiently manage all of the discovered signals (i.e., the Acquaintance Maintenance Problem of Section 4.3).

4. Problem Statement

In this chapter we analyse the problems that occur when a programmer uses existing reactive programming languages and frameworks to implement distributed reactive applications for open networks. Concretely we will study 3 problems: 2 problems occur when using a reactive programming language or framework "for distributed application development", and 1 problem occurs when doing so specifically "for open networks".

The first problem is called the **Reactive Thread Hijacking Problem** discussed in Section 4.1, where the data that enters a reactive program via the network can accidentally cause the program to be *no longer reactive*. This is problematic when interacting with devices on an open network because the various discovered devices are not necessarily part of the same codebase by the same developers (i.e., they cannot be controlled).

The second problem is called the **Reactive/Imperative Impedance Mismatch** discussed in Section 4.2, which encompasses the problems of combining imperative code with reactive code within the same application. Since both types of code are inevitable when programming distributed applications, we will study their combination in both directions, i.e., the embedding of reactive code within imperative code and vice-versa. Essentially, their combination is problematic because the semantics of the program become unclear and it becomes difficult to predict how the program will behave at run-time.

Finally, the third problem is called the **Acquaintance Maintenance Problem** discussed in Section 4.3, which occurs when writing reactive programs that react to an *open* number of data producers, which means that they are dynamic and cannot be determined beforehand. The current approaches to implement such programs either lead to code that exhibits accidental complexity and is error-prone, or the implementation is inefficient.

4.1. Reactive Thread Hijacking Problem

The word *reactive* in "reactive programming" denotes both the programming style (code) as well as the run-time behaviour of reactive programs. In this section we discuss the latter, where programs that are called *reactive* are expected to *react* in real-time to any changes that occur, i.e., they are expected to be responsive. To this end, the "Reactive Manifesto" highlighted the need to establish reliable upper bounds on program response times [23]. To better understand what it means for a program to be "responsive", we define different *levels* of reactivity that correspond to different kinds of "reliable upper bound" for a program's execution time.

4.1.1. Levels of Reactivity: Weak, Eventual, and Strong Reactivity

The core of the issue concerns program termination. More specifically the (non-)termination of an individual reaction of the reactive program. We identified 3 levels of reactivity that provide different termination guarantees: *weak* reactivity, *eventual* reactivity, and *strong* reactivity. We will discuss them in terms of the *reaction time* of a reactive program, i.e., the time it takes for a reactive program to react to an arbitrary input value.

Weak Reactivity: A programming language or framework is called *weakly reactive* when it provides no guarantees about the reaction time of an arbitrary program that is accepted as a valid program by the language or framework. Most reactive programming languages and frameworks are weakly reactive, because usually they allow applications to be programmed with the full power of a Turing-complete language.

Eventual Reactivity: A reactive programming language or framework is called *eventually reactive* when it can guarantee that any program accepted as a valid program by the language or framework has a finite reaction time. In other words, all reactions to any input value will eventually terminate.

Strong Reactivity: A reactive programming language or framework is called *strongly reactive* when it can guarantee that the reaction time of any program accepted as a valid program by the language or framework does not depend on the size of the input. In other words, the asymptotic worst-case reaction time is guaranteed to be in $\mathcal{O}(1)$.

A taxonomy of related work according to the different levels of reactivity will be provided in Chapter 7.

4.1.2. Hijacking Reactive Programs With Long Lasting Computations

Most reactive programming languages and frameworks are weakly reactive, and it is often easy to demonstrate a reactive program which (accidentally) becomes

```
1 val userInputSignal = Var("") // initial signal value is ""
2 val matches = Signal {
3   "(A+)*B".r.findFirstMatchIn(userInputSignal())
4 }
```

Listing 4.1: Matching strings to a regular expression in REScala.

no longer reactive for certain types of input. For example, consider the program in Listing 4.1 written in REScala [123], a state of the art FRP library for Scala, that checks whether user-provided input strings match a regular expression. Line 1 defines a new source called `userInputSignal` (initialized with the empty string), and line 2 derives a new signal called `matches`. Whenever the value of `userInputSignal` changes, the value of `matches` automatically reflects whether the string matches the given regular expression `(A+)*B` (e.g., `AB` and `AAAB`, but not `AAA`). The worst-case complexity of this program is $\mathcal{O}(2^n)$ with `n` being the size of the input string. Matching the string `AAAAA` fails after approximately 112 steps, and matching 50 `A`'s fails only after ∼3 quadrillion steps [115]. Thus, the program clearly cannot be called reactive with respect to *all* possible inputs. This is especially problematic when the inputs are provided externally and thus cannot be predicted, e.g., user input, or data from distributed services.

While the example may be a contrived case of catastrophic backtracking in regular expressions, a developer can easily and accidentally introduce computations into reactive programs that (occasionally) have unintended consequences for their reaction time. We call this problem the **Reactive Thread Hijacking Problem**, because long lasting computations (such as matching regular expressions) can completely "hijack" the thread of execution of a reactive program, thereby stopping the reactive program from being able to react to any new input.

4.1.3. Research Goal

While responsiveness is considered to be an important property of reactive systems, knowing that a program *should* be responsive is different from then also *ensuring* that it is responsive. Most existing reactive programming languages and frameworks sidestep the issue completely.

In summary:

Research Goal #1: Bounded-time Reactivity

The language design problem that needs to be solved is how to ensure that a "reliable upper bound" can be imposed on long lasting computations within reactive programs.

4.2. Reactive-Imperative Impedance Mismatch

Consider developing distributed reactive programs. It is inevitable that some parts of the reactive program will be responsible for handling traditional concerns of distributed programming such as networking. In the worst-case programmers resort to using low-level sockets, in the best case they have access to higher-level abstractions to perform network requests or pass messages. However, in all cases the act of sending data over a network is a side-effect executed via paradigmatically imperative code rather than reactive code. After all, IO is an effectful aspect of programs.

In analogy with the *Object-Relational Impedance Mismatch* for object-oriented programming [66], we identify the **Reactive-Imperative Impedance Mismatch** as the set of problems that occur when combining imperative code with reactive code. We discuss the problems of their combination in two directions. Namely the "embedding of imperative code within reactive code" (Section 4.2.1), and vice-versa, the "embedding of reactive code within imperative code" (Section 4.2.2).

4.2.1. Embedding Imperative Code in Reactive Code

Effectful computations are extremely tricky to understand and debug when they are embedded within the nodes of a reactive program's Directed Acyclic Graph (DAG) [42, 79], i.e., when imperative expressions are subexpressions of reactive expressions. In brief, this is because the update order of the DAG, and therefore the order of its side-effects, is *not* part of the semantics of a reactive program.

Reactive programming languages such as FrTime [31], Flapjax [88] and REScala [123] prevent glitches[1] by specifying that updates should be executed in a topological order of the DAG. Some implementations parallelise the execution of certain regions of the DAG, such as the conceptual propagation model of Elm [36] and a parallel version of the REScala update algorithm [40]. Streaming frameworks such as ReactiveX [113] and Akka Streams [119] do not feature such an algorithm, and instead only specify that parent nodes should be updated before their child nodes. Hence, a programmer cannot determine the order in which side-effects are executed by just reading the program's code, and they may cause additional issues such as race conditions [93] when the execution of the DAG is parallelised.

All of the aforementioned technologies allow multiple valid update orders to be used for a given program. This is important information for language implementers, because it gives them a lot of freedom to tweak and optimise (e.g., parallelise) how values propagate through the reactive program. However, for application developers this means that the concrete update order can vary across

[1] A glitch is a temporary inconsistency in the program, see Section 2.1.5 on page 16.

```
1 val counter = Var(0) // initial source value is 0
2 Signal { print("A" + counter() + " ") }
3 Signal { print("B" + counter() + " ") }
```

Listing 4.2: A REScala reactive program with side-effects.

different implementations or versions of the same language or framework. As we will demonstrate, the order can even change at run-time.

It is easy to demonstrate a reactive program where the **embedding of imperative code in reactive code** yields nondeterministic results when it is executed. Consider the reactive program (written in REScala) in Listing 4.2. Line 1 defines a new source called `counter`, and lines 2 and 3 define two signals that print either `A` or `B` to the console followed by the value of the counter. When this program is executed, the initial value of `counter` is propagated through the program, and "`A0 B0`" is printed to the console in the order of evaluation (from top to bottom). However, when the value of `counter` is updated to 1, approximately 50% of the time the program output is reversed and "`B1 A1`" is printed. Thus, effectful expressions leak information about the update order of subexpressions within reactive programs, and their correct execution may never rely on a specific order. Note that, in this case, the side-effecting `print` expressions are immediately obvious to a programmer, but in general side-effects can be hidden by multiple layers of abstraction (e.g., function or library calls).

The root of the problem are unconstrained side-effects within the reactive program. Not only can side-effects cause bugs that are difficult to find and reproduce because of an unlucky ordering in some propagations through the DAG, but they are also very difficult to coordinate and have a detrimental effect on behavioural composition [41]. Recognising these issues, most reactive programming languages and frameworks forbid side-effects within a reactive program, either through language design or via programmer guidelines (e.g., REScala guidelines [114]). However, as we will discuss next, they are rarely successful in banning side-effects completely.

4.2.2. Embedding Reactive Code in Imperative Code

Reactive programs do not exhibit the issues from the previous section when all embedded code is *purely* functional. However, we observe that this requirement is not met in practice. That is because reactive programs never exist in a vacuum, and parts of real-world programs naturally require side-effects, e.g., to provide input to the reactive program, to modify a GUI, or to push notifications to a user. Hence the **embedding of reactive code within imperative code** that performs those tasks is unavoidable.

Existing reactive languages and frameworks all tackle the problems caused by the embedding to some degree. We argue that their solutions either have limited applicability, or are ad hoc solutions with unclear semantics. What follows is a list of mechanisms that we identified in related work.

Built-in Primitives

Reactive programming languages and frameworks often incorporate built-in primitives to simplify the development of certain types of applications. For example, Flapjax [88] and Elm [36] provide a DSL for building GUIs where sources are automatically created and updated by GUI components, and the GUI automatically integrates with sinks of the reactive program. Similarly, the GUI library used by FrTime [31] consists of automatically generated wrappers (via macros) for Racket's object-oriented GUI toolkit [47, 65]. These wrappers automatically integrate with FrTime, and hide the (imperative) code to interact with an object-oriented GUI library. Such languages typically also feature built-in signals with narrow applicability, such as Elm's `Mouse.position` signal [36] that follows the coordinates of the cursor, and FrTime's `seconds` signal that follows Unix time.

Languages may also include special native functions or special forms that perform specific tasks. For example, Elm [36] is a *functional* reactive programming language built for programming web applications[2]. Because many parts of web applications require side-effects, Elm offers a built-in `syncGet` operation to execute synchronous HTTP requests (a side-effect), e.g., to fetch images.

Meta-constructs

Reactive programs always react to the values supplied to their sources, which originate from many different parts of the program such as the GUI, network sockets, or separate program threads. To bridge the gap between these parts and the DAG of the reactive program, reactive programming languages and frameworks often offer mechanisms to imperatively change the values of sources, or vice-versa, to perform an imperative action whenever a sink of the DAG changes. From the perspective of the reactive program code we call them meta-constructs, since this code is not part of the reactive program itself, and instead it is standard (non-reactive) code in the host language.

For example, FrTime [31] and REScala [123] offer built-in primitives to create and (destructively) modify the source of a reactive program. Their semantics is often unclear since these mechanisms are not part of the reactive programming

[2]We always refer to the originally published version of Elm described in [36]. As of Elm version 0.17 (released in 2016), Elm has replaced its functional reactive programming with an architectural pattern [35].

model, but are necessary to implement real-world programs. Streaming frameworks such as ReactiveX [113] and Akka Streams [119] include a wide range of built-in operators to transform the contents of data structures to a stream, and usually offer a special type of stream to which values can be imperatively pushed, e.g., a `Subject` in Rx.

(A)Synchronous Input/Output

A reactive runtime is responsible for propagating values through the reactive program (see Section 2.1.5 on page 17). Whenever the program thread that contains the reactive runtime is blocked, the reactive program (temporarily) stops reacting to any new input. Hence, reactive programming languages and frameworks often use multi-threading or asynchronous tasks to prevent blocking the reactive runtime, frequently to provide input values to the reactive program, or to process the output of the reactive program.

For example, FrTime [31] contains both asynchronous tasks and multi-threading. Asynchronous tasks are used in the implementation of FrTime to periodically update primitive signals such as `seconds`, which represents the current Unix time. An asynchronous task is scheduled by the implementation of FrTime to update its value every second. Multi-threading is used to ensure that the user does not block the reactive program via code that is entered into Racket's Read-Eval-Print Loop (REPL), which is used by FrTime developers to interact with a running FrTime program. A dedicated program thread manages the reactive runtime while the main program thread is responsible for the GUI and REPL. Developers may enter code into the REPL to imperatively change the values of source nodes of the DAG. From FrTime's implementation we distilled that the main thread asynchronously "sends" (via a message) these new input values to the reactive thread. Conversely, a programmer may monitor the value of a signal by entering its name in the REPL, which displays the value of the signal in the GUI. Behind the scenes a dependency is created from the REPL thread to the reactive program thread, such that the value of the specified signal is continuously "sent" to the REPL thread as it updates over time. However, multi-threading and concurrency are not part of the FrTime programming model.

The Elm [36] language incorporates asynchronous tasks as part of its execution model, and offers an `async` special form to compute blocking or long lasting computations asynchronously, e.g., to asynchronously fetch an image from a web address.

The REScala [123] library allows Scala code to modify the value of the sources of the DAG. In this case modifying sources is synchronous: the new value of a source is changed and immediately (synchronously) propagated through the DAG. While multi-threading is not part of REScala's semantics, from our experiments with the original variant of REScala we learned that multiple Scala threads may change

the same sources simultaneously. The implementation requires these threads to acquire a global lock to prevent race conditions. When multiple threads supply input values for sources, they may suffer from lock contention. A more recent implementation of REScala [40] changed these propagation semantics by allowing multiple updates to propagate through the reactive program concurrently as transactions.

4.2.3. Research Goal

The mixing of imperative and reactive code should not be an afterthought when designing a reactive programming language or framework. In one direction, the embedding of imperative code within reactive code can cause bugs that are difficult to find and reproduce because of an unlucky update order of the DAG, and side-effects within the reactive program have a detrimental effect on program composition. In the other direction, there is currently no semantically well-defined and sufficiently general mechanism to embed reactive code within imperative code *without* consequently also allowing imperative code to be embedded within reactive code.

In summary:

Research Goal #2: Reactive-Imperative Reconciliation

The language design problem that needs to be be solved is how to reconcile the reactive parts of reactive programs with the unavoidable imperative parts, without violating the invariants of the reactive code.

4.3. The Acquaintance Maintenance Problem

Remember from Section 1.1.3 that we used the term *prosumer* to denote a software component that both consumes and produces data, and that prosumers interact with each other via conceptual streams of data (e.g., sensor measurements). Furthermore, in Section 1.2.3 we introduced that every reactive program has to perform what we called *acquaintance management*, which is the combination of 2 mechanisms:

1. An *acquaintance discovery* mechanism to find acquaintances on the open network.
2. An *acquaintance maintenance* mechanism to subscribe to discovered streams in order to react as they appear, and to gracefully close the streams as they disappear.

We explore acquaintance management for reactive programs, and we will show that existing reactive programming languages and frameworks are either inef-

ficient, or require a complex and error-prone mix of code when implementing distributed reactive programs for open networks.

Throughout this section we use the running example of a reactive program that discovers thermometers and that computes their average temperature. We first discuss acquaintance discovery in Section 4.3.1 to find thermometers on the open network, and in Section 4.3.2 we discuss acquaintance maintenance, i.e., calculating their average temperature and maintaining the thermometers throughout the reactive program as thermometers appear and disappear.

4.3.1. Acquaintance Discovery: Extensional vs. Intensional

We classify acquaintance discovery mechanisms as either *extensional* or *intensional*. We use these terms in the traditional mathematical sense, namely that an *extensional definition* of a concept formulates its meaning by explicitly specifying every object that falls under the definition, and an *intensional definition* gives meaning to a concept by specifying the rules or conditions that objects have meet to be part of the definition. Applying these terms to acquaintance discovery:

Extensional acquaintance discovery: The developer *explicitly* specifies the acquaintances that are required by the program. This is demonstrated by the following example in RxJS, a JavaScript implementation of ReactiveX [113, 121], which explicitly names the URL to a specific thermometer and binds the resulting stream to `sensor1`.

```
const sensor1 = rxjs.webSocket("ws://sensor1.mysensors");
```

Intensional acquaintance discovery: The developer writes a *prescription* of the acquaintances that are required by the program. An example can be found in the AmbientTalk/R reactive programming language [27]. The following snippet discovers all thermometers that are accessible on the same (wireless) network. The expression `ambientBehavior:` creates a set of all objects on the network with the `ThermometerT` type tag. Its result is a signal that automatically updates whenever the underlying collection updates.

```
def allNearbySensors := ambientBehavior: ThermometerT @All;
```

When dealing with *open* networks, an intensional discovery mechanism is essential. By definition of an open network, the set of potential acquaintances is unknowable up front. Hence, extensionally enumerating all possible acquaintances is impossible. Because prosumers spontaneously come and go, the mechanism itself must also be reactive. However, existing reactive programming languages and frameworks such as Flapjax [88], REScala [123], Akka Streams [119], FS2 [51], Project Reactor [108], ReactiveX [113], and Streamz [135] only support extensional acquaintance discovery. The lack of built-in intensional acquaintance discovery

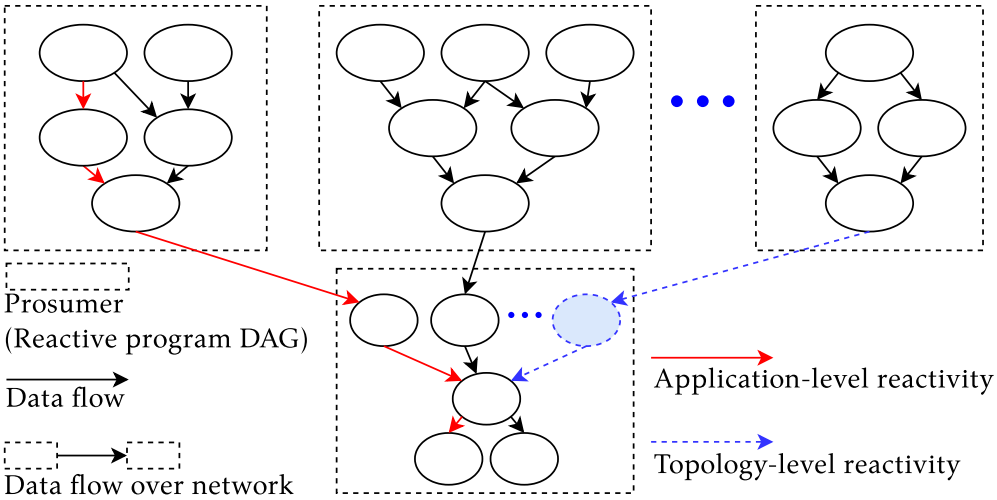


Figure 4.1.: Illustration of the 2 different levels of reactivity. Note that the distinction between producer and consumer is for clarity. In general both are prosumers.

mechanism leaves them in an unfavourable position, because there is no corresponding strategy to maintain the state of the reactive program as acquaintances appear and disappear.

4.3.2. Acquaintance Maintenance in Reactive Programs

An acquaintance maintenance is needed to correctly and efficiently *maintain* the state of the reactive program as acquaintances continuously appear and disappear. To show the different facets of doing so, we continue the example of computing the average temperature of a set of thermometers connected to a network, and we will assume an intensional acquaintance discovery mechanism that is capable of discovering them. Speaking in terms of streams, the `average` computation must appropriately react to streams appearing, disappearing, and updating with new values. In Figure 4.1 we illustrate these interactions between prosumers and their streams. Each of the dashed rectangles represents one prosumer whose reactive program is represented by a DAG. At the top, Figure 4.1 depicts `N` (possibly different) producers of data, e.g., thermometers. At the bottom, a sole consumer continuously reacts to data from the producers. We discriminate 2 levels of reactivity that constitute acquaintance maintenance.

Application-level reactivity: Whenever a source node of a DAG changes (i.e., at the top of the DAG), the reactive language or framework automatically recomputes the dependent parts of the program that are affected by the change. This is the "normal" propagation of values through the DAG as discussed in

Section 2.1.5 on page 16, e.g., an updated measurement of a thermometer causes the average to be updated as well. Figure 4.1 illustrates one such propagation path in red. We call this *application-level reactivity*.

Topology-level reactivity: A new kind of reactivity that we call *topology-level reactivity* occurs in consumers whose computations depend on an open number of producers, e.g., the `average` calculation. The topology of the DAG needs to be continuously reconfigured to accommodate the appearing or disappearing streams. This is illustrated in Figure 4.1 in blue, denoting the appearing and disappearing of a stream (and its dependencies) in the DAG of the consumer.

Application-level reactivity is well understood from existing reactive programming languages and frameworks, such as the ones introduced in Chapter 2 and Chapter 3. In the remainder of this section we analyse how such languages and frameworks handle topology-level reactivity. We will make the distinction between the 2 ways to represent a reactive value, namely as a (continuous) signal (see Section 2.1.1 on page 13) and as a stream (see Section 2.2 on page 18) Since they lead to a different programming style, we will discuss them separately.

Acquaintance Maintenance using Reactive Streams

The mainstream approach to writing reactive programs is based on **reactive streams**, where the basic unit of change are *events* (see Section 2.2 on page 18). Many reactive programming languages and frameworks offer abstractions that represent event streams [31, 43, 88, 123], and frameworks based on *reactive streams* (i.e., event streams) are often used in mainstream software development [113, 119].

Listing 4.3 implements the aforementioned computation that averages temperatures via an RxJS stream called `average$` (`$` is a naming convention for streams). To communicate the continuously appearing and disappearing sensors to the reactive program we used a stream called `sensorDiscove-ryService.sensors$` (not defined here) from which `average$` is derived. This stream propagates a new `sensor$` stream (containing temperature measurements) every time a sensor appears. In RxJS, the `average$` stream is defined by "piping" the values from `sensors$` through a sequence of RxJS stream operators. The operators in the example work as follows:

Line 2, `zipWithIndex`: Every value propagated by `sensors$` is a stream of temperature measurements that belongs to a particular thermometer. To be able to identify each thermometer downstream, line 2 adds an identifier to each thermometer's stream, yielding `[sensor$, id]` pairs where `sensor$` is the thermometer's stream.

Line 3, `flatMap`: The lambda given as argument to `flatMap` is invoked for every `[sensor$, id]` pair and generates a new stream. `flatMap` remembers all

```
const average$ = sensorDiscoveryService.sensors$.pipe(
  zipWithIndex,
  rxjs.flatMap(([sensor$, id]) => sensor$.pipe(
    rxjs.map((tmpr) => [id, tmpr]),
    rxjs.endWith([id, null]),
    rxjs.catchError(_ => rxjs.of([id, null])))),
  rxjs.scan((tracker, [id, tmpr]) =>
    tmpr === null ? tracker.remove(id) : tracker.update(id, tmpr),
    new AverageTracker()),
  rxjs.map(tracker => tracker.getAverage()));
```

Listing 4.3: Topology-level reactivity in RxJS, calculating an average of all sensors.

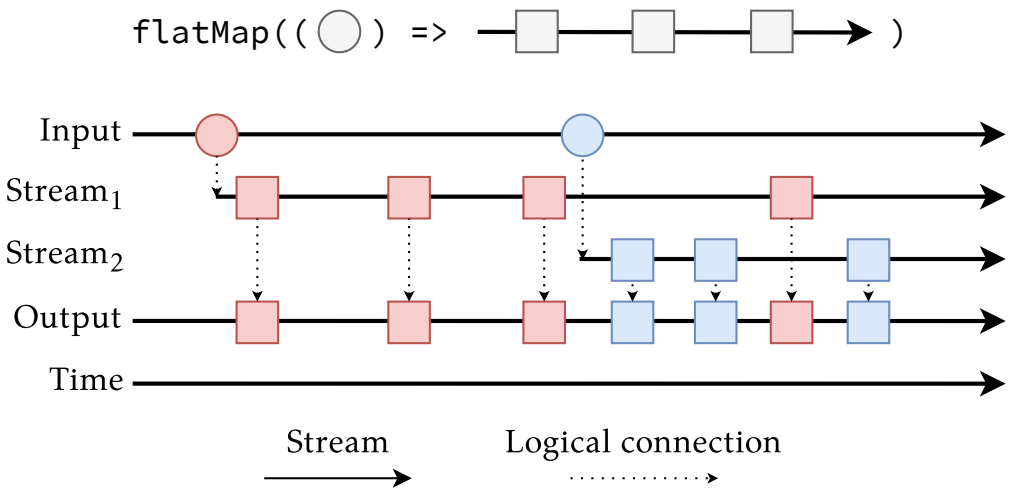


Figure 4.2.: Illustration of the `flatMap` operator in RxJS. The diagram is simplified for brevity (it does not consider higher-order streams, i.e., streams whose values are other streams).

streams it has generated and echoes their values on a first-in first-out basis. Since a textual explanation of `flatMap` is often difficult to comprehend, it is visualised in Figure 4.2 in terms of the various streams involved (time flows from left to right). At the top, we depict `flatMap` as an operation that transforms values of type "circle" to a stream of values of type "square". Whenever the input stream contains a red circle, then the lambda passed to `flatMap` generates a stream of red squares, which are echoed on the output stream. When the input stream contains a new value such as a blue circle, then `flatMap` remembers both the stream of red squares and blue squares, and echoes both of their values on its output stream.

We use `flatMap` to transform a stream of `[sensor$, id]` pairs to a stream of `[id, tmpr]` pairs where `id` is the same as before, and `tmpr` is its latest

temperature measurement. Whenever the `sensor$` stream is updated by said sensor, a new `[id, tmpr]` pair is propagated by `flatMap`. Lines 5 and 6 handle the removing of a thermometer's stream (gracefully or via an error). They ensure that an `[id, null]` pair is propagated to "clean up" the thermometer downstream.

Line 7, `scan`: Most of the application logic is tackled by `scan`, that essentially implements a `fold` operation for streams where the accumulator is emitted for every input value. The accumulator is a purely functional `AverageTracker` (implemented elsewhere) that tracks the latest temperatures of each sensor. The subsequent `map` (line 10) extracts the current average.

The main problem with this approach is its accidental complexity. More specifically:

- The code is error-prone. In our experience this problem translates to other frameworks like Akka Streams as well, which operate in a very similar manner. Besides the essential complexity of the application logic (averaging temperatures), the operations needed to handle topology-level reactivity to manage acquaintances (thermometers) are accidental complexity.
- Finding the correct combination of operators is difficult. The code snippet of Listing 4.3 was actually written by an anonymous reviewer of one of our papers about acquaintance management. The original code snippet written by us contained a semantic bug, because we thought of the solution in a different way (more similar to the solution based on signals which we show in the next section). The original code can still be found in Appendix A where we discuss the bug.

Acquaintance Maintenance using Signals

Functional Reactive Programming languages and frameworks use **signals** to represent time-varying values (see Section 2.1.1 on page 13). Those that support both signals and event streams (e.g., Fran [43], FrTime [31], Flapjax [88] and REScala [123]) also include operations to convert one to the other, and either one can be used to implement the other [30]. Whereas event streams use operations such as `map` and `filter`, signals are programmed by "lifting" regular functions, leading to code that is typically more declarative and compact.

Listing 4.4 implements the same `average` calculation from the previous section in REScala [90, 123], an FRP library in Scala. Line 1 declares a signal that holds values of type `Set[Sensor]`. Line 2 derives a new signal that computes their average value. The implementation is standard Scala: line 3 computes the sum total of all sensor values via a standard Scala `foldLeft`, and line 6 divides the total by the number of sensors. Note that each `.value` access automatically creates a dependency on a signal. In contrast to the RxJS code in Listing 4.3, the topology

```
1 val sensors: Signal[Set[Sensor]] = ... // omitted for brevity
2 val average: Signal[Int]  = Signal.dynamic {
3   val total = sensors.value.foldLeft(0) { (accum, sensor) =>
4     accum + sensor.measurement.value
5   }
6   total / sensors.value.size
7 }
```

Listing 4.4: Topology-level reactivity in REScala.

of the DAG that arises from this computation is automatically maintained by REScala, which correctly creates and removes dependencies as new sets of sensors are propagated.

Signal-based code is usually inefficient because of the propagation of (in this case) sets. Every time a change occurs at the topology-level (e.g., a new sensor appears) then a completely new `Set` is propagated through the reactive program, and the average temperature is computed from scratch via the `foldLeft`. Similarly, whenever a change occurs at the application-level (e.g., a new temperature measurement is produced) then this *also* causes the result to be computed from scratch, despite the majority of the work being identical (since only 1 value has changed). While in this case the computation is simple arithmetic, in general it can be much more complex and involve many prosumers.

Techniques such as *incremental data structures* [80] and *incremental behaviours* [116] have been proposed to speed up application-level reactivity. However, to the best of our knowledge these techniques have not been integrated to speed up topology-level reactivity.

4.3.3. Research Goal

Reactive programming languages and frameworks for open networks need to support all aspects of acquaintance management in a reactive way. This means support for a intensional acquaintance discovery, as well as acquaintance maintenance, which leads to topology-level reactivity. The **Acquaintance Maintenance Problem** can be summarised as follows.

Reactive Streams result in code that can react efficiently on both the application-level and the topology-level. However, event streams seem to engender larger and more complex code that is more difficult to write.

Signals result in code that is more idiomatic and seems to be easier to write and understand [124], but it is usually inefficient for both levels of reactivity.

In other words:

Research Goal #3: Acquaintance Maintenance

The language design problem that needs to be be solved is how to achieve correct acquaintance maintenance that is idiomatic (like signals) but also efficient.

4.4. Summary

In this chapter we have identified three problems that occur when programming distributed reactive programs for open networks.

First, the **Reactive Thread Hijacking Problem** in Section 4.1 occurs whenever input values of the reactive program (accidentally) cause long lasting computations that block the reactive program. This is particularly problematic when said inputs are unpredictable, e.g., provided by an external (distributed) program. We discussed various levels of reactivity, namely weak, eventual and strong reactivity, that each provide different guarantees with respect to the reaction time of a reactive program.

Second, the **Reactive-Imperative Impedance Mismatch** in Section 4.2 describes the problems that occur when combining imperative code and reactive code. This is especially pervasive in distributed reactive programs, because inevitably reactive code has to be coupled to imperative code to handle the traditional concerns of I/O (e.g., sockets). We discussed the embedding in two directions. In one direction, when embedding imperative code within reactive code, side-effects will expose the internal update order of the reactive program's DAG. In the other direction we discussed techniques to embed reactive code within imperative code, i.e., to couple imperative code to the sources and sinks of a reactive program.

Finally, the **Acquaintance Maintenance Problem** arises writing reactive programs for *open* networks. Neither of the 2 main approaches (based on reactive streams and signals) are suitable for performing acquaintance maintenance, which consists of application-level reactivity and topology-level reactivity. The existing approaches are either inefficient, or require a complex and error-prone mix of code.

5. Reactive Programming in Stella

This chapter tackles 2 of the 3 problems introduced in Chapter 4. Concretely we present a solution to the following 2 problems:

1. The *Reactive Thread Hijacking Problem* (Section 4.1 on page 50), where long lasting computations can completely hijack the thread of execution of a reactive program, thereby stopping the program from being able to react to any new inputs.
2. The *Reactive-Imperative Impedance Mismatch* (Section 4.2 on page 52), where the combination of imperative programming with reactive programming can cause issues for both paradigms.

We tackle these problems using a conceptual model that we call the *Actor-Reactor Model*. The model serves as the foundation of our solution, and will also serve as the basis for solving the 3rd and final problem in Chapter 6. We have implemented the Actor-Reactor Model in Stella, a new experimental programming language that serves as a linguistic vehicle to express the ideas developed in this dissertation. It is conceived as a continuation-passing style interpreter written in TypeScript with trampolines and (green) threads, programmed in the style of [50].

This chapter will serve as a reference for both Stella's programming model and the various language features that we used to build real applications using Stella.

5.1. Running Example: Bikey

Infrastructure for shared bicycles or electric scooters can be found in many cities around the world. These bikes or scooters are increasingly network-enabled to track their location, battery level, distance travelled, etc [69]. The application which we will implement is called "Bikey". A screenshot of Bikey is given in Figure 5.1, which provides an overview of all electric bikes in an area around the VUB's university campus in a web-based application. Since real-world data of this nature is difficult to come by, this implementation is controlled via buttons and context menus in the GUI. A user will mock data via the GUI by manually adding, removing, and moving bikes (via drag & drop). Note that, as we will show in Chapter 6, it is within Stella's capabilities to replace the GUI-based data entry of this application such that the data is served by a real distributed system (e.g., electric bikes).

Bikey is a prime example of a web-based distributed reactive program that necessarily includes both imperative and reactive code: Imperative code interacts with an ordinary, imperative web browser, and reactive code is responsible for correctly and timely computing metrics about bikes. More specifically, each bike continuously streams its current location to a server, and the server computes certain metrics for the user in real-time. A close-up of these metrics can be found in Figure 5.2. Based on the location of a bike, the reactive program computes the following metrics in real-time:

- Whenever a user rents a bike, its path is tracked from start to finish. Paths are visualised in Figure 5.1 via the thick coloured lines that track (moving) bikes.
- Users pay a fixed price of €1 to start a trip, followed by €0.25 per minute. The current price is shown in the popup of Figure 5.2.
- Under the price in the popup, users see for how long they have rented the bike and the total distance travelled.

After introducing the Actor-Reactor Model in Section 5.2, throughout the rest of this chapter we will explain Stella via code snippets that implement parts of the Bikey.

5.2. Stella Fundamentals: The Actor-Reactor Model

The Reactive Thread Hijacking Problem and the Reactive-Imperative Impedance Mismatch are caused by long lasting computations and side-effects respectively. Their problems will persist as long as they are allowed to be part of the reactive program. Hence the solution to both problems is similar, namely to completely disallow them to be a part of the reactive program. In other words, long lasting computations and side-effects should be evacuated into a *different* part of the

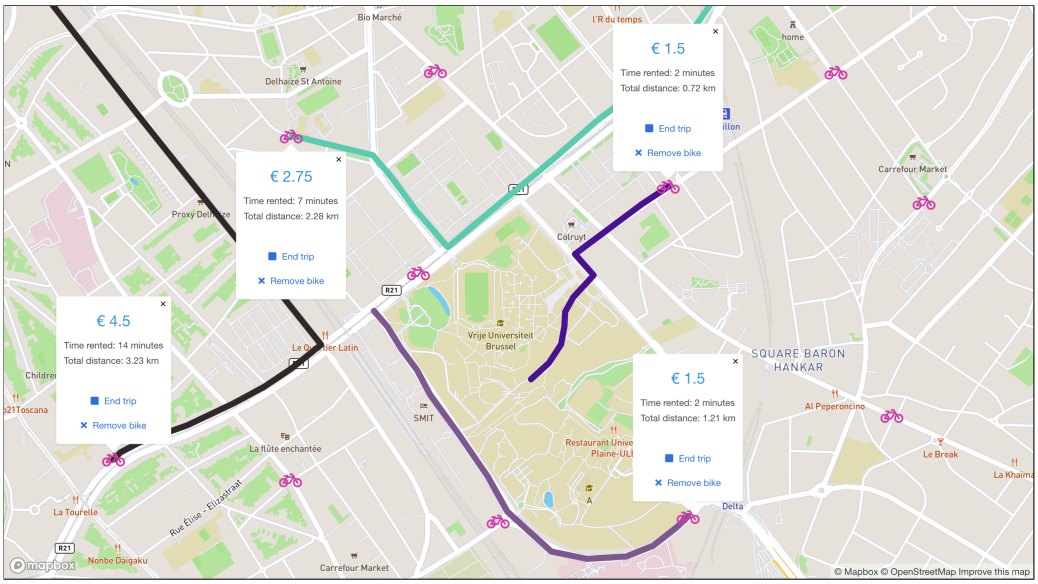


Figure 5.1.: Screenshot of Bikey (using mock bicycle data).

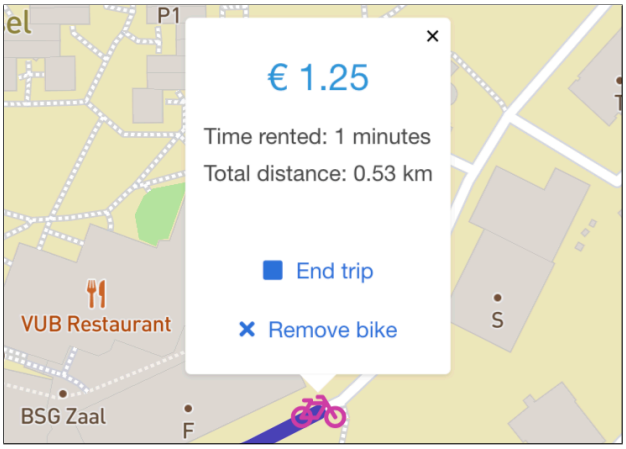


Figure 5.2.: Zoomed-in screenshot of Bikey, showing the metrics collected per bike. The pop-up reads "€1.25; Time rented: 1 minutes; Total distance: 0.53km".

program where they can no longer negatively affect the reactive program. We do so via a computational model called the *Actor-Reactor Model*.

In Section 5.2.1 we explain our so-called *actors* and *reactors*, and in Section 5.2.2 we explain the role of abstract data types to uphold their properties.

5.2.1. The Actor-Reactor Model

The Actor-Reactor Model is a programming model that treats *actors* and *reactors* as the basic primitives of concurrent "active" and "reactive" computations. It can be considered to be an extension of the traditional actor model for concurrent computations [70].

Actors are used to represent the **imperative** parts of reactive programs, i.e., they are programmed using imperative code. Since actors communicate solely via message passing, in response to a message they *can*:

- make nondeterministic local decisions on how to process the message,
- imperatively create ("spawn") other actors and reactors,
- imperatively send more messages,
- imperatively modify their own local state.

In the Actor-Reactor Model they are solely responsible for executing long lasting computations and side-effects.

Reactors are used to encapsulate **reactive** programs, i.e., they are programmed using purely functional and declarative code. Reactors also communicate solely via messages, and in response to a message they *will*:

1. deterministically process the message by propagating it through the reactive program[1],
2. declaratively send more messages.

Similar to other reactive programming languages, each reactor has its own Directed Acyclic Graph (DAG) and update thread to propagate values through the DAG. As we will discuss in Section 5.2.2, it is impossible for reactors to perform long lasting computations and side-effects.

Actors and reactors are the basic unit of composition between imperative and reactive code, which will communicate via the *streams* which they produce and consume. Streams are an abstraction on top of message passing where actors and reactors can subscribe to a particular "type" of message of another actor or reactor. They do so by subscribing to a stream that is exported by another (re)actor. Using streams, (re)actors can easily publish (i.e., "broadcast") certain types of messages to other (re)actors. Whereas actors support (1) the sending of point-to-point

[1] Since reactive programs can track state over time, functional purity is not enough to be deterministic. Thus, for reactors, deterministic processing of messages means that the same sequence of messages from the inception of the reactor will yield the same output.

messages, (2) broadcasting messages via streams, and (3) subscribing to streams to receive their messages, reactors only support (1) the receiving of messages, and (2) the automatic publishing of their computation results to their output stream.

5.2.2. Passing Application Data Between Actors and Reactors

Despite actors and reactors being programmed using paradigmatically different code, the messages that are passed between actors and reactors may contain arbitrary application data. To avoid code duplication, incompatibilities and inconsistencies between this data, actors and reactors share the same abstract data types. Concretely, in Stella these will be conceived as classes that are defined by an object-oriented base language.

Sharing abstract data types without restriction is potentially problematic, since the allowed sets of operations are different between actors and reactors. Whereas actors may use the full power of a Turing-complete language (unrestricted loops and side-effects), reactors are restricted to operations that are guaranteed to terminate eventually (see *eventual reactivity* in Section 4.1.1 on page 50) and that are free from side-effects. Hence, Stella's classes will feature two tiers of operations. The first tier are regular class methods which have no restrictions, i.e., they may contain side-effects and endless recursion. The second tier of operations are called *routines*, which are a novel kind of class method whose expressive power is restricted in a specific way. Actors will be able to call both methods and routines, whereas reactors can only invoke routines. To ensure reactors perform no side-effects and long lasting computations, routines enforce the following properties.

Properties of Routines

1. Routines have no side-effects. [Purity]
2. Routines always terminate. [Termination]
3. Routines can only invoke other routines. [Transitivity]

Routine Purity

Routine purity means that routines have no side-effects. In practice this means that any expressions with side-effects should be rejected by the interpreter or compiler. Since Stella is a dynamic language, its interpreter rejects expressions such as assignments and message sends in the body of routines. Furthermore, the operations of all native classes are defined as routines whenever possible.

Routine Termination

Various techniques and mechanisms exist to ensure that routines will always terminate. An overview of techniques used by related work will be given in Chapter 7. Since Stella is intended to be used for (distributed) web-based applications, its timing constraints are more relaxed compared to, e.g., real-time embedded systems. Hence, Stella's routines will still allow recursion as long as it is guaranteed to be finite.

We will enforce termination using a dynamic variant of **size-change termination (SCT)** [94]. This form of SCT checks at run-time whether the arguments of a (recursive) routine keep getting "smaller" in every step of the recursion. To check when a value is smaller than another, all Stella values can be ordered using a built-in routine. For example, numbers can be ordered via $|x| < |y|$, and lists can be ordered using the length of the list. A *size-change graph* is tracked at run-time to check how the arguments of a routine evolve over time. Before entering a new routine call, the so-called *size-change termination principle* is used to compare the argument values of the routine call to those of earlier calls to the same routine higher on the call stack. The routine call is permitted as long as (at least) one argument is decreasing in size over time. A run-time exception is thrown when all arguments remain the same or increase in size.

The size-change principle is a safe over-approximation of termination. In other words:

- Programs that satisfy the size-change principle will terminate. The authors of [94] developed a formal semantics where all programs terminate, and they formally prove that, if the size-change principle is satisfied, then the SCT algorithm does not change the result of the program compared to the same program without SCT checking.
- There are other programs that do not satisfy the size-change principle but which terminate. These programs cannot be correctly detected to terminate via size-change termination. Some of these programs can still be rewritten to aid the size-change principle (i.e., transform them to include decreasing arguments). For example, SCT has been used by its authors to implement several non-trivial programs [94]. The largest program is a 1100 lines of code R5RS Scheme interpreter that was used to run a Mergesort on a list of strings. Some of the other programs implemented by the authors could only be run with additional help for the size-change termination, e.g., by writing a custom ordering for data types, or by transforming conditional expressions to pattern matching.
- Some programs simply do not terminate.

Size-change termination errs on the side of caution and only accepts the first kind of programs, i.e., those which are known to terminate under the size-change principle. All other programs are rejected via a run-time error.

Routine Transitivity

The properties of routines are transitive. Once a routine is called by a regular method, then from that point onwards all invocations must also be invocations of routines. This property ensures that purity and termination are always upheld.

5.2.3. Using Actors and Reactors: Bikey Architecture

Every Stella application consists of actors and reactors that communicate via messages. As an example, Figure 5.3 depicts the architecture of Bikey. It shows the various types of actors and reactors involved in running the application, as well as the messages and streams that connect them.

Figure 5.3 depicts 3 actors called `Main`, `Bike` and `Time`, and a single reactor called `TripMonitor`. Note that we only draw a single `Bike` actor and `TripMonitor` reactor. In general there is one such (re)actor for each bike and bike trip in the application. While the implementation of these (re)actors will be explained later in this chapter, note how each of them interact via streams.

1. The `Main` actor is the main entrypoint of the program which processes messages such as `add-bike!` and `move-bike!` that are generated by the GUI (implemented in HTML + CSS + JavaScript). When processing such messages the `Main` actor may spawn other actors such as the `Bike` actor, e.g., when the user simulates the appearance of a bike through GUI actions.
2. A `Bike` actor represents a single bicycle which is continuously updating its location by emitting location messages via its `location` stream. In this case the location updates are caused by a user's GUI actions via the `Main` actor, but in general one can imagine these actors interfacing with a hardware positioning system (e.g., GPS) to autonomously update their location.
3. Whenever a bike's location updates, a "location update message" is immediately received by a `TripMonitor` reactor that is subscribed to a bike's `location` stream. The reactor processes messages from its input streams and reactively calculates application-level metrics such as the rental price and total distance travelled. Processing the messages causes the metrics to change, resulting in an output message that is broadcast and received by `Main`. Finally, the `Main` actor processes the output of `TripMonitor` by updating the GUI.

In the remainder of this chapter we introduce Stella by implementing the various components of Bikey, starting with the object-oriented base language.

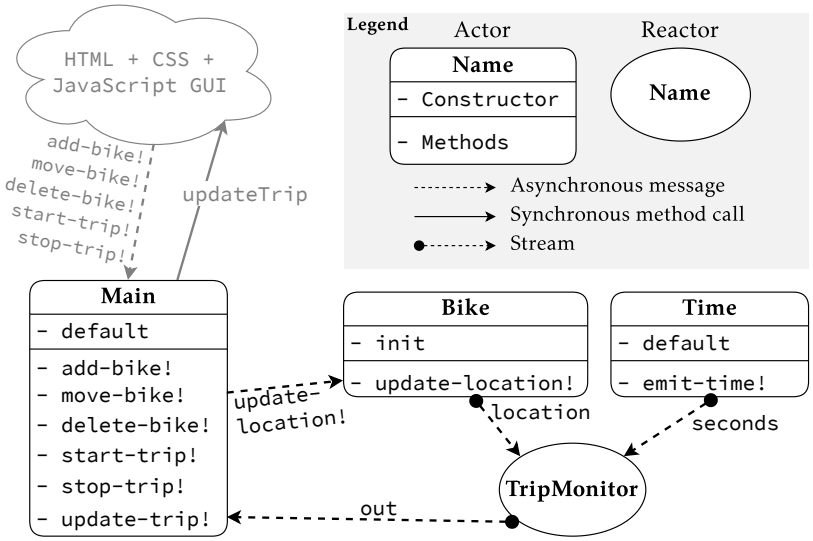


Figure 5.3.: Architecture of Bikey.

5.3. Sequential Object-Oriented Base Language

Stella is a class-based, dynamically typed language in which all run-time values are objects. It can be thought of as having two layers: a sequential object-oriented base language, and the concurrent level of actors and reactors that are programmed in terms of the base language. The sequential base language contains objects such as numbers, strings (e.g., `"hello"`) and symbols (e.g., `'hello`), as well as method invocations on objects and a number of special forms (e.g., to spawn actors and reactors). Keywords that denote objects will start with a #, such as the boolean values `#true` and `#false`. Throughout all Stella code snippets we will highlight keywords in bold, strings are surrounded by double quotes, and symbols start with a single quote and are italicised.

Stella uses S-expression syntax with operators in prefix notation. Dynamic method lookup in Stella's object-oriented model is single dispatch. For example, the expression `(println! "Hello World!")` looks up the `println!` method on the class `String` of the receiver object `"Hello World!"`. Similarly, the expression `(+ 1 2)` invokes `+` from the `Number` class on the receiver object, number 1, passing the number 2 as the argument. Some other examples from the base language are shown in Listing 5.1. Local variables are introduced via `def`, assignments use `set!`, and conditionals use `if`. Equality is implemented in the root class (`Object`): its method `eq?` tests for object reference equality. All values are `#true` except for `#false` and `#undefined`.

```
1 // (def <identifier> <expression>)
2 (def a 1)
3 // (set! <identifier> <expression>)
4 (set! a 2)
5 // (if <condition> <consequent> <alternative>)
6 (if (eq? a 1) (println! "yes!") (println! "no!"))
```

Listing 5.1: Examples of basic expressions in Stella

A Stella file may contain 4 types of top-level definitions: classes, *actor behaviours* ("the class of an actor"), *reactor behaviours* ("the class of a reactor"), and *flocks*. Classes are introduced in this section, actors in Section 5.4, and reactors in Section 5.5. Flocks will be introduced in Chapter 6 where we present Stella's features for distributed programming.

5.3.1. Classes

Classes are defined at the top level of a program file. For example, consider the `Pair` class defined in Listing 5.2 which can be used to represent linked lists. Local fields of the class are declared on line 2. Line 4 defines a constructor called `default` with 2 formal parameters called `initial-car` and `initial-cdr` that will initialize the fields of a new pair. Lines 8 and 9 define two routines called `first` and `second` with no arguments which are "getters" for the `car` and `cdr` field, and correspondingly, lines 10 and 11 define two methods `set-first!` and `set-second!` which are setters for these fields. Line 13 defines a routine called `length` that will compute the length of a linked list by calling `length` on the `cdr` field as long as it is also a pair[2].

Instantiating Classes

Classes are instantiated using the `new` special form. For example, the expression in Listing 5.3 instantiates `Pair` using its `default` constructor. The resulting object can be referenced via variable `p1`.

Note that there can be many constructors with different names, each with a different purpose. While multiple constructors allow for flexibility when instantiating a class, in practice we found that in many cases a class will have only one way to instantiate it, or that one way is used most of the time. Hence, Stella offers a variant of `new` called `newd` ("new default") that invokes the `default` constructor,

[2] Note that the `type-of` invocation in Listing 5.2 line 15 returns a symbol that represents the name of the class. This is because, unlike SmallTalk [56], classes are not reified as objects in our language. They cannot be referenced directly *except* via the `new` or `newd` special forms to create an instance.

```
(class Pair
  (def-fields car cdr)

  (def-constructor (default initial-car initial-cdr)
    (set! car initial-car)
    (set! cdr initial-cdr))

  (def-routine (first) car)
  (def-routine (second) cdr)
  (def-method (set-first! new-car) (set! car new-car))
  (def-method (set-second! new-cdr) (set! cdr new-cdr))

  (def-routine (length)
    (cond ((eq? cdr #undefined) 1)
          ((eq? (type-of cdr) 'Pair)  (+ 1 (length cdr)))
          (else 2))))
```

Listing 5.2: An implementation of a `Pair` class which demonstrates 2 different kinds of operations: methods and routines.

```
// (new <class name> <constructor name (symbol)> <...args>)
(def p1 (new Pair 'default 1 2))
```

Listing 5.3: Instantiating the `Pair` class.

which is called as such by convention. For example, the previous instantiation of `Pair` via its default constructor can be rewritten as:

```
(def p1 (newd Pair 1 2)) // (newd <class> <...args>)
```

When the programmer defines no constructors at all, the class compiler inserts a `default` constructor that does not initialise local fields (their value remains `#undefined`).

Enforcing Routine Properties

In Section 5.2.2 we introduced the properties of routine purity, termination, and transitivity. These properties are upheld by all `def-routine` definitions.

Routine purity: Routines that contain expressions with side-effects are rejected because of Stella's design. In our case they are `set!` and a couple of others[3]. A run-time error occurs when a routine calls a regular method, for example `println!` which prints to the console.

[3] Operations with side-effects end in a "!" by convention. They are: `set!`, `spawn-actor!`, `spawn-reactor!`, `send!`, `emit!`, `monitor!`, `react-to!`, `publish!` and `unpublish!`.

```
1 (def p1 (newd Pair 1 2))
2 (set-second! p1 p1)
3 (length p1) // successfully rejected via a run-time error
```

Listing 5.4: Creating a circular data structure with `Pair` of Listing 5.2.

Routine termination: The current implementation of Stella uses size-change termination [94] to enforce routine termination, which we briefly explained in Section 5.2.2. As an example, consider the excerpt of Stella code in Listing 5.4 that uses the `Pair` class of Listing 5.2. Here, a `Pair` is made to point to itself, creating a circular linked list. A subsequent call to `length` on the `Pair` gives rise to a recursive call to `length` on the same object. SCT rejects this call, since the argument values have not decreased since previous invocations (there are no arguments).

In general, the size-change property is checked for each routine at the level of the class, rather than at the level of individual objects. This is because we allow routines to create new objects, as long as the constructor only contains routine invocations and field initialisations (i.e., a very controlled form of `set!`). If the size-change principle were not checked at the class-level but at the object-level, then a routine could cause an infinite computation. For example, if the aforementioned `length` routine created a new pair and called `length` on this pair, which also creates a new pair that invokes `length` on that pair, and so forth.

Routine transitivity: Transitivity requires that routines can only invoke other routines. This is upheld by a run-time check, because Stella is a dynamic language.

Note that Stella's design is not coupled to any specific SCT algorithm, or even to SCT itself. Rather, we currently use it as a technique to enforce that routines will eventually terminate which works for high-level programming languages such as Stella, and unlike other approaches (such as specialised type systems), SCT requires no programmer assistance.

5.3.2. Built-in Classes

Since Stella is a research prototype, its standard library is limited and defined on a "by need" basis. The native classes include `Boolean`, `Date`, `Dictionary`, `Number`, `Vector`, `ImmutableVector`, `Number`, `Object`, `PriorityQueue`, `Random`, `Set`, `String`, and `Symbol`. We will show the interface of the `Dictionary`, `Vector` and `ImmutableVector` since they are the data structures that we use most often in this dissertation.

Name	Type	Signature	Description
Constructors			
default	Constructor	`() → Dictionary`	Creates an empty dictionary.
Methods and routines			
get	Routine	`Object → Object`	Given a key, retrieves its associated value (or `#undefined`).
put!	Method	`Object Object → ()`	Associates a key (1st argument) with a value (2nd argument).
remove!	Method	`Object → ()`	Removes a key.
contains?	Routine	`Object → Boolean`	Checks if the dictionary contains a key.
size	Routine	`() → Number`	Get the number of keys in the dictionary.
keys	Routine	`() → Vector`	Lists all keys in the dictionary.
values	Routine	`() → Vector`	Lists all values in the dictionary.
iterator	Routine	`() → Iterator`	Creates an iterator for the dictionary.

Table 5.1.: Interface of Stella's `Dictionary` class.

```
1 (def dict (newd Dictionary)) // create dict with its default constructor
2 (put! dict 'my-key 1) // insert 'my-key with value 1 into dictionary
3 (println! (get dict 'my-key)) // lookup the value of 'my-key, prints 1
4 (remove! dict 'my-key) // remove 'my-key from the dictionary
```

Listing 5.5: Examples of base language expressions to use a `Dictionary`.

Dictionaries

The interface of Stella's `Dictionary` class is given in Table 5.1. It contains a standard set of methods[4] to associate keys with values, to lookup keys, etc. Note that the type signatures of `get`, `put!`, `remove!` and `contains?` that take a key as argument only specify that the key has type `Object`. That is because any type of Stella object can be used as key. The equality of keys is based on strict "object reference equality", the same kind of equality that is used when comparing objects via the `eq?` method that is implemented by the `Object` class. Typically we will use symbols as keys, as two symbols that contain the same sequence of characters are always referentially equal.

An example of base language code that uses a dictionary is shown in Listing 5.5, where a key `'my-key` is added to the dictionary, its value is retrieved and printed, and then removed from the dictionary.

[4]Note that, since routines are a restricted version of methods, we include routines under the umbrella term of "methods" when talking about the interface of a class.

```
1 (def vec (newd Vector 1 2 3)) // create new vector with its default
      constructor, initialising it with values 1 2 3
2 (push! vec 4) // adds 4 to the end of the vector
3 (println! (get vec 0)) // prints 1, which is the value on index 0
4 (println! (last vec)) // prints 4, which is the last value of the vector
5 (pop! vec) // removes (and returns) the last element from the vector
```

Listing 5.6: Examples of base language expressions to use a `Vector`.

Vectors

Stella's `Vector` represents an indexed, growable array (of objects). It is the default data structure to store an ordered list or a collection of objects. The interface of `Vector` class is shown in Table 5.2. It contains methods to `get` and `put!` values from/on a specified index, as well as methods to manipulate the size of the vector by adding or removing values at both ends of the vector (the front and back)[5]. An example of using `Vector` can be found in Listing 5.6.

In addition to a mutable `Vector`, Stella contains an `ImmutableVector` that implements most of the same methods as `Vector`, but all of them are conceived as routines. Methods that modify the vector will return a new `ImmutableVector` instead. Offering both mutable and immutable versions of data structures occurs in other languages as well, e.g., in Scala [128]. The interface of `ImmutableVector` can be found in Appendix B (page 189).

5.3.3. JavaScript Foreign Function Interface

Stella has a practical foreign function interface that programmers can use to interact with JavaScript objects. E.g., in the architecture of Bikey in Figure 5.3 (page 72), GPS coordinates of bikes enter the Stella program from the HTML + CSS + JavaScript GUI. These GPS coordinates are JavaScript objects with a `lng` and `lat` field.

In general, JavaScript objects enter a Stella program in three ways:

1. When starting a program, via the `env` environment argument of the `Main` actor's constructor (shown later in Section 5.4). The contents of the `env` object is provided by the JavaScript code that starts the Stella interpreter.
2. JavaScript code sends an asynchronous message to a Stella actor that contains JavaScript objects. For example, JavaScript code can send a message to Stella's `Main` actor via a JavaScript function exposed by Stella's interpreter.

[5] An experienced JavaScript developer will notice that the interface of Vector is similar to JavaScript's `Array` [92]. This is not a coincidence, since Stella's `Vector` is implemented using JavaScript arrays. Since Stella runs in JavaScript, we have a lot of experience with the interface of JavaScript's `Array` and have adopted some of its features.

Name	Type	Signature	Description
Constructors			
default	Constructor	`Object…` → `Vector`	Creates a vector from the specified argument values.
from-vector	Constructor	`Vector` → `Vector`	Creates a (shallow) copy of the given argument vector.
Methods and routines			
get	Routine	`Number` → `Object`	Gets the value at the specified index.
put!	Method	`Number Object` → `()`	Replaces the value at the specified index.
push!	Method	`Object` → `()`	Adds a value at the end of the vector.
pop!	Method	`()` → `Object`	Removes and returns the last value.
shift!	Method	`()` → `Object`	Removes and returns the first value.
unshift!	Method	`Object` → `()`	Adds a value to the start.
delete!	Method	`Object` → `()`	Removes the specified value from the vector.
first, …, tenth	Routine	`()` → `Object`	Gets the n^{th} value (1st value, 2nd value, …).
penultimate, last	Routine	`()` → `Object`	Gets the penultimate or last value of the vector.
empty?	Routine	`()` → `Boolean`	Tests if the vector is empty.
append	Routine	`Vector` → `Vector`	Appends the current vector to the argument vector, and returns a new vector.
length	Routine	`()` → `Number`	Gets the length of the vector.
head	Routine	`()` → `Object`	Gets the first value.
tail	Routine	`()` → `Vector`	Returns a new vector with all values but the first.
fold	Routine	`Object Symbol` → `Object`	Invokes the given routine (2nd argument) on the accumulator (1st argument) with every value of the vector as argument, and returns the final accumulator.
iterator	Routine	`()` → `Iterator`	Creates an iterator for the vector.

Table 5.2.: Interface of Stella's `Vector` class.

```
1 {
2     value: 1,
3     fun: (x, y) => x + y
4 }
```

Listing 5.7: A JavaScript object with a `value` and `fun` field.

3. Stella code invokes a foreign function (e.g, passed via `env`) which returns a JavaScript object.

JavaScript objects are automatically wrapped as Stella objects when they enter a Stella program. More specifically, JavaScript's `number`, `boolean`, `string` and `undefined` map directly to Stella's `Number`, `Boolean`, `String` and `Undefined` classes, and JavaScript's `array` is mapped to Stella's `Vector`. All other JavaScript objects are wrapped with Stella's generic `JSObjectProxy` class.

Whereas Stella objects can only be interacted with by calling methods, JavaScript objects have multiple ways to interact with them, e.g., to get or set an object's fields. Stella's object wrapper needs to support these common interactions. Hence, 4 types of special methods are defined in `JSObjectProxy` to interact with the underlying JavaScript object, namely to test if a field is present, to get the value of a field, to set the value of a field, and to call a function stored in a field. We explain these interactions using the JavaScript object defined in Listing 5.7, which has a field called `value` (bound to number 1) and `fun` (bound to a JavaScript lambda).

Testers To test whether a field is present, `JSObjectProxy` accepts method calls of the form `has-*?` where `*` is the name of a field. For example, if `obj` denotes the wrapper for the JavaScript object in Listing 5.7, then executing `(has-value? obj)` in Stella will return `#true` since the `value` field is present in Listing 5.7. Since testing for field presence has no side-effects and always terminates, testing for field presence is a routine.

Getters To retrieve the value of a field, `JSObjectProxy` accepts method calls of the form `get-*` where `*` is the name of a field. The returned value is wrapped as a Stella object. For example, executing `(get-value obj)` in Stella on the wrapper for the object in Listing 5.7 will return the number 1 (of Stella's `Number` class). Since retrieving the value of a field has no side-effects, getters are routines.

Setters To set the value of a field, `JSObjectProxy` accepts method calls of the form `set-*!` where `*` is the name of a field. The assigned value is automatically unwrapped whenever possible, e.g., booleans, numbers, strings and vectors (arrays) are unwrapped to their JavaScript counterpart, as well as `JSObjectProxy` objects that were already JavaScript wrappers. Other Stella objects that do not have an immediate JavaScript counterpart are converted to a JavaScript object that only contains the object's fields and their (unwrapped) values. For

example, executing `(set-value! obj 2)` on the wrapper for the object of Listing 5.7 will assign the number `2` to its `value` field.

Callers To call a function on a JavaScript object, `JSObjectProxy` forwards any method calls that do not match the pattern of testers, getters and setters. Highlighting the fact that invoking JavaScript methods may cause side-effects, Stella's FFI adds an exclamation mark suffix to each method name[6]. For example, executing `(fun! obj 1 2)` on the wrapper for the object Listing 5.7 calls the JavaScript lambda `fun`. The arguments are automatically unwrapped before calling `fun` and the return value is automatically wrapped.

5.3.4. Tackled Research Goals

Stella's object-oriented base language fulfils the first research goal, namely to ensure bounded-time reactivity (see Section 4.1.3 on page 51). To this end, Stella's classes can define routines, which have no side-effects and are guaranteed to eventually terminate. While the base language itself is not yet reactive, Stella's reactors, which will be introduced in Section 5.5, will only be able to invoke routines on objects.

5.4. Actors and Streams

An actor is a process that has an *actor behaviour* and a mailbox [70] (a message queue). An actor behaviour describes the internal state and interface of an actor. Actors in Stella are based on the Active Objects model [21, 70, 154], where actor behaviours are defined similar to classes in object-oriented programming. In addition to receiving and processing messages, actors can be used to implement zero or more streams to which they can *emit* (publish) values. We explain how actor behaviours are defined, how streams are implemented using actors, and how actors monitor streams for changes.

5.4.1. Actor Behaviour

An actor behaviour is defined in the top-level scope. Similar to a class, it declares a number of local fields via `def-fields`, constructors via `def-constructor`, and methods via `def-method`. Additionally they may declare streams via `def-stream`.

Every Stella program must contain a `Main` actor behaviour that defines a constructor named `start`. This is exemplified by the "*Hello World!*" program in Listing 5.8. This `Main` actor behaviour declares no local fields, no streams, and no

[6] JavaScript functions cannot have an exclamation mark in their name, so it is safe for Stella to add it in the interface.

```
(def-actor Main
  (def-constructor (start env)
    (println! "Hello World!")))
```

Listing 5.8: A "Hello World!" program in Stella.

methods. The formal parameter of its `start` constructor called `env` is an object that contains environment variables passed from the JavaScript code that starts the Stella program. The body of the constructor is a single `println!` expression that prints `"Hello World"` to the console (remember that `println!` is invoked on the receiver object `"Hello World"` of type `String`).

5.4.2. Spawning Actors: "spawn-actor!"

Stella's actor model requires the explicit spawning (and killing) of actors. For this, Stella offers a `spawn-actor!` expression that spawns an actor from an actor behaviour, initialising the actor with one of its constructors. For example, in the running example, each bike is represented by an actor that has the `Bike` actor behaviour. A bike is spawned (e.g., in the `Main` actor) as follows.

```
// (spawn-actor! <identifier> <constructor (symbol)> <... args>)
(def bike (spawn-actor! Bike 'init gps-position))
```

The first argument of `spawn-actor` refers to the actor behaviour, and the second argument to the constructor of the actor. The remaining arguments correspond to the formal parameters expected by the constructor, which in this case is a variable `gps-position` which we assume to be in scope. Constructors are invoked in the process of the newly spawned actor. More specifically, the very first message inserted into the mailbox of the newly spawned actor will invoke the corresponding constructor. All subsequent messages will invoke methods defined in the actor behaviour via `def-method`.

Spawning an actor returns an `ActorReference` (ordinary) object that, as we will explain, can be used to refer to the actor's streams and to send asynchronous messages to the actor.

5.4.3. Emitting to Streams: "emit!"

As introduced in Section 5.2.1, every actor can implement data streams to broadcast values (via messages) to other actors and reactors. Streams are used to broadcast data that naturally evolves over time. For example, the running example contains an actor behaviour called `Bike` that represents a "digital twin" for every

```
1 (def-actor Bike
2   (def-stream location)
3
4   (def-constructor (init initial-location)
5     // (emit! <stream name> <values to emit>)
6     (emit! location initial-location))
7
8   (def-method (update-location! new-location)
9     (emit! location new-location)))
```

Listing 5.9: A simple "digital twin" for a bicycle in Bikey.

bicycle[7]. This actor behaviour is implemented in Listing 5.9. There are 3 declarations its body. Line 2 declares a stream called `location`, which means that all actors with the `Bike` behaviour have a stream called "location". Line 4 declares a constructor called `init` (there can be multiple), and line 8 declares a method called `update-location!`. Both the constructor and the method have 1 formal parameter and an `emit!` expression in their body that emits a location (a GPS coordinate object) to the `location` stream of the current actor. Every time an `emit!` expression is evaluated, a message will be broadcast to all other actors and reactors that are subscribed to the `location` stream of a particular `Bike` actor.

Note that emitting to streams exported by other actors (and reactors) is not possible thanks to the scoping rules of actor behaviours. The first argument of an `emit!` expression expects the name of a stream declared in the scope of the current actor behaviour. Furthermore it would be undesirable to do so, because then the actor that defines the stream has no control over the data emitted to its stream.

5.4.4. Qualification

To subscribe to streams exported by specific actors and reactors, (re)actors need a mechanism to refer to streams. Such a reference to a stream is obtained via a *qualification* expression that uses dot-notation. For example, given an actor `bike` (cf. Listing 5.9), then the expression `bike.location` designates an object of type `Stream`. Note that the dot-notation is exclusively used to address streams, and not fields, methods, etc.

[7] While `Bike` actors are conceptually running on the physical bikes that move around a city, in our implementation they are spawned locally in response to GUI events. As we will show in Chapter 6, Stella fully supports distributing actors such as `Bike` over a network.

5.4.5. Stream Arity

Streams in Stella have an *arity* that specifies the number of values that must be emitted to the stream in one emission. The previous example of the `Bike` actor behaviour (Listing 5.9) declared a stream of arity 1. For example, `Bike` declared a `location` stream as follows:

```
(def-stream location)
```

A `def-stream` expression takes an optional 2nd argument to denote its arity. The previous declaration is equivalent to:

```
(def-stream location 1)
```

An arity higher than 1 is used to emit multiple values that must change simultaneously. For example, a part of Bikey involves aggregating the path that a user has cycled and to calculate the total distance travelled. This path and its corresponding distance are computed by a reactor that emits both values simultaneously, i.e., on a stream with arity 2. Otherwise, if these values would be emitted independently, a consumer of the stream might first update the application with an extended path, and only after some time with the updated length of the path. After the first update the application would be in an inconsistent state, similar to a glitch in reactive programming (glitches are introduced in Section 2.1.5 on page 16). Furthermore, we will use stream arity to facilitate the composition of actors and reactors.

5.4.6. Monitoring a Stream: "monitor!"

Actors subscribe to streams by "monitoring" them for changes. Each `monitor!` expression that they execute establishes a subscription to a stream, meaning that any values emitted to the stream (i.e., a broadcasted message) will be received by the monitoring actor. As a concrete example, consider the `Main` actor behaviour in Listing 5.10 which continuously prints the current time to the console, as produced by some `Time` actor that exports a stream called `seconds` which tracks Unix time. On line 5 the `Main` actor subscribes to the `time.seconds` stream. Upon subscribing, the publisher immediately sends the latest value that was published to the stream, which enters the mailbox of the subscriber as a `print-time!` message. All future emissions of the `time` actor to its `seconds` stream are also received as `print-time!` messages. The number of formal parameters of the `print-time!` method must be equal to the arity of the monitored stream (in this case 1). `monitor!` returns a handle that can be used to cancel the subscription.

```
1 (def-actor Main
2   (def-constructor (start env)
3     (def time (spawn-actor! Time 'default))
4     // (monitor! <stream> <selector>)
5     (def handle (monitor! time.seconds 'print-time!)))
6
7   (def-method (print-time! current-time)
8     (println! current-time)))
```

Listing 5.10: Monitoring a stream.

5.4.7. Asynchronous Message Passing: "send-after!" and "send!"

Stella's actors communicate by sending each other asynchronous messages. The basic primitive for sending messages is an operation called `send-after!` that schedules a messages to be sent after a specified delay.

A `send-after!` expression can be used to send a message immediately by specifying the delay to be 0. Since this is often the standard way to send a message, Stella offers the a `send!` operation with an automatic delay of 0.

To demonstrate both forms of message sending in a practical example, consider the `Time` actor behaviour defined in Listing 5.11 that implements a stream that tracks Unix time. We use a `Time` actor in Bikey to track the total duration of a bicycle trip (cf. Bikey's architecture in Figure 5.3 on page 72). It exports a stream called `seconds`, has a constructor called `default`, and a method called `emit-time!`. In the body of its constructor, on line 6 a `Time` actor sends an asynchronous "`emit-time!`" message to itself via an immediate `send!`. We use the special keyword `#self` to denote a reference to the current actor (an object of type `ActorReference`). The corresponding `emit-time!` method is invoked when the message is dequeued from the mailbox.

The body of the `emit-time!` method creates a new `Date` object via its default constructor (line 9), which defaults to the current time. The following lines convert the current Unix time from milliseconds to seconds and emit it to the `seconds` stream. Finally, to emit the next time update, on line 15 the actor sends a recursive message to itself after 1000 milliseconds have passed. This message is only inserted into the receiver's mailbox after the specified time has passed.

5.5. Reactors: Functional Reactive Programming

Besides actors, Stella features *reactors*. A reactor is a process that encapsulates a functional reactive program that is programmed using functional reactive code rather than sequential, imperative code. By encapsulating imperative code in actors and reactive code in reactors, Stella avoids the issues that arise when

```
(def-actor Time
  (def-stream seconds)

  (def-constructor (default)
    // (send! <target> <selector> <...args>)
    (send! #self 'emit-time!))

  (def-method (emit-time!)
    (def time (newd Date))
    (def unix-time-ms (get-time time))
    (def unix-time-seconds (round (/ unix-time-ms 1000)))
    (emit! seconds unix-time-seconds)

    // (send-after! <target> <milliseconds> <selector> <...args>)
    (send-after! #self 1000 'emit-time!)))
```

Listing 5.11: The `Time` actor behaviour emits the Unix time.

combining imperative and reactive programming. Since reactors can only invoke routines on objects, it is guaranteed that their computations have no side-effects and that they terminate (eventually). Furthermore, since a reactor is a separate process, long lasting computations in other processes cannot accidentally block a reactor.

In this section we explain the various constituents of reactors. We first explain *reactor behaviours* (Section 5.5.1) and how to turn them into a reactor (Section 5.5.2). Then we further detail the relation between reactor behaviours and reactors by introducing the intermediary concept of *reactor deployments* (Section 5.5.4). The remainder of this section explains the various operations of reactors that we used to implement the running example, namely qualifications within reactors, sampling, and state accumulation.

5.5.1. Reactor Behaviours

Just like an actor is spawned from an actor behaviour, every reactor is spawned from a *reactor behaviour* that describes the reactor's program logic. The general structure of a reactor behaviour is shown in Figure 5.4, where we annotated the different parts of a `Plus` reactor behaviour that adds two numbers. Every reactor behaviour follows the same structure.

1. It has a unique name, e.g. `Plus`.
2. Its inputs (formal parameters) are declared after the name, e.g. `x` and `y`.
3. Its body may contain variable definitions (e.g., `result`), routine invocations (e.g., `+`), conditionals (via `if`), and reactor-specific expressions (e.g., the aforementioned qualification, sampling, and state accumulation).

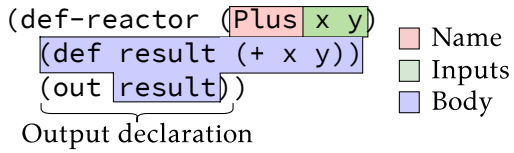


Figure 5.4.: Structure of a reactor behaviour.

```
1 (def-reactor (PriceCalc start-fee minutes cost-per-minute)
2   (out (+ start-fee (* minutes cost-per-minute))))
```

Listing 5.12: Calculating the price of a bike rental in Bikey

4. Its last line of code is always an output declaration given by `out` that denotes the outputs of the reactor behaviour, e.g., in Figure 5.4 the local variable `result` is the single output value.

An example of a reactor behaviour used in Bikey can be found in Listing 5.12, which defines `PriceCalc` that calculates the price of a bicycle trip. Its inputs are `start-fee`, `minutes` and `cost-per-minute`. In this case there are no local variable definitions, and the price calculation occurs immediately as part of the `out` declaration.

Reactor behaviours are written in a programming style that resembles Functional Reactive Programming (see Section 2.1 on page 13). Input variables such as `start-fee` and `minutes` behave similarly to signals (time-varying values) in other FRP languages. Similar to spreadsheet formulas, new signals are derived by invoking routines such as `+` and `*` on existing signals. The value of such a derived signal is the result of invoking the routine on the values of the argument signals. At run-time, whenever the value of any argument signal changes, e.g., for the `*` invocation they are `minutes` *or* `cost-per-minute`, then all dependent signals are automatically updated as well.

As is typical for FRP languages, the program logic of a reactor behaviour can be visually represented by a Directed Acyclic Graph (DAG) that is derived from the program text. For example, the corresponding DAG representation of `PriceCalc` is given in Figure 5.5, which shows how data flows through the reactive program. Due to the strong correspondence between the code and the DAG, we often call the inputs the "source nodes" ("sources" for short), the outputs are the "sink nodes" ("sinks" for short), and all other nodes are called internal nodes. For now we will explain reactors and their operations without considering the DAG, and in Section 5.6 we will revisit various reactor operations and explain them in terms of their DAG representation.

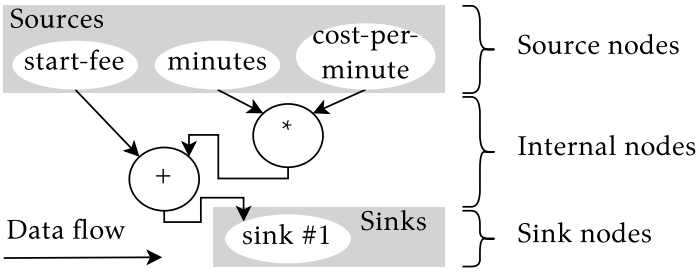


Figure 5.5.: DAG representation of `PriceCalc` (Listing 5.12).

5.5.2. Spawning Reactors: "spawn-reactor!"

Since spawning a reactor is in fact an effectful action, actors are responsible for spawning reactors. They do so via a `spawn-reactor!` expression that returns a reference (of type `ReactorReference`) to the spawned reactor. For example, an actor spawns a reactor from the aforementioned `PriceCalc` reactor behaviour (cf. Listing 5.12) as follows:

```
(def r (spawn-reactor! PriceCalc))
```

After spawning, a reactor perpetually waits for messages that change the value of its input signals, after which these values propagate through the reactor and cause newly computed values to be emitted by the reactor (the reactor's output). We will call the complete processing of a single message a *turn* of the reactor, i.e., the complete propagation of values from the input signals to the output signal.

5.5.3. Sending Values to Reactors: "react-to!"

The output of reactors is updated whenever the value of one or more of its input signals has changed. New values for input signals can be provided by other (re)actors in three ways:

1. **Direct propagation:** An actor sends a `'react-to!` message to the reactor to change the value of its input signals. For example, an actor can send such a message to the `PriceCalc` reactor `r` as follows:

   ```
   // start fee: 1 EUR, trip duration: 5 min., price per min.: 0.25 EUR
   (send! r 'react-to! 1 5 0.25)
   ```

   This `'react-to!` message is enqueued in the mailbox of the reactor. When the reactor processes this message it changes the values of its inputs to 1, 5 and 0.25 respectively, which also cause any derived values to be updated as well (in this case the price of a trip). Since sending a `'react-to!` message is a very common operation, Stella provides a built-in `react-to!` operation that is functionally equivalent:

```
(react-to! r 1 5 0.25)
```

Note that not all input signals are required to change on every `react-to!` message. A programmer may send the value `#undefined` for one or more inputs to signify that the value of the input signal should not change (i.e., it remains the same as the old value).

2. **Stream propagation:** Whenever an argument of `react-to!` is an object of type `Stream`, then the receiving reactor will subscribe to the stream and react to the stream's values. In other words, rather than changing the value of the corresponding input signal to the `Stream` object, the input signal will instead change to the values emitted to the stream by a different (re)actor.
3. **Qualification:** As we will explain later in this section, using a qualification expression in the body of a reactor will establish a subscription to the given stream, rather than propagating the `Stream` object.

We call the complete processing of a message from any of these sources a *turn* of the reactor, after which the output of the reactor is updated. This output is only accessible as a stream called `out` that every reactor exports. A reference to this stream is obtained via a regular qualification expression, e.g., `r.out`. Other (re)actors can subscribe to this stream via the means that we already explained, for example actors can use `monitor!` (see Section 5.4.6), and reactors use the aforementioned "stream propagation" or "qualification".

5.5.4. Reactor Deployments

Traditionally, the DAG of a reactive program (e.g., Figure 5.5 on page 87) is used (internally) to store the "current state". For example, consider a node with 2 dependencies such as the `*` node in Figure 5.5. Whenever the value of one of the dependencies changes (e.g., the `minutes` source), then the `*` routine is reapplied using the most recently computed value of the other input signals that did not change (e.g., `cost-per-minute`).

Stella's reactive programs consists of multiple *reusable* reactor behaviours. More concretely, a single reactor behaviour can be used to to spawn multiple reactors each with their own independent internal state and computations. Furthermore, as we will show, each reactor behaviour can also use other reactor behaviours to perform their computations. Hence Stella requires an intermediate abstraction to capture the run-time state of a particular "instance" of a reactor behaviour. We call this intermediate abstraction a *reactor deployment*.

A reactor deployment is an "instance" of a reactor behaviour that stores its run-time state, e.g., the most recent values that were propagated and dependencies on streams. A reactor deployment cannot be referred to directly (i.e., it cannot

"escape" a reactor), and there can be many deployments of reactor behaviours within the same reactor.

The Root Deployments

Every reactor behaviour that is used within the program logic of some reactor must be deployed, i.e. "instantiated", before it can be used by a reactor. The first deployment is created when the reactor is spawned. For example, following expression will spawn a reactor `r` that initially contains exactly 1 reactor deployment, namely the deployment of `PriceCalc`.

```
(def r (spawn-reactor! PriceCalc))
```

We call this the *root deployment* of that particular reactor since it lies at the "root" of (potentially) any other reactor deployments within the same reactor. Compared to the (potential) other reactor deployments it receives special treatment, namely:

- Any `'react-to!` message sent to the reactor changes the values of the root deployment's input signals.
- The output stream of a reactor (accessed via `r.out`) emits the values of the root deployment's output signals (the values of the signals in the `out` declaration of a reactor behaviour).

5.5.5. Creating Additional Deployments: "deploy"

Similar to how, in a functional programming language, functions call other functions, Stella's reactor behaviours can "call" other reactor behaviours. Reactor behaviours can use a `deploy` expression which can be seen as a function application for reactor behaviours, but instead of applying a function once, it deploys a reactor behaviour that (in this case) remains active throughout the lifetime of the reactor[8]. At run-time this means that one reactor deployment (e.g., the root deployment) gives rise to other reactor deployments.

As an example, consider the `DistanceBetween` reactor behaviour in Listing 5.13 that calculates the great-circle distance between two points using the Haversine formula [45]. It is used by Bikey to calculate the total distance travelled by a bike, which is a metric shown in the GUI. Most expressions in its body are variable definitions and routine invocations on objects. Since the formula requires some angles to be specified in radians, `DistanceBetween` makes use of the `DegreesToRadians` behaviour defined on line 16 by deploying an instance of it. Responsible for this is the `deploy` expression, for example on lines 7 and 8. Whenever the value of one of the given argument signals changes, this new value will propagate through

[8] Chapter 6 will define a `deploy-*` operation where reactor deployments spontaneously appear and disappear.

```
1  (def-reactor (DistanceBetween point1 point2)
2    (def lat1 (get-lat point1))
3    (def lon1 (get-lng point1))
4    (def lat2 (get-lat point2))
5    (def lon2 (get-lng point2))
6
7    (def dlat (deploy DegreesToRadians (- lat2 lat1)))
8    (def dlon (deploy DegreesToRadians (- lon2 lon1)))
9    (def a (+ (expt (sin (/ dlat 2)) 2)
10             (* (cos (deploy DegreesToRadians lat1))
11                (cos (deploy DegreesToRadians lat2))
12                (expt (sin (/ dlon 2)) 2))))
13   (def c (* 2 (atan2 (sqrt a) (sqrt (- 1 a)))))
14   (out (* 6367 c)))
15
16 (def-reactor (DegreesToRadians degrees)
17   (out (* degrees (/ #Pi 180))))
```

Listing 5.13: Deploying additional reactor behaviours.

the corresponding deployment of `DegreesToRadians`. Similarly, the value of the output signal of `DegreesToRadians` will propagate back to the deployment of `DistanceBetween`.

Note the difference between a reactor that contains multiple reactor deployments, and multiple reactors that are connected via streams. All propagation of values between multiple deployments in a single reactor occurs synchronously during the processing of a single message of the reactor's mailbox. On the other hand, multiple reactors that are connected via streams only communicate asynchronously via messages. When a reactor emits its output values to its `out` stream, a message is broadcast to all subscribers (e.g., other reactors) that receive that message in their mailbox. Eventually they will process this message by finding the reactor deployment that is subscribed to the stream, and changing the values of its input signals according to the contents of the message. Changing those signals thus causes the synchronous propagation of values within the reactor potentially across various reactor deployments, eventually leading to an update of the output stream.

5.5.6. Qualification Within Reactors

A crucial difference between a routine invocation (e.g., `get-lat` and `get-lng` in Listing 5.13) and a reactor deployment is that a routine invocation is (re)computed whenever the values of the input signals change (i.e., input → output), whereas a reactor deployment can produce new output values even when its apparent input signals did *not* change. This is because each reactor deployment can subscribe

```
1 (def-reactor (AccumulatePath bike)
2  (def path (deploy Accumulate bike.location (newd ImmutableVector) 'push))
3  (def distance-delta
4    (if (< (length path) 2)
5        0
6        (deploy DistanceBetween (penultimate path) (last path))))
7  (def exact-distance (deploy Accumulate distance-delta 0 '+))
8  (out path (round exact-distance 2)))
```

Listing 5.14: Accumulating the path of a bike trip in Bikey.

to streams by using qualification expressions, and they will update their output signals whenever the referred to streams emit new values.

As an example, consider the `AccumulatePath` reactor behaviour in Listing 5.14 that accumulates the path travelled by a rented bike as an immutable list of coordinates. A bike's path and the distance spanned by the path is visualised in Bikey's GUI (see Figure 5.1 on page 67). The only input signal of `AccumulatePath` is a reference to a bike, i.e., a reference to an actor with the `Bike` actor behaviour (from Listing 5.9 on page 82).

Line 2 of Listing 5.14 accumulates every location update in an `ImmutableVector`. In other words, an object that represents a GPS location will be added to the vector whenever the location of a bike updates, i.e., when its `location` stream emits a new value. The accumulation itself is implemented by the `Accumulate` reactor behaviour which we will show later (Section 5.5.9). Its first argument is a qualification expression that will supply location updates, the second argument an initial accumulator, and the third argument a method to invoke on the accumulator with a new location update as argument, yielding an updated accumulator. The qualification expression automatically subscribes to the referenced `location` stream, i.e., the value of `Accumulate`'s first input signal will change to the values emitted by the stream. Hence, whenever the location of a bike updates, the updated location automatically propagates through the deployment of `Accumulate`.

Note that the expression `bike.location` is not eternally tied to a specific actor and stream. In general, whenever the value of the `bike` signal is updated to be a different actor, then the reactor will stop propagating values from the old bike's stream (i.e., it unsubscribes from the stream), and start propagation values by the new bike's stream (i.e., it subscribes to the stream). In this specific application the value of `bike` is not changed because doing so is not part of the application logic. Regardless, allowing the actor or reactor reference to change at run-time suits the semantics of signals such as `bike` whose value can naturally change over time.

The expressions in the rest of the body of `AccumulatePath` in Listing 5.14 implement the logic to calculate the total distance spanned by the accumulated

```
1 (def-reactor (TripMonitor id bike time start-fee cost-per-minute)
2   (def time-elapsed (ceiling (deploy TripTime time.seconds)))
3   (def-values (path distance) (deploy AccumulatePath bike))
4   (def cost (deploy PriceCalc start-fee time-elapsed cost-per-minute))
5   (out id time-elapsed path distance cost))
```

Listing 5.15: The `TripMonitor` reactor behaviour collects all metrics of a trip: its duration, path, distance, and cost.

path. In short, for each update of `path`, lines 3 to 6 calculate the real-world distance between the two most recent additions to the vector (which are situated at the rear), or returns 0 if the vector is too small. The total distance travelled is accumulated on line 7, and rounded to 2 decimal places on line 8. Both the path and the rounded distance are declared as the output of `AccumulatePath`.

5.5.7. Receiving Multiple Output Values: "def-values"

Just like many programming languages offer mechanisms to return multiple values from a function, it is often the case that a reactor has multiple declared outputs. For example, the `AccumulatePath` reactor behaviour in Listing 5.14 outputs both the path and its real-world distance. Multiple output values can also arise when using a qualification expression, namely whenever the referenced stream has an arity larger than 1 (see Section 5.4.5), meaning that at least 2 values are emitted to the stream as a single message. Up until now we have not shown how to distinguish multiple output values.

The mechanism to capture (i.e., name) multiple output values is relatively simple. Besides a regular `def` expression to define a single local variable, reactor behaviours can use a `def-values` expression to define multiple variables at once. The number of variables must correspond exactly to the number of output values. An example can be found in Listing 5.15 which implements the `TripMonitor` reactor behaviour. Besides being the central reactor behaviour responsible for computing and outputting all metrics related to a trip[9], it shows how reactor behaviours can incorporate deployments with multiple outputs.

`TripMonitor` in Listing 5.15 has 5 input signals. Their expected values are, from left to right:

1. A unique identifier for the trip (a symbol).
2. A bike (a reference to a `Bike` actor from Listing 5.9 on page 82).
3. An actor that tracks the current time (a reference to a `Time` actor from Listing 5.11 on page 85).
4. The starting fee of renting a bike (€1).

[9]The architectural overview of Bikey can be found in Figure 5.3 on page 72.

```
1 (def-reactor (TimeElapsed current-time)
2   (def time-start (sample-once current-time))
3   (out (- current-time time-start)))
```

Listing 5.16: The `TimeElapsed` reactor behaviour tracks elapsed time.

5. The cost per minute of renting a bike (€0.25).

The body of `TripMonitor` deploys various other behaviours that implement the required calculations[10]. Among these, it deploys `AccumulatePath` on line 3 which had declared 2 outputs. We used the `deploy` expression in combination with a `def-values` expression, such that the 2 output signals can be referred to as `path` and `distance` respectively.

5.5.8. Sampling Signals: "sample-once" and "sample"

In digital signal processing, *sampling* is the reduction of a continuous electrical signal (e.g., a sound wave) to a sequence of samples, where each sample measures the amplitude or value of the electrical signal at a specified point in time as defined by the sampling rate (the rate at which values are measured). Similarly, in Functional Reactive Programming, sampling is the act of measuring the value of an FRP signal at the rate defined by another FRP signal.

Sampling a Signal Once

The simplest form of sampling is the one required by the running example. Here, the price to rent a bike is calculated based on a fixed cost of €1 followed by €0.25 per minute. To track the duration of a trip, a reactor needs to store the starting time and to calculate the difference between the start time and the current time. Defining a signal that perpetually holds the starting time requires us to sample the time signal once at the start of a trip.

Stella provides a `sample-once` expression that can be used in reactor behaviours. An example from Bikey is given in Listing 5.16, which implements the `TimeElapsed` reactor behaviour that will track how long a user has rented a bike. Its single input signal is `current-time`, whose values we expect to be the Unix time in seconds. The body contains a single signal definition of `time-start` that will sample the value of the given signal `current-time` once. In practice this will be whenever `TimeElapsed` is deployed, i.e., whenever the first value for `current-time` propagates through the reactor deployment of `TimeElapsed`. The value of `start-time` will be perpetually equal to this sampled value. Speaking in

[10] The implementation of the used reactor behaviours `AccumulatePath` and `PriceCalc` have been shown earlier in this section, and the implementation of `TripTime` will be shown in Section 5.6.5.

reactor turn	0	1	2	3
`current-time`	1636716691	1636716692	1636716693	1636716694
`start-time`	1636716691	1636716691	1636716691	1636716691

Table 5.3.: Example of the signal `current-time` being sampled once.

reactor turn	0	1	2	3	4
`s`	`'a`	`'b`	`'c`	`'c`	`'d`
`r`	0	0	1	2	3
`sampled-s`	`'a`	`'a`	`'c`	`'c`	`'d`

Table 5.4.: Example of a signal `s` being sampled at the rate of signal `r`.

terms of the turns of a reactor (i.e., the complete processing of a set of input signal updates), then the values of `current-time` and `start-time` evolve according to Table 5.3. Whereas the value of `current-time` is updated to the next Unix time every turn, the value of `start-time` does not change.

Sampling a Signal Multiple Times

In general, a signal `s` can be sampled at any rate as given by the rate at which another signal `r` changes. For example, consider the following `sample` expression that can be used in reactor behaviours.

```
(def sampled-s (sample s r))
```

The signal `sampled-s` carries the values of `s`, but only at the times that `r` changes. An example is given in Table 5.4 where we show the output of `sampled-s` whenever the value of `s` and `r` changes in a turn of the reactor. Here, signal `s` contains symbols of a single character, and `s`'s value is updated at reactor turns 0, 1, 2 and 4. Similarly, `r` is a signal that contains the natural numbers, which are updated in turns 0, 2, 3, and 4. The output signal `sampled-s` will carry the value of `s` whenever the signal `r` changes its value, i.e., at reactor turns 0, 2, 3 and 4. Note how the value of `sampled-s` does not change at time 1 because the value of `r` did not change. Also note that `sampled-s`'s value does not change at time 3 despite `r`'s value changing from 2 to 3, because the value of `s` did not change between time 2 and 3.

5.5.9. State Accumulation: "pre"

Statefulness is an important aspect of Bikey and reactive programs in general. For example, in an earlier listing (Listing 5.14 on page 91) we used an `Accumulate` reactor behaviour to accumulate the path and total distance travelled by a user.

reactor turn	0	1	2	3	4
`s`	`'a`	`'b`	`'c`	`'d`	`'e`
`(pre s)`	`#undefined`	`'a`	`'b`	`'c`	`'d`
`(pre s 'x)`	`'x`	`'a`	`'b`	`'c`	`'d`

Table 5.5.: Example use of `pre` within a reactor.

```
1 (def-reactor (Accumulate val initial-acc accumulate-routine)
2   (def acc
3     (if (eq? val (pre val))
4         (pre acc initial-acc)
5         (apply* accumulate-routine (pre acc initial-acc) val)))
6   (out acc))
```

Listing 5.17: Accumulating reactor behaviour.

This `Accumulate` reactor behaviour is not the most primitive form of state accumulation. It is implemented just like any other reactor behaviour, but using a more fundamental operation to track local state.

State is often introduced into a reactive program by using an operation that stores the "old" value of a signal, i.e., the value a signal had in the previous turn [127, 132, 142, 149, 150]. This is a relatively simple mechanism that allows a programmer to use the "old state" of a signal in computations, and to update this value with new information for the next turn. We define a special form called `pre` that captures this previous value of a signal.

The evolution of a signal's value using `pre` is exemplified in Table 5.5, which shows how the value of signals that use `pre` will evolve over time. Here, the value of signal `s` changes to a different symbol in each turn of the reactor. The signal defined by `(pre s)` echoes these values, but they are delayed by 1 turn. Note that the initial value of `(pre s)` remains `#undefined`. An optional 2nd argument of `pre` replaces the initial `#undefined` with a different value, e.g., `(pre s 'x)` replaces the initial value at reactor turn 0 with the symbol `'x`.

The implementation of the `Accumulate` reactor behaviour uses `pre` to store the accumulator. Its implementation is given in Listing 5.17. We define 3 input signals: `val` is the signal whose values are accumulated, `initial-acc` is the initial value of the accumulator, and `accumulate-routine` is the name (a symbol) of a routine to invoke on the accumulator to update its state. The accumulating variable is called `acc` (line 2), and is defined as follows:

- If the current value of the input signal `val` is equal to its value in the previous turn, then the accumulator `acc` is not updated (it keeps its old value). This

is to prevent the accumulator from updating every time the reactor activates, even when the value of `val` did not change[11].

- If the current value of `val` is *not* equal to its value in the previous turn, then the value of `acc` is updated by applying the `accumulate-routine` to the previous value of the accumulator and the new value. Note that `apply*` is a native method defined in the `Symbol` class. It calls itself (the value of `accumulate-routine`) on the object provided as its second argument (the previous accumulator) with the remaining arguments. E.g., `(apply* '+ 0 1)` calls `'+` on the object 0 with 1 as argument.

The value of `acc` is continuously updated and "passed" to the next turn of the reactor, and thus we are able to use it to accumulate changes over time.

5.5.10. Tackled Research Goals

Stella's implementation of the Actor-Reactor Model fulfils the second research goal from Section 4.2.3 (page 56), namely to reconcile imperative and reactive code. The Reactive-Imperative Impedance Mismatch is avoided by construction, by enforcing that all imperative code must be contained within actors, and all reactive code is contained within reactors. Speaking in terms of the embedding of imperative and reactive code:

Embedding imperative code in reactive code: Both side-effects and infinite loops cannot affect reactors. First, reactors can only invoke routines on objects, and the object-oriented base language enforces that routines do not contain side-effects and that they must eventually terminate. Second, side-effects from actors cannot affect reactors either since they are isolated to the actor's process: Any messages sent between (re)actors are passed by (deep) copy.

Embedding reactive code in imperative code: There is only 1 way for imperative code to interact with reactive code, namely via message passing. The semantics of message passing is well defined, namely: Input enters the reactor via messages that are inserted in the mailbox of a reactor are processed on a first-in first-out basis. The values emitted by reactors are sent onwards via messages that enter the mailbox of the receiving (re)actor.

5.6. Reactors as DAGs

In the previous sections we briefly mentioned that Stella's reactor behaviours are internally represented as a Directed Acyclic Graph (DAG). A DAG is not just an implementation technique to make reactive programs work, and that

[11] We have investigated other mechanisms to perform state accumulation that do not require such an `if`-test to eliminate erroneous updates. A more general method of tracking state is called a "trampoline variable" [99], but this method is currently not implemented in Stella.

```
1 (def-reactor (ToMinutes time-seconds)
2   (out (/ time-seconds 60)))
```

Listing 5.18: The `ToMinutes` reactor behaviour converts a time in seconds to minutes.

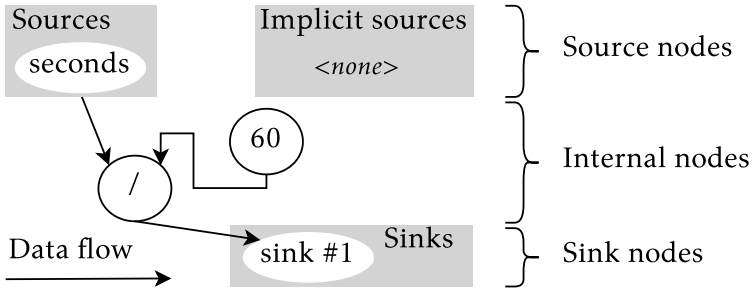


Figure 5.6.: The DAG representation of `ToMinutes` (Listing 5.18) shows all parts of our DAG visualisation.

programmers can also gain a better understanding of how a reactive program works by keeping in mind its DAG representation.

All operations of reactor behaviours discussed in the previous sections have a DAG representation that reveals additional info about how they work. In this section we discuss the operations which we think benefit the most from revealing their structure in the DAG, and finally we introduce a DAG *point-free* composition operation that follows directly from viewing reactor behaviours as DAGs.

5.6.1. DAG Representation of Reactor Behaviours

Every reactor behaviour can be drawn as a DAG. In Section 5.5.1 we previously showed the `PriceCalc` reactor behaviour of Listing 5.12 (page 86) and its DAG representation in Figure 5.5. Another example used in Bikey is the `ToMinutes` reactor behaviour in Listing 5.18 which converts a Unix timestamp in seconds to minutes. Its corresponding DAG representation is shown in Figure 5.6.

Every reactor behaviour's DAG consists of a set of (explicit) sources and *implicit* sources, which together make up the source nodes. The (explicit) sources correspond to the input signals of the reactor behaviour. The implicit sources are a new type of source node that represent implicit data inputs of the reactor behaviour. Essentially they represent signals that may cause the output to change, but which are not explicitly listed as input signals in the code of the reactor behaviour. Note that the DAG of `ToMinutes` has no implicit sources, since its body only contains a simple routine invocation. In general, as we will show when discussing the DAGs of `deploy` and qualification, implicit sources are automatically generated when using expressions that add an additional data source (as in "new input stream")

```
1 (def-reactor (Add a b)
2   (out (+ a b)))
3
4 (def-reactor (Increment x)
5   (out (deploy Add x 1)))
```

Listing 5.19: The `Add` and `Increment` reactor behaviours.

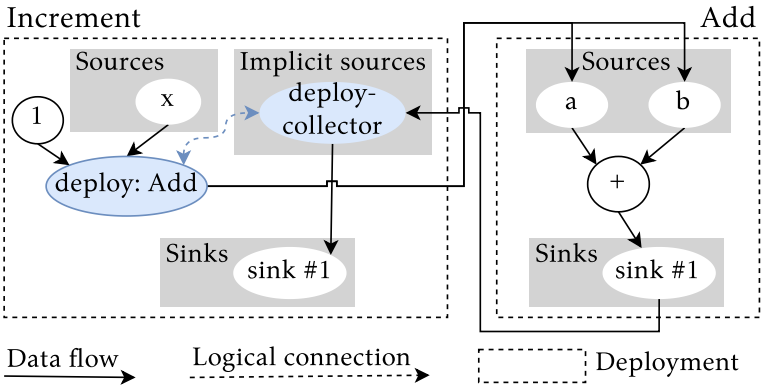


Figure 5.7.: DAG representation of `Increment` (Listing 5.19).

to the reactor behaviour. We will normally omit the implicit sources from the DAG visualisation when there are none. The internal nodes comprise the various operations within the DAG such as the routine invocation `/` (division). Finally, we draw a sink node for each entry in the `out` declaration of the reactor behaviour, in this case 1.

5.6.2. DAG Representation of Deploy

In Section 5.5.4 we discussed the `deploy` expression that allows a reactor behaviour to use other reactor behaviours a number of times. We used the example of a `DistanceBetween` reactor behaviour (Listing 5.13 on page 90) that deployed multiple "instances" of a `DegreesToRadians` reactor behaviour. Since the DAG of `DistanceBetween` is too big to visualise here, we will use the exemplary reactor behaviours `Add` and `Increment` defined in Listing 5.19. `Add` simply adds two numbers, and `Increment` deploys `Add` to increment a given number. While these reactor behaviours have a simple DAG visualisation, the technique we use to represent a `deploy` expression is the same for all reactor behaviours, including `DistanceBetween`.

The DAG representation of `Add` and `Increment` is shown in Figure 5.7. Rather than visualising their DAGs in isolation, we visualise two reactor deployments

using a rectangle with dashed border, one for `Increment` and one for `Add`. Notice how `Increment`'s `deploy` expression is compiled to two DAG nodes (highlighted in blue) that are logically connected.

1. An internal `deploy` node is connected to the arguments of the `deploy` expression, in this case source node `x` and constant node `1`.
2. An implicit source node is connected to the sink of the `Add` deployment. Its name "`deploy-collector`" is solely for the visualisation, and is not derived from code.

The internal `deploy` node is responsible for managing its corresponding deployment of `Add`. Whenever the value of the argument signals change – in this case `x` – then these changes propagate to the deployment of `Add`, i.e., they change the value of the input signals. In the other direction, whenever the value of the sink node of `Add` changes, the new value propagates to the implicit source node followed by any further dependents that use the result of the `deploy` expression (in this case, just `Increment`'s sink node).

Compiling a `deploy` expression to 2 DAG nodes was inspired by the `async` statement in Elm [36], and we found it to be a common pattern that we used for other statements as well, such as a qualification expression.

5.6.3. DAG Representation of Qualification

In Section 5.5.6 explained how a qualification expression is used within a reactor. Rather than propagating an object of type `Stream`, the reactor will create a dependency on the referred stream, and propagate the stream's values instead. To exemplify this we explained the `AccumulatePath` reactor behaviour (Listing 5.14 on page 91).

The DAG representation of `AccumulatePath` is given in Figure 5.8, which shows how a qualification expression is compiled to a DAG. For completeness we draw the entire DAG with all its internal nodes. We highlighted the nodes of the qualification in blue. Similar to `deploy`, a qualification expression is compiled to 2 nodes. The internal node called `bike.location` manages the dependency on the stream, i.e., establishing or removing the subscription, and ensuring that the stream's values change the value of the desired implicit source node. The implicit source node called `location` receives the values emitted by the stream. In this case the qualification expression refers to the `location` stream of a particular `bike` actor.

Representing a qualification expression via 2 DAG nodes highlights the subtle complexity of the operation. Namely, that there are potentially 2 axes of change that should be correctly reacted to by a reactor. First, any updates emitted to the `location` stream will change the value of the `location` implicit source node. Second, while it does not occur in this specific application due to its logic, reactors

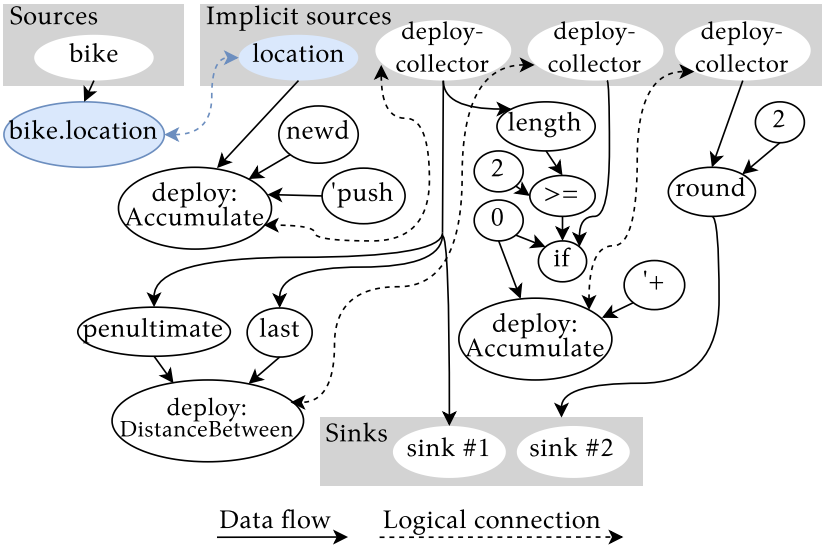


Figure 5.8.: DAG representation of `AccumulatePath` (Listing 5.14).

also handle the situation where the reference to the `bike` actor changes to a different actor. This means that the reactor should stop propagating values from the old bike's stream, and start propagating values from the new bike's stream.

5.6.4. DAG Representation of If

The control structure supported by reactors is an `if` expression. Up until now we used `if` expressions as if they were a normal expression, but it is important to know exactly how an `if` expression operates to know which signals are updated and when.

We will explain the `if` expression using the artificial `SwitchBetween` reactor behaviour in Listing 5.20. Depending on the current value of the condition, the result of the `if` expression will either be the value of the "`a`" signal or the "`b`" signal. The DAG representation of `SwitchBetween` is given in Figure 5.9, where the `if` node depends on the nodes that correspond with its condition, consequent and alternative. Depending on the value of the condition, the `if` node propagates either the latest value of the consequent or the alternative. Note that when the expressions used in the consequent or alternative branch are more complicated (e.g., nested routine invocations, stateful expressions, `deploy` expressions, qualifications, etc.), these expressions remain "active" even when their values are (temporarily) not used as the result of `if`.

```
1 (def-reactor (SwitchBetween condition a b)
2   (out (if condition
3            a // consequent
4            b))) // alternative
```

Listing 5.20: Using `if` in a reactor behaviour.

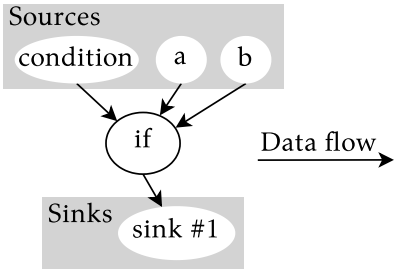


Figure 5.9.: DAG representation of `SwitchBetween` (Listing 5.20).

5.6.5. Point-free Reactor Behaviour Composition: "ror"

Reactor behaviours are composable in 2 ways. The first way is via a regular `deploy` expression, where one reactor behaviour uses another reactor behaviour. In analogy with functional programming we call this a **point-wise** composition of reactor behaviours. The second way is called **point-free** composition, where reactor behaviours can be composed by "glueing" them together.

We will exemplify both forms of composition by showing 2 equivalent implementations of the only remaining reactor behaviour used in Bikey that we have not yet shown, namely `TripTime` which tracks the duration of a bicycle trip. A point-wise implementation of `TripTime` is given in Listing 5.21. Its input signal `time-seconds` is expected to be the Unix time in seconds, and its single output is the duration of the trip. Tracking this duration is handled by deploying reactor behaviours that we have previously shown, namely `TimeElapsed` (Listing 5.16 on page 93) and `ToMinutes` (Listing 5.18 on page 97).

In mathematics, new functions can be defined point-free via a function composition operator ∘. The composition `f ∘ g` (pronounced "`f` after g") is a function `h` such that `h(x) = f(g(x))`. Similarly, Stella offers a `ror` operator ("reactor behaviour after reactor behaviour") that composes reactor behaviours: $r_1 \circ r_2$

```
1 (def-reactor (TripTime time-seconds)
2   (out (deploy ToMinutes (deploy TimeElapsed time-seconds))))
```

Listing 5.21: The `TripTime` reactor behaviour defined in point-wise style

```
1 (def-reactor TripTime (ror ToMinutes TimeElapsed))
```

Listing 5.22: The `TripTime` reactor behaviour defined in point-free style using `ror`

constructs a new behaviour where data is first propagated through $\texttt{r}_2$ and then through $\texttt{r}_1$. The DAG of this behaviour is the composition of $\texttt{r}_1$ and $\texttt{r}_2$, where the sinks of $\texttt{r}_2$ are connected to the sources of $\texttt{r}_1$.

As an example of point-free reactor behaviour composition, Listing 5.22 defines same `TripTime` from Listing 5.21 but in a point-free style using `ror`. `TripTime` is defined as a new reactor behaviour that combines the behaviours of `ToMinutes` and `TimeElapsed`. In this case, the constituent reactor behaviours can be easily composed because they each define only one source and one sink. If the reactor behaviours would not directly "fit together", intermediate reactor behaviours can take care of reordering sources and sinks or transforming data.

In general the `ror` operator is capable of connecting *multiple* "input" reactor behaviours to one "output" reactor behaviour. The following expression defines a new reactor behaviour `R` that is the composition of an output reactor behaviour R_{out} with input reactor behaviours R_1 to R_n.

```
(def-reactor R (ror R_out R_1 R_2 ... R_n))
```

Reactor behaviour `R` can be compiled as long as the number of sinks of all input reactor behaviours matches the number of sources in R_{out}. If this is the case, they will be connected in order from left to right to construct the reactor behaviour `R`. The sources of `R` will be the same as the sources of the inputs, ordered from left to right.

$$sources(R) \;:=\; sources(R_1) \;+\; sources(R_2) \;+\; \ldots \;+\; sources(R_n)$$

The sinks of `R` are the same as the sinks of R_{out}.

$$sinks(R) \;:=\; sinks(R_{out})$$

Additional point-free composition operators with different semantics that are not implemented in Stella are conceivable and useful, for example, the `parallel` and `parallel*` operators defined in [97]. In [98] various other kinds of point-free composition operators are used extensively to construct reactive sorting networks.

5.7. Tying Everything Together: Bikey's Main Actor

When combining the code snippets from this chapter in a single file, one obtains a complete, executable implementation of Bikey in Stella[12].

In this section we show the final piece of Bikey, namely the `Main` actor that responds to messages from the external HTML + CSS + JavaScript application. All user interactions will generate a message that is sent from JavaScript to the `Main` actor. Vice-versa, the `Main` actor sends updates of bike trips back to JavaScript by invoking an "update" JavaScript function via Stella's foreign function interface.

The `Main` actor for Bikey is defined in Listing 5.23. It declares four local fields on line 2 which are initialized in the constructor on lines 5 to 8.

`env` is a `JSProxyObject` provided by the external JavaScript world. It contains a method called `updateTrip!` that updates the GUI.

`bikes` is a dictionary that stores all `Bike` actors in the application. Its keys are identifiers generated by the outside JavaScript world to identify the GUI elements related to a particular bike, e.g., the small bike icons in Figure 5.1 on page 67.

`trips` is a dictionary that stores all `TripMonitor` reactors in the application. Similar to `bikes`, the keys are unique identifiers generated by JavaScript to identify the GUI elements related to a particular trip, e.g., the path a bike has travelled and the metrics of a trip (distance travelled, cost, etc.)

`time` is a `Time` actor (from Listing 5.11 on page 85) that provides Unix time as a stream. One `time` actor is shared among all `TripMonitor` reactors.

The methods in the body of `Main` handle messages sent from JavaScript in response to GUI events. They work as follows:

`add-bike!` (line 10) A bike was added in the GUI. Given an `id` and initial location for the bike, in response a new `Bike` actor (see Listing 5.9 on page 82) is spawned on line 11 and subsequently stored in the `bikes` dictionary. Since the identifier provided by JavaScript is a string, we convert it to a `Symbol` to use as a dictionary key. Otherwise, when using strings as key, there are equality issues where different `String` objects are not equal despite representing the same sequence of characters. Symbols do not have these issues.

`move-bike!` (line 14) A user drag & dropped a bike to a different location on the map, which represents a bike moving in the real world. The corresponding `Bike` actor is retrieved on line 15 and subsequently sent an `update-location!` message.

`delete-bike!` (line 18) A user removed a bike from the map. The corresponding `Bike` actor is retrieved on line 19, subsequently removed from the dictionary

[12] Note that we did not show the parts of the application not implemented in Stella, namely the HTML, JavaScript and CSS that power the webpage, and which communicate with the `Main` actor via messages and Stella's foreign function interface.

```
(def-actor Main
  (def-fields env bikes trips time)

  (def-constructor (start _env)
    (set! env _env)
    (set! bikes (newd Dictionary))
    (set! trips (newd Dictionary))
    (set! time (spawn-actor! Time 'default)))

  (def-method (add-bike! id initial-location)
    (def bike (spawn-actor! Bike 'init initial-location))
    (put! bikes (to-symbol id) bike))

  (def-method (move-bike! id new-location)
    (def bike (get bikes (to-symbol id)))
    (send! bike 'update-location! new-location))

  (def-method (delete-bike! id)
    (def bike (get bikes (to-symbol id)))
    (remove! bikes (to-symbol id))
    (terminate! bike)
    (if (contains? trips (to-symbol id))
        (let ((trip (get trips (to-symbol id))))
          (remove! trips (to-symbol id))
          (terminate! trip))))

  (def-method (start-trip! id)
    (def bike (get bikes (to-symbol id)))
    (def trip-monitor (spawn-reactor! TripMonitor))
    (react-to! trip-monitor id bike time 1 0.25)
    (put! trips (to-symbol id) trip-monitor)
    (monitor! trip-monitor.out 'update-trip!))

  (def-method (stop-trip! id)
    (def trip-monitor (get trips (to-symbol id)))
    (terminate! trip-monitor)
    (remove! trips (to-symbol id)))

  (def-method (update-trip! id trip-time path total-dist cost)
    (updateTrip! env id trip-time path total-dist cost)))
```

Listing 5.23: `Main` actor of the Bikey application.

and terminated entirely. Terminating an actor automatically cleans up any dependencies it has on streams, and vice-versa, dependencies from other actors and reactors on its own streams. If a trip was active while the bike got removed, the trip is terminated as well on lines 22 to 25. As a convenience we used the same `id` for a bike and its corresponding trip.

`start-trip!` (line 27) A user clicked a button in the GUI signalling to start a trip for a particular bike. In response, a new `TripMonitor` (see Listing 5.15 on page 92) is spawned on line 29, and subsequently instructed to react to the desired values on line 30. The trip monitor is stored and its output stream is monitored. Any updates by the reactor will be received by the `Main` actor as an `update-trip!` message.

`stop-trip!` (line 34) A user clicked a button in the GUI signalling to stop the trip of a particular bike. In response, the corresponding trip monitor is retrieved, terminated and removed.

`update-trip!` (line 39) Whenever any trip monitor reactor produces an update, the update is received as an `update-trip!` message because of the `monitor!` statement on line 32. In response, the outside JavaScript world is notified via a call to the `updateTrip!` method on `env` which invokes a foreign JavaScript function.

5.8. Stella Cheat Sheet

It is typical summarise a programming language in a so-called "cheat sheet" that provides a quick overview of a language's syntax and operations. Stella's cheat sheet is provided in Figures 5.10 to 5.14 on pages 106 to 108. It provides an overview of the object-oriented base language (Figure 5.10, actors and message passing (Figure 5.11), reactors (Figure 5.12), the composition of actors and reactors via streams (Figure 5.13), and *flocks* (Figure 5.14). Note that the reactor operator `deploy-*` and flocks are included in the cheat sheet, but will be introduced in Chapter 6.

5.9. Summary and Conclusion

In this section we set out to solve 2 of the problems discussed in Chapter 4, namely the Reactive Thread Hijacking Problem (Section 4.1 on page 50) where long lasting computations can stop a reactive program from being able to react, and the Reactive-Imperative Impedance Mismatch (Section 4.2 on page 52) which are the issues that arise when combining imperative and reactive code. Our proposed solution is based on the hypothesis that there is no panacea to write both imperative and reactive programs within a single unified language that

Stella Cheat Sheet: Object-Oriented Base Language

Define Classes

```
(def-class Foo
  (def-fields field1 field2 ...)
  (def-constructor (default arg1 arg2) <body>)
  (def-method      (Name1 arg1 ...)    <body>)
  (def-routine     (Name2 arg1 ...)    <body>))
```

Invoke Methods/Routines

```
(method-name receiver-object args...)
```

Make Objects

```
(new Foo 'default arg1 arg2) (newd Foo arg1 arg2)
```

Define Variables & Assign to Variables

```
(def id val)                 (set! id val)
```

If / Conditional Expressions

```
(if <condition>              (cond (<condition1>     <body>)
    <consequent>                   (<condition n...> <body>)
    <alternative>)                 (else <body>))
```

Figure 5.10.: Stella Cheat Sheet: Object-oriented base language.

Stella Cheat Sheet: Actors & Messages

Define Actor Behaviours

```
(def-actor Foo
  (def-stream stream-id arity1)
  (def-fields field1 field2 ...)
  (def-constructor (default args...) <body>)
  (def-method      (Name1 args...)   <body>))
```

Spawn Actors

```
(spawn-actor! <ActorBehaviour> 'constructor args...)
```

Send Messages

```
(send!       <actor ref>                'selector args...)
(send-after! <actor ref> <delay-in-ms> 'selector args...)
```

Figure 5.11.: Stella Cheat Sheet: Actors and message passing.

Stella Cheat Sheet: Reactors

Define Reactor Behaviours

```
(def-reactor (Foo signal1 signal2 ...)
  <body: def(-values)/if/new(d)/deploy(-*)/pre/bind/sample(-once)>
  (out signal1 signal2 ...))
```

Spawn Reactors

```
(spawn-reactor! <ReactorBehaviour>)
```

Deploy Reactor Behaviours

```
(deploy   <ReactorBehaviour> args...)
(deploy-* <ReactorBehaviour> arg)
```

Name Reactor Deployment's Multiple Output Signals

```
(def-values (signal1 signal2 ...) (deploy ...))
```

Capture Signal's "Previous Value"

```
(pre signal)
```

Partial Application

```
(bind <ReactorBehaviour> signal1 signal2...)
```

Sample Signals

```
(sample val-signal rate-signal) (sample-once signal)
```

Figure 5.12.: Stella Cheat Sheet: Reactors.

Stella Cheat Sheet: Actor-Reactor Composition & Streams

Emit to Actor's Streams

```
(emit! stream-id val)
```

Emit a value to a stream defined in the current actor. Number of arguments = arity of stream-id.

Change Reactor's Input Signals

```
(react-to! <reactor ref> val1 val2 ...)
```

Send a message to the designated reactor, changing the value of its input signals.

Use Stream in Reactors

`object.stream-name` **(qualification)**

A qualification expression in the body of a reactor creates an automatic dependency on the stream.

Use Stream in Actors

```
(monitor! <stream obj> 'selector)
```

Monitor a stream, and enter a 'selector message into the current actor's mailbox whenever the stream emits a value.

Compose Reactor Behaviours

```
(ror <Output ReactorBehaviour> <Input ReactorBehaviours...>)
```

Compose a new reactor behaviour out of the given reactor behaviours.

Figure 5.13.: Stella Cheat Sheet: Actor and reactor composition via streams.

Stella Cheat Sheet: Flocks

Define a Flock

```
(def-flock Foo)
```

Publish to a Flock

```
(publish! <flock> <local actor or reactor ref>)
```

Unpublish From Flock

```
(unpublish! <flock> <local actor or reactor ref>)
```

Figure 5.14.: Stella Cheat Sheet: Flocks.

exposes the same concepts and operations in both types of code. Instead, that imperative and reactive programs are fundamentally incompatible, and that they should be programmed whilst guaranteeing their own invariants. To this end the Actor-Reactor Model introduced in Section 5.2 serves as a new mental model to classify and design reactive systems.

Two aspects of Stella's design are key to using the Actor-Reactor Model work in practice. The first aspect is the object-oriented base language. Abstract data types are defined using classes that, besides regular methods, contain routines that always (eventually) terminate, and that are guaranteed to be free of side-effects. They allow actors and reactors to share the same data structures, while the set of operations they can invoke differ. Whereas actors can invoke both methods and routines, reactors are restricted to invoking only routines.

The second aspect is the separation of different parts of the program in either actors or reactors. Both are programmed in a different style. Actors are programming using traditional imperative code. Their code is executed from top to bottom, driven by the control flow of the program. Reactors are programmed in a style akin to Functional Reactive Programming. Their computations are not driven by the order of code statements, but by the flow of incoming data.

Our definition of reactors is atypical compared to most other Functional Reactive Programming languages and frameworks. We define distinct terminology for the different stages of a reactive program: A reactor behaviour represents the code of a reactive program, a reactor deployment is a specific "instance" of (part of) the reactive program, and a reactor is a process that contains a reactive engine that propagates values through one or more reactor deployments. Conceptually, the reactive programs in many existing reactive programming languages are analogous to one reactor with one deployment.

The Actor-Reactor Model avoids the Reactive Thread Hijacking Problem and the Reactive-Imperative Impedance Mismatch, and actors and reactors are a natural fit for distributed programming due to their isolated state and message passing semantics. How to use Stella in a distributed context is the topic of the next chapter.

6. Distributed Reactive Programming in Stella

This chapter tackles the third and final problem introduced in Chapter 4, namely the *Acquaintance Maintenance Problem* from Section 4.3 on page 56 which occurred when programming distributed reactive programs for *open* networks. Due to the open network assumption of Chapter 1, at every point in time, an application must know its set of acquaintances that can be reached over the network at that exact moment. Since the acquaintances vary throughout the lifetime of the application as devices join and become unreachable, a central aspect is acquaintance management, which we defined as the combination of 2 mechanisms:

1. An *acquaintance discovery* mechanism to discover acquaintances on the open network. In Stella, acquaintances consist of actors and reactors.
2. An *acquaintance maintenance* mechanism to subscribe to discovered streams in order to react as they appear, and to gracefully close the streams as they disappear.

In Section 4.3 we discussed acquaintance maintenance in state of the art programming languages and frameworks. This resulted in 2 different approaches based on signals and event streams. Both lead to a different code style, and different problems. We found that code written using event streams is efficient, but exhibits a lot of accidental complexity and seems to be more error-prone, whereas code written using signals tends to be more compact and idiomatic, but is inefficient.

The goal of this chapter is to define a mechanism that results in code that is idiomatic like signals, but that is also efficient. Acquaintance discovery will be conceived via a so-called *flock*, a new abstraction to automatically discover acquaintances. A flock communicates the joining and leaving of acquaintances via a stream, and a *topology-reactive* operator called `deploy-*` uses this information to implement efficient acquaintance maintenance.

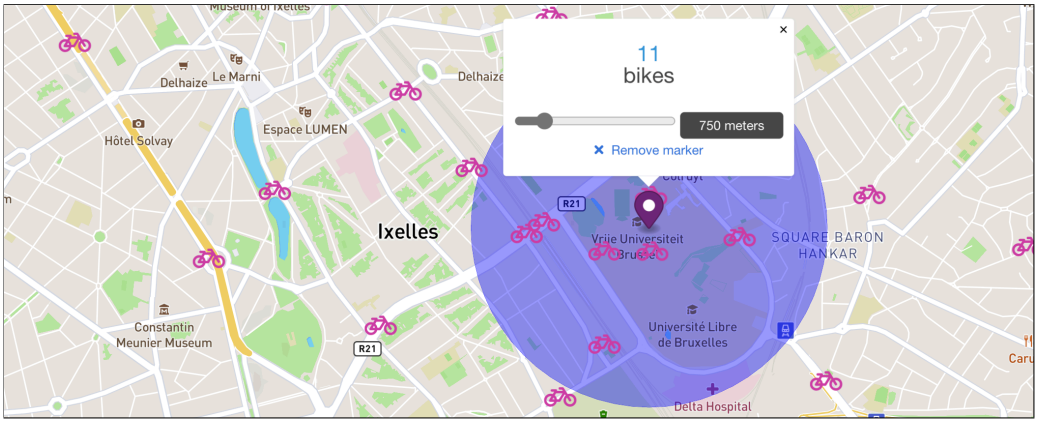


Figure 6.1.: Screenshot of Whereabikes (using mock bicycle data).

6.1. Running Example: Whereabikes

The running example used in this chapter is an extension to Bikey from Section 5.1 on page 66, namely a reactive bike counter called "Whereabikes" that counts all bikes that are available within a user-designated area. A screenshot of Whereabikes is given in Figure 6.1, where a user placed a "counting marker" to count all bikes within a radius of 750 meters of our university campus. Similar to Bikey, in the absence of real-world data, this implementation is also completely user-controlled. I.e., a user mocks data via the GUI by manually adding, removing, and moving (via drag & drop) bikes and counting markers.

The example is conceptually simple, but challenging to implement. The programmer must ensure that bike counters correctly update in the following circumstances:

1. Bikes constantly move in and out of the designated area highlighted in blue.
2. Bikes spontaneously appear and disappear as a result of changing network conditions, which is mocked by the user via GUI interactions.
3. A user may move the counting marker (drag & drop).
4. A user may change the radius wherein bikes are counted via a slider in the pop-up.

To give the reader a notion of scale, in 2019 *Villo!*, the public bike sharing program of Brussels, stationed 5000 bikes in the city dispersed over a total of 352 bike stations [109]. Consequently, the programmer must ensure that the computation is efficient.

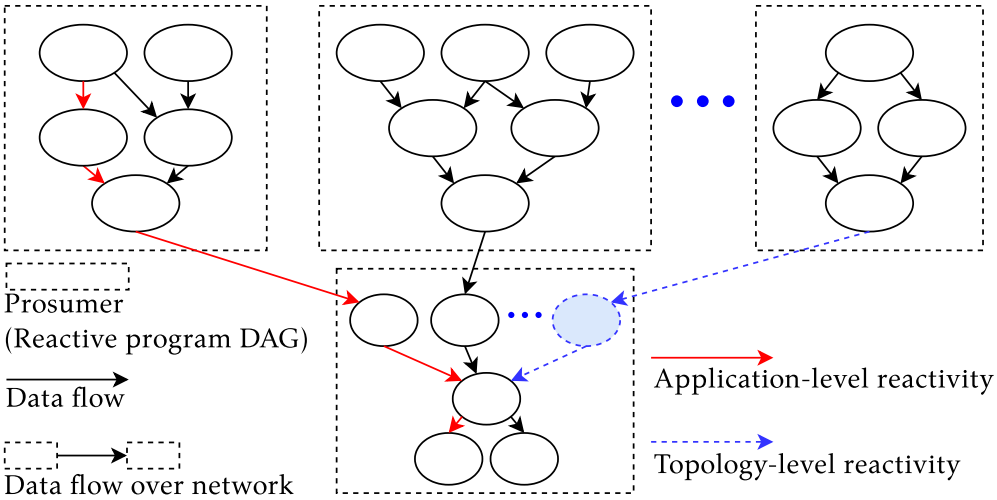


Figure 6.2.: Illustration of the 2 different levels of reactivity (recap).

6.2. Recap: Application-level Reactivity vs. Topology-level Reactivity

In Section 4.3.2 on page 58 we discussed acquaintance maintenance. To obtain efficient acquaintance maintenance, the language or framework needs to efficiently support two levels of reactivity, namely application-level reactivity and topology-level reactivity. We recap these two concepts since they are crucial to aid the understanding of this chapter.

Consider a reactive program that computes the average temperature of all thermometers that are connected to a network, and an acquaintance discovery mechanism that is capable of discovering them. Speaking in terms of streams, the `average` computation must appropriately react to streams appearing, disappearing, and updating with new values. In Figure 6.2 we illustrate these interactions between acquaintances and their streams. Each of the dashed rectangles represents one acquaintance, each containing a reactive program (visualised by a DAG). At the top we draw `N` (possibly different) producers of data, e.g., thermometers, and at the bottom we draw a sole consumer continuously reacts to data from the producers. We discriminated 2 levels of reactivity that constitute acquaintance maintenance:

Application-level reactivity: Whenever a source of one of the DAGs changes (at the top), the reactive language or framework automatically recomputes the dependent parts of the program that are affected by the change. We illustrate one such propagation path in red. We called this application-level reactivity,

```
1 // spawn bike using 'init constructor
2 (def bike (spawn-actor! Bike 'init (new LngLat 'at 4.3951313 50.822023)))
3 // publish bike actor to the Bikes flock
4 (publish! Bikes bike)
5 // remove bike actor from the Bikes flock
6 (unpublish! Bikes bike)
```

Listing 6.1: Publishing and unpublishing (re)actors to and from a flock.

which equates to the "normal" propagation of values in existing work, e.g., temperature measurements that flow through the program.

Topology-level reactivity: Topology-level reactivity occurs in consumers of which the computations depend on an open number of producers, e.g., the `average` calculation. The topology of the DAG needs to be continuously reconfigured to accommodate the appearing or disappearing streams. This is illustrated in Figure 4.1 in blue, denoting the appearing and disappearing of a stream (and its dependencies) in the DAG of the consumer.

The key to efficient acquaintance maintenance will be to achieve both efficient application-level reactivity *and* efficient topology-level reactivity.

6.3. Intensional Acquaintance Discovery via Flocks

In this section we propose a solution for acquaintance discovery, namely an acquaintance discovery abstraction called a *flock*. A flock is implemented as an actor that facilitates acquaintance discovery. In essence, a flock describes a collection of (re)actor references that are automatically shared over the network with other flocks that have the same name. Our definition of flocks was inspired by *volatile sets* [62] and flocks for ambient-oriented programming [22]. Both offer intensional acquaintance discovery in the same spirit as our flocks, but they have not been conceived for reactive programming.

Flocks are defined in top-level scope with a unique name. For example, the following code snippet defines a flock called `Bikes`.

```
(def-flock Bikes)
```

6.3.1. Publishing and Unpublishing Actors and Reactors

Local (re)actor references can be published to the network by adding them to a flock via `publish!`. For example, the devices that are running `Bike` actors from the previous chapter (Listing 5.9 on page 82) may (un)publish these actors as demonstrated in Listing 6.1:

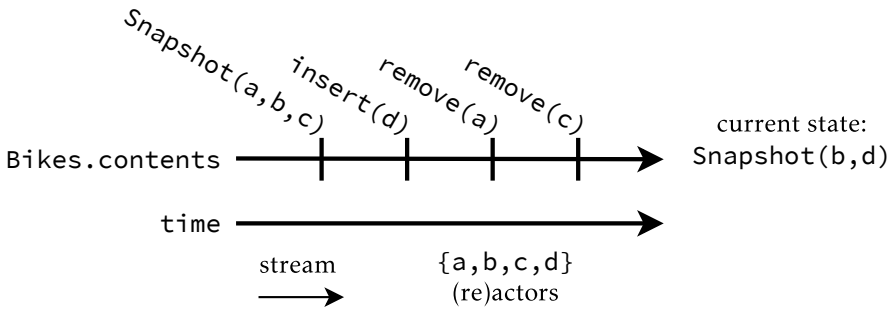


Figure 6.3.: Example of a stream that contains snapshots and patches.

Publishing or unpublishing (re)actors changes the contents of the flock, which are reactively communicated to other actors and reactors which are using the flock.

6.3.2. Reading the Contents of a Flock

Every flock exports a stream called `contents`, which is the primary means to access its continuously evolving acquaintances. Thus, a reference to this stream is obtained via the expression `Bikes.contents`.

The first value emitted by the flock to its `contents` stream is always a *snapshot*, and all subsequent values are a *patch*. We use these terms to more easily describe the use of a special `IncrementalBag` in combination with streams (inspired by [80, 116]).

Snapshot: The snapshot of a flock is an immutable `IncrementalBag` data structure (a set that may contain duplicate values) that records the contents of a flock at a specific moment in time.

Patch: A patch is emitted whenever the contents of a flock changes. There exist 3 types of patches, namely the `insert`, `update` or `remove` patches which can be applied to a snapshot. Concretely they are objects of the types `PatchInsert`, `PatchUpdate` and `PatchRemove`.

As an example, consider the `bikes.contents` stream that is drawn in Figure 6.3 which depicts a potential scenario using snapshots and patches. In this diagram time flows from left to right, meaning that the leftmost value is emitted first. The first value emitted by the stream is a snapshot containing 3 (re)actor references, followed by an insert patch and 2 remove patches. A receiver can use the patches to keep its view of the flock up-to-date, which in this case results in a snapshot containing only 2 (re)actor references `b` and `d`.

Snapshots and patches are used to obtain efficient application-level and topology-level reactivity. The semantics of snapshots and patches align with regular Stella

semantics for streams, where any (re)actor that subscribes to a stream immediately receives the most recently published value. For flocks this is always an up-to-date snapshot, rather than the latest patch.

6.3.3. Distributed Flocks

A defining property of flocks is that their contents are automatically shared over the network with all other flocks of the same type. Any additions via `publish!` or removals via `unpublish!` are automatically propagated to the other flock actors on the network, who incorporate those changes into their own contents. References to local (re)actors are automatically transformed to remote references when they are passed over the network. In the event of a peer or network failure, each peer individually determines the (re)actors that can no longer be reached, which are then automatically removed from that peer's flock. On a technical level we currently do so via heartbeats between all peers that share a flock. Whenever a disconnected peer rejoins the network, the removed (re)actors are re-added to the flock. This process of synchronising state and monitoring the network is always running in the background for every flock, hence why they are explicitly conceived as an actor.

The sharing of actor and reactor references is graphically depicted in Figure 6.4. At the top, we draw 2 flocks that are connected via a network. The left flock contains 3 local (re)actors in red, and the right flock contains 2 local (re)actors in blue. When there is an active network connection between the two flocks, then the flocks can exchange references to their local actors and reactors. The bottom depicts the same flocks and (re)actors, but here the connection between the two flocks was dropped. In this case each flock determines it can no longer reach the remote (re)actors (red and blue respectively), and thus removes them from their contents.

Our current implementation of Stella makes use of a discovery server to facilitate the discovery of the Stella interpreters running on different peers in the network. A signalling (discovery) server is required by WebRTC [26] (which is used by Stella) to exchange initial peer information without user interaction. Afterwards all communication is directly peer-to-peer [4]. The discovery server is also responsible for keeping a total overview of the complete contents of a flock, i.e., all (re)actors that a peer *should* be able to reach via the network.

Currently, any other Stella program can publish (re)actors to a flock. From a security point of view, we consider an authentication mechanism (e.g., to discover only authorised (re)actors) to be an orthogonal concern. Such mechanisms can be implemented on the meta-level (e.g., on the level of socket connections), via a type system, etc. At the time a (re)actor is included in a flock, we already assume that the other parties are trusted.

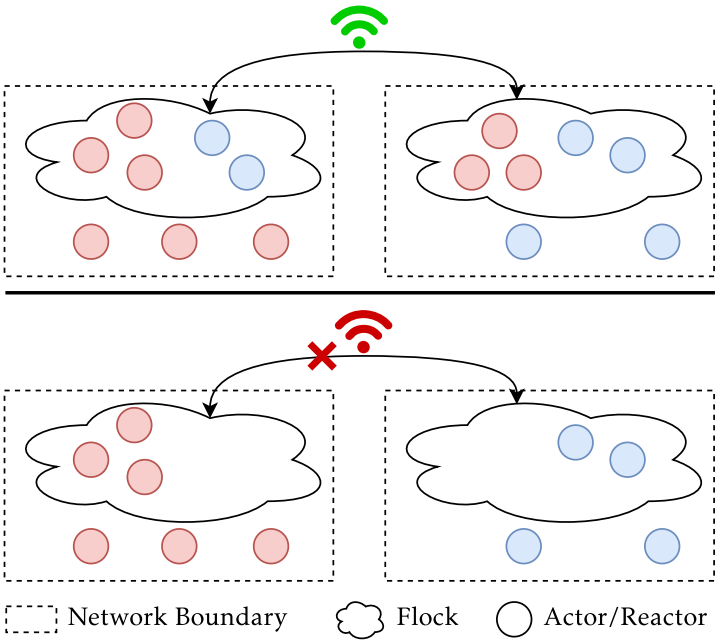


Figure 6.4.: Sharing actor and reactor references via a flock over the network.

6.4. A Topology-Reactive Operator: "deploy-*"

We show how reactors are used in combination with reactor deployments (cf. Section 5.5.4 on page 88) and flocks to support efficient application-level and topology-level reactivity. We introduce an efficient topology-reactive operator for reactors called `deploy-*`, and we demonstrate its semantics by implementing the bike counters from the running example.

6.4.1. Calculating Distance Between Bikes and Counting Markers

Before showing how to count all bikes using `deploy-*`, we define an auxiliary reactor behaviour called `IsBikeWithinRadius` that checks whether a bike is within the radius of the counting region, i.e., the blue area in Figure 6.1 on page 110. `IsBikeWithinRadius` is given in Listing 6.2. Its inputs are a point (GPS coordinate) at the centre of the radius, a radius in meters, and one bicycle (a reference to a `Bike` actor). The body calculates the distance between the point and the current location of the bike, and outputs the bike if the distance is smaller than the given radius.

Recall from Listing 5.9 on page 82 that the location of a bicycle was conceived as a `location` stream exported by a `Bike` actor. To calculate the distance to such a

```
1 (def-reactor (IsBikeWithinRadius point radius-meters bike)
2   (def distance-km (deploy DistanceBetween point bike.location))
3   (out (if (< (* distance-km 1000) radius-meters)
4            bike)))
```

Listing 6.2: The `IsBikeWithinRadius` reactor behaviour checks whether a bike falls within the radius of a counting marker.

```
1 (def-reactor (CountingMarker id location radius)
2   (def all-bikes Bikes.contents)
3   (def bikes-nearby
4     (deploy-* (bind IsBikeWithinRadius location radius) all-bikes))
5   (out id (size bikes-nearby)))
```

Listing 6.3: Counting all bikes within a radius

bike, `IsBikeWithinRadius` reuses the `DistanceBetween` behaviour that we have defined previously to calculate the great-circle distance between two coordinates (defined in Listing 5.13 on page 90). Note that the `if` test is lacking an alternative branch. In this case, the alternative branch is implied to be a constant value `#none`, an "empty value" used by reactors.

6.4.2. Topology-level Reactivity in Whereabikes

Listing 6.3 implements a reactor behaviour called `CountingMarker` that counts all bikes within an area. It has 3 sources: `id` is a unique identifier for the marker (a string generated by the GUI), `location` is the centre of the circular area wherein bikes are counted, and `radius` determines the radius (in meters) of the area. Its body is structured as follows. Line 2 defines a variable `all-bikes` that refers to the `Bikes.contents` stream, which emits all bikes that can be discovered on the network (a snapshot followed by patches). Line 3 defines `bikes-nearby` as the result of a `deploy-*` operation, which conceptually transforms a snapshot of all bikes to a snapshot that contains only the bikes that fall within the given radius. Line 5 declares 2 sinks, namely the `id` of the marker and the size of the `bikes-nearby` collection. These 2 values are published together on the `out` stream of the reactor whenever either of them changes.

6.4.3. The "deploy-*" Operator

The `deploy-*` operator can be thought of as a regular "map" operation for lists, but instead of mapping a function over a list, `deploy-*` deploys ("instantiates") a new reactor behaviour for every element in the given collection. In simplified terms, the type signature of `deploy-*` is given in Listing 6.4. Its first argument is the reactor

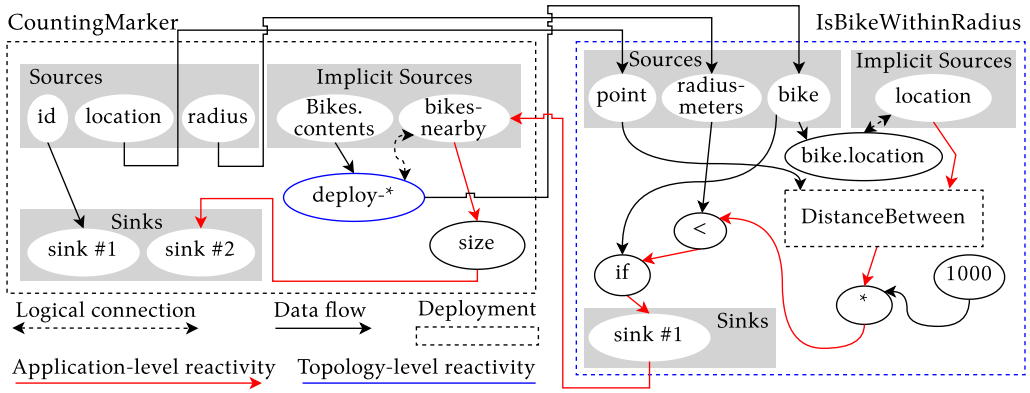


Figure 6.5.: `CountingMarker` DAG representation (Listing 6.3).

```
deploy-* :: ReactorBehaviour → Collection → Collection
```

Listing 6.4: The type signature of `deploy-*`.

behaviour to "map" over the collection, in this case given by a `bind` expression. The `bind` operator implements partial application for reactor behaviours (not to be confused with monadic bind). The first two sources of `IsBikeWithinRadius` (cf. Listing 6.2) will depend on `location` and `radius`, and its third source remains "unconnected". The second argument of `deploy-*` is a collection given by `all-bikes`. At run-time, a new deployment of `IsBikeWithinRadius` is created for each bike in the collection (in no particular order), and the corresponding bike is propagated via the sole unconnected source of the deployment. The result of `deploy-*` is a collection that contains the result of each created deployment, i.e., the value of their sinks (of which there must be exactly 1 for each deployment).

The creation of deployments is visualised in Figure 6.5 which depicts the DAG representation of `CountingMarker`. The blue `deploy-*` node has created one deployment of `IsBikeWithinRadius` whose sources are connected to the corresponding nodes in the `CountingMarker` deployment. The sink node of `IsBikeWithinRadius` is connected to the `bikes-nearby` implicit source. It is called "implicit" because it is generated by the DAG compiler to receive values from outside the current deployment, and in the case of `deploy-*`, it also constructs the correct output[1].

Intuitively, `deploy-*` offers "filter-map" semantics where values that signify "no value" (e.g., `#undefined`) are excluded from the result. This is the case in our example, since the implementation of `IsBikeWithinRadius` (Listing 6.2) did not

[1] Compiling a statement to 2 nodes is a common pattern that we also used to compile a `deploy` expression (see Section 5.6.2 on page 98) and a qualification (see Section 5.6.3 on page 99).

provide an alternative branch for the if-test. When this is not desirable, `deploy-*` accepts an optional third argument (not used here) to replace those empty values with a default.

6.5. Obtaining Efficient Application-level and Topology-level Reactivity

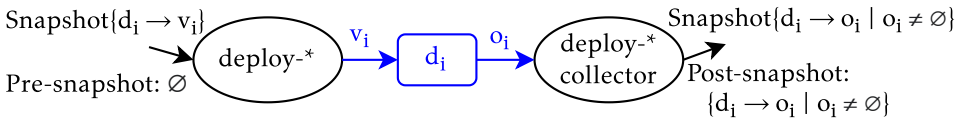
Efficient application-level and topology-level reactivity is achieved by meticulously propagating snapshots and patches whenever a change occurs. For example:

Topology-level patching: Whenever the `Bikes` flock emits an `insert` patch that adds a bike, then `deploy-*` also propagates an `insert` patch with the output of the newly created `IsBikeWithinRadius` deployment.

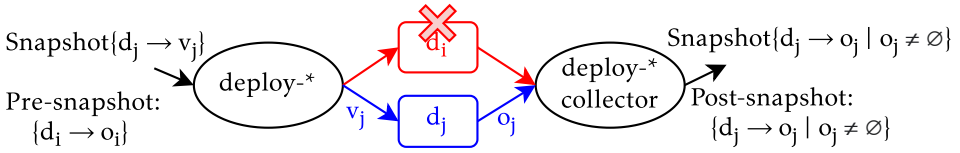
Application-level patching: Whenever a bike is within the counting radius and suddenly its location updates to being outside the counting radius, then `deploy-*` propagates a `remove` patch that removes the bike from the output.

We have devised a set of rules that show how `deploy-*` transforms its input to output, i.e., which snapshots or patches are propagated as output in response to a particular change in the input. As reactive programs are conventionally drawn as a DAG, we visualise these rules as a diagram that corresponds to a part of the DAG. Their structure can be explained using Figure 6.6a:

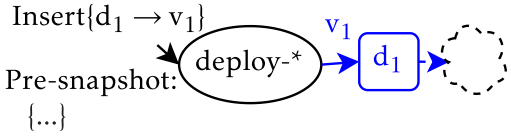
- Arrows denote the direction in which data flows, i.e., from left to right.
- Ellipses correspond to nodes in the DAG. The ellipse on the left corresponds to the main `deploy-*` node (the blue node in Figure 6.5), and the ellipse on the right corresponds to its implicit source node (called `bikes-nearby` in Figure 6.5).
- The input value of `deploy-*` is shown in the top left, and its output in the top right. All values v within snapshots and patches are associated with a *deployment key* d, denoted as $d \to v$. A deployment key is a unique identifier generated by the implementation of a snapshot (an `IncrementalBag`) when a value is inserted.
- Reactor deployments are drawn as rectangles and labelled with a deployment key that identifies the deployment.
- The run-time state of `deploy-*` is a snapshot. In the bottom left we show the pre-snapshot, i.e., the snapshot before the new input value has propagated through the DAG. The post-snapshot is given in the bottom right, i.e., `deploy-*`'s new state after propagating the input value.



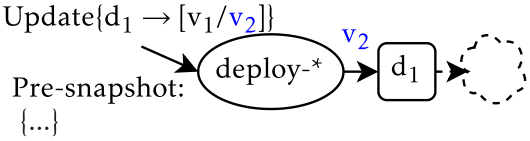
(a) Snapshot



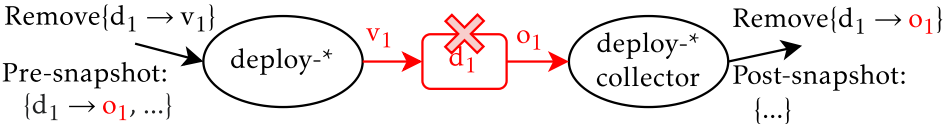
(b) Snapshot replace



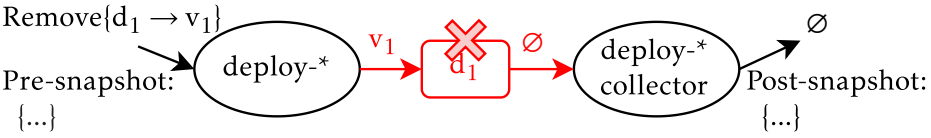
(c) Patch insert



(d) Patch update



(e) Patch remove



(f) Patch remove*

Figure 6.6.: Topology-level reactivity snapshot and patching rules.

6.5.1. Efficient Topology-level Reactivity

Snapshot (Figure 6.6a) The first value propagated by a flock is always a snapshot. This propagation is depicted in Figure 6.6a where the pre-snapshot is empty. A new deployment d_i is created for each deployment key $d_1 \dots d_i$ in the input snapshot, and the corresponding value $v_1 \dots v_i$ is propagated through the corresponding deployment. The outputs o_i are collected into a new snapshot with the same deployment key. Since deployments can produce "no value" (`#none` or `#undefined`), those values $\varnothing$ are omitted from the output and post-snapshot.

Snapshot replace (Figure 6.6b) While the situation does not arise when using flocks, in general the input of `deploy-*` can be replaced with an entirely new snapshot. In this case all existing deployments d_i from the pre-snapshot are replaced with new deployments d_j. The output is a new snapshot containing the outputs of the new deployments o_j, and the post-snapshot is adjusted accordingly.

Patch insert (Figure 6.6c) An `insert` patch adds a new deployment key d_1 with value v_1. A new deployment d_1 is created (highlighted in blue) with the new value v_1 as input. We omit the generated output and post-snapshot, denoted by the dashed cloud, as these follow the same rules as application-level reactivity (Section 6.5.2).

Patch update (Figure 6.6d) An `update` patch for a deployment key d_1 replaces its old value v_1 with a new value v_2. The new value v_2 is propagated through the existing deployment d_1. The generated output and post-snapshot follow the same rules as application-level reactivity (Section 6.5.2).

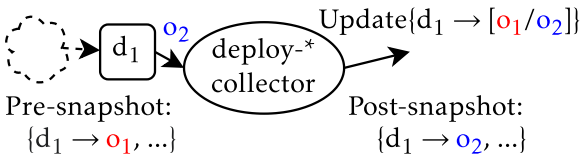
Patch remove (Figure 6.6e) To remove a value v_1, the corresponding deployment d_1 is removed. Since the pre-snapshot denotes that deployment d_1 previously contributed a value o_1 to the output, the new output is a `remove` patch with deployment key d_1 and value o_1.

Patch remove* (Figure 6.6f) A corner case of a `remove` patch is when the deployment d_1 did not contribute a value to the pre-snapshot. In this case no new `remove` patch is propagated as output (i.e., the old result remains valid).

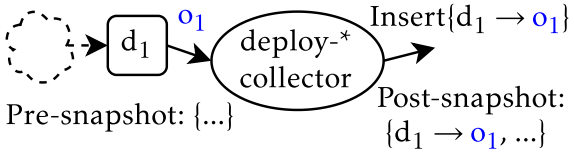
6.5.2. Efficient Application-level Reactivity

The set of rules in Figure 6.7 show the output that is generated after a reactor deployment produces a new value. Each diagram omits the left part (denoted by a dashed cloud).

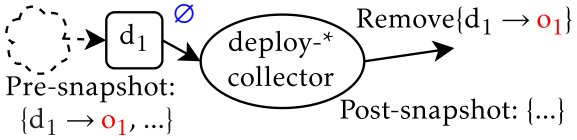
Production (Figure 6.7a) A deployment d_1 propagates a new value o_2, and d_1 is already associated with the output o_1 in the pre-snapshot. In this case,



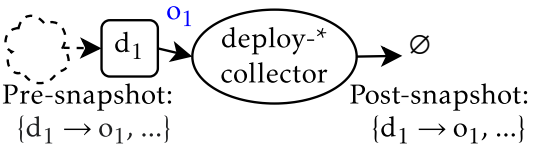
(a) Production



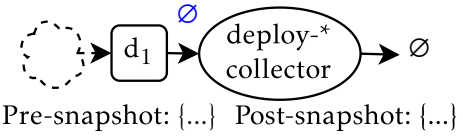
(b) Sudden production



(c) Remove production



(d) Equal production



(e) Empty production

Figure 6.7.: Application-level reactivity production rules.

propagate an `update` patch that replaces value o_1 with o_2 and update the post-snapshot.

Sudden production (Figure 6.7b) After a period of producing no value $\varnothing$, a deployment d_1 suddenly propagates a new value o_1. In this case, propagate an `insert` patch for o_1, and add this value to the post-snapshot. This situation arises, for example, when o_1 is the first value propagated by the deployment d_1 after its creation.

Remove production (Figure 6.7c) After a period of producing values, a deployment d_1 suddenly produces "no value" (`#undefined`). In this case, propagate a `remove` patch that contains o_1 and remove o_1 from the post-snapshot.

Equal production (Figure 6.7d) Since reactive programs are purely functional, typically only distinct values are propagated. Thus, if a deployment d_1 produces the same value o_1 included in the pre-snapshot, then no patch is propagated.

No production (Figure 6.7e) When a deployment d_1 produces "no value" and d_1 is not included in the pre-snapshot, then propagate nothing.

6.6. Discussion

There are a number of concerns and drawbacks related to our approach that should be taken into account.

Method calls When propagating patches, all operations that are executed on snapshots are automatically "patch-aware". For example, the invocation of `size` on `bikes-nearby` (in Listing 6.3 on line 5) automatically takes into account patches, such that previously computed results can be incrementally adapted. Some methods, such as a `fold`, require information about the values that were removed and replaced, hence why these values are included in `update` and `remove` patches.

Supported data structures In principle, `deploy-*` accepts any iterable collection as input value, but not all types are equally suitable. Stella currently supports a `Bag` and `IncrementalBag`. Particularly an `IncrementalBag` lends itself to an $\mathcal{O}(1)$ implementation of `deploy-*` for every `insert`, `update`, or `remove` patch. Other data structures such as lists and vectors are likely to introduce extra algorithmic complexity and computational overhead to the implementation of `deploy-*`, specifically because extra work is required to create and maintain order in the output collection [80], e.g., making sure indices in a vector are sequential and without gaps.

Developer convenience Unfortunately snapshots and patches are not entirely transparent, and programmers have to be aware of them. For example, consider the `AddAll` reactor behaviour in Listing 6.5 that sums all numbers in a snapshot. The `fold` method in the interface of `IncrementalBag` efficiently aggregates

```
1 (def-reactor (AddAll snapshot-of-numbers)
2   (out (fold snapshot-of-numbers 0 '+ '-)))
```

Listing 6.5: The `AddAll` reactor behaviour accumulates the values of an `IncrementalBag` by using a `fold`.

```
1 (def-flock Thermometers)
2 (def-reactor (SensorValue sensor) (out sensor.value))
3 (def-reactor (Average)
4   (def measurements (deploy-* SensorValue Thermometers.contents))
5   (def sum (fold measurements 0 '+ '-))
6   (out (/ sum (size measurements))))
```

Listing 6.6: Averaging the temperature of a set of thermometers in Stella.

its values. However, its expected arguments differ from a conventional `fold`: besides an initial accumulator and the `+` method name to add values to the accumulator, the programmer must also provide an operation to "reverse" the accumulator, in this case `-` (minus) for numbers. Otherwise it is not possible to efficiently update the accumulator after an `update` or `remove` patch.

6.7. Qualitative Evaluation: Comparison to State of the Art

In Section 4.3.2 on page 58 we implemented an `average` computation that calculates the average temperature of a set of thermometers in two styles using reactive streams (in RxJS) and signals (in REScala). We argued the RxJS code exhibits substantial accidental complexity, whereas the REScala code is more idiomatic, but inefficient. Our goal with Stella was to implement the example using idiomatic code like signals, but also efficient. In this section we show an implementation of the example in Stella. In Section 6.8 we will evaluate `deploy-*`'s performance.

Listing 6.6 implements the aforementioned `average` computation. It consists of a `Thermometers` flock (line 1), a `SensorValue` reactor behaviour (line 2), and the main logic which is implemented by the `Average` reactor behaviour (line 3). In a nutshell, `deploy-*` is used to transform a snapshot of thermometers to a snapshot of their latest measurements, which are subsequently averaged using the `fold` operation discussed in Section 6.6. Stella's programming style is more similar to the signal-based REScala code.

Compared to RxJS, `deploy-*` seems to be more restricted in use than RxJS' `flatMap` (used in Listing 4.3) and its siblings (e.g., `switchMap`). But these restrictions are no accident: the features and restrictions of `deploy-*` are due to it being

tailored specifically for supporting topology-level reactivity, whereas `flatMap` and its siblings operate at a broader level of abstraction. Hence, it is more difficult to use them for topology-level reactivity, e.g., when dealing with disappearing information (e.g., disconnecting thermometers).

The main difference between Stella's code and the REScala code from Listing 4.4 is that topology-level reactivity is *explicitly* managed via `deploy-*`, whereas in REScala it is implicitly managed as a consequence of using REScala in combination with Scala's `fold`. Using Stella's approach, efficient topology-level reactivity is achieved at the cost of one single line of code.

In both cases the guarantees provided by `deploy-*`, i.e., that application-level and topology-level reactivity are correct and efficient, are very difficult to emulate.

6.8. Quantitative Evaluation: Algorithmic Complexity

We provide experimental evidence that our approach leads to efficient application-level *and* topology-level reactivity. It is difficult to compare Stella to existing systems (e.g., REScala) in terms of raw performance, since they are completely different technological platforms. Therefore the benchmarks compare the algorithmic complexity of our approach and of the typical approach taken by state of the art RP languages such as REScala, but implemented using Stella.

6.8.1. System and Benchmarking Specifications

Experiments were run on Ubuntu 20.04.2 LTS and Node.js v14.16.0, with the command-line options `--trace-gc --max-old-space-size=65536`. While Stella is single-threaded, experiments were run on an AMD Ryzen™ Threadripper™ 3990X with 128GB of DDR4-3200 RAM, of which 64GB were available to Node.js (abundant for our application). We used Node.js' "performance measurement API", which implements the W3C recommendation for high resolution time with sub-millisecond precision [58].

6.8.2. Benchmarking Application-level Reactivity

We measure the time it takes (on average) for a value to propagate through a `CountingMarker` reactor from Listing 6.3. To this end, we ran two sets of experiments to propagate a varying number of bikes (from 1 to 1000) using an `IncrementalBag` and a (non-incremental) `Bag`. The `Bag` experiments replicate the algorithmic complexity of the same program implemented in an existing signal-based RP language such as REScala. Then, we measure the time it takes for the `CountingMarker` reactor to process 1 location update from a bike. To obtain statistically reliable results and to exceed any internal VM thresholds for

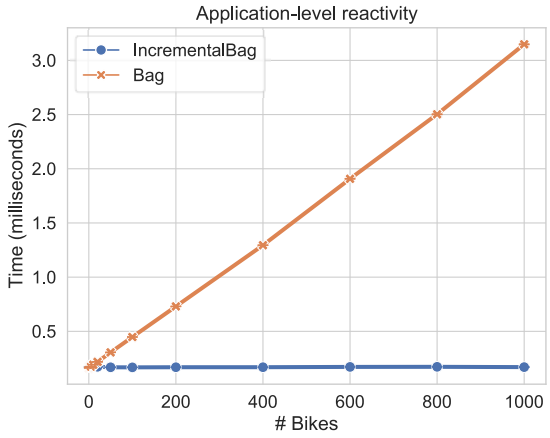


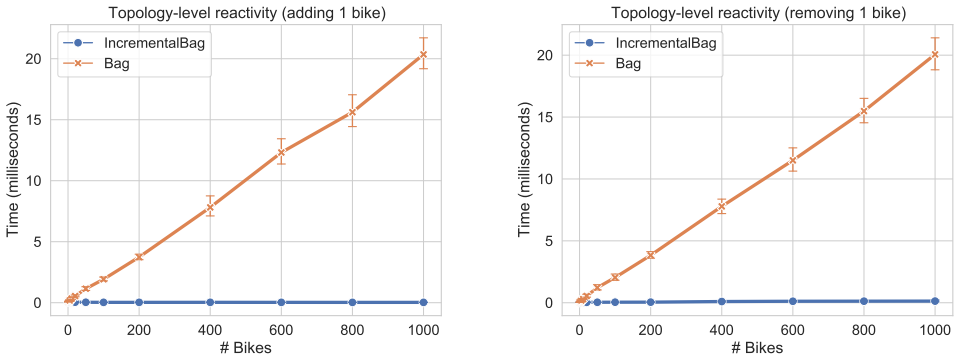
Figure 6.8.: Application-level reactivity experiments. Error bars = 99% confidence interval (drawn, but barely visible due to their small size).

optimisation [12], we measured 100,000 updates evenly spread over all bikes in the experiment. Total benchmark run-time ranged between 22 seconds (1 bike) and 6 minutes (1000 bikes).

If our performance claims are valid and if our proposed mechanisms are correct, then the measured execution time will grow with the number of bikes when using a `Bag`, but will remain constant when using an `IncrementalBag`.

Results

Based on a manual inspection of the data (via a scatterplot), between 50-500 updates are required by the JavaScript VM (V8) to warm up, so we removed the first 500 measurements for each experiment from the compiled results shown in Figure 6.8. The graph shows that the run-time using a `Bag` grows linearly with the number of bikes. This is as expected, since every time the location of one bike is updated, `deploy-*` constructs and propagates a new `Bag` (which is the case in existing signal-based RP literature such as the `fold` in Listing 4.4). In contrast, when using an `IncrementalBag`, run-times remain constant as the number of bikes increases because only a patch is propagated. Note that we draw error bars denoting a 99% confidence interval, but they are barely visible due to their small size. These results verify that application-level reactivity was implemented correctly and efficiently.



(a) Run-time cost of adding 1 bike.

(b) Run-time cost of removing 1 bike.

Figure 6.9.: Topology-level reactivity: comparison of the `Bag` and `IncrementalBag` experiments. Error bars = 99% confidence interval.

6.8.3. Benchmarking Topology-level Reactivity

We measure the time it takes (on average) to modify the topology of the DAG of the `CountingMarker` reactor (Listing 6.3) when the number of bikes increases or decreases. Similar to the application-level reactivity experiments, we ran two sets of experiments using an `IncrementalBag` and `Bag` with a varying number of bikes (from 1 to 1000). The `Bag` experiments replicate existing signal-based RP implementations.

Each experiment starts from a (fixed) number of `N` bikes (between 1 and 1000). First we propagate `N-1` bikes through the `CountingMarker` reactor (not measured). Then, we measure the time it takes for `deploy-*` to change the topology of the DAG when adding or removing the N^{th} bike.

If our performance claims are valid, then the experiments using a `Bag` exhibit longer execution times, because modifying the topology of the DAG involves more work. At the very least this requires a complete traversal over the `Bag` (i.e., `O(n)`).

To obtain statistically reliable results, we repeated each addition and removal of the N^{th} bike 500 times, except for the `Bag` experiments where `N = {400,600,800,1000}`, which were repeated 100 times to reduce total run-time. Total run-time ranged between 2 seconds (1 bike) and 4 hours (1000 bikes). Note that the run-time is high, because there can be a lot of application-level reactivity work between consecutive measurements. This is already an indication that inefficient topology-level reactivity can have compounding consequences for total application run-time.

Results

Based on a manual inspection of the data (via a scatterplot) we determined that, depending on the experiment (`Bag` or `IncrementalBag`) and number of bikes, between 25-100 updates are required by the JavaScript VM (V8) to warm up. We removed the first 100 repetitions for all experiments with 500 runs, and the first 25 for the `Bag` experiments with 100 runs (with `N = {400,600,800,1000}`).

We compiled the measurements of adding 1 bike and removing 1 bike in separate graphs, shown in Figures 6.9a and 6.9b respectively. Both show that the runtime of `deploy-*` in the `Bag` experiments grows linearly with the number of bikes, whereas it remains constant for `IncrementalBag`. These results verify that topology-level reactivity was also implemented correctly and efficiently.

6.9. Case Study on Scalable Distributed Programming: Villo!

We implemented a variation of Whereabikes (Section 6.1) based on real-world data from `Villo!`, the public bike-sharing programme of Brussels, Belgium[2]. This case study evaluates the scalability of `deploy-*` and flocks in a real prosumer application, and demonstrates the distributed capabilities of Stella.

The `Villo!` API only provides data about its ∼352 stations where bikes are picked up and dropped off, rather than data about individual bikes themselves. Thus, we will count how many bikes are available for pickup at any given time. A screenshot of the application running in a normal web browser is shown in Figure 6.10, where each small blue marker represents a bike station.

Villo! bike stations are simulated. For each station, an actor replays the events of bikes being taken and returned, which we recorded for an entire day. We evenly distributed these actors over a computer cluster of 160 Raspberry Pis (version 3, model B), a small single-board computer. A photo of the cluster is shown in Figure 6.11. At the time of writing, 147 of Raspberry Pis are operational for our experiments, each of which simulated 2 or 3 bike stations.

To simulate the application with a larger load, we also ran it on a real cluster that consists of 10 computers with an Intel® Xeon® E3-1240 v5 CPU and 32GB of RAM. Here, we simulated 3000 bike stations, which is approximately the largest bike sharing program in the world (2965 stations in Hangzhou, China [153]). We reused `Villo!` data by simulating the same stations multiple times but with random perturbations in the data. At this point interaction with the web browser became slow due to the large number of stations, as shown in Figure 6.12, partly because

[2] Villo! — https://www.villo.be/en/home

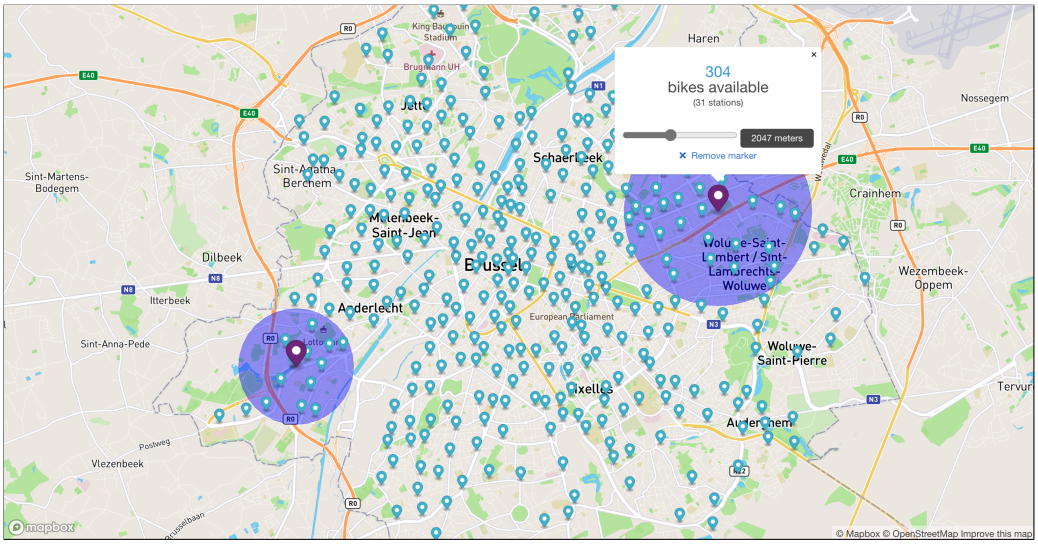


Figure 6.10.: Screenshot of the *Villo!* application. The popup reads "304 bikes available (31 stations) 2047 meters".

Figure 6.11.: Photo of the Raspberry Pi cluster. The Raspberry Pis are encased in LEGO®.

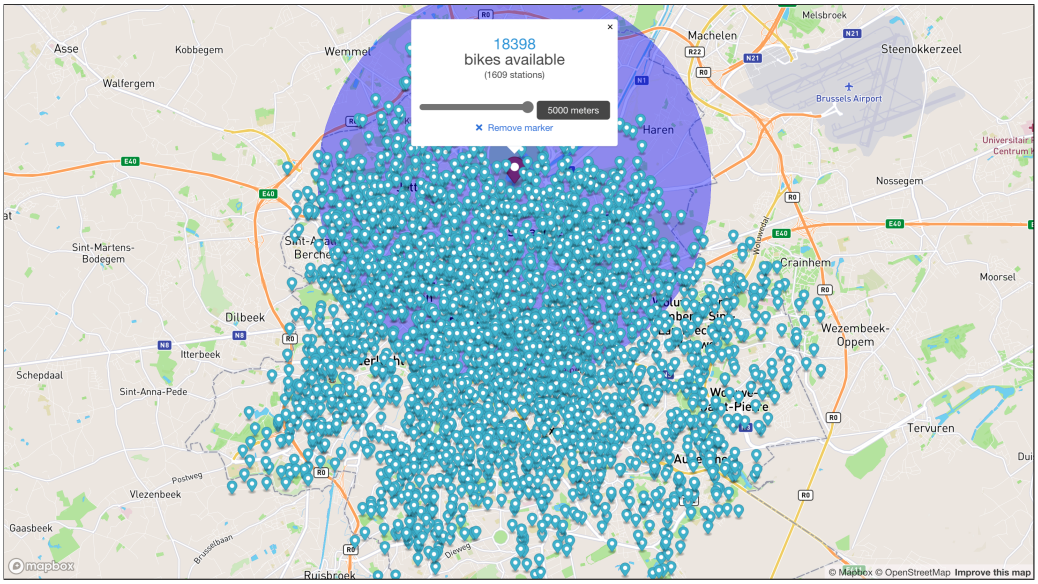


Figure 6.12.: Screenshot of the *Villo!* application with 3000 simulated bike stations. The pop-up reads "18398 bikes available (1609 stations) 5000 meters".

of the Stella interpreter, and partly because of the many GUI items rendered by the browser.

Lessons Learned We faced some practical network-related challenges when using flocks on the clusters.

Simultaneous connections: Whenever a new peer joins the network, all other peers with the same flocks simultaneously try to open a connection to the newly discovered peer. To prevent overwhelming the new peer (causing it to crash), connections have to be spread out over time, e.g., by introducing random delays.

Bidirectional discovery: When a new peer is discovered via a flock, situations can arise where 2 peers simultaneously open a socket connection to each other. Since WebRTC sockets are bidirectional, a sufficient solution is to have the peers deterministically decide on which of 2 sockets to use (e.g., using a peer's id) and which to close.

6.10. Summary

In this chapter we set out to provide a solution to the final problem presented in Chapter 4 which was related to correctly and efficiently managing acquaintances

throughout the reactive program. Managing acquaintances consisted of two mechanisms:

1. An acquaintance discovery mechanism to discover acquaintances on the open network.
2. An acquaintance maintenance mechanism to subscribe to discovered streams in order to react as they appear, and to gracefully close the streams as they disappear.

The main insight is that the problem should be tackled by carefully designing the acquaintance discovery mechanism to work with the acquaintance maintenance mechanism. As such, we defined flocks to intensionally discover (re)actors on the network. The key insight of flocks is that they offer a stream to the reactive program that propagates snapshots and patches. These patches are the key to efficient computations in reactors, which we conceived as a topology-reactive `deploy-*` operator that efficiently reacts to snapshots and patches. The resulting code is idiomatic like signals, and at the same time efficient compared to the traditional signal-based approaches.

7. Qualitative Evaluation: Comparison to the State of the Art

This chapter provides a classification of related work on reactive programming using the problems of Chapter 4 as main dimensions. We will show extensive evidence that the problems tackled in this dissertation are important to the domain of reactive programming, and that the problems are effectively present in existing reactive programming languages and frameworks.

A taxonomy of related work on FRP and stream-based reactive programming is given in Table 7.1 on page 132. The rows list the reactive programming languages and frameworks that we previously considered in Chapters 2 to 3 (Table 2.1 on page 12 and Table 3.1 on page 32). Every labelled column describes, to the best of our knowledge, a property of the language or framework according to the problems introduced in Chapter 4. We will discuss them from left to right in Sections 7.1 to 7.4. Afterwards, in Section 7.5, we discuss how the Actor-Reactor Model is already present in various related work.

The used acronyms and abbreviations are as follows:

RTHP Reactive Thread Hijacking Problem
RIIM Reactive/Imperative Impedance Mismatch
AD Acquaintance Discovery
AM Acquaintance Maintenance
I Imperative (code)
R Reactive (code)
Weak Weakly reactive
Eventual Eventually reactive
Strong Strongly reactive
M Meta-constructs
B Built-in primitives
A (A)Synchronous Input/Output

	RTHP	RIIM		AD	AM
		I ⊂ R	R ⊂ I		
Reactive programming for interactive applications or GUIs					
Dunai [102]	Weak	✓	M	-	-
Elm [36]	Weak	-	B+A	-	-
Flapjax [88]	Weak	-	M+B+A	Extensional	∼
Fran [43]	Weak	✓	A	-	-
Frappé [33]	Weak	-	M	-	-
FRPNow [106]	Weak	✓	A	-	-
FrTime [31, 65]	Weak	-	M+B+A	-	-
Haai [97, 98]	Strong	✓	A	-	-
Hokko [116]	Weak	-	M	-	✓
NewFran [44]	Weak	✓	A	-	-
RxJS [121]	Weak	-	M+A	Extensional	∼
Scala.React [79]	Weak	-	M	-	✓
Yampa [63]	Weak	✓	M	-	-
Reactive programming for embedded systems (e.g., microcontrollers, networks)					
Flask [81]	Strong	✓	M+B	-	-
Frenetic [48]	Weak	-	M	-	-
Nettle [143]	Weak	✓	M	-	-
CFRP [136]	Weak	-	M	-	-
EmFRP [127]	Strong	✓	M	-	-
Hae [150]	Weak	✓	M	-	-
ReactiFi [134]	Weak	-	B+A	-	-
RT-FRP [149]	Strong	✓	A	-	-
Distributed reactive programming					
Akka Streams [77, 119]	Weak	-	A	Intensional	∼
AmbientTalk/R [27]	Weak	-	M	Intensional	∼
Creek [138, 139]	Weak	-	✓	Intensional	-
DREAM [83, 84]	Weak	-	M	Extensional	-
Gavial [118]	Weak	-	M+A	Extensional	-
REScala [40, 90, 123]	Weak	-	M+A	Extensional	∼
ScalaLoci [152]	Weak	-	M	Intensional	∼
Stella [141, 147]	Eventual	✓	✓	Intensional	✓
XFRP [132, 151]	Weak	✓	✓	Extensional	-
Other					
ActiveSheets [142]	Eventual	✓	✓	Extensional*	-
Coherence [41, 42]	Weak	✓	✓	-	-
Lively RaTT [9]	Eventual	✓	✓	-	-

Table 7.1.: Classification of reactive programming languages and frameworks according to the problems introduced by Chapter 4.

7.1. Reactive Thread Hijacking Problem (RTHP)

The Reactive Thread Hijacking Problem was discussed in Section 4.1 on page 50. In summary, it describes the problem that occurs when some types of input data for the reactive program (accidentally) cause a long lasting computation that completely "hijacks" the reactive program's thread of execution. Consequently, this computation stops the reactive program from being able to react to any incoming data, and the run-time behaviour is essentially no longer *reactive*. In Section 4.1.1 we identified 3 levels of reactivity that reactive programming languages can adhere to.

Weak Reactivity A reactive programming language or framework is weakly reactive when it cannot bound the reaction time (i.e., the time to propagate one update) to arbitrary input data. In other words, the reaction time may be infinite.

Eventual Reactivity A reactive programming language or framework is eventually reactive when it can guarantee that the reaction time is bounded. I.e., a reaction will eventually terminate.

Strong Reactivity A reactive programming language or framework is strongly reactive when it can statically bound the reaction time to arbitrary input data. I.e., it will always react in $\mathcal{O}(1)$ regardless of the size or format of the input data.

We categorised related work in Table 7.1 according to *weak*, *eventual*, and *strong* reactivity.

7.1.1. Weakly Reactive Languages and Frameworks

Nearly all of the listed languages and frameworks are weakly reactive. Most of them are conceived as a library in a host language such as Haskell or Scala that imposes no restrictions on the complexity of expressions (e.g., while loops and infinite recursion). They are Dunai, Flapjax, Fran, Frappé, FRPNow, Hokko, NewFran, RxJS, Scala.React, Yampa, Frenetic, Nettle, Hae, ReactiFi, Akka Streams, AmbientTalk/R, DREAM, Gavial, REScala, and ScalaLoci. Other weakly reactive work is conceived as languages that allow unrestricted loops or recursion, such as Elm, FrTime, CFRP, XFRP and Coherence.

7.1.2. Eventually Reactive Languages and Frameworks

There are a couple of eventually reactive languages. Besides Stella, other eventually reactive languages include ActiveSheets and Lively RaTT. ActiveSheets is an FRP language where programs consist of Microsoft Excel spreadsheets. Many spreadsheet operations are always finite and in many cases $\mathcal{O}(1)$ (e.g., arithmetic).

However, we call ActiveSheets eventually reactive because, for some formulas, the reaction time of the program depends on the input given by an external program or user. Examples include `VLOOKUP` to look up a value in a table, and the `SEARCH` and `REPLACE` functions for searching in strings and replacing substrings. Thus, whenever unexpectedly large strings enter the program, the reaction time may be higher than expected, but still finite. Lively RaTT is a formalism that describes an FRP language that guarantees *liveness* [5], i.e., that "something good" must eventually happen.

7.1.3. Strongly Reactive Languages and Frameworks

The only strongly reactive programming language for interactive or distributed applications (like Stella) that we could find is Haai, a language designed by a colleague at our lab intended to further investigate reactive programming languages and strong reactivity. Note that other types of languages exist where strong reactivity is important, e.g., synchronous languages which we discussed in Section 2.3 (page 23).

A notable observation is that even languages and libraries such as Frenetic, Nettle, CFRP, Hae, and ReactiFi are weakly reactive, despite targeting embedded systems with limited processing power and memory. Frenetic and Nettle are used to program OpenFlow Software Defined Networking devices (e.g., routers and switches). They are conceived as a reactive DSL in Python and Haskell respectively. Since they allow ordinary Python or Haskell functions to be lifted to the level of reactive signals, there are no guarantees with respect to the processing time of these functions. CFRP is an FRP language for resource-constrained microcontrollers that is compiled to C++, which we classify as weakly reactive because it supports a foreign function interface that allows C++ functions to be called from within the reactive program. Similarly, Hae is a Haskell DSL that is transformed to C++ code, which allows (recursive) Haskell functions to be used in the reactive program. Finally, ReactiFi is a Scala DSL that generates C code to program Wi-Fi chips. While Wi-Fi chips have strict timing constraints, ReactiFi's authors note that its type checker cannot guarantee that the generated C functions terminate or use a bounded amount of memory.

7.1.4. Bounding Reaction Time

We briefly summarise existing tools and techniques that we found that a reactive programming language can use to enforce either eventual reactivity or strong reactivity.

Ensuring Eventual Reactivity

In general, there are both static and dynamic approaches to enforce eventual reactivity, i.e., to ensure that a program will eventually terminate. Each of the approaches has a varying level of restrictions that it imposes on the underlying program.

Static enforcement includes work such as total functional programming [140], primitive recursion [24], and Whalter recursion [85] that syntactically restrict recursive definitions such that they are guaranteed to (eventually) terminate. Furthermore, a program analysis such as T2 [25, 29] can be used to prove termination of programs written in C.

Techniques that dynamically enforce program termination can make use of both static information and run-time values, such as size-change termination [94] which is used by Stella.

Ensuring Strong Reactivity

To ensure strong reactivity, the language must statically limit the run-time cost of each execution step. For example, RT-FRP [149] statically limits the run-time cost of each execution step with a constant to guarantee their execution in real-time. EmFRP [127] excludes recursion from both its type and function definitions, and is thus able to statically calculate an upper limit on the run-time memory required to run a program.

Similar mechanisms are found in synchronous languages (cf. Section 2.3). For example, Esterel [17] disallows recursive definitions and disallows looping within the same reaction [137], and LUSTRE [59] allows recursion, but the number of recursive calls must be known at compile-time.

7.1.5. Concluding Remarks

Most of the reactive programming languages and frameworks listed in Table 7.1 (page 132) are weakly reactive, even those in application domains (e.g., embedded systems) that can benefit from stronger guarantees. Stella's approach to enforce eventual reactivity is novel compared to other general-purpose reactive programming languages and frameworks. Furthermore, the distinction between Stella's actors and reactors is an important feature to be able to enforce stricter constraints only on certain parts of the program (reactors). The alternative approaches either enforce no constraints at all (weakly reactive), or they enforce strict constraints across the entire language (strongly reactive), thereby limiting the kinds of applications that can be built.

7.2. Reactive/Imperative Impedance Mismatch (RIIM)

The Reactive/Imperative Impedance Mismatch was discussed in Section 4.2 on page 52. The term denotes the set of problems that occur when combining code written in the reactive programming paradigm (i.e., code that is driven by data) with code written in the traditional imperative programming paradigm (i.e., code that is driven by control flow). In Table 7.1 we distinguish between their combination in both directions, namely the embedding of imperative code within reactive code (denoted by $I \subset R$), and the embedding of reactive code within imperative code (denoted by $R \subset I$).

7.2.1. Embedding Imperative Within Reactive Code ($I \subset R$)

The embedding of imperative code within reactive code allows side-effects to be part of the reactive program's execution. As we discussed in Section 4.2.1, they are undesirable because they can cause tricky bugs due to the reactive program's update order, and have a detrimental effect on behaviour composition. Recognising these issues, many reactive programming languages already forbid side-effects. However, simply forbidding side-effects may not be enough. Before disseminating the taxonomy of Table 7.1, we first discuss how side-effects are handled in functionally pure FRP libraries.

Side-effects in Functionally Pure FRP Libraries

Reactive programming libraries for Haskell inherit their functional purity from Haskell itself. However, programs written in Haskell are not devoid of side effects and interaction with the outside world. To keep the programs pure, side-effects are wrapped in so-called *IO actions*, which are themselves immutable values of type `IO` that can be composed (e.g., via the IO monad [96]) into larger operations. IO actions are eventually executed by the interpreter. Haskell remains pure by using the type system to indicate which parts of the program are functionally pure, and which are not (i.e., parts of type `IO`).

Whenever FRP Haskell libraries need to interact with the outside world, they need to do so via such IO actions. Fran, the first FRP library, offers no general method to interact with the outside world via IO actions. To solve this issue (as well as other issues), Yampa introduces *arrowized* FRP where an FRP program is a function type `SF Input Output`. Here, `SF` represents the type of a *signal function*, which is a function that operates on time-varying values (i.e., signals). The `Input` type represents all of the input values for the signal function, and the `Output` type represents all output values. To perform `IO` actions using arrowized FRP, all IO actions must flow through the top-level "main" function before being executed by the interpreter. As depicted in Figure 7.1, IO input needs to be routed from the

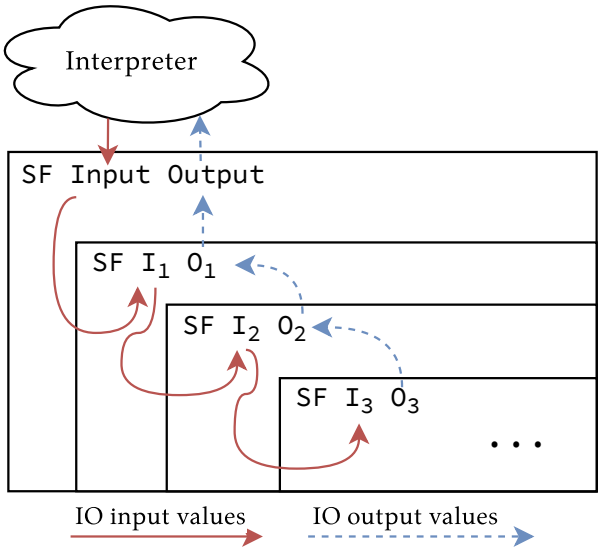


Figure 7.1.: Conceptual propagation of IO actions using Signal Functions `SF`.

top-level function to deep within the program, and any IO actions created deep within the program need to be routed to the top level function. Consequently, many type signatures of functions along the way need to be changed to incorporate the new IO action. This has a detrimental effect on software development (as noted by [106]), because adding an IO action deep within an existing program is contagious.

Alternative approaches to arrowized FRP which do not suffer from the aforementioned issue have been proposed, e.g., FRPNow [106] and the more general Monadic Stream Functions [103] (MSFs). For example, Monadic Stream Functions are functions of type `MSF m Input Output` where `m` is the type of a monadic context (e.g., Haskell's `IO` monad), and `Input` and `Output` are the type of the input and output values respectively. By including a monadic context for each `MSF`, IO actions no longer have to propagate to the top-level function, but can instead be passed to the interpreter directly, as depicted in Figure 7.2.

Since MSFs allow IO actions to be "executed" (passed to the interpreter) at arbitrary points in the reactive program, one must wonder whether the Reactive/Imperative Impedance Mismatch has been reintroduced. More specifically the problem where the relative order of side-effects cannot be easily determined by the programmer, and which can potentially cause bugs if related side-effects are not correctly ordered. In arrowized FRP this problem was not present, since the programmer has to consciously compose IO actions in a particular order to be able to propagate them to the top level. However, when using MSFs, the order in

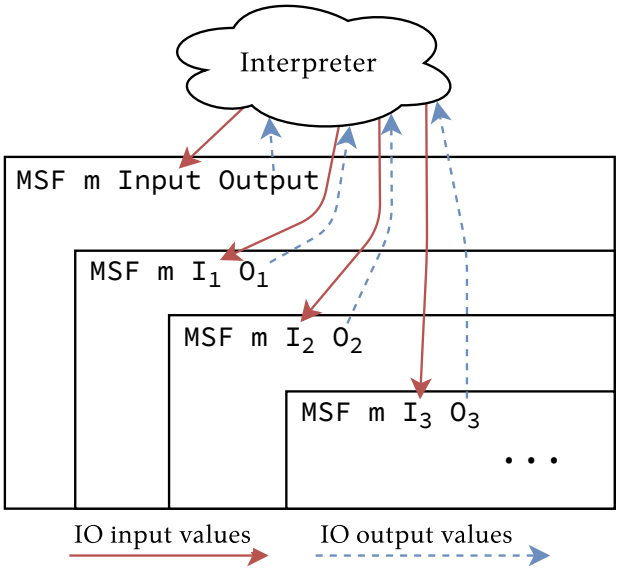


Figure 7.2.: Conceptual propagation of IO actions using Monadic Stream Functions

which MSFs are composed may depend on the MSF library. For example, consider an MSF that is built via the "parallel" MSF combinator &&&, such as `c = a &&& b`. This operator composes 2 MSFs (`a` and `b`), and broadcasts the input given to the composed MSF `c` to its 2 constituents in parallel. When both `a` and `b` generate IO actions, then the order in which they are executed depends on the implementation of &&&. I.e., first `a` and then `b`, first `b` then `a`, or `a` and `b` concurrently. In this case the programmer cannot tell in which order the IO actions are executed, since the order depends on the implementation of the library.

$I \subset R$ in Related Work

We categorised related work in Table 7.1 (page 132) with a checkmark when they do not allow side-effects to be a part of the reactive program's DAG. They are either conceived as Haskell libraries and thus functionally pure (Dunai[1], Fran, FRPNow, NewFan, Yampa, Flask[2] and Nettle), or they are conceived as fully

[1] Dunai avoids executing side-effects (IO actions) within the reactive program itself, which we currently consider to be a sufficient separation between imperative and reactive code for IO. We are currently unaware of a solution to also correctly order these side-effects when they are executed by the interpreter.

[2] Flask is implemented in the Red programming language [11, 110], which is syntactically equivalent to Haskell. Red imposes additional constraints on Haskell 98 [105], the main ones which are relevant to our discussion are the disallowing of recursive data types, closures, and recursive functions.

fledged programming languages that simply do not support operations with side-effects (Haai, EmFRP, Hae, RT-FRP, Stella, XFRP, ActiveSheets, Coherence, and Lively RaTT).

Of the various languages and frameworks that do not forbid side-effects, most of them are conceived as libraries written in imperative languages (Flapjax, Frappé, Hokko, RxJS, Scala.React, Frenetic, ReactiFi, Akka Streams, AmbientTalk/R, DREAM, Gavial, REScala, and ScalaLoci). Sometimes languages do not allow side-effects directly, but still allow them to be executed via a foreign function interface. For example, CFRP is able to import C++ functions, and FrTime tightly integrates with the Racket language [46]. Finally, Elm is an FRP language for developing web applications that appears to be purely functional. However, since side-effects are often essential, the language includes native functions such as `getImage` to fetch an image from a URL, which is clearly a side-effect on the network.

7.2.2. Embedding Reactive Within Imperative Code ($R \subset I$)

In practice, reactive code is *always* embedded within a program that is imperative. On the one hand, the imperative code (e.g., a network or GUI) is responsible for providing input values to the reactive program, e.g., by modifying the value of a source and thus giving rise to a turn of the reactive program. On the other hand, a reactive program that is purely functional does not do anything, and thus side-effects are required to act on the output of the reactive program, e.g., to modify a GUI, send a message over the network, push a notification to a user, etc. Often this coupling between the imperative code and the reactive code is semantically ill-defined, or relies on mechanisms that are not broadly applicable.

In Section 4.2.2 on page 53 we discussed three general mechanisms that we identified in related work to couple the imperative code to reactive code and vice-versa. We briefly recap them, and describe how these mechanisms manifest themselves in related work.

Built-in primitives (abbrev. "B")

Built-in primitives are language features that perform a specific task, and which a developer could not program otherwise by using the features of the language. Typically this only applies to dedicated reactive programming languages and DSLs.

There is some related work in Table 7.1 that offers built-in primitives. Elm includes primitive signals that are automagically driven by the language run-time, e.g., `Mouse.position`. Elm, FrTime and Flapjax include special primitives to construct GUI widgets that automatically produce signals and incorporate signals (e.g., the contents of a text input field). Flask includes a networking primitive to

broadcast signals to the network, and to receive signals that others have broadcast. Finally, ReactiFi has special primitives for programming Wi-Fi chips, e.g., to switch Wi-Fi channel, or to send data to the operating system.

Meta-constructs (abbrev. "M")

Reactive languages and frameworks often include non-reactive "meta" code that makes the reactive program work, and which is written using imperative code rather than reactive code. Crucially, the non-reactive (imperative) meta-code and the reactive code are intermingled, and are executed by the same program thread.

Most languages and frameworks in Table 7.1 include meta-constructs. In most cases they are libraries where non-reactive code creates the right library calls to provide new input to the reactive program, and to be notified (e.g., using callbacks) whenever new output is produced. For example, FrTime and REScala offer built-in primitives to create and (imperatively) modify input signals of a reactive program. Doing so causes a new propagation turn every time the value of an input signal is modified. Streaming frameworks such as RxJS and Akka Streams offer a special type of stream to which values can be imperatively pushed, e.g., a `Subject` in RxJS. Most other related work includes a variation of these mechanisms to drive the reactive program. We will briefly discuss the notable exceptions.

Flask is an FRP library for Reactive Wireless Sensor Networks. It includes a combinator called `adc` that samples the output of an external nesC [54] component[3] at a specified rate. Thus, Flask's meta program can be written in nesC.

CFRP and Hae are FRP languages for small-scale embedded systems. Their top-level input signal signal declarations directly correspond to C++ classes that are implemented by the developer to interface with external devices (e.g., sensors). Similarly, programs written in EmFRP also declare their input signals, from which the EmFRP compiler generates skeleton code in C with empty function bodies that should be completed by the developer to drive the input of the EmFRP program. Hence, the meta program that drives CFRP and Hae is written in C++, and the meta program of EmFRP is written in C.

(A)synchronous input/output (abbrev. "A")

Recognising the Reactive Thread Hijacking Problem, reactive programming languages and frameworks can make use of concurrency to separate different operations in their execution, e.g., to asynchronously execute long lasting computations. Crucially, the combination of these (a)synchronous input/output features can still cause issues for the reactive program, or the semantics of their interaction reactive

[3] nesC is a component-based, event-driven language to build applications for TinyOS.

program are unclear. There are a wide range of mechanisms employed by related work, which we summarise.

Elm is an FRP language for web applications, which features an `async` operation to asynchronously schedule and execute a blocking computation (e.g., fetching an image from a URL). A similar mechanism has been implemented in Gavial and FRPNow.

Flapjax and FrTime contain features such as `delayB` (called `delayBy` in FrTime) that delays the values of an input signal by a given number of milliseconds, i.e., it creates a new signal that asynchronously echoes the input signal, but with a wall-clock time delay.

Fran, FrTime and Haai contain a built-in signal that represents the current time (e.g., Unix time). Similarly, in ReactiFi a source of the reactive program can be constructed from a timer that fires at regular intervals (e.g., every 10ms). These time updates are driven by some (asynchronous) loop external to the reactive program.

FrTime is an FRP language that tightly integrates with Racket [46], and which developers can interact with via expressions entered into Racket's Read-Eval-Print Loop (REPL). By experimenting with FrTime and its implementation we found that the REPL and the reactive program run in different program threads, and that they interact via asynchronous message passing. Their interaction is not part of the FrTime programming model.

RxJS is a reactive streaming library in JavaScript (and 17 other languages [112]). The propagation of values through RxJS's reactive streams is typically synchronous from input signal to output signal. However, similar to other ReactiveX implementations, RxJS contains different *schedulers* (execution contexts) whereby streams scheduled on a different scheduler are updated concurrently [120]. By using schedulers, the propagation of values can be switched from synchronous to asynchronous, or a dynamic combination of both. Similarly, Akka Streams offers a mechanism to concurrently execute certain operations on a stream [75].

REScala [123] is an FRP library for Scala that allows Scala code to synchronously modify the value of the reactive program's sources (i.e., the program blocks until propagation is finished). While multi-threading is not part of REScala's semantics, from our experiments we learned that multiple Scala threads may change the same sources simultaneously. REScala's implementation requires these threads to acquire a lock to prevent race conditions. When multiple threads supply input values for sources, they may suffer from lock contention. Following REScala's original paper [123], the authors devised a mechanism whereby source updates from multiple threads can propagate through disjoint regions of the DAG concurrently to improve performance. Essentially this modified implementation of REScala does not change the language itself, but it changes the semantics of interactions between imperative and reactive code.

7.2.3. Concluding Remarks

There are a number of reactive languages and frameworks that completely avoid the Reactive/Imperative Impedance Mismatch. They are labelled with a checkmark in Table 7.1 for both `RIIM` columns. From these languages, ActiveSheets (Microsoft Excel) and Lively RaTT (a formalisation) are not general purpose languages. XFRP is an actor-based FRP language that separates IO code from reactive code using actors (just like Stella), which we will further discuss in Section 7.5. Coherence defines an exotic programming model where code is divided in *derivations* and *reactions*. Derivation is used to automatically compute the program output by deriving values from input via purely functional computations. The interaction between derivations and reactions seems to be sound, but there are many open questions before Coherence can be used to build distributed reactive programs.

Stella's approach, namely the Actor-Reactor Model, avoids the Reactive/Imperative Impedance Mismatch by construction. The main benefit of our approach is that some parts of applications are more natural to program either imperatively or reactively. Stella supports both using actors and reactors, and specifies precisely how they can interact with each other (via message passing).

7.3. Acquaintance Discovery (AD)

Acquaintance discovery was discussed in Section 4.3 on page 56. Here, we defined two kinds of acquaintance discovery: extensional and intensional. Extensional discovery encompassed techniques whereby a program obtains a reference to an acquaintance by explicitly naming it, e.g., via a hostname, URL, or other types of unique identifiers. Intensional discovery encompassed a range of discovery mechanisms whereby groups of acquaintances are automatically discovered on the network, e.g., based on common type tags or descriptions. We categorise the related work in Table 7.1 (column `AD`) according to the support for extensional or intensional discovery (or the lack thereof, indicated via a hyphen).

7.3.1. Extensional Discovery in Related Work

From the related work that supports extensional discovery, DREAM, REScala, and XFRP discover acquaintances via a unique name. They publish references to signals to a global registry such as Java's remote object registry, or Erlang's global process registry. A different approach is taken by Gavial, a multi-tier programming language that supports 3 tiers (client, server, and session). The supported tiers are referred to extensionally via standard Scala variable references.

Besides the languages and frameworks listed in the category of "distributed reactive programming", Flapjax, RxJS and ActiveSheets also support extensional acquaintance discovery. They are not included in the category of distributed reactive programming because their programming abstractions for reactive values (e.g., a signal or stream) cannot be passed over the network. Essentially, RxJS supports operators to create a new stream from a socket (e.g., a WebSocket) [87], but the abstraction of a stream itself cannot be passed over the network. Flapjax contains stream operations such as `evalForeignScriptValE` that continuously fetches new JavaScript files from remote URLs, evaluates them, and pushes the result of their computation on an output stream. Finally, users of ActiveSheets can store a live spreadsheet on a remote server via GUI interactions. Using the GUI, data from these live spreadsheets can be imported into other (local or distributed) spreadsheets.

7.3.2. Intensional Discovery in Related Work

We found a number of intensional acquaintance discovery mechanisms in related work. The common denominator is that they somehow allow multiple acquaintances (e.g., actors or signals) to be referred to via the same reference (e.g., a type or description). Akka Streams does so via a `Receptionist` abstraction [74] that allows multiple actors to be published to the same topic (i.e., a topic-based publish-subscribe mechanism). The crucial difference compared to extensional mechanisms is that actors published to the same topic can be retrieved as a collection, rather than discovering each actor individually. Similarly, AmbientTalk/R supports *ambient references* and *volatile sets* which are used to describe collections of remote (reactive) objects that share a common type tag.

Creek, an actor-based streaming library in Elixir, offers a global stream to the program that emits *join* and *leave* events whenever actors are discovered or disconnected. Collections of related actors can be created based on these events.

ScalaLoci is a multi-tier reactive programming language where the various tiers are joined via so-called ties. A single tier (e.g., a server tier) can be connected to multiple other tiers (e.g., client tiers). The connected tiers are communicated to the reactive program via a signal that carries them in a list. Additionally, event streams are provided that notify the application of peers that join and leave.

Stella's flocks are similar to other intensional discovery mechanisms, but they are better integrated with the reactive program by propagating a snapshots and patches that enable efficient reactions.

7.4. Acquaintance Maintenance (AM)

Acquaintance maintenance was discussed in Section 4.3.2 on page 58, which encompasses the mechanism to correctly and efficiently maintain the program state of the varying number of acquaintances throughout the reactive program. We discussed 2 mainstream approaches to perform acquaintance maintenance based on streams and signals. The stream-based approach revolved around processing a stream of *join* events whenever a new acquaintance is discovered on the network, and the programmer is responsible for manually managing the application's topology and state whenever acquaintances join, disconnect, or propagate new application-level updates. The signal-based approach exhibits much less code complexity than the stream-based approach since it offers a continuously updating list of connected acquaintances, which is typically processed inefficiently using list operations such as maps and folds.

We have divided the related work in Table 7.1 (column `AM`) in 3 categories. Note that we evaluated support for the mechanisms of acquaintance maintenance regardless of their support for distributed programming.

- A hyphen (-) indicates that we could find no evidence that the language or framework supports acquaintance maintenance.
- A tilde (∼) indicates that we could find some evidence of support for acquaintance maintenance, but that it is inefficient or complex in the same way as discussed in Section 4.3.2.
- A check (✓) indicates support for idiomatic and efficient and acquaintance maintenance.

Most languages and frameworks in Table 7.1 do not support acquaintance maintenance. Those that do have some support are Flapjax, RxJS, Akka Streams, AmbientTalk/R, REScala and ScalaLoci. RxJS and Akka Streams are frameworks based on reactive streams where acquaintance maintenance is supported but complex. Similarly, we believe that Flapjax has support for acquaintance maintenance due to its support for higher-order streams (streams whose values are other streams) in combination with its `switchE` operator to "flatten" those streams (similar to `flatMap` in RxJS). AmbientTalk/R, REScala and ScalaLoci support (inefficient) signal-based acquaintance maintenance using regular functional programming techniques such as maps and folds.

There are 2 libraries that support *efficient* acquaintance maintenance, namely Hokko and Scala.React. Both propose a mechanism that involves incremental data structures to obtain efficient computations within a reactive program. The design of Stella's snapshots and patches are inspired by both of their approaches. Note that Hokko and Scala.React do not support distributed programming, and we cannot verify whether their mechanisms still work when acquaintances are distributed over a network.

Stella's approach is conceived as the topology-reactive `deploy-*` operator which integrates with Stella's flocks. It ensures that the reactive program correctly reacts on both the application-level and the topology-level, which is a common type of computation when implementing reactive applications for open networks. Without `deploy-*`, a programmer has to implement similar functionality themselves, which we found to be difficult to do correctly.

7.5. Precursors to the Actor-Reactor Model

One of the main contributions of this dissertation is the Actor-Reactor Model as a new programming model for distributed reactive systems. There is some evidence that some weak form of actors and reactors are already present in related work, without them being identified as such. Hence, in this section we briefly discuss the precursors of the Actor-Reactor Model that we could identify in related work.

7.5.1. Akka Streams

The most notable example of actors and reactors in practice is Akka Streams [77, 119], a stream-based reactive programming library built on top of the Akka actor library for Scala, which we previously introduced in Section 2.2.2 (page 21). Akka Streams allows developers to construct and compose *flows* which correspond to a reactive program's logic. Flows are executed by dedicated actors that are responsible for propagating values through the flow. When using our terminology, a flow corresponds to a reactor behaviour, and an instance of a flow managed by a particular Akka actor corresponds to a reactor deployment. Finally, the actor that manages the data inputs and outputs of a flow typically has no other purpose than to process messages by propagating them through a flow, and to send messages to different actors which are the output of the flow. Hence, Akka Streams' actors that manage flows are conceptually similar to Stella's reactors. However, Akka Streams offers no guarantees that these actors only execute reactive code, i.e., they are subject to the Reactive Thread Hijacking Problem, the Reactive/Imperative Impedance Mismatch, and Acquaintance Maintenance remains a manual programmer effort.

7.5.2. Creek

Creek is a stream-based reactive programming library for Elixir in the domain of the Internet of Things. Similar to Akka Streams, Creek distinguishes between the application logic of streams, instances of streams, and the run-time processes (actors) that propagate values through the streams. Using our terminology, Creek's actors that manage the propagation of values through its streams fulfil the same

purpose as Stella's reactors. However, the code executed by Creek actors can contain a mix of imperative and reactive code, and they are subject to the problems discussed of this dissertation (cf. Table 7.1).

7.5.3. FrTime

FrTime is a reactive programming language that tightly integrates with Racket and its REPL. Racket's REPL is used by FrTime developers to interact with a running FrTime program. To ensure that the user does not block the reactive program via expressions that are evaluated in the REPL, FrTime makes use of multi-threading. A dedicated program thread manages the reactive engine while the main program thread is responsible for the GUI and REPL.

Developers may enter expressions into the REPL to imperatively change the values of source nodes of the DAG. When doing so, from FrTime's implementation we distilled that the main thread asynchronously "sends" (via a message) these new input values to the reactive thread. The reactive thread has a mailbox that continuously dequeues and processes messages. Conversely, a programmer may monitor the value of a signal by entering its name in the REPL, which displays the value of the signal in the GUI. Behind the scenes a dependency is created from the REPL thread to the reactive program thread, such that the value of the specified signal is continuously "sent" to the REPL thread as it updates over time. Hence, using our terminology, both the REPL thread can be seen as an actor, and the reactive program thread as a reactor. The REPL actor sends new values to the reactor to modify its input signals, and it may monitor the output signals of the reactor. However, the interaction between these different program threads is not part of the FrTime programming model, leading to (among other problems) the Reactive/Imperative Impedance Mismatch.

7.5.4. XFRP

XFRP is an actor-based reactive programming language for distributed applications that compiles to Erlang code. In essence, the reactive program is compiled to one or more Erlang actors which contain the necessary logic to propagate values through a DAG, i.e., using our terminology they are reactors. Since XFRP is purely functional, I/O code is not written in XFRP itself, but using Erlang actors that send asynchronous messages to the reactors in order to start a propagation turn. In the reverse direction, whenever the output signals of the reactive program update, the new values are sent to Erlang via an asynchronous message. However, XFRP is subject to the Reactive Thread Hijacking Problem and the Acquaintance Maintenance Problem (cf. Table 7.1). Furthermore, in this case the imperative actors are written in a different language than XFRP itself, namely Erlang. A benefit of integrating both actors and reactors in the same language is that it

enables them to share abstract data type definitions, e.g., Stella's classes which are shared between actors and reactors. Consequently, Stella's actors and reactors operate on the same data structures with the same methods and routines.

7.6. Summary and Conclusion

In this chapter we provided extensive evidence that the the problems discussed in Chapter 4 pop up in related work. Any reactive programming language or framework that is suitable for building distributed reactive programs for open networks has to provide a solution for:

Reactive Thread Hijacking Problem (RTHP) Weak reactivity is incompatible with our philosophy of reactive systems. Hence we advocate for any reactive language or framework to offer at least *some* guarantees with respect to their reaction time, i.e., **eventual or strong** reactivity. Stricter enforcement will result in reactive programs that have stronger guarantees with respect to the processing of their input. However, the trade-off is that the types of programs that can be written is also reduced.

Reactive/Imperative Impedance Mismatch (RIIM) The reactive and imperative programming paradigms are fundamentally incompatible, and their code should not be mixed to **completely avoid** the Reactive/Imperative Impedance Mismatch. Furthermore, language designers should be careful when adding a foreign function interface into their language, as allowing reactive programs to arbitrarily execute foreign functions may accidentally (re)introduce the problems of the Reactive/Imperative Impedance Mismatch.

Acquaintance Discovery (AD) Extensionally enumerating all possible acquaintances is impossible by definition of an open network. Hence, any reactive language or framework for open networks requires an **intensional** acquaintance discovery mechanism.

Acquaintance Maintenance (AM) Acquaintances are expected to continuously appear and disappear, and the reactive program should react to those events **correctly and efficiently**. Hence, a built-in acquaintance maintenance mechanism that does so is essential.

Based on our assessment of related work, to the best of our knowledge, Stella is the only language that meets these requirements.

8. Mira: A Meta Specification of Reactors in Stella

Stella's implementation spans 23064 lines of code and is not an accessible way to reproduce the research. Instead, we adopt the approach that is used by Abelson & Sussman [1] for Scheme, and we implemented (a part of) Stella within Stella itself. This meta implementation is called Mira, which is a meta implementation of reactors using actors that specifies the semantics of reactors. In particular, Mira sheds light on:

1. how the program text is compiled to a reactor behaviour, i.e., a Directed Acyclic Graph (DAG),
2. the semantics of spawning reactors, and the different steps involved in making the reactive program react to incoming values,
3. and the reactive engine that every reactor has, which schedules and propagates values without causing glitches.

Mira supports the complex features of reactors, which are the most difficult aspects of Stella to reproduce in another language or framework.

Compared to Stella's implementation of reactors which spans 4683 LOC, the complete implementation of Mira is 889 lines of Stella code (excluding comments and blank lines). Still, we cannot show the implementation in its entirety in the course of a chapter. However, we will explain the main parts of its architecture using code, and further highlight other parts of the implementation through diagrams. Mira in combination with this dissertation should allow the reader to reproduce Stella's reactors.

Mira's complete code is included in Appendix C (page 191).

We preface the implementation of Mira with a brief overview of Stella's implementation.

8.1. Preface: The Implementation of Stella

Stella is implemented as a parser, analyser, and interpreter in TypeScript. Over the years Stella's implementation has grown to 23064 lines of code (without blank lines, comments, and generated code files) spread over 443 files. Stella can be run in standard web browsers and Node.js [49] on laptops, desktop computers and servers. The smallest device that we tested is a Raspberry Pi Model 3B[1] used in Chapter 6, which currently costs around €38.

Stella's interpreter is written in Continuation-passing Style (CPS) and is based on the CPS Scheme interpreter (with trampolines) presented by Friedman & Wand [50]. This approach facilitates the concurrent execution of actors and reactors on a target platform (JavaScript) where concurrency is otherwise difficult to achieve. More specifically, it is important for Stella that any (potentially infinite) computation performed a process such as an actor cannot block reactors. Therefore the interpreter must be able to preempt the execution of processes, which is made possible by using a CPS interpreter.

Most of Stella's 23064 lines of code (LOC) are devoted to implement said CPS interpreter. They can be broken down as follows:

Parser Stella's parser is written using the PEG.js parser generator [82]. The parser that is generated by PEG.js spans around 10980 LOC, which we did not include when counting Stella's LOC.

Analyser Before executing a Stella program we analyse the parsed AST to check for undefined variable references, and to compile actor behaviours, reactor behaviours, and classes. The analyser contributes 2693 LOC (82 files).

Interpreter Stella's interpreter spans 17349 LOC (319 files). We can further break down the lines of code in the following main categories which cover the interpreter's main functionality:

- Object-oriented base language: 2971 LOC (102 files).
- Native classes and foreign function interface: 7049 LOC (75 files).
- Actors and messages: 1036 LOC (20 files), excluding code that is shared with reactors (generic processes).
- Reactors: 4117 LOC (72 files), excluding code shared with actors and code to construct a DAG (556 LOC that are a part of the analyser).
- Networking and discovery: 913 LOC (7 files).

We briefly discuss in more detail what the categories of the interpreter encompass.

[1] A single-board computer with a Quad Core 1.2GHz Broadcom BCM2837 64bit CPU and 1GB of RAM.

8.1.1. Object-oriented Base Language, Native Classes, and Foreign Function Interface

The object-oriented base language concerns the definition of classes, objects, and all expressions that are evaluated synchronously, e.g., spawning actors and reactors, sending messages, creating objects, invoking methods, conditionals, etc.

The size-change termination algorithm discussed in Section 5.2.2 (page 70) is implemented on the level of classes. Whenever a routine is invoked, the interpreter tracks a call stack analogous to the original Scheme-based implementation by the algorithm's authors [94], which is used to check the size-change principle every time a routine is called. Anecdotally, we found that the Scheme implementation of the authors translates well to another language such as TypeScript if one can read past the (often verbose) Racket[2] macros used in the original implementation.

Stella's library of native classes was implemented on a "by need" basis. Although it looks extensive based on the high number of lines of code, currently implementing native classes (in TypeScript) requires a lot of boilerplate code. Furthermore, we often found ourselves writing very similar code for different classes (e.g., to support iterators), hence inflating the number of lines of code.

Stella's foreign function interface is implemented as a native class (of the type `JSObjectProxy`) that contributes only 236 LOC. The class's only purpose is to identify the desired operation based on the structure of the method call (i.e., field get, set, or function call, see Section 5.3.3 on page 77), and to use JavaScript's reflection API to execute the correct action on JavaScript objects.

8.1.2. Actors, Reactors and Messages

The implementation of actors includes the representation of actor behaviours, the actor itself which processes messages, and the necessary bookkeeping abstractions to manage an actor's exported streams and the streams that an actor is monitoring (i.e., publish-subscribe bookkeeping).

A main part of Stella's interpreter, both in terms of lines of code and the density of logic, is to support the features of reactors which were introduced in Chapters 5 and 6. These features include qualification expressions where a reactor can react to the values of other (re)actor's streams, reactor deployments where a reactor can deploy multiple instances of a reactor behaviour, and the topology-reactive operator `deploy-*` which dynamically creates and removes reactor deployments. Together with the analyser's logic to construct reactor behaviours, the logic to support reactors spans 4683 LOC. Compared to the other features of Stella we consider this part of the implementation to contain most of the "black box magic" for which there exist no well-known principles on how to implement them (compared

[2] A dialect of Scheme [46].

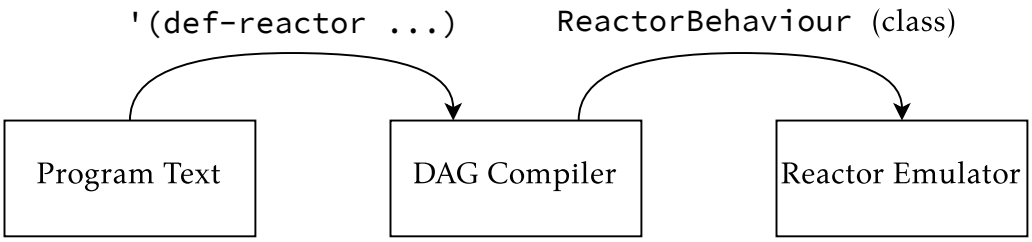


Figure 8.1.: General architecture of Mira.

to actors or an object-oriented language). To this end, Stella's implementation in 4683 LOC will not help either. Therefore, in this chapter we present Mira, an implementation of the same features which is not written in CPS, and which does not concern itself with other implementation details of an interpreter. In contrast to the TypeScript implementation of reactors, the Mira implementation spans 889 LOC.

8.2. Mira Architecture Overview

The general architecture of Mira is shown in Figure 8.1. It consists of 3 main parts, from left to right:

1. The program text of a reactor behaviour, specified as a Stella value.
2. A compiler transforms the program text to Mira's representation of a reactor behaviour. The different nodes of the DAG contain the logic of different types of expressions in the reactive program, e.g., routine invocations, qualifications, `deploy` expressions, etc.
3. An actor is used to emulate a reactor. This actor turns Mira's representation of a reactor behaviour into a running reactive program. Just like Stella's reactors, the emulated reactor responds to messages sent by other (re)actors.

The `Main` actor in Listing 8.1 shows the different steps of Figure 8.1 but expressed in Stella. It emulates a simple `Plus` reactor that adds 2 numbers. The code contains the following parts:

The environment Line 3 defines a simple environment (a dictionary) that is used during the compilation process to store compiled reactor behaviours. Any subsequently compiled reactor behaviours (none in this example) can use this environment to lookup reactor behaviours when they are used at compile-time, e.g., in a `deploy` expression.

The program AST The "program text" of a reactor behaviour is an s-expression that contains symbols, numbers, booleans and strings. Stella's supports literal expressions of the form `'(...)`, which is shorthand for constructing a `Vector` object. For example, the program text specified on line 4 corresponds to the

```
1  (def-actor Main
2    (def-constructor (start env)
3      (def env (newd Dictionary))
4      (def program '(def-reactor (Plus x y) (out (+ x y))))
5      (def behaviour (new ReactorBehaviour 'compile program env))
6      (def reactor (spawn-actor! Reactor 'init behaviour))
7
8      // change x and y sources
9      (send! reactor 'react! '(1 1))
10     (send! reactor 'react! '(2 2))
11     (send! reactor 'react! '(3 3))))
```

Listing 8.1: Running a `Plus` reactor in Mira.

usual Stella syntax for a `Plus` reactor behaviour. All identifiers in the body of the expression (e.g., `def-reactor`, `Plus`, …) are symbols in the resulting vector (`'def-reactor`, `'Plus`, …). Numbers, booleans and strings remain present in the vector as-is, and nesting parenthesis creates nested vectors, thus producing a reactor behaviour's Abstract Syntax Tree (AST).

The meta reactor behaviour The class `ReactorBehaviour` is Mira's representation of a reactor behaviour, and it also implements the logic to construct DAGs. Line 5 instantiates the class using the `compile` constructor that accepts a program's AST and environment as input, and transforms the AST to a DAG that is stored within the instantiated `ReactorBehaviour` object.

The emulated reactor A reactor is emulated by an actor. The actor behaviour that implements the logic of the emulation is called "`Reactor`". Line 6 spawns this actor using its `init` constructor. The constructor accepts 1 argument, namely the aforementioned `ReactorBehaviour` object. This will serve as the root deployment of the emulated reactor.

Propagating values To propagate values, the emulated meta reactor accepts `'react` messages[3]. For example, lines 9 to 11 send various values to the reactor. The sole value of a `'react` message is a `Vector` that contains the new values for the input signals. The vectors contain 2 values since the `Plus` reactor in this example has 2 input signals, namely `x` and `y`.

In the following sections we will explain the core parts of Mira that turn the program text to a running program. Section 8.3 explains how the program text is transformed to a DAG. Section 8.4 explains how reactors are emulated using actors, and in Section 8.5 we further detail the reactive engine that is used to propagate values.

[3] Stella's reactors process `react-to!` messages instead of `react!` messages. We could not use `react-to!` in Mira for technical reasons, namely because "`react-to!`" is reserved as a keyword in Stella's current parser. This means that an actor cannot define a method to process `react-to!` messages.

```
1  (class ReactorBehaviour
2    (def-fields name all-nodes sources sinks env)
3
4    (def-constructor (compile sexp _env)
5      (def header (head (tail sexp)))
6      (def body (tail (tail sexp)))
7      (def footer (pop! body))
8
9      (def localenv (newd Dictionary))
10     (make-sources! #this localenv header)
11     (make-body! #this localenv body)
12     (make-sinks! #this localenv footer)
13     (put! env name #this))
14
15
16   (def-method (make-sources! localenv header)     ...)
17   (def-method (make-body!    localenv body)       ...)
18   (def-method (make-sinks!   localenv expression) ...)
19   (def-method (make-node!    localenv expression) ...)
20
21   // methods omitted: get-name, get-sources, get-sinks, get-all-nodes,
       make-define-node!, make-constant-node!, make-qualification-node!,
       make-application-node!, make-deploy-node!, make-deploy-star-node!
22 )
```

Listing 8.2: General structure of the `ReactorBehaviour` class in Mira.

8.3. Constructing the DAG

The `ReactorBehaviour` class represents a reactor behaviour in Mira and contains the necessary code to transform program text (specified as a `Vector`) to a DAG. The general idea of its implementation is shown in Listing 8.2. Every reactor behaviour has a number of local fields declared on line 2. These fields contain the following values:

`name` The name of the reactor behaviour

`all-nodes` A vector of all nodes in the DAG.

`sources` A vector of source nodes. Note that this vector only contains the nodes that correspond to the input signals, and not the implicit source nodes.

`sinks` A vector of sink nodes.

`env` The aforementioned environment that stores already compiled reactor behaviours.

The body of the `compile` constructor on lines 4 to 13 initialises those fields by deconstructing the program AST and creating the DAG of the reactive program. We will show the general idea of doing so.

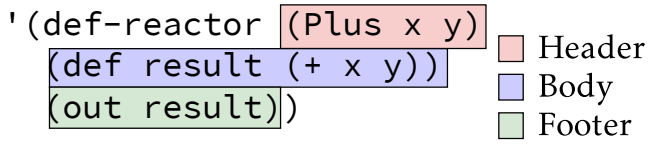


Figure 8.2.: Structure of an s-expression reactor behaviour in Mira.

The first part of constructing the DAG involves deconstructing the AST into its constituents. Listing 8.2 deconstructs the AST on lines 5 to 7 into a header, body and footer. For example, these parts are labelled in Figure 8.2 using a `Plus` reactor behaviour.

1. The header of the reactor behaviour is highlighted in (light) red. This is the second value in the AST's vector. It is referred to on line 5 by extracting the `head` (returns the first element) from the `tail` (returns a new vector with all elements except the first) of the program text[4]. The header itself is also a vector whose first value is the reactor behaviour's name, and the rest of the values are the input signals.
2. The body denotes all expressions between the header and the footer, in this case a single value definition highlighted in (lavender) blue.
3. The footer is the `out` expression that contains the output signals of the reactor behaviour. In this case there is a single output signal, namely `result`. It is extracted from the body via `pop!`, which is a destructive vector operation that removes and returns the last element of a vector.

Lines 9 to 12 turn the header, body and footer into a DAG. A local environment is used to map variables used within the reactor behaviour to their nodes, e.g., the input signals of the `Plus` reactor behaviour are called `x` and `y`, which will be stored in the local environment with the generated source node for those signals. Similarly, local variable definitions such as `result` are mapped to the DAG node that results from compiling the defined expression.

8.3.1. Compiling Expressions

We now present the implementation of the `make-sources!`, `make-sinks!`, `make-body!` and `make-node!` methods that were omitted from Listing 8.2. They show how the various parts of the AST are transformed to connected DAG nodes.

[4]The complete interface of Stella's `Vector` can be found in Table 5.2 on page 78.

```
1 (def-method (make-sources! localenv header)
2   (set! sources (newd Vector))
3   // (for/iterable (variable iterable-obj) body ...)
4   (for/iterable (signal-name (tail header))
5     (def node (newd SourceNode signal-name))
6     (put! localenv signal-name node)
7     (push! sources node)
8     (push! all-nodes node)))
```

Listing 8.3: The implementation of a `ReactorBehaviour`'s `make-sources!` method.

Compiling Input Signals

The implementation of `make-sources!` is given in Listing 8.3. Essentially it loops over each input signal in the header using Stella's `for/iterable` loop. A `for/iterable` expression is used to iterate over all entires of a collection, such as a vector. It defines a variable to be used in each iteration, in this case `signal-name`, and the collection of values to iterate over, in this case the tail of the header[5]. The rest of the expressions are its body which is evaluated for each entry in the iterable object. The body creates a new `SourceNode` object for each input signal. This source node is stored in the local environment as well as the vectors `sources` and `all-nodes` (fields of the `ReactorBehaviour` class).

Compiling Output Signals

The implementation of `make-sinks!` is given in Listing 8.4. Its purpose is the same as `make-sources!` (but for sink nodes), and its implementation is very similar. There are 2 notable differences. Firstly, we keep a counter `sink-id` that is used for debugging purposes to identify the position of a sink node (first, second, …). Secondly, each expression in the footer besides the `out` keyword can contain arbitrary expressions such as routine invocations. Note that these expressions are still part of the reactor behaviour's body. They are compiled via a call to `make-node!` on line 5 which compiles body expressions. The return value is a node that supplies the values for the sink node.

Compiling Body Expressions

Listing 8.5 provides the implementation of `make-body!`. Compared to `make-sources!` and `make-sinks!` its implementation is simpler, because compiling the body amounts to creating a node for each expression in the body via the `make-node!` method.

[5]The first element is skipped because it is the reactor behaviour's name

```
1 (def-method (make-sinks! localenv footer)
2   (set! sinks (newd Vector))
3   (def sink-id 1) // for debugging purposes
4   (for/iterable (sink-expression (tail footer))
5     (def expr-node (make-node! #this localenv sink-expression))
6     (def sink-node (newd SinkNode sink-id expr-node))
7     (set! sink-id (+ sink-id 1))
8     (push! all-nodes sink-node)
9     (push! sinks sink-node)))
```

Listing 8.4: The implementation of a `ReactorBehaviour`'s `make-sinks!` method.

```
1 (def-method (make-body! localenv body)
2   (for/iterable (expression body)
3     (make-node! #this localenv expression)))
```

Listing 8.5: The implementation of a `ReactorBehaviour`'s `make-body!` method.

The `make-node!` method is a recursive method that compiles all possible expressions in the body of a reactor behaviour. Its implementation is shown in Listing 8.6. There are 2 formal parameters: `expression` corresponds to a part of the program text (e.g., `(def a b)`), and `environment` is the aforementioned dictionary for local variables definitions (a mapping of symbols to DAG nodes).

Expressions and sub-expressions are compiled similar to how a metacircular evaluator of Scheme evaluates S-expressions [2]. In essence, it dispatches on the type of the expression and recursively compiles any constituent expressions. The conditional expression in Listing 8.6 implements this dispatch based on the type of expression[6]. For example, when the expression is a symbol (i.e., a variable in the body of the program text), then line 4 looks up the corresponding node in the environment. Otherwise, methods such as `make-constant-node!` (line 10) and `make-application-node!` (line 25) will process the given expression.

Note that Mira does not support the dot-notation for qualification expressions that is used by Stella. Instead, as shown on line 14, we use an expression whose first value is the symbol `'qualification`, its second value is an expression which yields the origin of the stream, and the third value yields the name of the stream.

We will not show the compilation of all types of expressions, because in all cases the code is quite tedious. It essentially boils down to traversing the program's AST, compiling any subexpressions via a call to `make-node!`, and instantiating the correct type of DAG node. For example, the compilation of a routine invocation is shown in Listing 8.7. A routine invocation is an expression of the form `(selector obj ...args)` where the selector denotes the routine to invoke, `obj` is the receiver

[6]The syntax of `cond` in Stella is shared with the `cond` expression in Scheme [3]

```
(def-method (make-node! environment expression)
  (cond
    // identifier: lookup identifier in env
    ((eq? (type-of expression) 'Symbol)
     (get environment expression))
    // constants
    ((or (eq? (type-of expression) 'Number)
         (eq? (type-of expression) 'String)
         (eq? (type-of expression) 'Boolean))
     (make-constant-node! #this environment expression))
    // (def var expr)
    ((eq? (first expression) 'def)
     (make-define-node! #this environment expression))
    // (qualification expr stream-name)
    ((eq? (first expression) 'qualification)
     (make-qualification-node! #this environment expression))
    // (deploy behaviour-name ...exprs)
    ((eq? (first expression) 'deploy)
     (make-deploy-node! #this environment expression))
    // (deploy-* behaviour-name expr)
    ((eq? (first expression) 'deploy-*)
     (make-deploy-star-node! #this environment expression))
    // (symbol obj ...args)
    ((eq? (type-of (first expression)) 'Symbol)
     (make-application-node! #this environment expression))
    (else (println! "Unsupported expression: " expression))))
```

Listing 8.6: The `make-node!` method of the `ReactorBehaviour` class dispatches on the type of expression to compile the various types of DAG nodes.

```
1  (def-method (make-application-node! environment expression)
2    (def operator (first expression))
3    (def operands (tail expression))
4    (def operand-nodes (newd Vector))
5    (for/iterable (operand operands)
6      (push! operand-nodes (make-node! #this environment operand)))
7    (newd ApplicationNode operator operand-nodes))
```

Listing 8.7: The `make-application-node!` method compiles a routine invocation to an `ApplicationNode`.

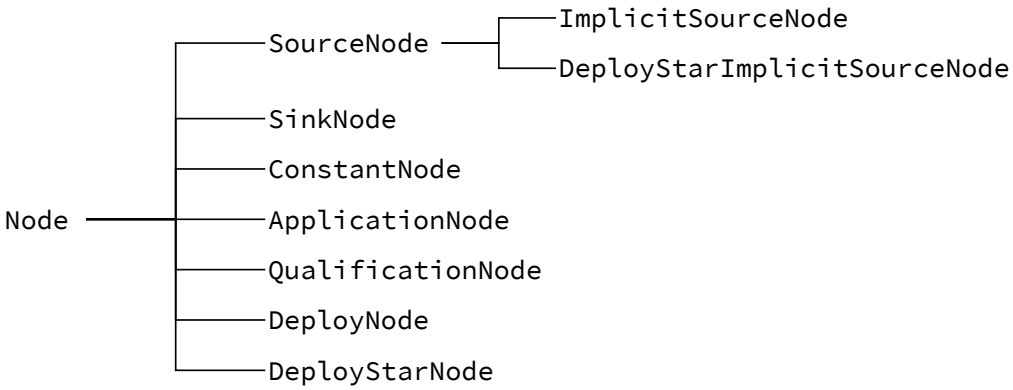


Figure 8.3.: Class hierarchy of DAG nodes in Mira.

object, and the rest are the arguments. These values are extracted from the vector on lines 2 to 3. Since the receiver object and the invocation arguments are subexpressions, they are compiled on line 6 via a recursive call to `make-node!`. Finally, a new `ApplicationNode` is created on line 7 with the operator and compiled receiver object and arguments as input.

8.3.2. The Node Class

Every node in the DAG performs a specific function that is related to the application logic of the expression. E.g., a node of type `ApplicationNode` performs a routine invocation every time the receiver object changes, or when any of the argument values changes. The complete class hierarchy of DAG nodes is shown in Figure 8.3.

In general, all types of DAG nodes share 2 main methods that are implemented in the `Node` superclass:

1. Every node stores its dependencies (nodes that supply a node's data) and dependents (nodes that receive a node's data).
2. Nodes can be ordered via a < method.

```
1  (class Node
2    (def-fields name id height dependencies dependents)
3
4    (def-constructor (default _name _dependencies)
5      (set! name _name)
6      (set! id (make-uuid (newd Random)))
7      (set! dependencies _dependencies)
8      (set! dependents (newd Vector))
9
10     (def computed-height 1)
11     (for/iterable (dependency dependencies)
12       (add-dependent! dependency #this)
13       (set! computed-height
14         (max computed-height (get-height dependency))))
15     (set! height (+ computed-height 1)))
16
17   (def-routine (< other-node) (< height (get-height other-node)))
18   (def-method (compute! deployment) #true)
19
20   // methods omitted: get-id, get-name, get-height, get-dependents,
       add-dependent!, get-dependencies, collect-dependency-values, is-source?,
       to-string, is-computable?
21   )
```

Listing 8.8: The `Node` class in Mira is the superclass of all DAG nodes.

Additionally, every subclass of `Node` should provide a `compute!` method that recomputes the current value of a node, which depends on the type of node.

The excerpt of `Node` in Listing 8.8 presents the essence of how nodes are used. The constructor on line 4 accepts a name (for debugging purposes) and a vector of dependencies, i.e., an ordered list of nodes on which the new node depends. The `for/iterable` loop on lines 10 to 15 performs 2 tasks. First, it registers the current node as a dependent of all nodes in the `dependencies` vector (line 12). Second, it computes a *height* of the current node in the DAG, which will be used to topologically sort them (see Section 2.1.5 on page 17). We will further discuss node height in Section 8.5, where we discuss Mira's strategy to propagate values through the DAG.

8.3.3. The ApplicationNode Subclass

Every subclass of `Node` overrides the `compute!` method that recomputes the value of a node of that class. For brevity we will only show the implementation of `ApplicationNode` that represents a routine invocation in the DAG. All other subclasses of `Node` are implemented in a similar way, but often with more elaborate implementations (e.g. for `deploy-*`).

The complete implementation of `ApplicationNode` is given in Listing 8.9. The `default` constructor accepts 2 arguments:

1. The first argument `op` is the name of the method to invoke, which is expected to be a symbol.
2. The second argument `ops` is a vector of operands, where each operand is a DAG node that is the result of compiling the corresponding expression in the program text.

The body of the constructor initialises the superclass with its `default` constructor and 2 arguments, namely the node's name (for debugging purposes) and a vector of nodes on which the `ApplicationNode` depends (the operands).

The `compute!` method on lines 11 to 15 has exactly 1 argument, namely the reactor deployment which the `ApplicationNode` is a part of when it is deployed in a reactor. Essentially, access to the reactor deployment allows a node to retrieve the most recently computed value of other nodes, such as its dependencies. Additionally, although it is not used by `ApplicationNode`, the reactor deployment can also be used to store additional run-time information that a node needs to operate correctly (e.g., created deployments in the case of `deploy` and `deploy-*`). This information must be stored in the reactor *deployment* because when the same reactor behaviour (which includes the DAG nodes) is used multiple times, each instance has a different run-time state.

The body of `ApplicationNode`'s `compute!` method is straightforward. First, line 12 retrieves a vector of the latest computed values of each of the dependencies[7]. Next, the receiver object and the arguments are extracted from the vector. Finally, on line 15, the `apply` method of the operator (a symbol) is called. Its first argument denotes the receiver object on which the method will be called, and the second argument is a vector of arguments of the method call.

8.4. Emulating Reactors

An actor is used to emulate a reactor. Concretely, Mira has an actor behaviour called `Reactor` that implements all of the logic that is usually hidden by the implementation of Stella.

The general architecture of an actor spawned with the `Reactor` behaviour is depicted in Figure 8.4. We depict a single instance (an actor) that processes asynchronous messages from its mailbox (top left). Figure 8.4 also depicts (as rectangles) the objects referred to by the actor and their interactions to propagate values through the DAG. Whenever the actor behaviour processes a message, this

[7] Note that, due to the order in which the reactive engine computes nodes, all dependency nodes are guaranteed to be updated to the latest value before the `compute!` method of the current node is called.

```
1  (class ApplicationNode
2    (extends Node)
3    (def-fields operator)
4
5    (def-constructor (default op ops)
6      (super 'default
7             (append "<ApplicationNode " (to-string op) ">")
8             ops)
9      (set! operator op))
10
11   (def-method (compute! deployment)
12     (def dependency-values (collect-dependency-values #this deployment))
13     (def receiver (head dependency-values))
14     (def arguments (tail dependency-values))
15     (apply operator receiver arguments)))
```

Listing 8.9: The `ApplicationNode` class in Mira.

leads to the invocation of various methods in the (object-based) implementation of a reactor.

In this section we show the implementation of the actor behaviour called `Reactor` and the classes depicted in Figure 8.4. They will show the different steps involved in processing a `react!` message from the mailbox of the actor. The next section (Section 8.5) will further detail the reactive engine which implements the DAG propagation algorithm.

8.4.1. The "Reactor" Actor Behaviour

The `Reactor` actor behaviour is responsible for all asynchronous interactions with other actors and reactors. Its 3 main tasks are:

1. to process `react!` messages, i.e., to initiate the propagation of values through the emulated reactor,
2. to handle subscribing and unsubscribing from streams, which are indicated by qualification expressions in the body of the emulated reactor, and
3. to emit the output of the reactor that is being emulated.

The implementation of `Reactor` is given in Listing 8.10. It declares 1 stream called `output` (equivalent to the `out` stream of Stella's reactors), and 2 local fields: `sync-reactor` stores an instance of the aforementioned `SynchronousReactor` class, and `subscription-handles` is a dictionary that stores the subscriptions to streams that are created by the emulated reactor. The `init` constructor on line 5 expects a `ReactorBehaviour` object as input and initialises those local fields.

Once the actor is spawned, it accepts 3 messages.

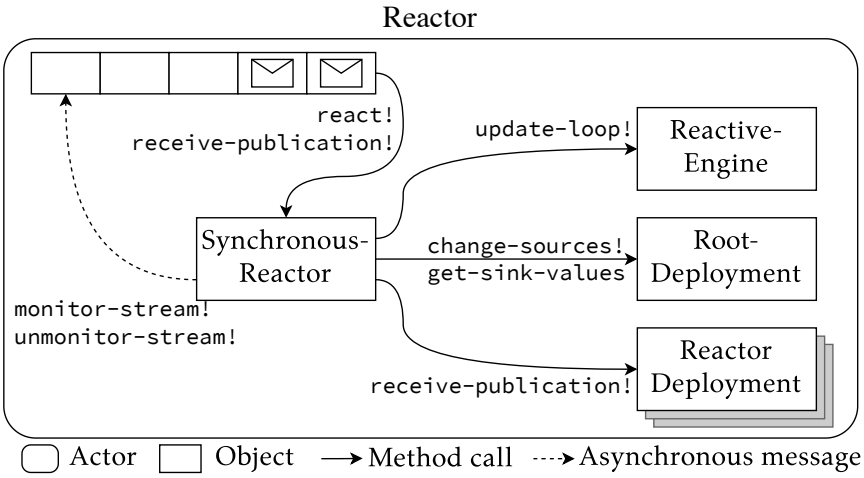


Figure 8.4.: Overview of a simulated reactor's synchronous and asynchronous interactions.

```
1  (def-actor Reactor
2    (def-stream output 1)
3    (def-fields sync-reactor subscription-handles)
4
5    (def-constructor (init behaviour)
6      (set! sync-reactor (newd SynchronousReactor behaviour))
7      (set! subscription-handles (newd Dictionary)))
8
9    (def-method (react! new-values)
10     (emit! output (react! sync-reactor new-values)))
11
12   (def-method (monitor-stream! id stream)
13     (def wrapper (spawn-actor! StreamWrapper 'init stream id))
14     (def handle (monitor! wrapper.output 'receive-publication!))
15     (put! subscription-handles id handle))
16
17   (def-method (receive-publication! id value)
18     (emit! output (receive-publication! sync-reactor id value)))
19
20   // method omitted: unmonitor-stream!
21 )
```

Listing 8.10: The actor behaviour "`Reactor`" emulates reactors.

`react!` Sending a `react!` message to an emulated reactor is the Mira equivalent of using `react-to!` in Stella. Whenever the emulated reactor receives such a message, it is processed by the `react!` method on line 9 which synchronously propagates the given values through the DAG. The updated output of the DAG is emitted to the emulated reactor's `output` stream.

`monitor-stream!` Whenever a qualification expression is used within the reactive program, the DAG's `QualificationNode` establishes a subscription to the given stream. Establishing the subscription is asynchronous (just like in Stella's reactors). To initiate the process, the `QualificationNode` sends a `monitor-stream!` message to the current actor.

The `monitor-stream!` method on line 12 accepts 2 arguments, namely a unique identifier and the stream to monitor. Since, in general, there is no way to identify the origin of an emission, the given `id` is manually added by Mira to each publication. Essentially, the stream is wrapped by a new `StreamWrapper` actor that echoes the values emitted by the given stream together with the `id`. A "handle" of the subscription is stored such that the subscription can later be cancelled (and the wrapper actor killed) when the subscription is stopped by the same `QualificationNode` (omitted from Listing 8.10 for brevity).

`receive-publication!` Stream emissions will be received as a `receive-publication!` message (line 17). The new values are synchronously propagated through the DAG, and the new output of the reactor is emitted to the emulated reactor's `output` stream.

8.4.2. The "SynchronousReactor" Class

The `SynchronousReactor` class tracks reactor deployments and prepares the synchronous propagation of values through the DAG. Its implementation is shown in Listing 8.11, which we briefly summarise.

The `default` constructor on line 4 accepts a reactor behaviour which was given when spawning the emulated reactor. The body initialises a reactive engine and deploys the behaviour by instantiating Mira's `ReactorDeployment` class[8]. Finally, every reactor deployment is tracked in a `Vector` that initially contains the root deployment. Two bookkeeping methods that we omitted from Listing 8.11 are used by the DAG nodes (namely those generated for `deploy` and `deploy-*` expressions) whenever deployments are added or removed, such that the `SynchronousReactor` always has a reference to every deployment within the emulated reactor.

[8] We will not further detail the various constructor arguments given to `ReactorDeployment`, which are used to track dependencies between reactor deployments.

```
(class SynchronousReactor
  (def-fields reactive-engine root-deployment deployments)

  (def-constructor (default behaviour)
    (set! reactive-engine (newd ReactiveEngine))
    (set! root-deployment
      (newd ReactorDeployment
            behaviour reactive-engine #this #false #false))
    (set! deployments (newd Vector root-deployment)))

  // omitted methods: add-deployment!, remove-deployment!, monitor-stream!,
    unmonitor-stream!

  (def-method (react! new-values)
    (change-sources! root-deployment new-values)
    (update-loop! reactive-engine)
    (get-sink-values root-deployment))

  (def-method (receive-publication! subscription-id value)
    (for/iterable (deployment deployments)
      (receive-publication! deployment subscription-id value))
    (update-loop! reactive-engine)
    (get-sink-values root-deployment)))
```

Listing 8.11: The SynchronousReactor class.

The `react!` and `receive-publiction!` methods are responsible for modifying the value(s) of source node(s), starting a propagation turn, and returning the new values of the sink nodes.

`react!` The `react!` method on line 13 changes the source nodes of the root deployment. In this case a vector of new values is provided, where each value in the vector corresponds to the new value for a source node. For example, the `Plus` reactor behaviour in Figure 8.2 on page 155 has 2 source nodes, so a possible vector of values is `'(1 2)`. Line 14 notifies the root deployment to change the value of its sources, and line 15 propagates those values by starting the update loop of the reactive engine. The return value of `react!` are the values of the sink nodes of the root deployment.

`receive-publication!` The `receive-publication!` method on line 18 is invoked by the `Reactor` actor behaviour whenever a stream to which the emulated reactor is subscribed emits a new value. On line 19, every deployment in the reactor is notified of the publication, which consists of an `id` and a value. Each deployment will decide whether a publication with the specified `id` is destined for a node in their DAG, and if not they simply ignore it. Note that we notify each deployment only to simplify Mira's code. The inefficiency is easily removed via additionally bookkeeping (e.g., using a dictionary) that tracks which deployment should receive publications with the given identifier.

8.4.3. The "ReactorDeployment" Class

Reactor deployments are implemented via the `ReactorDeployment` class. As introduced in Section 5.5.4 on page 88, a reactor deployment stores all run-time state of a reactor behaviour. For brevity we will not show the implementation of `ReactorDeployment` because it is mostly bookkeeping code. However, we will detail the information that is stored by a reactor deployment and the tasks it performs.

Every reactor deployment tracks the following information.

Reactor behaviour The reactor behaviour of which the reactor deployment is an "instance".

Node values Every node in the reactor behaviour has a value that is tracked by the reactor deployment (initially `#undefined`). This value is updated whenever a node is recomputed.

Stream subscriptions Nodes in the DAG can create subscriptions to streams, e.g., whenever the DAG contains a qualification node. Since these subscriptions can be different for each use of a reactor behaviour, they are tracked by the reactor deployment. More specifically, for any incoming publication (containing an `id` and a `value`), the reactor deployment can look up which node in the DAG is affected by the publication, and schedule this node to recompute

its value. As a concrete example, remember that a qualification expression is compiled to 2 nodes, namely a `QualificationNode` that manages the subscription, and an `ImplicitSourceNode` that receives the values emitted by the stream. The `QualificationNode` will register its subscriptions with the reactor deployment, thereby indicating to the reactor deployment that any publications received from the subscription should change the value of the implicit source node. Whenever such a publication is received, the implicit source node is scheduled for recomputation.

Cross-deployment dependents Besides dependencies between the nodes of the DAG (which are stored by the nodes themselves), the `deploy` and `deploy-*` expressions introduce dependencies between nodes of *different* reactor deployments. Both dependencies to other reactor deployments as well as dependent reactor deployments are tracked. To more easily discuss why this is so, we will call the reactor deployment that creates a new deployment the *deployer*, and the created reactor deployment is called the *deployee*.

The relation between the nodes of the deployer and the deployee is depicted in Figure 8.5. Whenever new values are propagated through the deployee, then the deployer will be updated as well. The internal mechanism that causes data to propagate from the deployee to the deployer is the registering of these so-called cross-deployment dependencies. While the concrete behaviour behind these deployments is unimportant, `Deployee` stores the relationship between its sink `x` and the implicit source `y` in `Deployer`. Thus, whenever the value of `x` changes, then this value is propagated to node `y` in `Deployer`. Note that the relation is also tracked in the reverse direction, which is necessary to clean up any data structures when a reactor deployment is removed.

Priority vector Every deployment stores a so-called *priority vector*. As we will discuss in Section 8.5, this is deployment-specific information to determine a correct topological order between the DAG nodes of *different* deployments.

8.5. The Reactive Engine

A reactive engine drives the propagation of values through the reactive program.

Recompute nodes: Whenever the value of a node in the DAG changes, then the reactive engine should also recompute the dependent nodes. E.g., when a source node is updated, the reactive engine recompute its dependents, the dependents of the dependents and so forth, until every affected node in the DAG has been updated to the latest program state. The process stops at the sink nodes which have no dependents. Similar to other reactive languages, the order in which nodes are recomputed should respect the topological order of the DAG to prevent glitches.

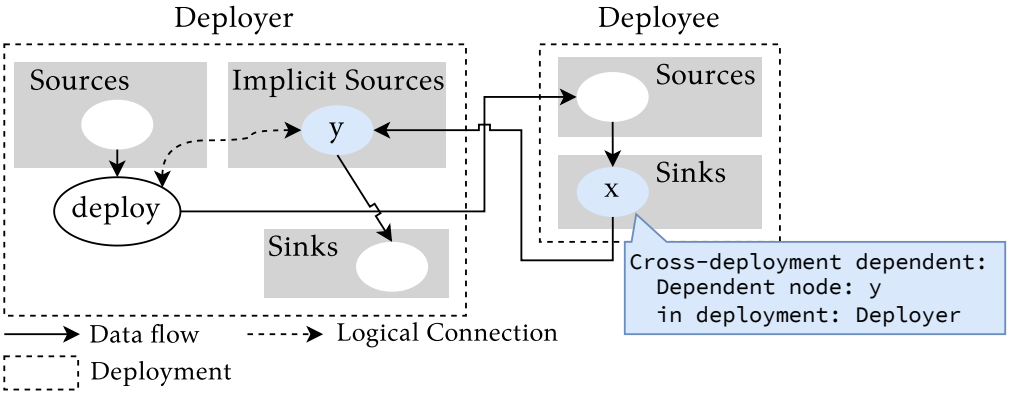


Figure 8.5.: Depiction of cross-deployment dependencies.

We will first recap the basic strategy used by other reactive programming languages and frameworks in Section 8.5.1 (previously introduced in Section 2.1.5 on page 16). We then adapt this strategy to work with Stella's reactor deployments in Section 8.5.2, and finally we show the implementation of the reactive engine in Section 8.5.3.

8.5.1. A Conventional DAG Propagation Algorithm

We discussed glitches in Section 2.1.5 (page 16). In summary, a glitch is caused when the value of a DAG node is computed multiple times during a single propagation cycle, and the strategy to prevent glitches is to delay the recomputing of a node as long as possible, i.e., until *all* of its predecessors (the dependencies) have been recomputed. To this end, reactive programming languages usually use a smart propagation algorithm that prevents glitches. For example, REScala [123] and Flapjax [88] use a propagation algorithm based on FrTime [31], where glitches are avoided by assigning a height to every node in the DAG. We previously showed that Mira also assigns such a height to each node during the construction of the DAG. For example, Figure 8.6 draws the DAG of a `Plus` reactor behaviour that adds 2 numbers, where the number besides each node depicts its (statically computed) height in the DAG.

Nodes can be sorted at run-time by scheduling them in a priority queue, where the priority is determined by the height assigned to the node. This means that the recomputation of a node is naturally delayed until all nodes with a higher priority (i.e., its dependencies) have already been processed. However, this propagation algorithm no longer prevents glitches in the context of Stella (and Mira) with reusable reactor behaviours and reactor deployments. We will first show the problem at hand, and then formulate a solution.

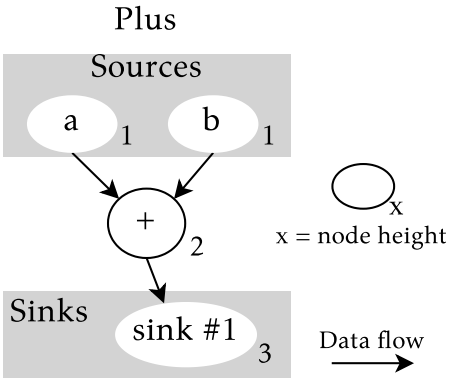


Figure 8.6.: The statically computed heights of nodes in the DAG of a `Plus` reactor.

Consider the DAG of an arbitrary reactive program depicted in Figure 8.7, where every node is labelled with its statically computed height in the DAG of their reactor behaviour (note that the dashed-line boxes are deployments of two different reactor behaviours). The DAG of `Deployer` contains a `deploy` expression that results in `Deployee` (the double-bordered node on the blue path). As we previously showed in Section 5.6.2 (page 98), a `deploy` expression is split up in a node that manages the created deployment, and an implicit source node that is responsible for receiving the values from the sink of the deployee. When nodes are recomputed according to their height in the DAG, a glitch occurs in the node highlighted in red. Concretely, its computation is executed two times instead of only once. The first computation is redundant and should be avoided. As a visual aid, we highlighted the two paths of execution in red and blue that both cause an update of the red node at different moments in the propagation cycle.

Nodes with the same height (priority) may be executed in an arbitrary order, and a lower number denotes a higher priority. The problem occurs when the source node of deployment `A` has a new value, and gradually all dependents on the red and blue paths are scheduled. Since the red node has a priority of 3, it is computed once after all nodes of priority 2 have been computed. However, via the blue path it indirectly depends on a sink node with the lower priority 4. Whenever this sink node changes, it will schedule its dependents for recomputation, and eventually the red node is recomputed once again.

8.5.2. Stella's DAG Propagation Algorithm

Stella's (and thus Mira's) propagation algorithm is still based on a priority queue, but we modify the way priorities are determined to also take into account a correct ordering of nodes across multiple reactor deployments.

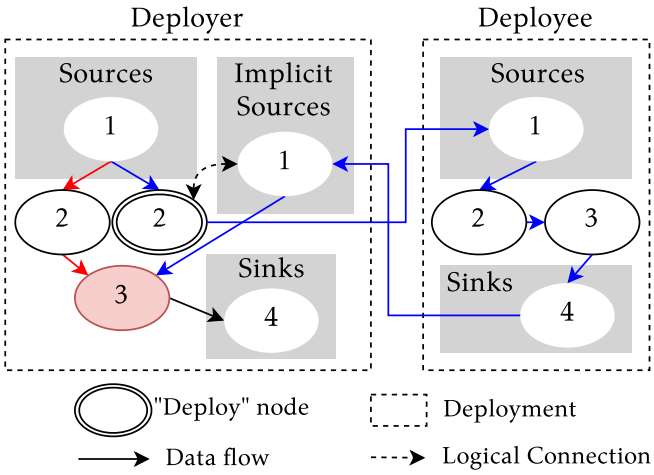


Figure 8.7.: Example of 2 reactor deployments where using the node height as priority can causes glitches.

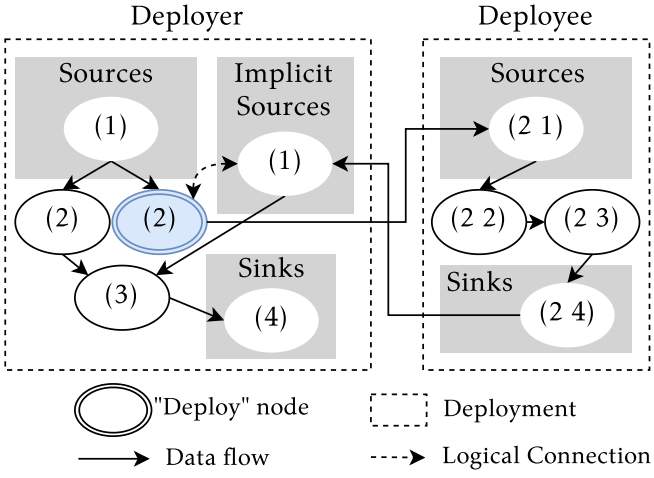


Figure 8.8.: Example of 2 reactor deployments where using priority vectors can prevent glitches.

```
1 (def-routine (priority<? priority1 priority2)
2   (if (or (empty? priority1) (empty? priority2))
3       (< (length priority) (length priority2))
4       (or (< (first priority1) (first priority2))
5           (priority<? #this (tail priority1) (tail priority2)))))
```

Listing 8.12: Routine to compare priority vectors.

Our solution is to ensure that there is a "globally" (within the same reactor) correct ordering between nodes of different reactor deployments. Instead of using a single number (height) to represent priority, priorities now consist of a vector of heights which includes the priority of the predecessing deployment(s). Consider the diagram in Figure 8.8, which is the same diagram as Figure 8.7, but using our system of assigning priorities. If we assume `Deployer` is the root deployment of the reactor, then the priorities of its nodes are now represented by singleton vectors with the same heights as Figure 8.7. Whenever a new deployment is created, such as `Deployee` which is created by the blue `deploy` node, then the priorities of all nodes in the deployee are prefixed with the `deploy` node's priority. In this case the prefix is the vector `'(2)`, which is added in front of the statically computed heights of the deployee's DAG.

To determine the order of nodes in the priority queue, priority vectors are compared left-to-right according to the Stella routine in Listing 8.12. The routine `priority<?` returns whether its first argument `priority1` has a higher priority than its second argument `priority2`.

- When either of the priority vectors is empty, then the smallest of the vectors should be ordered earlier in the priority queue. For example, this means nodes such as the aforementioned `deploy` node with priority `'(2)` take precedence over all of reactor deployment `B`'s nodes, e.g., the source node `'(2 1)`.
- When both priority vectors contain values, then those values are compared pairwise from left to right. The vector `priority1` has a higher priority whenever a height is found that is smaller than the height on the same index in `priority2`. For example, the vector `'(3 4 2)` has a higher priority than `'(3 5 2)` because the height on index 1 in `priority1` is smaller the height on index 1 in `priority2`.

Note that the length of the vector depends on how deep reactor behaviours are nested. While most of the reactor behaviours presented in this dissertation are shallow, it remains future work to optimise the comparison of priorities for deeply nested reactor behaviours.

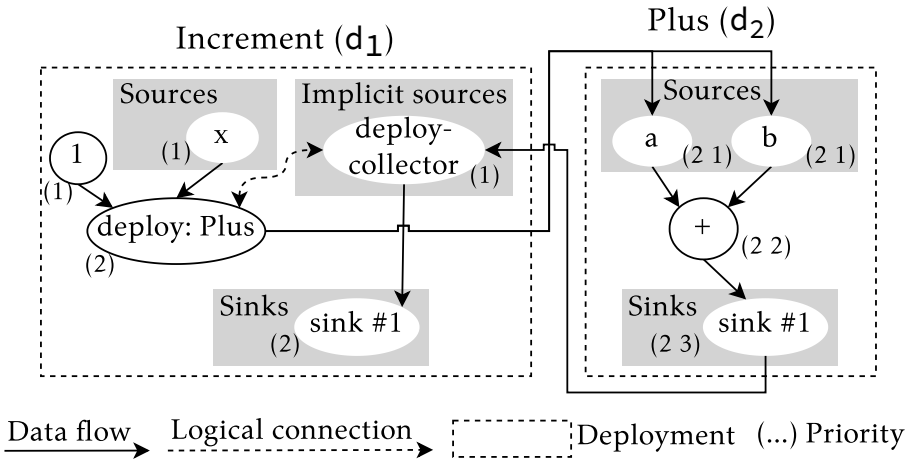


Figure 8.9.: Example of 2 reactor deployments, where an `Increment` reactor behaviour (d_1) uses a `Plus` reactor behaviour (d_2) to increment its input.

8.5.3. The Reactive Engine's Implementation

The implementation of the reactive engine is given in Listing 8.13, which we will explain throughout this section. It defines 2 local fields on line 2, which are initialised in the constructor (not shown in Listing 8.13). The aforementioned priority queue is stored in the `pq` field. Since our implementation of a priority queue does not contain methods to efficiently retrieve entries in the queue, we store an additional dictionary in the `scheduled-nodes` fields that will store all entries that are currently scheduled in the priority queue by a unique `id`.

We will discuss each of the methods in the body of `ReactiveEngine`, which are responsible for the following logical steps that comprise the reactive engine's update loop:

1. Scheduling nodes
2. Recomputing nodes
3. Storing node values and scheduling dependents

We explain each of these steps by using the example of a reactor that contains the deployments of Figure 8.9, where deployment d_1 is the root deployment.

Scheduling Nodes

Suppose that the reactor receives a `'react` message with a value of `50`. This message should change the value of the corresponding source node `x` of deployment d_1 (from Figure 8.9). The first step is to schedule the node in the reactive engine's (initially empty) priority queue. In general nodes are scheduled via the reactive

```
(class ReactiveEngine
  (def-fields pq scheduled-nodes)

  (def-method (schedule-node! node deployment)
    (def node-id (get-id node deployment))
    (if (not (contains? scheduled-nodes node-id))
        (let ((scheduled-node (newd ScheduledNode node deployment)))
          (put! scheduled-nodes (get-id node deployment) scheduled-node)
          (enqueue! pq scheduled-node))))

  (def-method (update-loop!)
    (when (not (empty? pq))
      (recompute-next! #this)
      (update-loop! #this)))

  (def-method (recompute-next!)
    (def scheduled-node (serve! pq))
    (def node (get-node scheduled-node))
    (def deployment (get-deployment scheduled-node))
    (remove! scheduled-nodes (get-id node deployment))
    (if (is-source? node)
        (let ((activations (get-activations scheduled-node)))
          (def new-val (activate-source! node deployment activations))
          (store-and-schedule! #this deployment node new-val))
        (when (is-computable? node deployment)
          (let ((result (compute! node deployment)))
            (store-and-schedule! #this deployment node result)))))

  (def-method (store-and-schedule! deployment node new-value)
    (def old-value (get-node-value deployment node))
    (if (and (eq? (type-of old-value) 'IncrementalBag)
             (eq? (type-of new-value) 'IncrementalDatastructureDeltaList))
        (set! new-value (apply-patch old-value new-value)))
    (set-node-value! deployment node new-value)
    (schedule-dependents! #this deployment node))

  (def-method (schedule-dependents! deployment node)
    (def dependents (get-dependents node))
    (for/iterable (dependent dependents)
      (schedule-node! #this dependent deployment))
    (schedule-cross-deployment-dependents! deployment node))

  // omitted methods: schedule-source-node!, schedule-source-node*!,
    schedule-source-activation!
)
```

Listing 8.13: The `ReactiveEngine` class.

engine's `schedule-node!` method defined in Listing 8.13 on lines 4 to 9. To avoid scheduling the same node node multiple times, we first check whether the node is already in the priority queue using a `node-id` which uniquely identifies a particular node in a particular reactor deployment. If the node has not been scheduled already, it is wrapped in a `ScheduledNode` object that also tracks which reactor deployment the node belongs to. The wrapped object is then scheduled in the priority queue `pq`, and added to the `scheduled-nodes` dictionary.

Source nodes are not scheduled via the `schedule-node!` method, but via other methods that we omitted from Listing 8.13. Their implementation is very similar, but source nodes are wrapped in a `ScheduledSourceNode` object instead of `ScheduledNode`. This wrapper additionally tracks a vector of so-called *activations*. Each activation is an object that contains the new value for the source node and information about the origin of the data, i.e., which reactor deployment and which sink node the data originated from. This information is important to implement `deploy-*` efficiently, such that `deploy-*` can efficiently update its output value whenever a reactor deployment produces a new value. The contents of the priority queue after scheduling the `x` source node with a value of `50` is depicted in the top left of Figure 8.10 (the area labelled "`loop 1`"). Since the source node was "activated" by a message sent to the reactor, the information about the origin of the data is `#false`.

Recomputing Nodes

Once the first source node is scheduled, the update loop will propagate this value through the reactor. The update loop is implemented in Listing 8.13 on lines 11 to 14 as a classic recursive method that keeps looping until the priority queue is empty, which means all nodes from source to sink were recomputed[9].

The logic to recompute a node is implemented by the `recompute-next!` method on lines 16 to 27. Lines 17 to 19 serve a `ScheduledNode` (or `ScheduledSourceNode`) object from the priority queue and extract the DAG node and its reactor deployment. The logic to process the node is different for source nodes and the other types of nodes.

Source nodes: The source node is "activated" on line 23 with a vector of activations which we previously discussed. In a nutshell, there will be a single object in this vector for each value provided to the source node. Currently the only case where a source node may be provided with multiple activations is `deploy-*`, whose corresponding (implicit) source node reacts to the values produced by multiple deployments. Activating the source on line 23 returns the new value computed by the source node. In most cases this will be the

[9] Stella's `(when condition exprs...)` expression is syntactic sugar for `(if condition (begin exprs...))`.

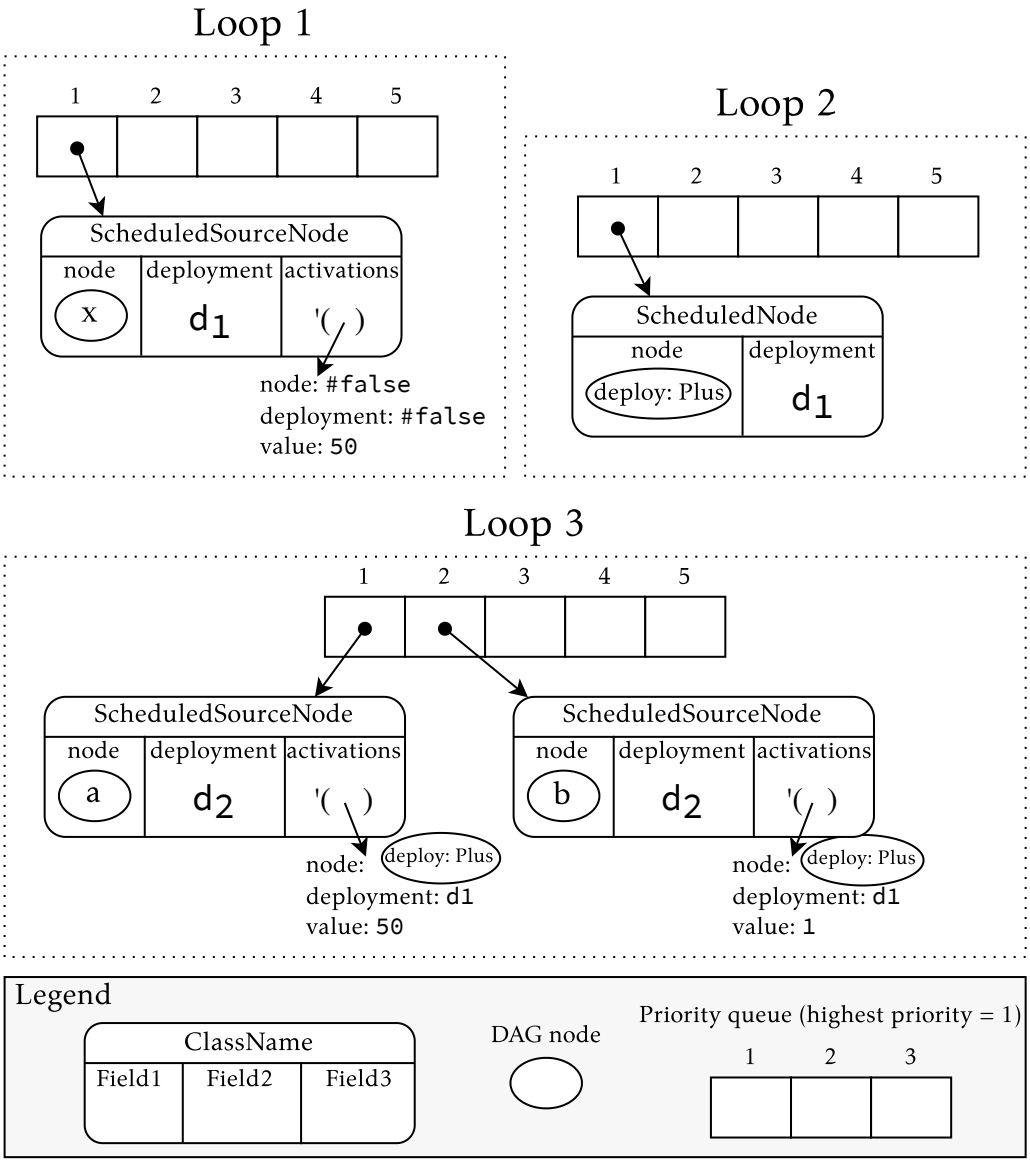


Figure 8.10.: Graphical depiction of the contents of the reactive engine's priority queue when an `Increment` reactor (of Figure 8.9) processes a `'react` message with value `50`.

value contained in the sole activation, but in the case of `deploy-*` this will be an updated collection. Finally, on line 24 the reactive engine stores this value and schedules the dependent nodes.

Internal & sink nodes: Internal nodes and sink nodes are treated similarly to source nodes, but instead of "activating" them, the reactive engine calls their `compute!` method on line 26. We have previously shown one such implementation for the `ApplicationNode` in Section 8.3.3. Just like source nodes, this new value will be stored and any dependent nodes will be scheduled.

Storing Values and Scheduling Dependents

After a node has been recomputed, the reactive engine stores the newly computed value via the `store-and-schedule!` method defined on lines 29 to 35 of Listing 8.13. Its arguments are the reactor deployment of the recomputed node, the node itself, and the newly computed value. In principe the only required action is to store the value unmodified in the corresponding reactor deployment. However, we implement a special case on lines 31 to 33 to support incremental data structures which are used by `deploy-*`. Whenever the node's old value was a snapshot of type `IncrementalBag` and the newly computed value is a patch of type `IncrementalDatastructureDeltaList`, then the patch is applied to the data, yielding an updated collection.

Updating the value of a node means that all dependent nodes should be scheduled in the priority queue. This scheduling is initiated on line 35 via the `schedule-dependent!` method. The implementation of the method on lines 37 to 41 schedules both direct dependent nodes within the DAG of the same deployment, as well as any cross deployment dependents, i.e., potential dependent source nodes within different deployments.

Continuing the example given in Figure 8.10 (page 175), the area labelled as "loop 2" (update loop iteration 2) depicts the content of the priority queue after scheduling the dependents of the `x` source node (the corresponding DAG is given in Figure 8.9). In this case there is 1 dependent `deploy` node within the same deployment. When this node is processed, the content of the priority queue just before "loop 3" shows that 2 nodes from a different reactor deployment d_2 were scheduled. Note in particular that the information about the origin of the data is no longer `#false`, but that the origin is the `deploy` node in reactor deployment d_1.

The update loop continues until the priority queue is empty, after which all affected nodes in the reactive program were updated, and the values of the sink nodes can be emitted to the reactor's output stream.

8.6. Summary and Conclusion

In this section we presented Mira, a meta implementation of reactors using actors. The implementation covers important features of reactors such as:

Constructing a DAG The program text of a Mira program is specified as a vector of symbols. We specified a compiler that turns the program text into a DAG. Just like Stella, the DAG construction occurs once when a reactor behaviour is created (i.e., at compile-time). Mira implements the same types of DAG nodes as Stella. While we have not shown the compilation and implementation of nodes such as the `QualificationNode`, `DeployNode` and `DeployStarNode`, their implementation aligns with the diagrams and descriptions that we provided when explaining the corresponding features[10].

Emulating reactors We showed how an actor can be used to emulate a reactor. More specifically, Mira shows how reactors process messages from their mailbox, and how they asynchronously (via messages) establish subscriptions to the streams of other (re)actors. The semantics of emulated reactors in Mira accurately reflect the implementation of Stella.

Propagating values A reactive program is a collection of DAGs within various connected reactor deployments. The concept of reactor deployments is, in our opinion, a key enabler of operation such as `deploy-*`. Using reactor deployments means that the implementation of Stella needs to carefully ensure that glitches cannot occur. To this end, Mira reimplements Stella's reactive engine, and we detailed the method used by Stella to determine the order in which nodes are recomputed.

The complete code is included in Appendix C (page 191).

[10] The compilation of qualification and `deploy` was discussed in Section 5.6, and `deploy-*` was discussed in Section 6.4.

9. Conclusion

This dissertation investigated various concerns of building distributed reactive software for open networks. In general, we focussed on 2 research areas. First, we investigated the combination of imperative code and reactive code which occurs in all reactive programs, and second, we investigated reactive programming to program applications whose distribution model involves open networks. In this final chapter we summarise this dissertation and we will discuss avenues for future work.

9.1. Summary

We started the dissertation by describing the need for reactive programming to develop modern software. Devices such as laptops, smartphones, smartwatches, and home automation devices connect to each other on open networks in order to continuously share information. Users expect the software that is running on these devices to *behave reactively*, such that the information presented to them is always up-to-date. Traditional approaches to develop reactive software such as callbacks and the Observer pattern have well-known drawbacks. The emerging reactive programming paradigm aims to avoid these issues, and has been used in many different application domains, such as embedded systems with extremely limited resources, and distributed systems.

When using reactive programming to develop distributed reactive applications, we identified 2 different levels on which the reactive program needs to react to change. First, *application-level reactivity* denoted the flow of application-level values such as sensor measurements that flow from one device to another, which is the main type of reactivity supported by existing reactive programming languages and frameworks. Second, *topology-level reactivity* is a new type of reactivity which occurs on an open network, where a modern reactive program continuously interacts with a constellation of devices which continuously join and leave the network. Topology-level reactivity denotes how the structure (i.e., the topology) of the reactive program adapts to the changing constellation of devices, which is not supported by most existing reactive programming languages and frameworks.

In Chapter 4 (page 49) we identified 3 problems that occur in existing reactive programming languages and frameworks. In the context of application-level react-

ivity, we identified the Reactive Thread Hijacking Problem and the Reactive/Imperative Impedance Mismatch which are caused by the uncontrolled combination of imperative and reactive code. In the context of topology-level reactivity, we identified the Acquaintance Maintenance Problem which occurs when a programmer tries to use existing reactive programming languages and frameworks for open network applications, despite the lack of built-in support. In summary:

Reactive Thread Hijacking Problem Long lasting computations that are part of a reactive program can (accidentally) block the reactive program's update thread, thus stopping said program from being able to react to any other input values. This is especially problematic when dealing with open networks, since the input values originates from devices that are not necessarily part of the same application.

Reactive/Imperative Impedance Mismatch We investigated the combination of imperative code and reactive code in both directions of their embedding. In one direction, expressions that contain side-effects (e.g., sockets, message passing, or simple assignments) can cause issues when they are embedded in subexpressions of a reactive program. That is because the order of their execution cannot be determined beforehand, they are very difficult to coordinate, and have a detrimental effect on program composition. In the other direction, existing reactive programming languages and frameworks have no well-defined and sufficiently general mechanism to embed reactive code within imperative code *without* (accidentally) allowing imperative code to be embedded within reactive code.

Acquaintance Maintenance Problem The existing reactive programming languages and frameworks are not designed to manage the ever changing acquaintances (i.e., devices) throughout the reactive program. When doing so, they are either inefficient, or require a complex and error-prone mix of code when performing acquaintance maintenance.

The basis of our solution to these problems emerged from the fundamental assumption that imperative code and reactive code cannot be reconciled in a single, unified programming model. Hence, we designed the *Actor-Reactor Model* to keep them separate in actors and reactors respectively, and to carefully specify the semantics of their interaction via message passing.

Stella's implementation of the Actor-Reactor Model demonstrates a practical way in which real applications can be built using actors and reactors. In Chapter 5 we implemented Bikey, an application to rent shared bicycles which demonstrated actors and reactors. An extension of this application called Whereabikes was used in Chapter 6 to demonstrate Stella's solution for acquaintance discovery and acquaintance maintenance, namely flocks and the topology-reactive operator `deploy-*`. Furthermore, we demonstrated Stella's capabilities to program real distributed applications via a case study using real data from "Villo!", Brussels

bike sharing program, whose network we simulated on a cluster computer of 240 Raspberry Pi's.

9.2. Restating the Contributions

Our first contribution is a new **taxonomy of reactive programming languages and frameworks**. We provided a taxonomy of the state of the art in reactive programming in Chapter 2, and distributed reactive programming in Chapter 3. Furthermore, Chapter 7 classifies these languages and frameworks according to the problems highlighted by this dissertation. We found that these problems are indeed prevalent, and are important to solve.

The second contribution is a thorough **problem analysis** that identifies the three problems formulated in this dissertation. We specified terms such as application-level reactivity and topology-level reactivity which are two distinct forms of reactivity that occur when developing reactive applications for open networks. Furthermore, we used the terms weak reactivity, eventual reactivity and strong reactivity to describe the degree to which a reactive program guarantees that it will be able to react to new data.

Third, we proposed the **Actor-Reactor Model** as a programming model to describe and implement reactive programs. Besides a practical implementation of the Actor-Reactor Model in Stella, in Chapter 7 we found precursors of the model in related work. This indicates that a programming model such as the Actor-Reactor Model naturally arises in other implementations of reactive programs as well, without being identified as such.

Fourth, we design **flocks** as a new programming abstraction to discover actors and reactors on an open network, and we introduced `deploy-*`, a new topology-reactive operator for reactive programs which complements flocks, and which is used to implement correct and efficient computations based on constantly fluctuating actor and reactors.

Our final contribution is a result of an **artefact-based research method**. Besides the scientific formulation of concepts such as the Actor-Reactor Model, flocks and **`deploy-*`**, we implemented and tested the proposed solutions in Stella to verify that they work in practice as well as in theory. Furthermore, since Stella's reactors hide a lot of application complexity as a black box, in Chapter 8 we provided a meta-implementation of reactors in Stella itself.

9.3. Limitations and Future Work

We will briefly discuss Stella's limitations, and highlight avenues for future research.

9.3.1. Stella's Technical Limitations

Building a completely new programming language is difficult, especially when the design and the scope of the language is continuously evolving over time. Hence, there are certain features that are expected from (production-ready) programming languages which are not supported by Stella.

No Exception Handling While Stella's interpreter is capable of correctly executing correct Stella problems, it currently has no features for exception handling (e.g., `throw` and `try/catch` expressions).

Limited Standard Library Stella's standard library was implemented on a "by need" basis, and as such it only supports a limited set of data structures. Features such as GUIs and (file) IO can be accessed via Stella's foreign function interface to JavaScript.

9.3.2. Reactive Exception Handling

Similar to how Stella is specified in two levels, namely a sequential (object-oriented) base language and the concurrent level of actors and reactors, an exception handling mechanism for Stella will likely exist on both levels. Firstly, a traditional `throw` and `try/catch` mechanism can be implemented for the sequential base language, which is not a research problem. Secondly, we envision that there is a need for *reactive* exception handling that deals with exceptions that occur on the level of actors and reactors. Since they are connected via streams, it can be beneficial when a receiver of data (e.g., a reactor) can signal a producer of data (e.g., an actor) that an exception occurred, such that the producer of data can (automatically) adapt such that the error no longer occurs.

A possible use case for reactive exception handling is known as *backpressure*, which means that a downstream component (e.g., a reactor) is not fast enough to process all of the data that it receives. In Stella this currently means that the (unbounded) mailbox of the receiver fills up until the program is out of memory and crashes. Reactive exception handling can be used to avoid that a receiver is overwhelmed with messages by throwing an exception, after which producers of data can slow down the rate at which messages are sent, or risk that some of their messages are dropped.

9.3.3. Security

Stella's distribution model assumes that, if other actors and reactors can be discovered, that they are trustworthy and can be interacted with. An avenue for future research is to no longer assume such trust, for which we envision the following research avenues.

A first avenue is that not all actors and reactors should be discoverable via a flock by the entire network. For example, an application could make its (re)actors discoverable to only other (re)actors within a trusted network. A second avenue is to allow certain kinds of communication between untrusted parties, but to restrict on the language level which kinds of data can be communicated via messages. For example, the language can restrict via security policies (e.g., both statically and dynamically) that sensitive credit card information cannot flow from (re)actors on a trusted network to (re)actors in an untrusted network. There are research opportunities to make such security policies reactive as well, e.g., to learn what kinds of data is typically passed between two actors, and to automatically flag whenever a kind of data is passed over a stream that is not expected from the producing (re)actors.

9.4. Closing Remarks With Respect to Applicability

Throughout this dissertation we have extensively discussed and shown the prevalence of the identified problems in existing reactive programming languages and frameworks, and we proposed a solution to solve those problems. We envision that other reactive programming languages and frameworks can adopt the solution proposed in this dissertation as follows:

Reactive Thread Hijacking Problem This problem is arguably the most tricky to solve when implementing a reactive programming language, and especially when implementing reactive programming features as a library in a different language. Regardless, other reactive programming languages exist that already enforce eventual reactivity or weak reactivity on the programs that are accepted by the language. These languages can continue to enforce those constraints by using reactors, and also expand their application domain by using actors to build the parts of the application which cannot be expressed within the constraints of the reactive parts of the language. Designers of reactive programming frameworks will have more difficulties enforcing eventual reactivity or strong reactivity since they cannot control all aspects of the used programming language. They may be able to use static analysis tools to provide programmer hints (e.g., in the development environment) that indicate whether a certain part of a reactive program has the potential to be unexpectedly slow (e.g., infinitely loop) for certain types of input values.

Reactive/Imperative Impedance Mismatch Our solution is based on a key technology – the actor model – that is already used in many other programming languages, either as a built-in feature or as a library. Therefore, we believe that it is feasible for other reactive programming languages and frameworks to adopt the Actor-Reactor Model as well, such that they can also avoid the Reactive/Imperative Impedance Mismatch.

Acquaintance Maintenance Problem Not all reactive programming languages and frameworks will be used to program reactive applications for open networks. However, extensions can be built for existing distributed reactive languages and frameworks (e.g., ReactiveX or Akka Streams) to incorporate a discovery mechanism such as a flock, together with a complementary topology-reactive operator such as `deploy-*` to correctly and efficiently manage the reactive program's application state.

In this dissertation we have identified and tackled problems that have been a part of reactive programming since its inception in 1997 (Fran [43]), and we have broadened the application to also include distributed reactive programming for open networks. By naming these issues and providing a feasible path to solve them in other languages and frameworks, we aim to further support the adoption of reactive programming to build reactive applications.

A. Topology-level Reactivity in RxJS: A Semantic Bug

An early paper submission on topology-level reactivity in RxJS featured a different solution to the problem at hand used in Section 4.3.2, namely that of averaging the measurements of thermometers. The solution contained a semantic bug. We believe it is realistic that other developers will make same mistake as well, possibly leading to a solution that contains such a bug. Arguably, we think that the (at least) two different ways to think about the problem may actually be part of the problem itself, i.e., part of the problem of accidental complexity when using topology-level reactivity. In this appendix we explain our buggy implementation, and at the end explain why it is buggy.

The main ideas can be explained via the `average` computation that computes the average value of all thermometers connected to a network, which we implemented in RxJS [121], a state of the art JavaScript framework based on reactive streams (see Section 2.2 on page 18). Listing A.1 implements this computation via an RxJS stream called `average$` (`$` is a naming convention for streams). To communicate the continuously appearing and disappearing sensors to the reactive program, we used a stream called `sensorDiscoveryService.sensors$` (not defined here) from which `average$` is derived. This stream propagates an updated `Set` of all sensors every time a sensor appears or disappears. In RxJS, the `average$` stream is defined by "piping" the values from `sensors$` through a sequence of RxJS stream operators. The operators in the example work as follows:

```
1 const average$ = sensorDiscoveryService.sensors$.pipe(
2   rxjs.switchMap((allSensors) =>
3     rxjs.from(allSensors).pipe(
4       rxjs.flatMap((sensor) =>
5         sensor.value$.pipe(
6           rxjs.map((val) => [sensor.id, val]))),
7       rxjs.scan((tracker, [id, val]) => tracker.update(id, val),
8           new AverageTracker()),
9       rxjs.map((tracker) => tracker.getAverage()))));
```

Listing A.1: Topology-level reactivity in RxJS, calculating an average of all sensors.

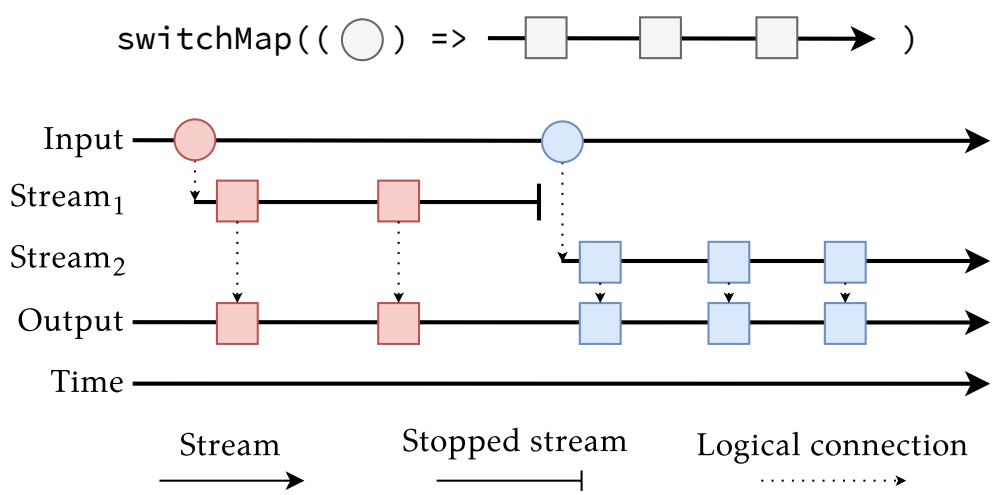


Figure A.1.: Illustration of the `switchMap` operator in RxJS. The diagram is simplified for brevity (it does not consider higher-order streams, i.e., streams whose values are other streams).

Line 2, `switchMap`: The lambda passed to `switchMap` accepts a set of sensors as argument, and will generate a new stream whenever a new set of sensors is propagated. `switchMap` echoes the values of the most recently generated stream, in this case the average of all sensor values. Since a textual explanation of `switchMap` is often difficult to comprehend, it is visualised in Figure A.1 in terms of the different streams involved (time flows from left to right). At the top, we depict `switchMap` as an operation that transforms values of type "circle" to a stream of values of type "square". Whenever the input stream contains a red circle, then the lambda passed to `switchMap` generates a stream of red squares, which are echoed on the output stream. As soon as the input stream contains a new value such as a blue circle, then a new stream of blue squares is generated. Consequently the stream of red squares is forgotten, and the output stream now echoes blue squares instead.

The body of the lambda on line 2 generates a stream via the `from` operator (line 3) that transforms a collection of sensors to a stream that emits the elements in the collection one by one. This stream is further transformed to eventually compute the average of all sensor values.

Line 3, `flatMap`: The lambda passed to `flatMap` generates a stream. `flatMap` remembers all streams it has generated and echoes their values on a first-in first-out basis. In this case we use it to transform a stream of sensors to a stream of tuples `[id, val]` where `id` is the unique identifier of the sensor and `val` is its latest value. A new `[id, val]` tuple is propagated every time the `sensor.value$` stream contains a new temperature measurement.

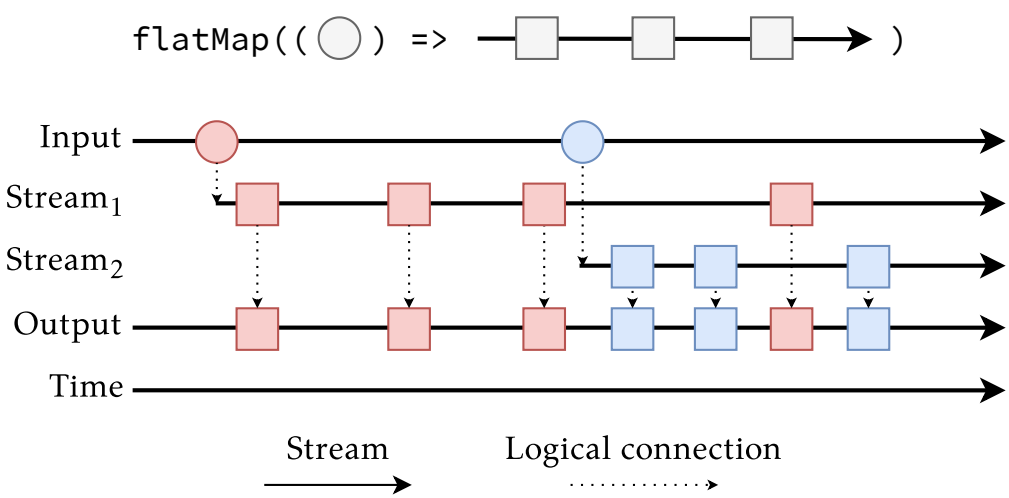


Figure A.2.: Illustration of the `flatMap` operator in RxJS. The diagram is simplified for brevity (it does not consider higher-order streams, i.e., streams whose values are other streams).

Similar to how we visualised `switchMap`, `flatMap` is visualised in Figure A.2. The main difference is that when the input stream contains a new value such as a blue circle, then the previously generated stream of red squares is not stopped. Instead `flatMap` remembers both the stream of red squares and blue squares, and echoes both of their values on its output stream. Hence, whenever *any* `sensor.value$` stream is updated, a new `[id, val]` tuple is propagated by `flatMap`.

Line 7, `scan`: Most of the actual application logic is tackled by the `scan` operator that essentially implements a `fold` operation for streams where the accumulator is emitted every time the input changes. The accumulator is a purely functional `AverageTracker` (implemented elsewhere) that tracks the latest value produced by each sensor. The subsequent `map` (line 9) extracts the current average.

This code incorrectly implements the desired computation. The bug is caused by the propagation of a `Set` of sensors – a solution that is conceptually similar to the solution in signal-based reactive programming languages (discussed in Section 4.3.2). This is because an entirely new `Set` of sensors is propagated every time a sensor is added or removed by the discovery service, and the entire result is recomputed from scratch. This means that, internally, all dependencies to the "old" sensors are removed, and new dependencies are established to the "new" sensors (despite their overlap). Because the sensors are connected via a network, this swapping of dependencies would currently cause unnecessary network traffic and delays. Worse, every time the set of sensors changes, the final output stream

emits all intermediate values until all sensors are known (again) to `flatMap`, which is semantically incorrect.

B. The ImmutableVector Class

The interface of the `ImmutableVector` class can be found in Table B.1.

Name	Type	Signature	Description
Constructors			
default	Constructor	`Object…` → `ImVec`	Creates a vector from the specified argument values.
from-vector	Constructor	`ImVec` → `ImVec`	Creates a (shallow) copy of the given argument vector.
Methods and routines			
get	Routine	`Number` → `Object`	Gets the value at the specified index.
put	Routine	`Number Object` → `ImVec`	Replaces the value at the specified index.
push	Routine	`Object` → `ImVec`	Adds a value at the end of the vector.
pop	Routine	`()` → `Object`	Removes and returns the last value.
unshift	Routine	`Object` → `ImVec`	Adds a value to the start.
delete	Routine	`Object` → `ImVec`	Removes the specified value from the vector.
first, …, tenth	Routine	`()` → `Object`	Gets the n^{th} value (1st value, 2nd value, …).
penultimate, last	Routine	`()` → `Object`	Gets the penultimate or last value of the vector.
empty?	Routine	`()` → `Boolean`	Tests if the vector is empty.
append	Routine	`ImVec` → `ImVec`	Appends the current vector to the argument vector, and returns a new vector.
length	Routine	`()` → `Number`	Gets the length of the vector.
head	Routine	`()` → `Object`	Gets the first value.
tail	Routine	`()` → `ImVec`	Returns a new vector with all values but the first.
fold	Routine	`Object Symbol` → `Object`	Invokes the given routine (2nd argument) on the accumulator (1st argument) with every value of the vector as argument, and returns the final accumulator.
iterator	Routine	`()` → `Iterator`	Creates an iterator for the vector.

Table B.1.: Interface of Stella's `ImmutableVector` class. Note that due to space constraints, in this table `ImmutableVector` was shortened to `ImVec`.

C. Complete Code of Mira

The complete implementation of Mira is provided in Listing C.1.

```
(def-actor Main
  (def-constructor (start env)
    (send! #self 'run-qualification-reactor))

  (def-method (run-plus-reactor)
    (def env (newd Dictionary))
    (def program '(def-reactor (Plus x y) (out (+ x y))))
    (def behaviour (new ReactorBehaviour 'compile program env))
    (def reactor (spawn-actor! Reactor 'init behaviour))
    (send! reactor 'react! '(1 1))
    (send! reactor 'react! '(2 2))
    (send! reactor 'react! '(3 3)))

  (def-method (run-qualification-reactor)
    (def env (newd Dictionary))
    (def number-producer (spawn-actor! NumberProducer 'init))
    (def number-producer2 (spawn-actor! NumberProducer 'init))
    (def program '(def-reactor (Increment x) (out (+ (qualification x number) 1))))
    (def behaviour (new ReactorBehaviour 'compile program env))
    (def reactor (spawn-actor! Reactor 'init behaviour))
    (send! reactor 'react! (newd Vector number-producer))
    (send! number-producer 'emit-number 5)
    (send! number-producer2 'emit-number 6)
    (send! reactor 'react! (newd Vector number-producer2)))

  (def-method (run-deploy-reactor)
    (def env (newd Dictionary))
    (def increment (new ReactorBehaviour 'compile
      '(def-reactor (Increment x) (out (+ (qualification x number) 1))) env))
    (def plus (new ReactorBehaviour 'compile
      '(def-reactor (Plus x) (out (+ (deploy Increment x) 1))) env))
    (def reactor (spawn-actor! Reactor 'init plus))

    (def number-producer (spawn-actor! NumberProducer 'init))
    (send! reactor 'react! (newd Vector number-producer))
    (send! number-producer 'emit-number 5)
    (send! number-producer 'emit-number 6))

    (def-method (run-deploy-star-reactor-non-incremental)
      (def env (newd Dictionary))
      (def add-1 (new ReactorBehaviour 'compile
        '(def-reactor (Add1 x) (out (+ (qualification x number) 1))) env))
      (def add-1-to-all (new ReactorBehaviour 'compile
        '(def-reactor (Add1ToAll xs) (out (deploy-* Add1 xs))) env))
      (def reactor (spawn-actor! Reactor 'init add-1-to-all))

      (def number-producer1 (spawn-actor! NumberProducer 'init))
```

```
48        (def number-producer2 (spawn-actor! NumberProducer 'init))
49        (send! reactor 'react! (newd Vector (newd Bag number-producer1 number-producer2))
        )
50        (send! number-producer1 'emit-number 1)
51        (send! number-producer2 'emit-number 1)
52
53        (def number-producer3 (spawn-actor! NumberProducer 'init))
54        (def number-producer4 (spawn-actor! NumberProducer 'init))
55        (send! reactor 'react! (newd Vector (newd Bag number-producer3 number-producer4))
        )
56        (send! number-producer3 'emit-number 10)
57        (send! number-producer4 'emit-number 10))
58
59      (def-method (run-deploy-star-reactor-incremental)
60        (def env (newd Dictionary))
61        (def add-1 (new ReactorBehaviour 'compile
62          '(def-reactor (Add1 x) (out (+ (qualification x number) 1))) env))
63
64        (def add-1-to-all (new ReactorBehaviour 'compile
65          '(def-reactor (Add1ToAll xs) (out (deploy-* Add1 xs))) env))
66        (def reactor (spawn-actor! Reactor 'init add-1-to-all))
67
68        (def number-producer1 (spawn-actor! NumberProducer 'init))
69        (def inc-bag (newd IncrementalBag number-producer1))
70        (send! reactor 'react! (newd Vector inc-bag))
71        (send! number-producer1 'emit-number 10)
72
73        (def number-producer2 (spawn-actor! NumberProducer 'init))
74        (def inc-bag2 (add inc-bag number-producer2))
75        (def patch (get-patch inc-bag2))
76        (send! reactor 'react! (newd Vector patch))
77        (send! number-producer2 'emit-number 20)))
78
79  (def-actor NumberProducer
80    (def-stream number)
81    (def-constructor (init) #true)
82    (def-method (emit-number x)
83      (emit! number x)))
84
85  (class ReactorBehaviour
86    (def-fields name all-nodes sources sinks env)
87
88    (def-constructor (compile sexp _env)
89      (def header (head (tail sexp)))
90      (def body (tail (tail sexp)))
91      (set! name (head header))
92      (set! all-nodes (newd Vector))
93      (set! env _env)
94      (def footer (pop! body))
95      (def localenv (newd Dictionary))
96      (make-sources! #this localenv header)
97      (make-body! #this localenv body)
98      (make-sinks! #this localenv footer)
99      (put! env name #this))
100
101   (def-routine (get-name) name)
102   (def-routine (get-sources) sources)
103   (def-routine (get-sinks) sinks)
104   (def-routine (get-all-nodes) all-nodes)
```

```
105
106  (def-method (make-sources! localenv header)
107    (set! sources (newd Vector))
108    (for/iterable (signal-name (tail header))
109      (def node (newd SourceNode signal-name))
110      (put! localenv signal-name node)
111      (push! sources node)
112      (push! all-nodes node)))
113
114  (def-method (make-body! localenv body)
115    (for/iterable (expression body)
116      (make-node! #this localenv expression)))
117
118  (def-method (make-sinks! localenv footer)
119    (set! sinks (newd Vector))
120    (def sink-expressions (tail footer))
121    (def sink-id 1)
122    (for/iterable (sink-expression sink-expressions)
123      (def expr-node (make-node! #this localenv sink-expression))
124      (def sink-node (newd SinkNode sink-id expr-node))
125      (set! sink-id (+ sink-id 1))
126      (push! all-nodes sink-node)
127      (push! sinks sink-node)))
128
129  (def-method (make-node! localenv expression)
130    (cond
131          ((eq? (type-of expression) 'Symbol)
132           (get localenv expression))
133          ((or (eq? (type-of expression) 'Number)
134               (eq? (type-of expression) 'String)
135               (eq? (type-of expression) 'Boolean))
136           (make-constant-node! #this localenv expression))
137          ((eq? (first expression) 'def)
138           (make-define-node! #this localenv expression))
139          ((eq? (first expression) 'qualification)
140           (make-qualification-node! #this localenv expression))
141          ((eq? (first expression) 'deploy)
142           (make-deploy-node! #this localenv expression))
143          ((eq? (first expression) 'deploy-*)
144           (make-deploy-star-node! #this localenv expression))
145          ((eq? (type-of (first expression)) 'Symbol)
146           (make-application-node! #this localenv expression))
147          (else (println! "!!!! Unsupported node type: " expression))))
148
149
150  (def-method (make-define-node! localenv expression)
151   (def identifier (second expression))
152   (def body (third expression))
153   (def body-node (make-node! #this localenv body))
154   (put! localenv identifier body-node)
155   body-node)
156
157  (def-method (make-constant-node! localenv expression)
158    (def node (newd ConstantNode expression))
159    (push! all-nodes node)
160    node)
161
162  (def-method (make-qualification-node! localenv expression)
163    (def source (second expression))
```

```
164     (def stream-name (third expression))
165     (def implicit-source (newd ImplicitSourceNode (to-string stream-name)))
166     (def data-source (make-node! #this localenv source))
167     (def qualification-node (newd QualificationNode data-source stream-name
        implicit-source))
168     (push! all-nodes qualification-node)
169     (push! all-nodes implicit-source)
170     implicit-source)
171
172   (def-method (make-application-node! localenv expression)
173     (def operator (first expression))
174     (def operands (tail expression))
175     (def operand-nodes (newd Vector))
176     (for/iterable (operand operands)
177       (push! operand-nodes (make-node! #this localenv operand)))
178     (def application-node (newd ApplicationNode operator operand-nodes))
179     (push! all-nodes application-node)
180     application-node)
181
182   (def-method (make-deploy-node! localenv expression)
183     (def behaviour-name (second expression))
184     (def behaviour-to-deploy (get env behaviour-name))
185     (def arguments (newd Vector))
186     (for/iterable (arg (tail (tail expression)))
187       (push! arguments (make-node! #this localenv arg)))
188     (def implicit-source (newd ImplicitSourceNode "Deploy"))
189     (def deploy-node (newd DeployNode behaviour-to-deploy arguments implicit-source))
190     (push! all-nodes deploy-node)
191     (push! all-nodes implicit-source)
192     implicit-source)
193
194   (def-method (make-deploy-star-node! localenv expression)
195     (def behaviour-name (second expression))
196     (def behaviour-to-deploy (get env behaviour-name))
197     (def data-source (make-node! #this localenv (third expression)))
198     (def implicit-source (newd DeployStarImplicitSourceNode))
199     (def deploy-star-node (newd DeployStarNode behaviour-to-deploy data-source
        implicit-source))
200     (push! all-nodes deploy-star-node)
201     (push! all-nodes implicit-source)
202     implicit-source))
203
204
205 (class Node
206   (def-fields name id height dependencies dependents)
207
208   (def-constructor (default _name _dependencies)
209     (set! name _name)
210     (set! id (make-uuid (newd Random)))
211     (set! dependencies _dependencies)
212     (set! dependents (newd Vector))
213
214     (def computed-height 1)
215     (for/iterable (dependency dependencies)
216       (add-dependent! dependency #this)
217       (set! computed-height (max computed-height (get-height dependency))))
218     (set! height (+ computed-height 1)))
219
220   (def-routine (get-id deployment) (append id "-" (get-id deployment)))
```

```
221   (def-routine (get-name) name)
222   (def-routine (get-height) height)
223   (def-routine (get-dependents) dependents)
224   (def-routine (get-dependencies) dependencies)
225   (def-routine (< other-node) (< height (get-height other-node)))
226
227   (def-method (add-dependent! node) (push! dependents node))
228
229   (def-method (collect-dependency-values deployment)
230     (def values (newd Vector))
231     (for/iterable (dependency dependencies)
232       (push! values (get-node-value deployment dependency)))
233     values)
234
235   (def-method (compute! deployment) #true)
236   (def-routine (is-source?) #false)
237   (def-routine (to-string)
238     (append "[" (to-string (type-of #this)) " | " (get-name #this) "]"))
239
240   (def-method (is-computable? deployment)
241     (def is-computable #true)
242     (for/iterable (input (collect-dependency-values #this deployment))
243       (if (eq? (type-of input) 'Undefined)
244           (set! is-computable #false)))
245     is-computable))
246
247 (class SourceNode
248   (extends Node)
249   (def-constructor (default name)
250     (super 'default (append "<SourceNode " (to-string name) ">") '()))
251
252   (def-method (activate-source! deployment activations)
253     (def activation (first activations))
254     (get-value activation))
255   (def-routine (is-source?) #true))
256
257 (class ImplicitSourceNode
258   (extends SourceNode)
259   (def-constructor (default name)
260     (super 'default (append "ImplicitSourceNode " (to-string name)))))
261
262 (class SinkNode
263   (extends Node)
264   (def-constructor (default sink-nr dependency)
265     (super 'default
266            (append "<SinkNode #" (to-string sink-nr) ">")
267            (newd Vector dependency)))
268
269   (def-method (compute! deployment)
270     (def dependency-values (collect-dependency-values #this deployment))
271     (get dependency-values 0)))
272
273 (class ConstantNode
274   (extends Node)
275   (def-fields value)
276   (def-constructor (default _value)
277     (super 'default
278            (append "<ConstantNode " (to-string value) ">")
279            (newd Vector))
```

```
280     (set! value _value))
281   (def-routine (get-value) value))
282
283 (class ApplicationNode
284   (extends Node)
285   (def-fields operator)
286
287   (def-constructor (default op ops)
288     (super 'default
289            (append "<ApplicationNode " (to-string op) ">")
290            ops)
291     (set! operator op))
292
293   (def-method (compute! deployment)
294     (def dependency-values (collect-dependency-values #this deployment))
295     (def receiver (head dependency-values))
296     (def arguments (tail dependency-values))
297     (apply operator receiver arguments)))
298
299 (class QualificationNodeDeploymentInfo
300   (def-fields subscription-id)
301
302   (def-method (set-subscription-id! new-sub-id)
303     (set! subscription-id new-sub-id))
304   (def-routine (get-subscription-id) subscription-id))
305
306
307 (class QualificationNode
308   (extends Node)
309   (def-fields implicit-source stream-name)
310
311   (def-constructor (default data-source _stream-name _implicit-source)
312     (super 'default
313            (append "<QualificationNode " _stream-name ">")
314            (newd Vector data-source))
315     (set! implicit-source _implicit-source)
316     (set! stream-name _stream-name))
317
318   (def-method (compute! deployment)
319     (def dependency-values (collect-dependency-values #this deployment))
320     (def new-data-source (first dependency-values))
321     (def deployment-info (get-deployment-info #this deployment))
322     (def old-subscription-id (get-subscription-id deployment-info))
323     (if old-subscription-id
324         (unsubscribe-from-stream! deployment old-subscription-id))
325     (def new-subscription-id (subscribe-to-stream! deployment new-data-source
        stream-name implicit-source))
326     (set-subscription-id! deployment-info new-subscription-id)
327     #undefined)
328
329   (def-method (get-deployment-info deployment)
330     (def existing-deployment-info (get-deployment-info deployment #this))
331     (if existing-deployment-info
332         existing-deployment-info
333         (let ((new-deployment-info (newd QualificationNodeDeploymentInfo)))
334           (save-deployment-info! deployment #this new-deployment-info)
335           new-deployment-info))))
336
337 (class DeployNodeDeploymentInfo
```

```
338   (def-fields deployment)
339
340   (def-constructor (default _deployment)
341     (set! deployment _deployment))
342
343   (def-routine (get-deployment) deployment))
344
345 (class DeployNode
346   (extends Node)
347   (def-fields behaviour implicit-source)
348
349   (def-constructor (default _behaviour _args _source)
350     (super 'default
351            (append "<DeployNode " (get-name _behaviour) ">")
352            _args)
353     (set! behaviour _behaviour)
354     (set! implicit-source _source))
355
356   (def-method (compute! deployment)
357     (def dependencies (get-dependencies #this))
358     (def dependent-deployment (get-or-create-deployment! #this deployment))
359     (def argument-values (collect-dependency-values #this deployment))
360     (def sources (get-sources (get-behaviour dependent-deployment)))
361     (for (i 0 (< i (length argument-values)) (+ i 1))
362       (def data-origin (get dependencies i))
363       (def new-value (get argument-values i))
364       (def source (get sources i))
365       (change-source*! dependent-deployment data-origin deployment source new-value))
366     #undefined)
367
368   (def-method (get-or-create-deployment! deployment)
369     (def deployment-info (get-deployment-info deployment #this))
370     (if deployment-info
371         (get-deployment deployment-info)
372         (let ((new-deployment (make-deployment! #this deployment)))
373           (def deployment-info (newd DeployNodeDeploymentInfo new-deployment))
374           (save-deployment-info! deployment #this deployment-info)
375           (link-deployments! #this deployment new-deployment)
376           new-deployment)))
377
378   (def-method (make-deployment! deployment)
379     (def reactive-engine (get-reactive-engine deployment))
380     (def owner (get-owner deployment))
381     (def new-deployment (newd ReactorDeployment behaviour reactive-engine owner
        deployment #this))
382     (add-deployment! owner new-deployment)
383     (add-derived-deployment! deployment new-deployment)
384     new-deployment)
385
386   (def-method (link-deployments! this-deployment dependency-deployment)
387     (def sink (first (get-sinks (get-behaviour dependency-deployment))))
388     (add-cross-deployment-dependency! this-deployment implicit-source sink
        dependency-deployment)))
389
390
391 (class DeployStarDeploymentManager
392   (def-fields behaviour
393               current-deployment
394               reactor
```

```
395             reactive-engine
396             deploy-star-node
397             implicit-source-node
398             deployments
399             deployment-keys)
400
401   (def-constructor (default _behaviour _deployment _deploy-star-node _implicit-source)
402     (set! behaviour _behaviour)
403     (set! current-deployment _deployment)
404     (set! reactor (get-owner current-deployment))
405     (set! reactive-engine (get-reactive-engine current-deployment))
406     (set! deploy-star-node _deploy-star-node)
407     (set! implicit-source-node _implicit-source)
408     (set! deployments (newd Dictionary))
409     (set! deployment-keys (newd Dictionary)))
410
411   (def-method (get-all-deployments)
412     (values deployments))
413
414   (def-method (new-deployment! key input-value)
415     (def deployment (newd ReactorDeployment behaviour reactive-engine reactor
        current-deployment deploy-star-node))
416     (add-deployment! reactor deployment)
417     (add-derived-deployment! current-deployment deployment)
418     (def sink (first (get-sinks behaviour)))
419     (add-cross-deployment-dependency! current-deployment implicit-source-node sink
        deployment)
420     (put! deployments key deployment)
421     (put! deployment-keys deployment key)
422     (update-deployment! #this key input-value))
423
424   (def-method (update-deployment! key input-value)
425     (def deployment (get deployments key))
426     (def source (first (get-sources behaviour)))
427     (change-source*! deployment deploy-star-node current-deployment source input-value)
        )
428
429   (def-method (remove-deployment! key)
430     (def deployment (get deployments key))
431     (def sink (first (get-sinks behaviour)))
432     (remove! deployments deployment)
433     (remove! deployment-keys key)
434     (remove-cross-deployment-dependency! current-deployment implicit-source-node sink
        deployment)
435     (clean-up! deployment))
436
437   (def-method (remove-all-deployments!)
438     (for/iterable (deployment (values deployments))
439       (def key (get deployment-keys deployment))
440       (remove-deployment! #this key)))
441
442   (def-method (has-deployment? key)
443     (contains? deployments key))
444
445   (def-method (get-deployment-key deployment)
446     (get deployment-keys deployment))
447
448   (def-method (get-all-deployment-keys)
449     (values deployment-keys)))
```

```
450
451 (class DeployStarDeploymentInfo
452   (def-fields input-data deployment-manager)
453   (def-constructor (default _manager)
454     (set! deployment-manager _manager))
455   (def-method (get-deployment-manager) deployment-manager)
456   (def-method (get-input) input-data)
457   (def-method (set-input! new-input)
458     (set! input-data new-input)))
459
460 (class DeployStarActivation
461   (def-fields changed-input type)
462   (def-constructor (default _changed-input _type)
463     (set! changed-input _changed-input)
464     (set! type _type))
465   (def-routine (get-changed-input) changed-input)
466   (def-routine (get-type) type))
467
468 (class DeployStarNode
469   (extends Node)
470
471   (def-fields behaviour implicit-source)
472
473   (def-constructor (default _behaviour _data-source _implicit-source)
474     (super 'default
475            (append "<DeployStarNode " (get-name _behaviour) ">")
476            (newd Vector _data-source))
477     (set! behaviour _behaviour)
478     (set! implicit-source _implicit-source))
479
480   (def-method (compute! deployment)
481     (def deployment-info (get-deployment-info! #this deployment))
482     (def deployment-manager (get-deployment-manager deployment-info))
483
484     (def old-input (get-input deployment-info))
485     (def new-input (first (collect-dependency-values #this deployment)))
486     (set-input! deployment-info new-input)
487
488     (cond ((and (eq? (type-of new-input) 'IncrementalBag)
489                 (is-derived-from? new-input old-input))
490            (map-patch! #this (get-patch new-input) deployment deployment-info))
491           ((or (eq? (type-of new-input) 'Bag)
492                (eq? (type-of new-input) 'IncrementalBag))
493            (map-new! #this new-input deployment deployment-info)))
494           (else "!!! DeployStarNode cannot process input !!!"))
495
496     #undefined)
497
498   (def-method (get-deployment-info! deployment)
499     (def deployment-info (get-deployment-info deployment #this))
500     (if deployment-info
501         deployment-info
502         (let ((deployment-manager (newd DeployStarDeploymentManager behaviour
      deployment #this implicit-source))
503               (new-deployment-info (newd DeployStarDeploymentInfo deployment-manager)))
504           (save-deployment-info! deployment #this new-deployment-info)
505           (save-deployment-info! deployment implicit-source new-deployment-info)
506           new-deployment-info)))
507
```

```
508   (def-method (map-new! collection deployment deployment-info)
509     (def deployment-manager (get-deployment-manager deployment-info))
510     (remove-all-deployments! deployment-manager)
511     (change-source-with-activation! deployment implicit-source (newd
        DeployStarActivation collection 'Total))
512
513     (for/iterable (key value collection)
514       (new-deployment! deployment-manager key value)))
515
516
517   (def-method (map-patch! patches deployment deployment-info)
518     (def deployment-manager (get-deployment-manager deployment-info))
519     (change-source-with-activation! deployment implicit-source (newd
        DeployStarActivation patches 'Incremental))
520
521     (for/iterable (patch patches)
522       (def patch-key (get-key patch))
523       (def patch-value (get-value patch))
524
525       (cond ((eq? (patch-type patch) 'insert)
526              (new-deployment! deployment-manager patch-key patch-value))
527             ((eq? (patch-type patch) 'update)
528              (update-deployment! deployment-manager patch-key patch-value))
529             ((eq? (patch-type patch) 'remove)
530              (remove-deployment! patch-key))))))
531
532
533
534
535 (class DeployStarImplicitSourceNode
536   (extends SourceNode)
537
538   (def-constructor (default)
539     (super 'default "DeployStarNode"))
540
541   (def-method (activate-source! deployment activations)
542     (if (eq? (type-of (first activations)) 'DeployStarActivation)
543         (let ((deploy-star-activation (first activations))
544               (other-activations (tail activations)))
545           (change-output-after-input-change! #this deployment deploy-star-activation
        other-activations))
546         (change-output-after-deployment-change! #this deployment activations)))
547
548   (def-method (change-output-after-input-change! deployment deploy-star-activation
        other-activations)
549     (cond ((eq? (get-type deploy-star-activation) 'Total)
550            (change-output-after-total-input-change! #this deployment))
551           ((eq? (get-type deploy-star-activation) 'Incremental)
552             (def changed-input (get-changed-input deploy-star-activation))
553             (change-output-after-incremental-input-change! #this deployment
        changed-input other-activations))
554           (else (println! "!!! Unknown activation type !!!"))))
555
556   (def-method (change-output-after-total-input-change! deployment)
557     (def deployment-info (get-deployment-info deployment #this))
558     (def deploy-star-input (get-input deployment-info))
559     (def all-values (get-all-values #this deployment))
560     (cond ((eq? (type-of deploy-star-input) 'Bag)
561            (new Bag 'from-vector all-values))
```

```
562           ((eq? (type-of deploy-star-input) 'IncrementalBag)
563            (new IncrementalBag 'from-vector all-values))
564           (else (println! "!!! Unknown input data structure !!!"))))
565
566   (def-method (change-output-after-incremental-input-change! deployment patches
        activations)
567     (def old-output (get-node-value deployment #this))
568     (def deployment-info (get-deployment-info deployment #this))
569     (def deployment-manager (get-deployment-manager deployment-info))
570
571
572     (def activation-values (newd Dictionary))
573     (for/iterable (activation activations)
574       (def source-deployment (get-source-deployment activation))
575       (def key (get-deployment-key deployment-manager deployment))
576       (put! activation-values key (get-value activation)))
577
578     (def output-patches (newd Vector))
579     (for/iterable (patch patches)
580       (def patch-key (get-key patch))
581       (cond ((eq? (patch-type patch) 'insert)
582              (let ((new-value (get activation-values patch-key)))
583                (if (and (not (eq? new-value #undefined)) (not (eq? new-value #none)))
584                    (push! output-patches (newd IncrementalDatastructureDeltaInsert
        patch-key new-value)))))
585             ((eq? (patch-type patch) 'update)
586              (let ((new-value (get activation-values patch-key))
587                    (replaced-value (_get-value-by-key old-output patch-key)))
588                (if (and (not (eq? new-value #undefined)) (not (eq? new-value #none)))
589                    (push! output-patches (newd IncrementalDatastructureDeltaUpdate
        patch-key replaced-value new-value))
590                    (push! output-patches (newd IncrementalDatastructureDeltaRemove
        patch-key replaced-value)))))
591             ((eq? (patch-type patch) 'delete)
592              (let ((removed-value (_get-value-by-key old-output patch-key)))
593                (push! output-patches (newd IncrementalDatastructureDeltaRemove
        patch-key removed-value))))))
594     (if (empty? output-patches)
595         old-output
596         (new IncrementalDatastructureDeltaList 'from-vector output-patches)))
597
598   (def-method (change-output-after-deployment-change! deployment activations)
599     (def deployment-info (get-deployment-info deployment #this))
600     (def deployment-manager (get-deployment-manager deployment-info))
601     (def old-output (get-node-value deployment #this))
602     (cond ((eq? (type-of old-output) 'Bag)
603            (def all-values (get-all-values #this deployment))
604            (new Bag 'from-vector all-values))
605           ((eq? (type-of old-output) 'IncrementalBag)
606            (def new-deltas (newd Vector))
607            (for/iterable (activation activations)
608              (def source-deployment (get-source-deployment activation))
609              (def key (get-deployment-key deployment-manager source-deployment))
610              (def value-in-old-output (_get-value-by-key old-output key))
611              (def new-value (get-value activation))
612
613              (if (and (not (eq? new-value #undefined))
614                       (not (eq? new-value #none)))
615
```

```
616                 (if (_contains-key? old-output key)
617                     (push! new-deltas (newd IncrementalDatastructureDeltaUpdate key
    value-in-old-output new-value))
618                     (push! new-deltas (newd IncrementalDatastructureDeltaInsert key
    new-value)))
619
620                 (if (_contains-key? old-output key)
621                     (push! new-deltas (newd IncrementalDatastructureDeltaRemove key
    value-in-old-output)))))
622             (if (empty? new-deltas)
623                 old-output
624                 (new IncrementalDatastructureDeltaList 'from-vector new-deltas)))
625           (else (println! "!!! Unknown output data structure !!!"))))
626
627   (def-method (get-all-values deployment)
628     (def deployment-info (get-deployment-info deployment #this))
629     (def deployment-manager (get-deployment-manager deployment-info))
630     (def all-deployments (get-all-deployments deployment-manager))
631     (def values (newd Vector))
632     (for/iterable (deployment all-deployments)
633       (def sink-values (get-sink-values deployment))
634       (def sink-value (first sink-values))
635       (if (and (not (eq? sink-value #undefined))
636                (not (eq? sink-value #none)))
637           (push! values (first sink-values))))
638     values))
639
640
641 (class CrossDeploymentDependent
642   (def-fields dependent-node dependent-deployment)
643   (def-constructor (default node deployment)
644     (set! dependent-node node)
645     (set! dependent-deployment deployment))
646   (def-routine (get-dependent-node) dependent-node)
647   (def-routine (get-deployment) dependent-deployment))
648
649 (class CrossDeploymentDependentsManager
650   (def-fields dependents)
651   (def-constructor (default)
652     (set! dependents (newd Dictionary)))
653
654   (def-method (add-dependent! node dependent-node dependent-deployment)
655     (def dependent (newd CrossDeploymentDependent dependent-node dependent-deployment))
656     (if (contains? dependents node)
657         (push! (get dependents node) dependent)
658         (put! dependents node (newd Vector dependent))))
659
660   (def-method (has-dependents? node)
661     (contains? dependents node))
662
663   (def-method (get-dependents node)
664     (if (has-dependents? #this node)
665         (get dependents node)
666         (newd Vector)))
667
668   (def-method (remove-dependent! node dependent-node dependent-deployment)
669     (def node-dependents (get dependents node))
670     (for/iterable (dependent node-dependents)
671       (if (and (eq? dependent-deployment (get-deployment dependent))
```

```
672                 (eq? dependent-node (get-dependent-node dependent)))
673             (delete! node-dependents dependent)))))
674
675
676 (class CrossDeploymentDependency
677   (def-fields dependency-node dependency-deployment)
678   (def-constructor (default node deployment)
679     (set! dependency-node node)
680     (set! dependency-deployment deployment))
681   (def-routine (get-dependency-node) dependency-node)
682   (def-routine (get-deployment) dependency-deployment))
683
684 (class CrossDeploymentDependencyManager
685   (def-fields dependencies)
686   (def-constructor (default)
687     (set! dependencies (newd Dictionary)))
688
689   (def-method (add-dependency! dependency-node dependency-deployment dependent-node)
690     (def dependency (newd CrossDeploymentDependency dependency-node
        dependency-deployment))
691     (if (contains? dependencies dependent-node)
692         (push! (get dependencies dependent-node) dependency)
693         (put! dependencies dependent-node (newd Vector dependency))))
694
695   (def-method (has-dependencies? node)
696     (contains? dependencies node))
697
698   (def-method (get-dependencies node)
699     (if (has-dependencies? #this node)
700         (get dependencies node)
701         (newd Vector)))
702
703   (def-method (remove-dependency! dependent-node dependency-node dependency-deployment)
704     (def node-dependencies (get dependencies dependent-node))
705     (for/iterable (dependency node-dependencies)
706       (if (and (eq? dependency-deployment (get-deployment dependency))
707                (eq? dependency-node (get-dependency-node dependency)))
708           (delete! node-dependencies dependency)))))
709
710 (class DeploymentSubscriptionManager
711   (def-fields subscription-to-node)
712
713   (def-constructor (default)
714     (set! subscription-to-node (newd Dictionary)))
715
716   (def-method (store-subscription! subscription-id data-source source-node)
717     (put! subscription-to-node subscription-id source-node))
718   (def-method (get-node subscription-id)
719     (get subscription-to-node subscription-id))
720   (def-routine (has-subscription? subscription-id)
721     (contains? subscription-to-node subscription-id))
722   (def-method (remove-subscription! subscription-id)
723     (remove! subscription-to-node subscription-id))
724   (def-method (get-subscriptions) (keys subscription-to-node)))
725
726 (class ReactorDeployment
727   (def-fields id behaviour reactive-engine node-values subscription-manager owner
        node-deployment-info cross-deployment-dependencies cross-deployment-dependents
        height-prefix derived-deployments)
```

```
728
729   (def-constructor (default _behaviour _reactive-engine _owner parent-deployment
          parent-node)
730     (set! id (make-uuid (newd Random)))
731     (set! node-values (newd Dictionary))
732     (set! behaviour _behaviour)
733     (set! reactive-engine _reactive-engine)
734     (set! subscription-manager (newd DeploymentSubscriptionManager))
735     (set! node-deployment-info (newd Dictionary))
736     (set! owner _owner)
737     (set! cross-deployment-dependencies (newd CrossDeploymentDependencyManager))
738     (set! cross-deployment-dependents (newd CrossDeploymentDependentsManager))
739     (set! derived-deployments (newd Vector))
740
741     (if parent-deployment
742         (let ((their-height-prefix (get-height-prefix parent-deployment)))
743           (set! height-prefix (new Vector 'from-vector their-height-prefix))
744           (push! height-prefix (get-height parent-node)))
745         (set! height-prefix (newd Vector)))
746
747     (for/iterable (node (get-all-nodes behaviour))
748       (if (eq? (type-of node) 'ConstantNode)
749           (set-node-value! #this node (get-value node)))))
750
751   (def-routine (get-id) id)
752   (def-routine (get-behaviour) behaviour)
753   (def-routine (get-reactive-engine) reactive-engine)
754   (def-routine (get-owner) owner)
755   (def-routine (get-node-value node) (get node-values node))
756   (def-routine (get-height-prefix) height-prefix)
757
758   (def-method (set-node-value! node new-value) (put! node-values node new-value))
759   (def-routine (get-node-value node) (get node-values node))
760
761   (def-method (get-sink-values)
762     (def values (newd Vector))
763     (def sinks (get-sinks behaviour))
764     (for/iterable (sink sinks)
765       (push! values (get-node-value #this sink)))
766     values)
767
768   (def-method (change-sources! new-values)
769     (def sources (get-sources behaviour))
770     (for (i 0 (< i (length sources)) (+ i 1))
771       (def source (get sources i))
772       (def new-value (get new-values i))
773       (if (not (eq? new-value #undefined))
774           (change-source! #this source new-value))))
775
776   (def-method (change-source! source new-value)
777     (schedule-source-node! reactive-engine source #this new-value))
778
779   (def-method (change-source*! origin-node origin-deployment source-node new-value)
780     (schedule-source-node*! reactive-engine origin-node origin-deployment source-node
          #this new-value))
781
782   (def-method (change-source-with-activation! source activation)
783     (schedule-source-activation! reactive-engine source #this activation))
784
```

```
785  (def-method (subscribe-to-stream! publisher stream-name source-node)
786    (def stream (newd LocalStream publisher stream-name))
787    (def subscription-id (make-uuid (newd Random)))
788    (store-subscription! subscription-manager subscription-id stream source-node)
789    (monitor-stream! owner subscription-id stream)
790    subscription-id)
791
792  (def-method (unsubscribe-from-stream! subscription-id)
793    (remove-subscription! subscription-manager subscription-id)
794    (unmonitor-stream! owner subscription-id))
795
796
797  (def-method (add-derived-deployment! deployment)
798    (push! derived-deployments deployment))
799
800  (def-method (clean-up!)
801    (def subscriptions (get-subscriptions subscription-manager))
802    (for/iterable (subscription subscriptions)
803      (unsubscribe-from-stream! #this subscription))
804
805    (remove-deployment! owner #this)
806    (for/iterable (deployment derived-deployments)
807      (clean-up! deployment)))
808
809  (def-method (receive-publication! subscription-id value)
810    (if (has-subscription? subscription-manager subscription-id)
811        (let ((source-node (get-node subscription-manager subscription-id)))
812          (change-source! #this source-node value))
813        (println! "deployment does not have subscription")))
814
815  (def-method (save-deployment-info! node info)
816    (put! node-deployment-info node info))
817
818  (def-method (get-deployment-info node)
819    (get node-deployment-info node))
820
821
822  (def-method (add-cross-deployment-dependency! dependent-node dependency-node
       dependency-deployment)
823    (on-deployment-dependent-added! dependency-deployment dependency-node
       dependent-node #this)
824    (add-dependency! cross-deployment-dependencies dependency-node
       dependency-deployment dependent-node))
825
826  (def-method (remove-cross-deployment-dependency! dependent-node dependency-node
       dependency-deployment)
827    (remove-dependency! cross-deployment-dependencies dependent-node dependency-node
       dependency-deployment)
828    (on-deployment-dependent-removed! dependency-deployment dependency-node
       dependent-node #this))
829
830  (def-method (on-deployment-dependent-added! dependency-node dependent-node
       dependent-deployment)
831    (add-dependent! cross-deployment-dependents dependency-node dependent-node
       dependent-deployment))
832
833  (def-method (on-deployment-dependent-removed! dependency-node dependent-node
       dependent-deployment)
```

```
834     (remove-dependent! cross-deployment-dependents dependency-node dependent-node
        dependent-deployment))
835
836   (def-method (schedule-cross-deployment-dependents! node)
837     (def node-value (get-node-value #this node))
838     (def dependents (get-dependents cross-deployment-dependents node))
839     (for/iterable (dependent dependents)
840       (def dependent-node (get-dependent-node dependent))
841       (def dependent-deployment (get-deployment dependent))
842       (schedule-source-node*! reactive-engine node #this dependent-node
        dependent-deployment node-value))))
843
844 (class ScheduledNode
845   (def-fields node deployment priority)
846
847   (def-constructor (default _node _deployment)
848     (set! node _node)
849     (set! deployment _deployment)
850     (def deployment-priority (get-height-prefix deployment))
851     (def node-priority (get-height node))
852     (set! priority (new Vector 'from-vector deployment-priority))
853     (push! priority node-priority))
854
855   (def-routine (get-node) node)
856   (def-routine (get-deployment) deployment)
857   (def-routine (get-priority) priority)
858
859   (def-routine (< other-scheduled-node)
860     (def other-node-priority (get-priority other-scheduled-node))
861     (priority<? #this priority other-node-priority))
862
863   (def-routine (priority<? priority1 priority2)
864     (if (or (empty? priority1) (empty? priority2))
865         (< (length priority) (length priority2))
866         (or (< (first priority1) (first priority2))
867             (priority<? #this (tail priority1) (tail priority2))))))
868
869 (class ScheduledSourceNode
870   (extends ScheduledNode)
871   (def-fields activations)
872   (def-constructor (default node deployment activation)
873     (super 'default node deployment)
874     (set! activations (newd Vector activation)))
875   (def-routine (get-activations) activations)
876   (def-method (add-activation! activation)
877     (push! activations activation)))
878
879 (class NodeActivation
880   (def-fields source-deployment source-node value)
881
882   (def-constructor (default _source-deployment _source-node _value)
883     (set! source-deployment _source-deployment)
884     (set! source-node _source-node)
885     (set! value _value))
886
887   (def-routine (get-source-deployment) source-deployment)
888   (def-routine (get-source-node) source-node)
889   (def-routine (get-value) value))
890
```

```
891
892 (class ReactiveEngine
893   (def-fields pq scheduled-nodes)
894
895   (def-constructor (default)
896     (set! pq (newd PriorityQueue '<))
897     (set! scheduled-nodes (newd Dictionary)))
898
899   (def-method (schedule-node! node deployment)
900     (def node-id (get-id node deployment))
901     (if (not (contains? scheduled-nodes node-id))
902         (let ((scheduled-node (newd ScheduledNode node deployment)))
903           (put! scheduled-nodes (get-id node deployment) scheduled-node)
904           (enqueue! pq scheduled-node))))
905
906   (def-method (schedule-source-node! node deployment new-value)
907     (def activation (newd NodeActivation #false #false new-value))
908     (schedule-source-activation! #this node deployment activation))
909
910   (def-method (schedule-source-node*! source-node source-deployment dependent-node
       dependent-deployment new-value)
911     (def activation (newd NodeActivation source-deployment source-node new-value))
912     (schedule-source-activation! #this dependent-node dependent-deployment activation))
913
914   (def-method (schedule-source-activation! node deployment activation)
915     (def node-id (get-id node deployment))
916     (if (contains? scheduled-nodes node-id)
917         (let ((scheduled-node (get scheduled-nodes node-id)))
918           (add-activation! scheduled-node activation))
919         (let ((scheduled-node (newd ScheduledSourceNode node deployment activation)))
920           (enqueue! pq scheduled-node)
921           (put! scheduled-nodes (get-id node deployment) scheduled-node))))
922
923   (def-method (update-loop!)
924     (when (not (empty? pq))
925       (recompute-next! #this)
926       (update-loop! #this)))
927
928   (def-method (recompute-next!)
929     (def scheduled-node (serve! pq))
930     (def node (get-node scheduled-node))
931     (def deployment (get-deployment scheduled-node))
932     (remove! scheduled-nodes (get-id node deployment))
933
934     (if (is-source? node)
935         (let ((activations (get-activations scheduled-node)))
936           (def new-source-value (activate-source! node deployment activations))
937           (store-and-schedule! #this deployment node new-source-value))
938         (when (is-computable? node deployment)
939           (let ((result (compute! node deployment)))
940             (store-and-schedule! #this deployment node result)))))
941
942   (def-method (store-and-schedule! deployment node new-value)
943     (def old-value (get-node-value deployment node))
944     (if (and (eq? (type-of old-value) 'IncrementalBag)
945              (eq? (type-of new-value) 'IncrementalDatastructureDeltaList))
946         (set! new-value (apply-patch old-value new-value)))
947     (set-node-value! deployment node new-value)
948     (schedule-dependents! #this deployment node))
```

```
949
950   (def-method (schedule-dependents! deployment node)
951     (def dependents (get-dependents node))
952     (for/iterable (dependent dependents)
953       (schedule-node! #this dependent deployment))
954     (schedule-cross-deployment-dependents! deployment node)))
955
956
957 (def-actor StreamWrapper
958   (def-stream output 2)
959   (def-fields id)
960   (def-constructor (init source-stream _id)
961     (set! id _id)
962     (monitor! source-stream 'emit-value!))
963   (def-method (emit-value! original-value)
964     (emit! output id original-value)))
965
966
967 (class SynchronousReactor
968   (def-fields reactive-engine root-deployment deployments)
969
970   (def-constructor (default behaviour)
971     (set! reactive-engine (newd ReactiveEngine))
972     (set! root-deployment (newd ReactorDeployment behaviour reactive-engine #this
        #false #false))
973     (set! deployments (newd Vector root-deployment)))
974
975   (def-method (react! new-values)
976     (change-sources! root-deployment new-values)
977     (update-loop! reactive-engine)
978     (get-sink-values #this))
979
980   (def-method (receive-publication! subscription-id value)
981     (for/iterable (deployment deployments)
982       (receive-publication! deployment subscription-id value))
983     (update-loop! reactive-engine)
984     (get-sink-values #this))
985
986   (def-method (get-sink-values)
987     (get-sink-values root-deployment))
988
989   (def-method (add-deployment! deployment)
990     (push! deployments deployment))
991
992   (def-method (remove-deployment! deployment)
993     (delete! deployments deployment))
994
995   (def-method (monitor-stream! subscription-id stream)
996     (send! #self 'monitor-stream! subscription-id stream))
997   (def-method (unmonitor-stream! subscription-id)
998     (send! #self 'unmonitor-stream! subscription-id)))
999
1000 (def-actor Reactor
1001   (def-stream output 1)
1002   (def-fields sync-reactor subscription-handles)
1003
1004   (def-constructor (init behaviour)
1005     (set! sync-reactor (newd SynchronousReactor behaviour))
1006     (set! subscription-handles (newd Dictionary)))
```

```
(def-method (monitor-stream! subscription-id stream)
  (def wrapper (spawn-actor! StreamWrapper 'init stream subscription-id))
  (def handle (monitor! wrapper.output 'receive-publication!))
  (put! subscription-handles subscription-id handle))

(def-method (unmonitor-stream! subscription-id)
  (def handle (get subscription-handles subscription-id))
  (unsubscribe! handle)
  (terminate! (get-source handle))
  (remove! subscription-handles subscription-id))


(def-method (react! new-values)
  (def sink-values (react! sync-reactor new-values))
  (println! "*** Reactor output: " sink-values)
  (emit! output sink-values))

(def-method (receive-publication! subscription-id value)
  (def sink-values (receive-publication! sync-reactor subscription-id value))
  (println! "*** Reactor output: " sink-values)
  (emit! output sink-values)))
```

Listing C.1: The complete implementation of Mira from Chapter 8.

Glossary

Actor In Stella, an actor is a process with a mailbox that continuously processes messages from its mailbox. Analogous to the well known actor model in other language (e.g., Erlang). See Section 5.4 (page 80).

Actor behaviour In Stella, an actor behaviour can be seen as the class of an actor, i.e., the program logic that describes the local state of an actor, the streams that an actor exports, and the types of messages that an actor can process. Spawning an actor behaviour yields an actor (the process) with the given actor behaviour. See Section 5.4 (page 80).

API Application Programming Interface

Behaviour A type of time-varying value in reactive programming languages and frameworks. The literature on reactive programming often makes a distinction between "events" and "behaviours" to distinguish between discrete and continuous time-varying values. In this dissertation we instead use the terminology of a signal, and reserve the term *behaviour* for actors and reactors (cf. *actor behaviour* and *reactor behaviour*).

CPS Continuation-passing Style.

CRDT Conflict-free Replicated Data Type. See Section 3.2 (page 34).

DAG Directed Acyclic Graph.

DSL Domain Specific Language.

Eventually reactive, eventual reactivity See Section 4.1.1 (page 50).

FIFO First in, first out.

FRP Functional Reactive Programming, a programming paradigm for reactive programming that combines functions (e.g., in a functional programming language such as Haskell) with time-varying values (signals) to build reactive programs.

Glitch A glitch is an incorrectly computed result by an FRP language or framework. It occurs when the reactive programming language or framework does not recompute parts of the reactive program in the correct order. See Section 2.1.5 (page 16).

IoT Internet of Things.

Lifting In the context of functional reactive programming, lifting is a technique applied by FRP languages and frameworks to "lift" non-reactive functions or methods in the base language to the level of the reactive framework [43]. Given a function `f` that works on non-reactive values, the process of lifting

creates a new function `lifted`$_f$ that can be applied to signals, and which will be recomputed automatically whenever the value of one the signals changes. See Section 2.1.2 (page 13).

LOC Lines of code.

Reaction time The time it takes for a value to propagate completely through a reactive program (from start to completion), i.e., the time it takes for the program to react to an incoming value. See Section 4.1.1 (page 50).

Reactor In Stella, a process with a mailbox that encapsulates a reactive program, and which continuously propagates messages from its mailbox by propagating the message's values through the reactive program. See Section 5.5 (page 84).

Reactor behaviour In Stella, a reactor behaviour can be seen as the class of a reactor, i.e., the program logic that describes how a reactor processes values. A reactor behaviour is a DAG that describes how values flow through the reactive program. See Section 5.5 (page 84).

Reactor deployment In Stella, reactor behaviours can be composed and used by multiple reactors. Since each use of a reactor behaviour within a reactor has a different application state at run-time, we call a reactor deployment a particular instance of a reactor behaviour. See Section 5.5.4 (page 88).

REPL Read-Eval-Print Loop.

SCT Size-Change Termination, a technique used to detect whether a function call will eventually terminate [94]. In the context of Stella, see Section 5.3.1 (page 73).

Strongly reactive, strong reactivity See Section 4.1.1 (page 50).

Time-varying value An umbrella term used to denote an abstraction of a reactive language or framework for a program value whose value can (automatically) change over time. In this dissertation we prefer to use the term signal.

Weakly reactive, weak reactivity See Section 4.1.1 (page 50).

Bibliography

[1] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. 2nd ed. Cambridge, MA, USA: MIT Press, 1996. ISBN: 0-262-01153-0.

[2] Harold Abelson and Gerald Jay Sussman. 'Structure and Interpretation of Computer Programs'. In: 2nd ed. Cambridge, MA, USA: MIT Press, 1996. Chap. 4. ISBN: 0-262-01153-0.

[3] Harold Abelson and Gerald Jay Sussman. 'Structure and Interpretation of Computer Programs'. In: 2nd ed. Cambridge, MA, USA: MIT Press, 1996. Chap. 1.1.6. ISBN: 0-262-01153-0.

[4] Feross Aboukhadijeh. *simple-peer: Simple WebRTC video, voice, and data channels*. Accessed on 2021-02-16. URL: http://web.archive.org/web/20210216115846/https://github.com/feross/simple-peer.

[5] Bowen Alpern and Fred B. Schneider. 'Defining Liveness'. In: *Information Processing Letters* 21.4 (1985), pp. 181–185. ISSN: 0020-0190. DOI: 10.1016/0020-0190(85)90056-0.

[6] Ioannis Arapakis, Souneil Park and Martin Pielot. 'Impact of Response Latency on User Behaviour in Mobile Web Search'. In: *CHIIR '21: ACM SIGIR Conference on Human Information Interaction and Retrieval, Canberra, ACT, Australia, March 14-19, 2021*. Ed. by Falk Scholer, Paul Thomas, David Elsweiler, Hideo Joho, Noriko Kando and Catherine Smith. New York, NY, USA: Association for Computing Machinery, 2021, pp. 279–283. ISBN: 978-1-4503-8055-3. DOI: 10.1145/3406522.3446038.

[7] Joe Armstrong. 'Erlang'. In: *Communications of the ACM* 53.9 (Sept. 2010), pp. 68–75. ISSN: 0001-0782. URL: https://doi.org/10.1145/1810891.1810910.

[8] Edward A. Ashcroft and William W. Wadge. *Lucid, the Data Flow Programming Language*. Cambridge, Massachusetts, USA: Academic Press, Jan. 1985. ISBN: 978-0-12-729650-0.

[9] Patrick Bahr, Christian Uldal Graulund and Rasmus Ejlers Møgelberg. 'Diamonds are not forever: liveness in reactive programming with guarded recursion'. In: *Proceedings of the ACM on Programming Languages* 5.POPL (2021), pp. 1–28. DOI: 10.1145/3434283.

[10] Engineer Bainomugisha, Andoni Lombide Carreton, Tom Van Cutsem, Stijn Mostinckx and Wolfgang De Meuter. 'A survey on reactive programming'. In: *ACM Computing Surveys* 45.4 (2013), 52:1–52:34. DOI: 10.1145/2501654.2501666.

[11] Ivo Balbaert. *Learn Red - Fundamentals of Red: Get up and running with the Red language for full-stack development*. 1st ed. Birmingham, United Kingdom: Packt Publishing, May 2018. ISBN: 978-1-78913-070-6.

[12] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount and Laurence Tratt. 'Virtual machine warmup blows hot and cold'. In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA (2017), 52:1–52:27. DOI: 10.1145/3133876.

[13] Albert Benveniste. 'Synchronous Languages and Reactive System Design'. In: *IFAC Proceedings Volumes* 31.15 (1998). 9th IFAC Symposium on Information Control in Manufacturing 1998 (INCOM '98), Nancy, France, 24-26 June, pp. 35–46. ISSN: 1474-6670. DOI: 10.1016/S1474-6670(17)40526-X.

[14] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic and Robert de Simone. 'The synchronous languages 12 years later'. In: *Proceedings of the IEEE* 91.1 (2003), pp. 64–83. DOI: 10.1109/JPROC.2002.805826.

[15] Phil Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot and Jorgen Thelin. *Orleans: Distributed Virtual Actors for Programmability and Scalability*. Tech. rep. MSR-TR-2014-41. Mar. 2014. URL: http://web.archive.org/web/20220111125054/https://www.microsoft.com/en-us/research/publication/orleans-distributed-virtual-actors-for-programmability-and-scalability/.

[16] Gérard Berry. *The constructive semantics of pure Esterel*. Draft book, version 3. Accessed on 2021-07-07. July 1999. URL: http://web.archive.org/web/20210707162633/https://www-sop.inria.fr/members/Gerard.Berry/Papers/EsterelConstructiveBook.pdf.

[17] Gérard Berry. 'The foundations of Esterel'. In: *Proof, Language, and Interaction, Essays in Honour of Robin Milner*. Ed. by Gordon D. Plotkin, Colin Stirling and Mads Tofte. Cambridge, Massachusetts, USA: MIT Press, 2000, pp. 425–454.

[18] Gérard Berry and Georges Gonthier. 'The Esterel Synchronous Programming Language: Design, Semantics, Implementation'. In: *Science of Computer Programming* 19.2 (1992), pp. 87–152. DOI: 10.1016/0167-6423(92)90005-V.

[19] Gérard Berry and Manuel Serrano. 'HipHop.Js: (A)Synchronous Reactive Web Programming'. In: *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. Ed. by Alastair F. Donaldson and Emina Torlak. New York, NY, USA: Association for Computing Machinery, 2020, pp. 533–545. ISBN: 978-1-4503-7613-6. DOI: 10.1145/3385412.3385984.

[20] Gérard Berry and Manuel Serrano. 'HipHop.js: (A)Synchronous reactive web programming'. In: *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. Ed. by Alastair F. Donaldson and Emina Torlak. Association for Computing Machinery, 2020, pp. 533–545. DOI: 10.1145/3385412.3385984.

[21] Frank S. de Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes and Albert Mingkun Yang. 'A Survey of Active Object Languages'. In: *ACM Computing Surveys* 50.5 (2017), 76:1–76:39. DOI: 10.1145/3122848.

[22] Elisa Gonzalez Boix, Andoni Lombide Carreton, Christophe Scholliers, Tom Van Cutsem, Wolfgang De Meuter and Theo D'Hondt. 'Flocks: enabling dynamic group interactions in mobile social networking applications'. In: *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC), TaiChung, Taiwan, March 21 - 24, 2011*. Ed. by William C. Chu, W. Eric Wong, Mathew J. Palakal and Chih-Cheng Hung. New York, NY, USA: Association for Computing Machinery, 2011, pp. 425–432. DOI: 10.1145/1982185.1982277.

[23] Jonas Bonér, Dave Farley, Roland Kuhn and Martin Thompson. *The Reactive Manifesto*. Accessed on 2019-12-10. URL: https://web.archive.org/web/20191210084324/https://www.reactivemanifesto.org/.

[24] Walter S Brainerd and Lawrence H Landweber. 'Theory of computation'. In: John Wiley & Sons, Inc., 1974. Chap. 3. ISBN: 978-0-471-09585-9. URL: https://archive.org/details/theoryofcomputat00brai.

[25] Marc Brockschmidt, Byron Cook, Samin Ishtiaq, Heidy Khlaaf and Nir Piterman. 'T2: Temporal Property Verification'. In: *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, Netherlands, April 2-8, 2016, Proceedings*. Ed. by Marsha Chechik and Jean-François Raskin. Vol. 9636. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer-Verlag, 2016, pp. 387–393. DOI: 10.1007/978-3-662-49674-9_22.

[26] Jan-Ivar Bruaroey, Cullen Jennings and Henrik Boström. *WebRTC 1.0: Real-Time Communication Between Browsers*. W3C Recommendation. Accessed on 2021-02-10. W3C, Jan. 2021. URL: http://web.archive.org/web/20210210094400/https://www.w3.org/TR/2021/REC-webrtc-20210126/.

[27] Andoni Lombide Carreton, Stijn Mostinckx, Tom Van Cutsem and Wolfgang De Meuter. 'Loosely-Coupled Distributed Reactive Programming in Mobile Ad Hoc Networks'. In: *Objects, Models, Components, Patterns, 48th International Conference, TOOLS 2010, Málaga, Spain, June 28 - July 2, 2010. Proceedings*. Ed. by Jan Vitek. Vol. 6141. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 41–60. DOI: 10.1007/978-3-642-13953-6_3.

[28] Paul Caspi, Christine Mazuet, Rym Salem and Daniel Weber. 'Formal Design of Distributed Control Systems with Lustre'. In: *Computer Safety, Reliability and Security, 18th International Conference, SAFECOMP'99, Toulouse, France, September, 1999, Proceedings*. Ed. by Massimo Felici, Karama Kanoun and Alberto Pasquini. Vol. 1698. Lecture Notes in Computer Science. Springer, 1999, pp. 396–409. DOI: 10.1007/3-540-48249-0_34.

[29] Byron Cook, Andreas Podelski and Andrey Rybalchenko. 'Proving program termination'. In: *Communications of the ACM* 54.5 (2011), pp. 88–98. DOI: 10.1145/1941487.1941509.

[30] Gregory H. Cooper. 'Integrating Dataflow Evaluation into a Practical Higher-Order Call-by-Value Language'. PhD thesis. Providence, Rhode Island: Brown University, May 2008. Chap. 2.10.

[31] Gregory H. Cooper and Shriram Krishnamurthi. 'Embedding Dynamic Dataflow in a Call-by-Value Language'. In: *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*. Ed. by Peter Sestoft. Vol. 3924. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 294–308. DOI: 10.1007/11693024_20.

[32] Oracle Corporation. *JEP 266: More Concurrency Updates*. https://web.archive.org/web/20191009093608/https://openjdk.java.net/jeps/266. Accessed on 2019-10-09.

[33] Antony Courtney. 'Frappé: Functional Reactive Programming in Java'. In: *Practical Aspects of Declarative Languages, Third International Symposium, PADL 2001, Las Vegas, Nevada, USA, March 11-12, 2001, Proceedings*. Ed. by I. V. Ramakrishnan. Vol. 1990. Lecture Notes in Computer Science.

Berlin, Heidelberg: Springer-Verlag, 2001, pp. 29–44. DOI: 10.1007/3-540-45241-9_3.

[34] 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'. In: *Computer Languages, Systems and Structures* 40.3-4 (2014), pp. 112–136. ISSN: 1477-8424. DOI: 10.1016/j.cl.2014.05.002.

[35] Evan Czaplicki. *A Farewell to FRP: Making signals unnecessary with The Elm Architecture*. Accessed on 2021-07-09. May 2016. URL: http://web.archive.org/web/20210709142428/https://elm-lang.org/news/farewell-to-frp.

[36] Evan Czaplicki and Stephen Chong. 'Asynchronous functional reactive programming for GUIs'. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. Ed. by Hans-Juergen Boehm and Cormac Flanagan. New York, NY, USA: Association for Computing Machinery, 2013, pp. 411–422. DOI: 10.1145/2491956.2462161.

[37] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D'Hondt and Wolfgang De Meuter. 'Ambient-Oriented Programming in AmbientTalk'. In: *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings*. Ed. by Dave Thomas. Vol. 4067. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 230–254. DOI: 10.1007/11785477_16.

[38] Cloé Descheemaeker, Sam Van den Vonder, Thierry Renaux and Wolfgang De Meuter. 'Poker: Visual Instrumentation of Reactive Programs with Programmable Probes'. In: *REBLS 2021: Proceedings of the 8th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems, Chicago, IL, USA, 18 October 2021*. Ed. by Louis Mandel. New York, NY, USA: Association for Computing Machinery, 2021, pp. 14–26. ISBN: 978-1-4503-9108-5. DOI: 10.1145/3486605.3486785.

[39] Terrence Dorsey. *Preview of Reactive Extensions for .NET Available*. Accessed on 2021-06-28. Nov. 2009. URL: http://web.archive.org/web/20210628072108/https://visualstudiomagazine.com/articles/2009/11/19/preview-of-reactive-extensions-for-.net-available.aspx.

[40] Joscha Drechsler, Ragnar Mogk, Guido Salvaneschi and Mira Mezini. 'Thread-safe reactive programming'. In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018), 107:1–107:30. DOI: 10.1145/3276477.

[41] Jonathan Edwards. *Coherent Reaction*. Tech. rep. MIT-CSAIL-TR-2009-024. Computer Science and Artificial Intelligence Laboratory, Cambridge, 02139 Massachusetts, USA: Massachusetts Institute of Technology, June 2009. URL: http://web.archive.org/web/20181103183154/http://dspace.mit.edu/bitstream/handle/1721.1/45563/MIT-CSAIL-TR-2009-024.pdf?sequence=1.

[42] Jonathan Edwards. 'Coherent reaction'. In: *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*. Ed. by Shail Arora and Gary T. Leavens. New York, NY, USA: Association for Computing Machinery, 2009, pp. 925–932. DOI: 10.1145/1639950.1640058.

[43] Conal Elliott and Paul Hudak. 'Functional Reactive Animation'. In: *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, Netherlands, June 9-11, 1997*. Ed. by Simon L. Peyton Jones, Mads Tofte and A. Michael Berman. New York, NY, USA: Association for Computing Machinery, 1997, pp. 263–273. DOI: 10.1145/258948.258973.

[44] Conal M. Elliott. 'Push-pull functional reactive programming'. In: *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell, Haskell 2009, Edinburgh, Scotland, UK, 3 September 2009*. Ed. by Stephanie Weirich. New York, NY, USA: Association for Computing Machinery, 2009, pp. 25–36. DOI: 10.1145/1596638.1596643.

[45] U.S. Census Bureau Geographic Information Systems FAQ. *GIS FAQ Q5.1: Great circle distance between 2 points*. Accessed on 2021-02-10. URL: http://web.archive.org/web/20201029002559/http://www.movable-type.co.uk/scripts/gis-faq-5.1.html.

[46] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay A. McCarthy and Sam Tobin-Hochstadt. 'A programmable programming language'. In: *Communications of the ACM* 61.3 (2018), pp. 62–71. DOI: 10.1145/3127323.

[47] Matthew Flatt, Robert Bruce Findler, Shriram Krishnamurthi and Matthias Felleisen. 'Programming Languages as Operating Systems (or Revenge of the Son of the Lisp Machine)'. In: *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99), Paris, France, September 27-29, 1999*. Ed. by Didier Rémy and Peter Lee. New York, NY, USA: Association for Computing Machinery, 1999, pp. 138–147. DOI: 10.1145/317636.317793.

[48] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story and David Walker. 'Frenetic: a network programming language'. In: *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*. Ed. by Manuel M. T. Chakravarty, Zhenjiang Hu and Olivier Danvy. New York, NY, USA: Association for Computing Machinery, 2011, pp. 279–291. DOI: 10.1145/2034773.2034812.

[49] OpenJS Foundation. *PEG.js: Parser Generator for JavaScript*. Accessed on 2022-05-23. URL: http://web.archive.org/web/20220523103613/https://nodejs.org/en/.

[50] Daniel P. Friedman and Mitchell Wand. 'Essentials of programming languages'. In: 3rd ed. Cambridge, Massachusetts, USA: MIT Press, 2008. Chap. 5. ISBN: 978-0-262-06279-4.

[51] *FS2: The Official Guide*. Accessed on 2020-08-24. URL: http://web.archive.org/web/20200824074626/https://fs2.io/guide.html.

[52] Abdoulaye Gamatié and Thierry Gautier. 'The Signal Synchronous Multiclock Approach to the Design of Distributed Embedded System'. In: *IEEE Transactions on Parallel and Distributed Systems* 21.5 (May 2010), pp. 641–657. DOI: 10.1109/TPDS.2009.125.

[53] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st ed. Boston, Massachusetts, USA: Addison-Wesley Professional, 1994. ISBN: 0-201-63361-2.

[54] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer and David Culler. 'The NesC Language: A Holistic Approach to Networked Embedded Systems'. In: *ACM SIGPLAN Notices* 49.4S (July 2014), pp. 41–51. ISSN: 0362-1340. DOI: 10.1145/2641638.2641652.

[55] Alain Girault. 'A Survey of Automatic Distribution Method for Synchronous Programs'. In: *International Workshop on Synchronous Languages, Applications and Programs, SLAP'05*. Ed. by F. Maraninchi, M. Pouzet and V. Roy. ENTCS. Edinburgh, UK: Elsevier Science, Apr. 2005.

[56] Adele Goldberg and David Robson. 'Smalltalk-80: The Language and Its Implementation'. In: Boston, Massachusetts, USA: Addison-Wesley Longman Publishing Co., Inc., Jan. 1983. Chap. 5. ISBN: 978-0-201-11371-6.

[57] Ido Green. *Web Workers: Multithreaded Programs in JavaScript*. Sebastopol, CA, USA: O'Reilly Media, 2012. ISBN: 978-1-4493-2213-7.

[58] Ilya Grigorik. *High Resolution Time Level 2*. W3C Recommendation. Accessed on 2021-03-25. W3C, Nov. 2019. URL: https://web.archive.org/web/20210325154905/https://www.w3.org/TR/2019/REC-hr-time-2-20191121/.

[59] Nicholas Halbwachs, Paul Caspi, Pascal Raymond and Daniel Pilaud. 'The synchronous data flow programming language LUSTRE'. In: *Proceedings of the IEEE* 79.9 (1991), pp. 1305–1320. DOI: 10.1109/5.97300.

[60] Nicolas Halbwachs. 'A synchronous language at work: the story of Lustre'. In: *3rd ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2005), 11-14 July 2005, Verona, Italy, Proceedings*. Washington, District of Columbia, USA: IEEE Computer Society, 2005, pp. 3–11. DOI: 10.1109/MEMCOD.2005.1487884.

[61] Nicolas Halbwachs. 'Synchronous Programming of Reactive Systems'. In: 1st ed. Dordrecht, Netherlands: Springer Science+Business Media, 1993. Chap. 1.4. ISBN: 978-1-4757-2231-4.

[62] Dries Harnie, Elisa Gonzalez Boix, Andoni Lombide Carreton, Christophe Scholliers and Wolfgang De Meuter. 'Volatile Sets: Event-driven Collections for Mobile Ad-Hoc Applications'. In: *Electronic Communications of the EASST* 43 (2011). DOI: 10.14279/tuj.eceasst.43.585.

[63] Paul Hudak, Antony Courtney, Henrik Nilsson and John Peterson. 'Arrows, Robots, and Functional Reactive Programming'. In: *Advanced Functional Programming, 4th International School, AFP 2002, Oxford, UK, August 19-24, 2002, Revised Lectures*. Ed. by Johan Jeuring and Simon L. Peyton Jones. Vol. 2638. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer-Verlag, 2002, pp. 159–187. DOI: 10.1007/978-3-540-44833-4_6.

[64] Paul Hudak and Donya Quick. *The Haskell School of Music: From Signals to Symphonies*. Cambridge, United Kingdom: Cambridge University Press, Sept. 2018. ISBN: 978-1-108-24186-1. DOI: 10.1017/9781108241861.

[65] Daniel Ignatoff, Gregory H. Cooper and Shriram Krishnamurthi. 'Crossing State Lines: Adapting Object-Oriented Frameworks to Functional Reactive Languages'. In: *Functional and Logic Programming, 8th International Symposium, FLOPS 2006, Fuji-Susono, Japan, April 24-26, 2006, Proceedings*. Ed. by Masami Hagiya and Philip Wadler. Vol. 3945. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 259–276. DOI: 10.1007/11737414_18.

[66] Christopher Ireland, David Bowers, Michael Newton and Kevin Waugh. 'A Classification of Object-Relational Impedance Mismatch'. In: *The First International Conference on Advances in Databases, Knowledge, and Data Applications, DBKDS 2009, Gosier, Guadeloupe, France, 1-6 March 2009*.

Ed. by Qiming Chen, Alfredo Cuzzocrea, Takahiro Hara, Ela Hunt and Manuela Popescu. Washington, District of Columbia, USA: IEEE Computer Society, 2009, pp. 36–43. DOI: 10.1109/DBKDA.2009.11.

[67] Saša Jurić. *Elixir in Action*. 2nd ed. Shelter Island, NY, USA: Manning Publications, Jan. 2019. ISBN: 978-1-61729-502-7.

[68] Kennedy Kambona, Elisa Gonzalez Boix and Wolfgang De Meuter. 'An Evaluation of Reactive Programming and Promises for Structuring Collaborative Web Applications'. In: *Proceedings of the 7th Workshop on Dynamic Languages and Applications, DYLA 2013, 1 July 2013, Montpellier, France*. New York, NY, USA: Association for Computing Machinery, 2013. ISBN: 978-1-4503-2041-2. DOI: 10.1145/2489798.2489802.

[69] Rick Kawamura. *How Hard is it to Build an IoT Electric Scooter Fleet like Bird or Lime?* Accessed on 2021-01-19. Sept. 2019. URL: http://web.archive.org/web/20210119133924/https://www.soracom.io/blog/how-hard-is-it-to-build-an-electric-scooter-fleet-like-bird-or-lime/.

[70] Joeri De Koster, Tom Van Cutsem and Wolfgang De Meuter. '43 years of actors: a taxonomy of actor models and their key properties'. In: *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE 2016, Amsterdam, Netherlands, October 30, 2016*. Ed. by Sylvan Clebsch, Travis Desell, Philipp Haller and Alessandro Ricci. New York, NY, USA: Association for Computing Machinery, 2016, pp. 31–40. DOI: 10.1145/3001886.3001890.

[71] Neelakantan R. Krishnaswami. 'Higher-order functional reactive programming without spacetime leaks'. In: *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*. Ed. by Greg Morrisett and Tarmo Uustalu. New York, NY, USA: Association for Computing Machinery, 2013, pp. 221–232. DOI: 10.1145/2500365.2500588.

[72] Roland Kuhn, Brian Hanafee and Jamie Allen. 'Reactive Design Patterns'. In: 1st ed. Greenwich, Connecticut, USA: Manning Publications Co., 2017. Chap. 1. ISBN: 978-1-61729-180-7.

[73] Paul Le Guernic, Thierry Gautier, Michel Le Borgne and Claude Le Maire. 'Programming Real-Time Applications with Signal'. In: *Proceedings of the IEEE*. Another look at Real-time programming 79.9 (1991), pp. 1321–1336. DOI: 10.1109/5.97301.

[74] Lightbend Inc. *Actor discovery – Akka Documentation*. Accessed on 2021-09-21. URL: http://web.archive.org/web/20210921133010/https://doc.akka.io/docs/akka/2.5.32/typed/actor-discovery.html.

[75] Lightbend Inc. *Pipelining and Parallelism – Akka Documentation*. Accessed on 2021-09-21. URL: http://web.archive.org/web/20210921082808/https://doc.akka.io/docs/akka/current/stream/stream-parallelism.html.

[76] Lightbend Inc. *Source.actorRef – Akka Documentation*. Accessed on 2022-01-13. URL: http://web.archive.org/web/20220113105109/https://doc.akka.io/docs/akka/current/stream/operators/Source/actorRef.html.

[77] Lightbend Inc. *Streams – Akka Documentation*. Accessed on 2021-09-16. URL: http://web.archive.org/web/20210814110535/https://doc.akka.io/docs/akka/current/stream/index.html.

[78] Lightbend Inc. *Working with Graphs: Constructing Graphs*. Accessed on 2022-01-06. URL: http://web.archive.org/web/20220106141630/https://doc.akka.io/docs/akka/current/stream/stream-graphs.html.

[79] Ingo Maier and Martin Odersky. *Deprecating the Observer Pattern with Scala.React*. Tech. rep. EPFL-REPORT-176887. EPFL IC IINFCOM LAMP, Station 14, 1015 Lausanne: École Polytechnique Fédérale de Lausanne, 2012, p. 20. URL: http://web.archive.org/web/20200522141109/https://infoscience.epfl.ch/record/176887.

[80] Ingo Maier and Martin Odersky. 'Higher-Order Reactive Programming with Incremental Lists'. In: *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings*. Ed. by Giuseppe Castagna. Vol. 7920. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 707–731. DOI: 10.1007/978-3-642-39038-8_29.

[81] Geoffrey Mainland, Greg Morrisett and Matt Welsh. 'Flask: Staged Functional Programming for Sensor Networks'. In: *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*. Ed. by James Hook and Peter Thiemann. ICFP '08. New York, NY, USA: Association for Computing Machinery, 2008, pp. 335–346. ISBN: 9781595939197. DOI: 10.1145/1411204.1411251.

[82] David Majda and Futago-za Ryuu. *Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine.* Accessed on 2022-05-24. URL: http://web.archive.org/web/20220524122217/https://pegjs.org/.

[83] Alessandro Margara and Guido Salvaneschi. 'On the Semantics of Distributed Reactive Programming: The Cost of Consistency'. In: *IEEE Transactions on Software Engineering* 44.7 (2018), pp. 689–711. DOI: 10.1109/TSE.2018.2833109.

[84] Alessandro Margara and Guido Salvaneschi. 'We have a DREAM: distributed reactive programming with consistency guarantees'. In: *The 8th ACM International Conference on Distributed Event-Based Systems, DEBS '14, Mumbai, India, May 26-29, 2014*. Ed. by Umesh Bellur and Ravi Kothari. New York, NY, USA: Association for Computing Machinery, 2014, pp. 142–153. DOI: 10.1145/2611286.2611290.

[85] David A. McAllester and Kostas Arkoudas. 'Walther Recursion'. In: *Automated Deduction - CADE-13, 13th International Conference on Automated Deduction, New Brunswick, NJ, USA, July 30 - August 3, 1996, Proceedings*. Ed. by Michael A. McRobbie and John K. Slaney. Vol. 1104. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 643–657. DOI: 10.1007/3-540-61511-3_119.

[86] Erik Meijer. *MS Open Tech Open Sources Rx (Reactive Extensions) - a Cure for Asynchronous Data Streams in Cloud Programming*. Accessed on 2021-06-28. Nov. 2012. URL: http://web.archive.org/web/20210628072148/https://docs.microsoft.com/en-gb/archive/blogs/interoperability/ms-open-tech-open-sources-rx-reactive-extensions-a-cure-for-asynchronous-data-streams-in-cloud-programming.

[87] Alexey Melnikov and Ian Fette. *The WebSocket Protocol*. RFC 6455. Dec. 2011. DOI: 10.17487/RFC6455. URL: https://rfc-editor.org/rfc/rfc6455.txt.

[88] 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: *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*. Ed. by Shail Arora and Gary T. Leavens. New York, NY, USA: Association for Computing Machinery, 2009, pp. 1–20. DOI: 10.1145/1640089.1640091.

[89] Tommi Mikkonen and Antero Taivalsaari. 'Web Applications - Spaghetti Code for the 21st Century'. In: *Proceedings of the 6th ACIS International Conference on Software Engineering Research, Management and Applications, SERA 2008, 20-22 August 2008, Prague, Czech Republic*. Ed. by Walter Dosch, Roger Y. Lee, Petr Tuma and Thierry Coupaye. IEEE Computer Society, 2008, pp. 319–328. DOI: 10.1109/SERA.2008.16.

[90] Ragnar Mogk, Lars Baumgärtner, Guido Salvaneschi, Bernd Freisleben and Mira Mezini. 'Fault-tolerant Distributed Reactive Programming'. In: *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, Netherlands*. Ed. by Todd D. Millstein. Vol. 109. LIPIcs Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 1:1–1:26. DOI: 10.4230/LIPIcs.ECOOP.2018.1.

[91] *Monix: Asynchronous Programming for Scala and Scala.js*. Accessed on 2021-06-28. URL: http://web.archive.org/web/20210628091425/https://monix.io/.

[92] Mozilla and individual contributors. *Array - JavaScript | MDN*. Accessed on 2021-10-27. URL: http://web.archive.org/web/20211016220924/https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array.

[93] Robert H. B. Netzer and Barton P. Miller. 'What Are Race Conditions? Some Issues and Formalizations'. In: *ACM Letters on Programming Languages and Systems* 1.1 (Mar. 1992), pp. 74–88. ISSN: 1057-4514. DOI: 10.1145/130616.130623.

[94] Phuc C. Nguyen, Thomas Gilray, Sam Tobin-Hochstadt and David Van Horn. 'Size-change termination as a contract: dynamically and statically enforcing termination for higher-order programs'. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. Ed. by Kathryn S. McKinley and Kathleen Fisher. New York, NY, USA: Association for Computing Machinery, 2019, pp. 845–859. DOI: 10.1145/3314221.3314643.

[95] Henrik Nilsson, John Peterson and Paul Hudak. 'Functional Hybrid Modeling'. In: *Practical Aspects of Declarative Languages, 5th International Symposium, PADL 2003, New Orleans, LA, USA, January 13-14, 2003, Proceedings*. Ed. by Verónica Dahl and Philip Wadler. Vol. 2562. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 376–390. DOI: 10.1007/3-540-36388-2_25.

[96] Bryan O'Sullivan, John Goerzen and Don Stewart. 'Real world Haskell: code you can believe in'. In: Sebastopol, California, USA: O'Reilly Media, Inc, Nov. 2008. Chap. 14. ISBN: 978-0-596-51498-3. URL: http://book.realworldhaskell.org/.

[97] Bjarno Oeyen, Humberto Rodríguez-Avila, Sam Van den Vonder and Wolfgang De Meuter. 'Composable higher-order reactors as the basis for a live reactive programming environment'. In: *Proceedings of the 5th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems, REBLS@SPLASH 2018, Boston, MA, USA, November 4, 2018*. Ed. by

Guido Salvaneschi, Wolfgang De Meuter, Patrick Eugster, Lukasz Ziarek and Francisco Sant'Anna. New York, NY, USA: Association for Computing Machinery, 2018, pp. 51–60. DOI: 10.1145/3281278.3281284.

[98] Bjarno Oeyen, Sam Van den Vonder and Wolfgang De Meuter. 'Reactive Sorting Networks'. In: *Proceedings of the 7th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*. REBLS 2020. Virtual, USA: Association for Computing Machinery, 2020, pp. 38–50. ISBN: 978-1-4503-8188-8. DOI: 10.1145/3427763.3428316.

[99] Bjarno Oeyen, Sam Van den Vonder and Wolfgang De Meuter. 'Trampoline Variables: A General Method for State Accumulation in Reactive Programming'. In: *Proceedings of the 8th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*. Ed. by Louis Mandel. REBLS 2021. Chicago, IL, USA: Association for Computing Machinery, 2021, pp. 27–40. ISBN: 978-1-4503-9108-5. DOI: 10.1145/3486605.3486787.

[100] Oracle. *Java Remote Method Invocation Specification: 2 - Distributed Object Model*. Accessed on 2022-01-07. URL: http://web.archive.org/web/20220107155832/https://docs.oracle.com/en/java/javase/17/docs/specs/rmi/objmodel.html.

[101] Sean Parent. 'A Possible Future for Software Development'. BoostCon 2007 keynote, accessed on 2021-06-24. May 2007. URL: http://web.archive.org/web/20210624121558/https://stlab.cc/legacy/figures/Boostcon_possible_future.pdf.

[102] Ivan Perez, Manuel Bärenz and Henrik Nilsson. 'Functional Reactive Programming, Refactored'. In: *SIGPLAN Not.* 51.12 (Sept. 2016), pp. 33–44. ISSN: 0362-1340. DOI: 10.1145/3241625.2976010.

[103] Ivan Perez, Manuel Bärenz and Henrik Nilsson. 'Functional reactive programming, refactored'. In: *Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016*. Ed. by Geoffrey Mainland. New York, NY, USA: Association for Computing Machinery, 2016, pp. 33–44. ISBN: 9781450344340. DOI: 10.1145/2976002.2976010.

[104] John Peterson, Paul Hudak, Alastair Reid and Gregory D. Hager. 'FVision: A Declarative Language for Visual Tracking'. In: *Practical Aspects of Declarative Languages, Third International Symposium, PADL 2001, Las Vegas, Nevada, USA, March 11-12, 2001, Proceedings*. Ed. by I. V. Ramakrishnan. Vol. 1990. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 304–321. DOI: 10.1007/3-540-45241-9_21.

[105] Simon Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. 1st ed. Cambridge, United Kingdom: Cambridge University Press, May 2003. ISBN: 978-0-521-82614-3.

[106] Atze van der Ploeg and Koen Claessen. 'Practical principled FRP: forget the past, change the future, FRPNow!' In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. Ed. by Kathleen Fisher and John H. Reppy. New York, NY, USA: Association for Computing Machinery, 2015, pp. 302–314. DOI: 10.1145/2784731.2784752.

[107] José Proença and Carlos Baquero. 'Quality-Aware Reactive Programming for the Internet of Things'. In: *Fundamentals of Software Engineering - 7th International Conference, FSEN 2017, Tehran, Iran, April 26-28, 2017, Revised Selected Papers*. Ed. by Mehdi Dastani and Marjan Sirjani. Vol. 10522. Lecture Notes in Computer Science. Cham, Switzerland: Springer International Publishing, 2017, pp. 180–195. DOI: 10.1007/978-3-319-68972-2_12.

[108] *Project Reactor: Create Efficient Reactive Systems*. Accessed on 2020-09-08. URL: http://web.archive.org/web/20200817051359/https://projectreactor.io/.

[109] Belga / N. Quintelier. *Villo lanceert 1.800 elektrische stadsfietsen in Brussel*. Accessed on 2021-01-19. Nov. 2019. URL: http://web.archive.org/web/20210119143206/https://nl.metrotime.be/2019/11/30/must-read/villo-lanceert-1-800-elektrische-stadsfietsen-in-brussel/.

[110] Nenad Rakocevic and Red Foundation. *Red Programming Language*. Accessed on 2021-09-17. URL: http://web.archive.org/web/20210917113253/https://www.red-lang.org/.

[111] *Reactive Streams*. Accessed on 2019-10-09. URL: http://web.archive.org/web/20191009093755/https://www.reactive-streams.org/.

[112] *ReactiveX - Languages*. Accessed on 2021-06-28. URL: http://web.archive.org/web/20210628072948/http://reactivex.io/languages.html.

[113] *ReactiveX: An API for asynchronous programming with observable streams*. Accessed on 2019-10-09. URL: http://web.archive.org/web/20191009085652/http://reactivex.io/.

[114] *REScala Manual*. Section 1.5.3. Accessed on 2019-11-26. URL: https://web.archive.org/web/20191126124033/http://www.rescala-lang.com/manual/.

[115] RexEgg. *The Explosive Quantifier Trap*. Accessed on 2019-12-23. URL: https://web.archive.org/web/20191223155226/https://www.rexegg.com/regex-explosive-quantifiers.html.

[116] Bob Reynders and Dominique Devriese. 'Efficient Functional Reactive Programming Through Incremental Behaviors'. In: *Programming Languages and Systems - 15th Asian Symposium, APLAS 2017, Suzhou, China, November 27-29, 2017, Proceedings*. Ed. by Bor-Yuh Evan Chang. Vol. 10695. Lecture Notes in Computer Science. Switzerland: Springer International Publishing AG, 2017, pp. 321–338. DOI: 10.1007/978-3-319-71237-6_16.

[117] Bob Reynders, Dominique Devriese and Frank Piessens. 'Multi-Tier Functional Reactive Programming for the Web'. In: *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*. Ed. by Andrew P. Black, Shriram Krishnamurthi, Bernd Bruegge and Joseph N. Ruskiewicz. New York, NY, USA: Association for Computing Machinery, 2014, pp. 55–68. DOI: 10.1145/2661136.2661140.

[118] Bob Reynders, Frank Piessens and Dominique Devriese. 'Gavial: Programming the web with multi-tier FRP'. In: *The Art, Science, and Engineering of Programming* 4.3 (2020), p. 6. ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2020/4/6.

[119] Raymond Roestenburg, Rob Bakker and Rob Williams. 'Akka in action'. In: 1st ed. Greenwich, Connecticut, USA: Manning Publications Co., 2016. Chap. 13. ISBN: 978-1-61729-101-2.

[120] *RxJS Documentation - Scheduler*. Accessed on 2021-09-21. URL: http://web.archive.org/web/20210921080341/https://rxjs.dev/guide/scheduler.

[121] *RxJS: Reactive Extensions For JavaScript*. Accessed on 2020-08-20. URL: http://web.archive.org/web/20200820080335/https://github.com/ReactiveX/rxjs.

[122] Esterel Technologies SA. *SCADE Success Stories*. Accessed on 2021-07-06. URL: http://web.archive.org/web/20190130201836/http://www.esterel-technologies.com/success-stories/.

[123] Guido Salvaneschi, Gerold Hintz and Mira Mezini. 'REScala: bridging between object-oriented and functional style in reactive applications'. In: *13th International Conference on Modularity, MODULARITY '14, Lugano, Switzerland, April 22-26, 2014*. Ed. by Walter Binder, Erik Ernst, Achille Peternier and Robert Hirschfeld. New York, NY, USA: Association for Computing Machinery, 2014, pp. 25–36. DOI: 10.1145/2577080.2577083.

[124] Guido Salvaneschi, Sebastian Proksch, Sven Amann, Sarah Nadi and Mira Mezini. 'On the Positive Effect of Reactive Programming on Software Comprehension: An Empirical Study'. In: *IEEE Transactions on Software Engineering* 43.12 (2017), pp. 1125–1143. DOI: 10.1109/TSE.2017.2655524.

[125] Francisco Sant'Anna, Roberto Ierusalimschy, Noemi de La Rocque Rodriguez, Silvana Rossetto and Adriano Branco. 'The Design and Implementation of the Synchronous Language CÉU'. In: *ACM Transactions on Embedded Computing Systems* 16.4 (2017), 98:1–98:26. DOI: 10.1145/3035544.

[126] Francisco Sant'anna, Roberto Ierusalimschy, Noemi Rodriguez, Silvana Rossetto and Adriano Branco. 'The Design and Implementation of the Synchronous Language CÉU'. In: *ACM Transactions on Embedded Computing Systems* 16.4 (July 2017). ISSN: 1539-9087. DOI: 10.1145/3035544.

[127] Kensuke Sawada and Takuo Watanabe. 'Emfrp: a functional reactive programming language for small-scale embedded systems'. In: *Companion Proceedings of the 15th International Conference on Modularity, Málaga, Spain, March 14 - 18, 2016*. Ed. by Lidia Fuentes, Don S. Batory and Krzysztof Czarnecki. New York, NY, USA: Association for Computing Machinery, 2016, pp. 36–44. DOI: 10.1145/2892664.2892670.

[128] Scala-lang. *Mutable and Immutable Collections | Collections | Scala Documentation*. Accessed on 2021-10-27. URL: http://web.archive.org/web/20210106103257/https://docs.scala-lang.org/overviews/collections-2.13/overview.html.

[129] Manuel Serrano and Gérard Berry. 'Multitier Programming in Hop'. In: *Communications of the ACM* 55.8 (Aug. 2012), pp. 53–59. ISSN: 0001-0782. DOI: 10.1145/2240236.2240253.

[130] 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, Jan. 2011, p. 50. URL: http://web.archive.org/web/20220110090523/https://hal.inria.fr/inria-00555588.

[131] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero and Marek Zawirski. 'Conflict-Free Replicated Data Types'. In: *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*. Ed. by Xavier Défago, Franck Petit and Vincent Villain. Vol. 6976. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 386–400. DOI: 10.1007/978-3-642-24550-3_29.

[132] Kazuhiro Shibanai and Takuo Watanabe. 'Distributed functional reactive programming on actor-based runtime'. In: *Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE!@SPLASH 2018, Boston, MA, USA, November 5, 2018*. Ed. by Joeri De Koster, Federico Bergenti and Juliana Franco. New York, NY, USA: Association for Computing Machinery, 2018, pp. 13–22. DOI: 10.1145/3281366.3281370.

[133] Michael Sperber. 'Developing a Stage Lighting System from Scratch'. In: *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001*. Ed. by Benjamin C. Pierce. New York, NY, USA: Association for Computing Machinery, 2001, pp. 122–133. DOI: 10.1145/507635.507652.

[134] 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'. In: *Art Sci. Eng. Program.* 5.2 (2021), p. 4. DOI: 10.22152/programming-journal.org/2021/5/4.

[135] *Streamz: Streamz Documentation*. Accessed on 2020-09-08. URL: http://web.archive.org/web/20200908160021/https://streamz.readthedocs.io/en/latest/.

[136] Kohei Suzuki, Kanato Nagayama, Kensuke Sawada and Takuo Watanabe. 'CFRP: A functional reactive programming language for small-scale embedded systems'. In: *Proceedings of the 6th Workshop on Computing: Theory and Practice WCTP2016, Cebu City, The Philippines, September 21-22, 2016*. Ed. by Shin-ya Nishizaki, Masayuki Numao and Jaime D L Caro Merlin Teodosia C Suarez. Singapore: World Scientific, Dec. 2017, pp. 1–13. ISBN: 978-981-3234-08-6. DOI: 10.1142/9789813234079_0001.

[137] Olivier Tardieu and Robert de Simone. 'Loops in esterel'. In: *ACM Transactions on Embedded Computing Systems* 4.4 (2005), pp. 708–750. DOI: 10.1145/1113830.1113832.

[138] Christophe De Troyer, Jens Nicolay and Wolfgang De Meuter. 'Building IoT Systems Using Distributed First-Class Reactive Programming'. In: *2018 IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2018, Nicosia, Cyprus, December 10-13, 2018*. Washington, D.C., USA: IEEE Computer Society, 2018, pp. 185–192. DOI: 10.1109/CloudCom2018.2018.00045.

[139] Christophe De Troyer, Jens Nicolay and Wolfgang De Meuter. 'The Art of the Meta Stream Protocol: Torrents of Streams'. In: *The Art, Science, and Engineering of Programming* 6.1 (2022), p. 2. DOI: 10.22152/programming-journal.org/2022/6/2.

[140] D. A. Turner. 'Total Functional Programming'. In: *Journal of Universal Computer Science* 10.7 (2004), pp. 751–768. DOI: 10.3217/jucs-010-07-0751.

[141] Sam Van den Vonder, Thierry Renaux and Wolfgang De Meuter. 'Topology-level Reactivity in Distributed Reactive Programs: Reactive Acquaintance Management using Flocks'. In: *The Art, Science, and Engineering of Programming* 6.3 (2022), 14:1–14:37. ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2022/6/14.

[142] Mandana Vaziri, Olivier Tardieu, Rodric Rabbah, Philippe Suter and Martin Hirzel. 'Stream Processing with a Spreadsheet'. In: *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*. Ed. by Richard E. Jones. Vol. 8586. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer-Verlag, 2014, pp. 360–384. DOI: 10.1007/978-3-662-44202-9_15.

[143] Andreas Voellmy and Paul Hudak. 'Nettle: Taking the Sting Out of Programming Network Routers'. In: *Practical Aspects of Declarative Languages - 13th International Symposium, PADL 2011, Austin, TX, USA, January 24-25, 2011. Proceedings*. Ed. by Ricardo Rocha and John Launchbury. Vol. 6539. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 235–249. DOI: 10.1007/978-3-642-18378-2_19.

[144] Sam Van den Vonder, Joeri De Koster and Wolfgang De Meuter. 'Composable Actor Behaviour'. In: *Distributed Applications and Interoperable Systems - 19th IFIP WG 6.1 International Conference, DAIS 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17-21, 2019, Proceedings*. Ed. by José Pereira and Laura Ricci. Vol. 11534. Lecture Notes in Computer Science. Cham, Switzerland: Springer Nature Switzerland AG, 2019, pp. 57–73. DOI: 10.1007/978-3-030-22496-7_4.

[145] Sam Van den Vonder, Joeri De Koster, Florian Myter and Wolfgang De Meuter. 'Tackling the awkward squad for reactive programming: the actor-reactor model'. In: *Proceedings of the 4th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems, Vancouver, BC, Canada, October 23, 2017*. Ed. by Guido Salvaneschi, Wolfgang De Meuter, Patrick Eugster and Lukasz Ziarek. New York, NY, USA: Association for Computing Machinery, 2017, pp. 27–33. DOI: 10.1145/3141858.3141863.

[146] Sam Van den Vonder, Florian Myter, Joeri De Koster and Wolfgang De Meuter. 'Enriching the Internet By Acting and Reacting'. In: *Companion to the first International Conference on the Art, Science and Engineering of Programming, Programming 2017, Brussels, Belgium, April 3-6, 2017*. Ed. by

Jennifer B. Sartor, Theo D'Hondt and Wolfgang De Meuter. New York, NY, USA: Association for Computing Machinery, 2017, 24:1–24:6. DOI: 10.1145/3079368.3079407.

[147] Sam Van den Vonder, Thierry Renaux, Bjarno Oeyen, Joeri De Koster and Wolfgang De Meuter. 'Tackling the Awkward Squad for Reactive Programming: The Actor-Reactor Model'. In: *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference)*. Ed. by Robert Hirschfeld and Tobias Pape. Vol. 166. LIPIcs Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 19:1–19:29. DOI: 10.4230/LIPIcs.ECOOP.2020.19.

[148] Sam Van den Vonder, Thierry Renaux, Bjarno Oeyen, Joeri De Koster and Wolfgang De Meuter. 'Tackling the Awkward Squad for Reactive Programming: The Actor-Reactor Model (Artifact)'. In: *Dagstuhl Artifacts Ser.* 6.2 (2020), 07:1–07:4. DOI: 10.4230/DARTS.6.2.7.

[149] Zhanyong Wan, Walid Taha and Paul Hudak. 'Real-Time FRP'. In: *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001*. Ed. by Benjamin C. Pierce. New York, NY, USA: Association for Computing Machinery, 2001, pp. 146–156. DOI: 10.1145/507635.507654.

[150] Sheng Wang and Takuo Watanabe. 'Functional Reactive EDSL with Asynchronous Execution for Resource-Constrained Embedded Systems'. In: *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, SNPD 2019, Toyama, Japan, July 8-11, 2019*. Ed. by Roger Lee. Vol. 850. Cham, Switzerland: Springer Nature Switzerland AG, 2020, pp. 171–190. ISBN: 978-3-030-26428-4. DOI: 10.1007/978-3-030-26428-4_12.

[151] Takuo Watanabe and Kazuhiro Shibanai. 'Towards a Functional Reactive Programming Model for Developing WSANs'. In: *Asia Pacific Conference on Robot IoT System Development and Platform (APRIS 2020)* (Nov. 2020), pp. 1–5.

[152] Pascal Weisenburger, Mirko Köhler and Guido Salvaneschi. 'Distributed system development with ScalaLoci'. In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018), 129:1–129:30. DOI: 10.1145/3276499.

[153] Wikipedia. *List of bicycle-sharing systems - Wikipedia, The Free Encyclopedia*. http://web.archive.org/web/20210402144924/https://en.wikipedia.org/wiki/List_of_bicycle-sharing_systems. Accessed on 2021-04-02. 2021.

[154] Akinori Yonezawa, Jean-Pierre Briot and Etsuya Shibayama. 'Object-Oriented Concurrent Programming in ABCL/1'. In: *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86), Portland, Oregon, USA, Proceedings*. Ed. by Norman K. Meyrowitz. New York, NY, USA: Association for Computing Machinery, 1986, pp. 258–268. DOI: 10.1145/28697.28722.