RTBScript: A High-Level Language for Hybrid Domain Modeling using Actors

Filipe Miguel Duarte Cristóvão

Dissertação para obtenção do Grau de Mestre em Engenharia Informática e de Computadores

Júri

Presidente: Prof.ª Ana Maria Severino de Almeida e Paiva
Orientador: Prof. António Paulo Teles de Menezes Correia Leitão
Vogal: Prof. João Manuel Pinheiro Cachopo

Outubro 2009
Acknowledgements

I would like to express my sincere thanks to my thesis adviser, Prof. António Leitão. His trust, support, helpful discussions and infinite patience helped me to accomplish this work successfully. Again, my profound thanks.

I would also like to thank the VILT team, namely Eng. Tiago Gavinhos and Eng. Christophe Conceição for their feedback and support in the development of this project.

A very special thanks to my friends: to Wilson, Jorge and Letícia who always supported and believed in me, even when distance kept us apart; to Filipe and Élton, who had (and still have) to put up with me during many stressful moments and, even so, helped me so much in so many occasions, including in the revision of this work; to Carlos, Emanuel, André and Pedro, thanks for all the long and interesting chats during lunch breaks and other occasions. I hold you all much dear to me!

I cannot forget to mention Joel since, without him, I would have never started this project, and Filipe C., whose comments influenced this work.

Last but not least, I thank my parents and brothers for all the support and for the right words in the right times. Thanks for the “hoe”: I’ll try to use it wisely. I treasure you all deeply!
Abstract and Keywords

In several domains the evolution of a system is entirely dependent on events, where an event is an occurrence at a discrete point in time. However, discrete models are not always sufficient because many systems also evolve continuously. Presently, it is considered that these kinds of hybrid domains are appropriately modelled by Functional Reactive Programming (FRP).

The key concepts of FRP are behaviors and events: behaviors are time-varying, reactive values, whereas events are time-ordered sequences of discrete-time occurrences. FRP has been successfully applied in programming reactive animations, vision tracking, and robotics.

Recently, a version of FRP based on the Arrow combinators has been developed: the Arrowized FRP (AFRP). In AFRP, the key concepts are signals and the operators that transform them, the signal transformers.

This work presents RTBScript, an high-level language for hybrid domain modeling inspired in the AFRP concepts. RTBScript uses the RealTimeBiz (RTB) Platform capabilities of real-time event gathering of different systems.

Since AFRP has been mainly developed in functional languages, its application to mainstream languages, such as Java, has been quite crude. RTBScript brings the AFRP concepts closer to mainstream languages.

This work also discusses the AFRP implementation using a novel approach: the Actors model.

Keywords: high-level language, hybrid domain modeling, events, continuity, functional reactive programming, actors.
Resumo e Palavras-chave

Em vários domínios a evolução de um sistema é inteiramente dependente de eventos, onde um evento é uma ocorrência num ponto discreto no tempo. Porém, modelos discretos nem sempre são suficientes visto que muitos sistemas também evoluem de forma contínua. Actualmente, considera-se que este tipo de domínios híbridos são apropriadamente modelados por Programação Funcional Reactiva (em inglês, FRP).

Os conceitos-chave da FRP são behaviors e eventos: behaviors são valores reactivos, variáveis no tempo, enquanto que os eventos correspondem a sequências de ocorrências discretas ordenadas no tempo. A FRP tem sido aplicada com sucesso em programação de animações reactivas, processamento de visão e robótica.

Recentemente, uma versão da FRP baseada em Arrow combinators foi desenvolvida: a Arrowized FRP (AFRP). Na AFRP, os conceitos-chave são os sinais e os operadores que os transformam, os transformadores de sinais.

Este trabalho apresenta o RTBScript, uma linguagem de alto nível para modelação de domínios híbridos inspirada nos conceitos da AFRP. O RTBScript usa as capacidades de recolha em tempo real de eventos de diferentes sistemas disponibilizadas pela Plataforma RealTimeBiz (RTB).

Visto que a AFRP tem sido principalmente desenvolvida em linguagens funcionais, a sua aplicação em linguagens mais largamente utilizadas, como Java, tem sido muito rudimentar. O RTBScript traz os conceitos da AFRP para mais próximo das linguagens mais utilizadas.

Neste trabalho também se discute a implementação da AFRP através de uma nova abordagem: a utilização do modelo de Actores.

**Palavras-chave:** linguagem de alto nível, modelação de domínios híbridos, eventos, continuidade, programação funcional reactiva, actores.
## Contents

Acknowledgements .......................................................... iii  
List Of Figures ..................................................................... xi  
List Of Tables ....................................................................... xiii  
Acronyms ............................................................................. xv  

1 Introduction ........................................................................ 1  
  1.1 Prologue ......................................................................... 1  
  1.2 Context ........................................................................... 1  
    1.2.1 Discrete Event System ............................................. 2  
    1.2.2 The RealTimeBiz Platform ..................................... 3  
    1.2.3 The Scala Language .............................................. 5  

2 Solution Requirements ....................................................... 7  
  2.1 Motivation ........................................................................ 7  
  2.2 Goals .............................................................................. 8  
    2.2.1 Main Goal ............................................................. 8  
    2.2.2 Design Goals ....................................................... 8  

3 Related Work ...................................................................... 11  
  3.1 Functional Reactive Programming .................................. 11  
    3.1.1 Fran: Functional Reactive Animation .................... 12  
    3.1.2 FrTime: Embedding Dynamic Dataflow in a Call-by-Value Language .................. 13  
    3.1.3 Frappé: Functional Reactive Programming in Java .......... 14  
    3.1.4 FlapJax: Functional Reactive Web Programming .......... 15  
    3.1.5 Functional Reactive Programming, Continued - Arrowized Functional Reactive Programming .................................................. 16  
    3.1.6 Other Related Work and Summary ......................... 17  
  3.2 Actors ............................................................................. 20  
    3.2.1 Scala Actors ......................................................... 20  

4 Development Roadmap ..................................................... 23  
  4.1 The idea of RTBScript .................................................. 23  
  4.2 Integration in the RTB Platform: the host language problem ........................................... 23  
  4.3 Localised computing and performance .......................... 24  
  4.4 External data sources .................................................... 24
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>RealTimeBiz Platform Technical Architecture</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>RealTimeBiz Management Console report examples</td>
<td>4</td>
</tr>
<tr>
<td>3.1</td>
<td>Arrow Combinators</td>
<td>17</td>
</tr>
<tr>
<td>3.2</td>
<td>Throughput (number of token passes per second) for a fixed number of 10 tokens</td>
<td>21</td>
</tr>
<tr>
<td>5.1</td>
<td>RTBScript in the RealTimeBiz (RTB) Platform Technical Architecture</td>
<td>26</td>
</tr>
<tr>
<td>5.2</td>
<td>RTBScript detailed conceptual architecture</td>
<td>27</td>
</tr>
<tr>
<td>5.3</td>
<td>RTBScript Engines architecture</td>
<td>28</td>
</tr>
<tr>
<td>5.4</td>
<td>Groups of Events in the RealTimeBiz Management Console</td>
<td>29</td>
</tr>
<tr>
<td>5.5</td>
<td>UML representation of the classes involved in the translation process</td>
<td>30</td>
</tr>
<tr>
<td>5.6</td>
<td>RTBScript combinators</td>
<td>35</td>
</tr>
<tr>
<td>5.7</td>
<td>Visual representation of program 5.7</td>
<td>37</td>
</tr>
<tr>
<td>5.8</td>
<td>Visual representation of program 5.8</td>
<td>38</td>
</tr>
<tr>
<td>5.9</td>
<td>Real implementation of program 5.8</td>
<td>39</td>
</tr>
<tr>
<td>5.10</td>
<td>Real implementation of combinator <code>first</code></td>
<td>40</td>
</tr>
<tr>
<td>5.11</td>
<td>Separate channel chains for common definitions of signal functions</td>
<td>41</td>
</tr>
<tr>
<td>5.12</td>
<td>The overall architecture of RTBScript regarding actors.</td>
<td>44</td>
</tr>
<tr>
<td>5.13</td>
<td>Visual representation for the <code>hold</code> and <code>changes</code> functions</td>
<td>46</td>
</tr>
<tr>
<td>7.1</td>
<td>RealTimeBiz Platform Distributed Architecture example</td>
<td>62</td>
</tr>
<tr>
<td>7.2</td>
<td>Signal flow diagrams for <code>tenPercent</code></td>
<td>65</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Summary of different advantages and disadvantages introduced by each implementation regarding FRP ................................................. 19
## Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADBS</td>
<td>Active Database System</td>
</tr>
<tr>
<td>AFRP</td>
<td>Arrowized Functional Reactive Programming</td>
</tr>
<tr>
<td>CPO</td>
<td>Complete Partial Order</td>
</tr>
<tr>
<td>DBMS</td>
<td>Database Management System</td>
</tr>
<tr>
<td>DES</td>
<td>Discrete Event System</td>
</tr>
<tr>
<td>DSEL</td>
<td>Domain Specific Embedded Language</td>
</tr>
<tr>
<td>ECA</td>
<td>Event-Condition-Action</td>
</tr>
<tr>
<td>EPFL</td>
<td>École Polytechnique Fédérale de Lausanne</td>
</tr>
<tr>
<td>ESL</td>
<td>Event Subscription Language</td>
</tr>
<tr>
<td>FRP</td>
<td>Functional Reactive Programming</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>I/O</td>
<td>Input/Output</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>REPL</td>
<td>Read-Eval-Print loop</td>
</tr>
<tr>
<td>RMI</td>
<td>Remote Method Invocation</td>
</tr>
<tr>
<td>RTB</td>
<td>RealTimeBiz</td>
</tr>
<tr>
<td>RTBMC</td>
<td>RealTimeBiz Management Console</td>
</tr>
<tr>
<td>SPS</td>
<td>Stream Processing System</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Prologue

For some time now, different companies have been using different kinds of computer systems and applications to aid their types of businesses. Database servers, network systems, firewalls, content management systems, web application servers, billing and orders processing systems, just to name a very few, are key systems to keep each business up and running.

In the past, many of these systems were isolated from each other, each one doing what it was programmed for.

In one side, as businesses relied more and more in these systems, unscheduled downtimes meant losses. With the need for fast detection and repair of the failed systems came the development of monitoring and notification systems. In this way, systems were watching for other systems.

On the other side, as the Internet developed, several companies have brought their businesses to the global web. Also, as these online businesses grew increasingly connected, so did other sources of data, rendering them increasingly easy to access.

So, nowadays, it is important to keep track of several distributed sources of data and information, as they grow in number and in importance to each business. Also, more and more, corporations need real-time information to increase their overall efficiency.

In this work, the main goal will be the design of a new high-level scripting language aiming to easily extract, analyse and present data from different, distributed sources of information. Abstracting these sources into both discrete and continuous streams of data allows their processing in a high-level way, with real-time analysis. As such, this language should allow the creation and management of actions to be executed upon some kind of conditions being met.

This thesis will show that it is possible to model continuous and discrete domains using Functional Reactive Programming (FRP), bringing it closer to mainstream languages. Since FRP has been mainly developed in functional languages, its application to mainstream languages, such as Java, have been quite crude (with the example of Frappé [Cou01], excessively verbose and with no type safety). Due to performance and concurrent execution considerations, we will also investigate its implementation in a novel way, using Actors.

1.2 Context

In this section we present some important concepts introducing this work and its motivation. We introduce the Discrete Event Systems and the platform where RTBScript will be integrated into: the
RTB Platform. Due to the interaction with the RTB Platform, written in Java, we present Scala, a Java-compatible language, in which RTBScript is developed.

1.2.1 Discrete Event System

Several types of systems can be modelled by certain discrete occurrences and their evolution through time. These discrete occurrences, usually called events, can represent different types of data, such as the stock price of a company reaching some value, the arrival of a customer to a queue or the transmission of a packet in a network.

For several decades now, different research has been conducted about the systems whose state changes are entirely dependent of the occurrence of discrete events that can happen at possibly unknown irregular intervals [Cas05, CL06, RW89]. These systems were called Discrete Event Systems (DESs) and they were used in different domains, such as communication networks, automated manufacturing systems, air traffic control systems, distributed software systems, among others [CL06].

One kind of systems that greatly benefited from this research were the Database Management Systems (DBMSs), because, previously, they were unable to automatically monitor and trigger notifications about some kind of domain (e.g., in inventory control, detecting when a product’s quantity gets below some threshold) [CA08]. To suppress these inadequacies, the Active Database Systems (ADBSs) were developed. The separation of an event (an occurrence of interest) from condition (corresponding to queries) and action (corresponding to transactions) in ADBSs led to the Event-Condition-Action (ECA) model [CA08]. In this model, the user defines a set of ECA rules, each one separating the event (the occurrence we are interested in), from the conditions (corresponding to a query to the database) and the actions (a transaction to the database): when an event occurs, the condition is evaluated and, if it holds true, the action is executed [Mar06]. Several projects relating to ADBS appeared, and several Event Subscription Languages (ESLs) emerged, with expressiveness as the primary goal in event detection: different languages came with different event operators (e.g., Snoop [CM94] had the ANY operator: \( \text{ANY}(m, E_1, E_2, ..., E_n) \) with \( m < n \), which generated an event whenever \( m \) events out of \( n \) distinct events have been triggered [Mar06]).

With the development of the DESs, it became evident that different entities have different needs of information: the meteorologist is not interested in the latest development in combustion engines, and the computer science engineer is not (usually) interested in the weather in the south pacific. The need of an unique system that could deal with different interests led to a new paradigm: the Publish/Subscribe paradigm. In it, there are different publishers generating events (some kind of information that may interest others), and different subscribers, which show interest in some kind of events and are notified when such events occur. In this paradigm, there is usually some kind of broker, the event service, responsible for managing the subscriptions, the storage and the delivery of events. Subscribers register their interests to this broker and the publishers notify the broker of some event, which is then delivered to the subscribers by the latter, according to their subscription [Mar06]. As a result, this paradigm provides decoupling both in time (as participants may not be active at the same time) and in space (as participants don’t need to know each other) for publishers and subscribers [Mar06]. Additionally, different ways of specifying events of interest led to the development several subscription schemes, as Channel-, Topic-, Content- or Type-based subscriptions. A description of these subscription schemes can be found in [Mar06].

Nevertheless, although the DES concept suffices to model many domains, it misses an important part of several other: the continuous ones! Time itself is continuous and so are many other physical domains (positions, velocities, and so on), although discrete events are still important in such domains. The need for modeling these kind of hybrid systems, that combine discrete and continuous domains, led to the
development to a new high-level programming paradigm: the FRP. In this paradigm, in addition to the concept of event (as in DES), it is introduced the concept of behavior, representing a value that varies over continuous time [EH97, WH00]. In section 3 we detail this paradigm and present some related work.

1.2.2 The RealTimeBiz Platform

This work, due to its corporate environment, will have at its aim a contribution to an already existing platform of the VILT company: the RTB Platform. This platform is a VILT product that allows to collect several events from different systems, although the platform is being mainly used with web applications. Through the use of different modules, it is capable of other functions, for example, datamining of the recorded data.

With the contributions of Kiron Middleware [Con06], RTB is capable of seamlessly integrate different technologies. For web applications, due to Kiron, RTB is capable of client- (e.g., with HTML, Javascript or Flash) and server- (e.g., with Java or C#) side monitoring.

In figure 1.1 we present the technical architecture of the RTB Platform, with its different components. Using Kiron, it is possible for different systems built with different technologies (e.g., Java or .Net) seamlessly interact with the RTB Data Collector, acting as collector of all the events generated by the systems, and registering them in some DBMS (as Oracle or MySQL). As a result, all the communication between the different systems and the RTB Data Collector is done through the Kiron Middleware.

![Figure 1.1: RealTimeBiz Platform Technical Architecture](image)

One of the fundamental components of RTB, also presented in figure 1.1, is the RealTimeBiz Management Console (RTBMC), responsible for the creation of different graphical reports according to the events collected. Examples of reports generated by the RTBMC are presented in figure 1.2.

Although RTBMC already provides different types of reports, and with different presentations (e.g., with charts or with tables), it could use some other means to create different reports, using event collection capabilities of the platform. This work will focus in creating a scripting language, detailed in section 2, that will also allow the generation of different reports.
Figure 1.2: RealTimeBiz Management Console report examples

(a) Errors in a server

(b) Visits in a web site
1.2.3 The Scala Language

The Scala\textsuperscript{1} language [Ode09] was designed by Martin Odersky, a professor at École Polytechnique Fédérale de Lausanne (EPFL), whose research interests focus on programming languages, more specifically languages for object-oriented and functional programming. Martin Odersky also influenced the development of Java as the original author of the current \texttt{javac} reference compiler and as a co-designer of Java generics.

Scala is “a multi-paradigm programming language designed to express common programming patterns in a concise, elegant, and type-safe way. It smoothly integrates features of object-oriented and functional languages” [Mic03, OSV08]

This language has several key-features to this work:

- Scala programs run on the Java Virtual Machine (JVM) [LY99] and are bytecode compatible with Java. So, it is possible to make full use of existing Java libraries or existing application code. Also, it is possible to call Scala from Java and you can call Java from Scala;

- Scala is a pure object-oriented language in the sense that every value is an object. Types and behavior of objects are described by classes and traits;

- Scala is also a functional language in the sense that every function is a value. Scala allows for the definition of anonymous functions, supports higher-order functions, allows functions to be nested, and supports currying;

- Scala is “scalable” in the sense that it allows to add new language constructs in the form of libraries using different mechanisms, including that any method may be used as an infix or postfix operator.

In sections 4 and 5, each of these features will be re-addressed to explain its importance to this work.

\textsuperscript{1}http://www.scala-lang.org/
Chapter 2

Solution Requirements

In the corporate environment, many businesses are being brought to the Internet as different technologies were developed to aid in this task.

On one hand, as web applications get more complex and personalised to each customer (or group of them), business owners want to know how each customer acts towards the online business. This requirement led to real-time monitoring of clients’ actions and notification of business owners if some kind of actions of interest is performed by clients.

On the other hand, as more and more operations are available to the user, the infrastructure also has to grow to support the many applications. With personalised services gaining greater importance, it is now important to keep track of several distributed sources of events. Not only there is the need to monitor client actions, but there is also the need to monitor the systems in which applications rely and notify the respective maintainers.

In this project we intend that some party is able to express its interest in the evolution of distributed sources of data and, through the definition of small scripts, is able to define actions when certain conditions are met. The next sections present this idea in more detail.

2.1 Motivation

Nowadays, several systems exist in a myriad of business contexts, working with different technologies. As these systems became increasingly connected between themselves, so did the importance of the information they provided to each business owner, creating the need for the effective processing of the data that is being generated by the different and distributed systems.

In a corporate environment, it isn’t difficult to imagine several examples where business owners might benefit from (near) real-time interaction with the different systems they already have running. Here we present just a few:

- In a sales company, “when the total volume of daily sales reach €10,000 and it is Christmas time, apply a discount to a subset of products”;

- On a web portal of some corporation, “when login errors exceed 5% of the total visitors, alert the developer team”;

- In a stock exchange corporation, “the first time that a share price of certain corporation is lower than 50, buy it”;

- In a systems monitoring context, “when the CPU usage of a server is greater than 90% for longer than 10 minutes, alert the system administrator and reboot the server”.
From the examples we can see that they cover a wide range of domains and include not only elements that are conceptually *discrete*, such as each sale in a company, but also elements that are conceptually *continuous*, such as the CPU usage of a server. It is also perceivable that these examples are *reactive*, in the sense that actions are performed *when* some conditions are met.

The motivation to this work is to find and expressive way of modeling this wide range of domains, considering both discrete and continuous elements. We will detail this motivation in the next sections.

2.2 Goals

In this section we present the main goals to achieve in this work, as well as the design goals to the final solution described later in this document.

2.2.1 Main Goal

The main goal for this work using the data collection capabilities of the RTB Platform.

The key features of the language are:

- event extraction from and insertion into the RTB Platform;
- creation and management of *rules*, allowing the expression of business rules such as “when the daily sales achieve €10,000, apply a 10% discount to all products”, or “when the number of sales in the last hour is greater than 10 send an SMS to the product manager”. Conceptually, these rules are composed from *premises* (or *conditions*), allowing to model when to perform some computation, and *actions*, procedures to be executed upon a condition;

With these capabilities, RTBScript should be able to interact with other systems, effectively allowing to react to real-time data, ranging from domains like monitoring and notification to business-related applications, e.g., sales management, inventory control or stock trading changes.

2.2.2 Design Goals

Due to the different requirements of this work, we now introduce the several design goals that influenced its development:

**Ease of use**

The main intent of this project is to be able to easily model different information based on a set of events. As such, one of the main goals is the ease of use. Also, the syntax of this language should be simple enough to not impair the development of scripts.

**Expressiveness**

Despite being easy to use, the language expressiveness is a key concept, as the language may be required to deal with different domains, from stock sales to systems monitoring. We consider embedding the declarative approach of the hybrid reactive component of the language in an imperative paradigm, as nowadays users are more familiarised with the latter.
Modularity

A key concept in application development is code modularity. Composition of modules will be a main goal as a form of empowering development through the reuse of already developed modules. Arrowized Functional Reactive Programming (AFRP) eases module composition, since allows for the composition of simpler functions into complex ones, which can again be composed seamlessly with others (in section 3.1.5 this aspect is looked into in more detail).

Continuity Modeling

Continuity modeling is specially useful in applications that interact with the physical world [NCP02]. Given that a computation can only simulate continuity, it is advantageous to abstract its discrete representation, hiding it from the user.

Event Modeling

The RTB platform was developed having events as its focus. As such, this aspect of modeling should be optimized for, both in expressiveness (in the case of a choice between expressiveness with continuity or with events, the latter should be the chosen) and at the operational level.

Declarative approach

The reactive component of RTBScript will try to follow a declarative modeling, instead of using an imperative approach, where event modeling is usually done in terms of state changes [EH97].

Causality

In [NCP02] it is emphasised that a Signal Transformer’s result at instant $t$ should be “uniquely determined by the input signal on the interval $[0, t]$”. This is known as causality, considered to be fundamental in previous work, such as LUSTRE [CPHP87], and, as stated in [Ste95], distinguishing it from its predecessor, LUCID [WA85]. This means that no function can depend on “future” values of others, that is, the result for an input signal at instant $t$ cannot depend on values of instants greater than $t$ (LUCID had an operator, next, that would allow this, where the sample at $t$ could depend of the sample $t + 1$ of other stream). This aspect is mandatory “to ensure that signal functions are properly executable” [NCP02].

Incremental Program Development

One of RTBScript’s goals is the incremental construction of programs, that is, the support of an interactive console (as in a Read-Eval-Print loop [CK06]) while the reactive component continues to operate. Hopefully, this will allow a much more smooth interaction, as users see immediately what they are programming.
Chapter 3

Related Work

As we introduced in chapter 1, FRP is a paradigm suited for hybrid domains modeling, considering both discrete and continuous elements. In the next sections we present different works and contributions already done regarding FRP. Later, we present some related work regarding Actors as well, an important part of this work.

3.1 Functional Reactive Programming

In this section we present the different works and contributions relating to Functional Reactive Programming. As it was already brought out in chapter 1, FRP introduces the concept of behavior, representing values that vary over continuous time. As such, a behavior can be thought of as an infinite sequence of mappings from time to data. These infinite sequences of data are usually referred to as streams and different systems deal with these.

A Stream Processing System (SPS) is a model for computation comprised of a collection of modules that communicate through a series of channels where streams are passed. The modules are usually divided into 1) filters (or agents), where some kind of computation is done; and 2) sources and sinks, where data is brought into and out of the system, respectively [Ste95]. The research of SPS includes, among others, dataflow systems, reactive systems, and signal processing systems, being used in computational models for, e.g., artificial neural networks, operating systems or safety critical systems [Ste95].

In [Ste95, section 3], the author suggests a classification for different SPSs according to three main characteristics:

- Either synchronous or asynchronous filters: if filters compute synchronously towards other filters, or with no synchronization with respect to the others;
- Either deterministic or non-deterministic filters: if filters do or do not compute a function;
- Either uni-directional or bi-directional channels.

In the interest of contextualization, FRP is, according to this author’s classification, a synchronous, uni-directional, and deterministic dataflow language [NCP02]. The more interested reader can refer to [RS09] as a bi-directional dataflow language example.

Instead of being implemented from scratch, with all the complexity that it brings, FRP has been using the infrastructure of other general purpose languages (influenced by all their design decisions) and being able to use the tools provided to each. As such, FRP is considered a Domain Specific Embedded Language (DSEL) [Hud98], that is, a programming language that is implemented using the terms of a host language. Usually, FRP has been embedded on Haskell [JHA+], a polymorphic, typified, purely

11
functional, and lazy-evaluated language. In the next sections, we’ll see works that used other host languages as well.

Now that we introduced the FRP, and its classification in the SPSs, we introduce in the next sections several contributions to FRP and, more recently, to AFRP.

3.1.1 Fran: Functional Reactive Animation

The key concepts behind FRP were introduced in Fran [EH97], using behaviors and events associated to the creation of reactive animations. The authors present the lack of sufficiently high-level abstractions for animation as motivation for their work, suggesting for the clear distinction between modeling and presentation of animation, or, as they say it, “between what an animation is and how it should be presented”. By focusing on the modelling, the developer is free from the low-level display elements, and it also allows for presentation optimization, level-of-detail regulation, and portability. The authors suggest several data types and a rich set of operators, but they also admit their suggestions do not form a programming language per se, although providing a formal denotational semantics.

As the time domain is fundamental to the authors work, they provide semantics to allow precise reasoning about it. They also provide a Complete Partial Order (CPO) of real time, allowing them to reason about events before they occur, as needed for predicate event detection, described ahead. Basically, this CPO introduces the possibility of temporal instants of, for example, \( \geq 42 \) (meaning “at least 42”), without really knowing the instants’ final value.

In this work, the authors define behaviors as representing a value that (possibly) varies over continuous time, while events represent a time-ordered sequence of event occurrences, each one carrying a value [WH00].

Regarding behaviors, we can consider the type \( \text{Behavior } \alpha \) as representing the values of type \( \alpha \) varying in the continuous time. The simplest time-varying value is time itself: we could represent time as a value of type \( \text{Behavior Real} \).\(^1\) As an example, consider the (always available) mouse’s current position \((x, y)\) in the screen: if \( \text{Point2D} \) represents the type for two-dimensional points, then we could represent the time-varying mouse position as a \( \text{Behavior Point2D} \). Another behavior example is the type \( \text{Behavior Picture} \), where animations are considered as time-varying pictures. However, although behaviors represent values that can vary over time, that doesn’t mean that they have to vary: we can easily picture a constant \( \text{Behavior Int} \) that could always yield the value 1 all the time.

Regarding events, the type \( \text{Event } \alpha \) represents a time-ordered sequence of values of type \( \alpha \). Considering the type \( \text{Button} \) as representing each mouse button (e.g., left, right or middle), a common type of event representing the mouse button presses would be \( \text{Event Button} \). Similarly, \( \text{Event Char} \) could represent the keyboard presses. An important element to notice is that these event types represent a sequence of all the occurrences, not just one. As already noted in section 3.1, these types can be seen as mappings from time to some value: in the case of \( \text{Event } \alpha \), only the instants that would map to an occurrence would have a value of type \( \alpha \).

Finally, additionally to the behaviors and events, the authors also define several operators used to compose behaviors and events from existing ones.

Regarding to behaviors semantics, in this work the authors define:

- \( \text{time} \) itself, as the simplest behavior, yielding the time value at each instant;
- the lifting operator, allowing static values or non-FRP functions to be “lifted” to the corresponding FRP elements. This is an important operator, as it allows to bring continuity to other types. As

\(^1\)With the peculiarity of the finite representation of \( \text{Real} \) forbiding a true representation of continuity, and that time is represented only by the non-negative real values.
an example, we can consider the lifted version of the Int value $3$ (resulting in a constant behavior) and the lifted version of the common subtraction operation, allowing us to express ‘\texttt{time - 3}’ as representing a delay in time of 3 seconds;

- the \textit{time transformation} operator, allowing to express, e.g., slower behaviors, such as \texttt{timeTransform b (time/2)}, or delayed ones, such as \texttt{timeTransform b (time - 2)};
- the \texttt{integral} operator, allowing for the \textit{integration} of behaviors, e.g., \texttt{integral 1} would yield the value of time itself;
- the \texttt{untilB} operator, expressing the \textit{reactivity} between behaviors and events, as \texttt{b untilB e} yields behavior \texttt{b} until \texttt{e} occurs, switching to the behavior associated to \texttt{e} afterwards.

Concerning \textit{events}, the authors define another set of operators:

- \textit{event handling}, allowing for the transformation of events of one type to another;
- \textit{constant events} creation, as \texttt{constEv 10 x} would create an event at time 10 with value \texttt{x};
- only one kind of \textit{external events}, considering only the mouse button presses;
- \textit{predicates}, as a way of creating events such as the first time that a boolean behavior becomes true, after a given time. For example, \texttt{predicate (sin time = 0.5) t0} generates an event the first time that \texttt{sin time} is 0.5 after the instant \texttt{t0}. The detection of these predicates is not a trivial task (thence the formal definition of time already introduced) and the authors needed to resort to \textit{interval analysis} presented elsewhere, suggesting an algorithm’s implementation in Haskell;
- the \textit{choice} between events, allowing to choose the earlier of two events with the \texttt{. | .} operator;
- \textit{snapshooting} (saving) a behavior’s value when an event occurs;
- \textit{event sequencing}, allowing to use one event occurrence to generate another.

In one last note, in this work behaviors and events are first-class values (i.e., they can be stored in variables, passed as arguments to functions or returned as results from functions), and a FRP \textit{program} is just a set of recursive behaviors and events, built upon non-time-varying values (using the lift operation) and/or other behaviors and events [WH00].

3.1.2 FrTime: Embedding Dynamic Dataflow in a Call-by-Value Language

FrTime [CK06] is an extension of the Scheme [KCR98] language designed for writing interactive applications. The main goals to this work were 1) the possibility to process and respond to events from external sources, 2) the \textit{incremental} program development (where programmers should be able to write expressions in a Read-Eval-Print loop (REPL), observe and name their values, use them to build larger expressions, and so on), and 3) to reuse as much of an existing evaluator as possible.

In this work, \textit{signals} (the time-varying values) are the fundamental unit, with behaviors being represented as signals defined at every point in time, while events are signals carrying sequences of discrete values. Additionally, some primitives are introduced, allowing to convert behaviors into events and vice-versa. An important detail in this language, considering the previous work, is that application of functions to behaviors only causes the application of the function to the initial value, and each time that value \textit{changes}. As such, it is no longer possible to express \texttt{integral 1} (since 1 is a \textit{constant} behavior), as it was in Fran [EH97].
In FrTime, the computation is triggered by arrival of events, and changes cause dependent parts to update. This is known as the push-driven\textsuperscript{2} update mechanism. Running a FrTime program creates a graph of its dataflow dependencies where nodes represent expressions and arcs indicate the flow of values between expressions. Since evaluation is push-driven, reactivity starts in external sources of events such as timers, mouses, and keyboards.

Since evaluation uses graphs, FrTime’s primitives have to maintain the graph and some awkwardness may arise: considering \texttt{seconds} as the behaviour displaying the seconds since some past occurrence, the expression \((\lt \texttt{seconds} (+1 \texttt{seconds}))\) should always evaluate to true. However it could evaluate to false if the update of \texttt{seconds} causes the evaluation of \((\lt \texttt{seconds} x)\), where \(x\) is the not yet updated value of \((+1 \texttt{seconds})\). This situation, where a signal is evaluated before all of its subordinate signals are up-to-date, is called a \textit{glitch}. Graph usage raises some other issues concerning the update of the graph during evaluation or the creation of cycles, albeit they are addressed in the work.

In this work, the authors also define a set of operators supported. However, these operators are mostly based in lifting the already existent operators of the host language. Due to its novel approach using graphs, it also defines a set of rules for graph construction and maintenance according to the referred operators.

### 3.1.3 Frappé: Functional Reactive Programming in Java

Frappé [Cou01] is an FRP implementation for Java using the Java Beans\textsuperscript{3} model. In the Java Beans model, a Java class specifies a set of \texttt{events} (occurrences causing notification of other objects upon certain conditions) and \texttt{properties} (named attributes mutable through its accessors - getters and setters). According to the authors, the main motivation for the work was to explore the similarities between the two models, as both have a notion of events and of time-changing values: behaviors in FRP and properties in Beans.

Again, this work uses graphs in its implementation, with each FRP behavior combinator being a Java class, whose instances constitute the nodes of the graph and fields represent the edges. As so, each operator becomes a mere \texttt{listener} of the objects that represent the behaviors. As of events, these are mapped directly to the Bean events.

Frappé’s limitations include no static type-safety and assume event processing as single-threaded and synchronous, though this has been a common assumption (yet, FrTime [CK06] addressed this issue with graph reconfiguration during updates). Also, it is unable to detect instantaneous predicate events (e.g., \texttt{predicate (time == 1)}, which is only true at instant time=1), despite Fran’s [EH97] use of interval analysis to solve such problem. At last, Frappé does not support time transformations, as described in section 3.1.1.

Finally, it is worth noticing that Frappé is still excessively verbose, comparing to its counterparts in Haskell. Since Frappé was designed as a library for Java, it suffers from Java’s explicit type declarations and verbose method and class names. In listing 3.1 we can see a Fran [EH97] example of a red ball following the mouse position with its counterpart in Frappé in program 3.2.

In the Fran example, there is a \texttt{ball} as a static red circle (scaled by a 0.3 factor). In the second line we can see the lifting operator (described in 3.1.1) to the function move. As such, \texttt{anim} will be a behavior, which is then passed to \texttt{animate} to produce the animation.

As for the Frappé example, we can see that the creation of the behavior \texttt{anim} is somewhat verbose (the references to \texttt{Drawable} elements refer to Java2D library).

\textsuperscript{2}In [Ste95, section 3.3] it is stated that, in an operational perspective, computation might be divided into \textit{data driven} (eager evaluation), when computation is performed as soon as data is available, or \textit{demand driven} (lazy evaluation), when data is requested from its sources only upon computation, as evaluation takes place. In FrTime [CK06], this is known as \textit{push-driven} and \textit{pull-driven} implementations, respectively.

### 3.1.4 FlapJax: Functional Reactive Web Programming

FlapJax presents itself as a contribution introducing a “JavaScript library and language extension that supports more declarative descriptions of rich web applications through the use of functional reactive constructs”. As more web applications are replacing desktop applications, the authors provide a core library based in FRP, having the web programming as domain of interest.

FlapJax has an event oriented design, considering each behavior as an initial value followed by the value at each sample in time. However, it considers each behavior as a stream of samples, as opposed to a stream of changes (as in FrTime [CK06]). This is an important difference, as different results may occur. For example, if we consider a stream of ones (the constant behaviour 1), and an accumulator, considering only the changes, we would never increase the counter, but considering the samples, the counter would steadily increase. Nevertheless, the authors provide operators to swap between stream representations: hold, from changes to samples, and changes, from samples to changes.

Just like FrTime [CK06], FlapJax uses graphs for the evaluation of FRP, using topological sorting to prevent glitches (presented in section 3.1.2). It also uses a compiler for transparent lifting of the already existing functions of JavaScript. Along with the implementation of this compiler, the authors also introduced some code optimization.

The authors give particular attention to the Document Object Model (DOM, a modifiable representation of a web page layout used by the browsers), creating a set of operators to integrate FRP with DOM modeling. Also, the authors abstract web services (what they call “web services transparency”) considering each web service as a stream function (with an input request stream and an output response stream) and using them in FRP combinators. Many other web related issues are discussed in the work, although not particularly relevant to the development of our work.

Another particular issue the authors mention is the propagation of changes in collections. For example, consider a generated array where each element is the successor of the corresponding element of an other array: if one entry in the dependent array changes, only one entry in the generated array needs to be changed. However, the authors do not implement any optimization for this case.
3.1.5 Functional Reactive Programming, Continued - Arrowized Functional Reactive Programming

As development progressed in FRP it was noted that, due to the first-class nature of behaviors and events in FRP, some problems relating to “time-space leaks” could arise [Ell98]. A “time-space leak” occurs when the implementation needs the complete time-history of a signal to compute one sample value, that is, to compute a value at instant \( t \), it is required the value at \( t - 1 \), which in turn requires the value at \( t - 2 \), and so on until \( t_0 \).

As a result, with the introduction of Arrows\(^4\) [Hug00], a new version of FRP was developed: the Arrowized FRP (AFRP) [CE01, NCP02]. There are two key concepts in AFRP: the Signals, able to represent both behaviors and events, and the Signal Transformers.

The type \( \text{Signal} \ \alpha \) can be seen as the representation of a function from \( \text{Time}^5 \) to a value of some type \( \alpha \):

\[
\text{Signal} \ \alpha = \text{Time} \rightarrow \alpha
\]

Consider again the example of section 3.1.1 with the mouse’s current position in the screen: if \( \text{Point2D} \) represents the type for the two-dimensional points, then we could represent the time-varying mouse position as a \( \text{Signal} \ \text{Point2D} \) (corresponding to \( \text{Behavior} \ \text{Point2D} \), as in Fran [EH97]).

But, since signals represent a function from time to some value, this raises the question of event representation: returning to the mouse button clicks example of section 3.1.1, we can consider that when there is a button click, there is also a \( \text{Button} \) value, and when there is no button click, then nothing happened. The Haskell language provides a type precisely for representing this situation: the type \( \text{Maybe}^6 \) \( \alpha \) represents values in the form \( \text{Just} \ x \), where \( x \) is a value of the type \( \alpha \), or in the form of \( \text{Nothing} \).

So, the signal representation of the mouse button clicks could be represented as \( \text{Signal} \ \text{Maybe} \ \text{Button} \), where only when there were button clicks would the value of this signal differ from \( \text{Nothing} \), yielding which button was clicked.

Regarding signal transformers, we can consider the \( \text{SignalTransformer} \ \alpha \ \beta \) type as representing a function from a value of type \( \text{Signal} \ \alpha \) to one of type \( \text{Signal} \ \beta \):

\[
\text{SignalTransformer} \ \alpha \ \beta = \text{Signal} \ \alpha \rightarrow \text{Signal} \ \beta
\]

These signal transformers can be seen as the different operators existing for FRP. As an example, we could consider a filter for events of type integer that discards those events whose value exceeds some predefined threshold. In this case, the type of this filter would be \( \text{SignalTransformer} \ \text{Maybe} \ \text{Integer} \) \( \text{Maybe} \ \text{Integer} \), so that the discarded events are replaced by the \( \text{Nothing} \) value.

In this work, the Signal Transformers (or signal functions) are first-class objects, although signals, to avoid time-space leaks, only exist indirectly (as non first-class entities), through the signal transformers. As a result, it is no longer possible to express collections of signals (as it was in Fran [EH97]).

The core primitives for composing signal functions are all the standard arrow combinators of [Hug00]. A graphical representation of these is in figure 3.1.

For example, the \( \text{arr} \) combinator (the corresponding to the \( \text{lift} \) operator in Fran [EH97]) allows for point-wise application of a function to a signal, while the \( \text{>>>} \) allows for function composition. Nevertheless, the functions based on the \( \text{arr} \) combinator allow only for stateless functions: the value of the output signal at instant \( t \) depends only on the input signal at instant \( t \). As this surely was insufficient, AFRP.

\(^4\)The author defines the type \( \text{Arrow} \ \alpha \ \beta \ \gamma \) “as representing a ‘computation [of an \( \alpha \) type] with input of type \( \beta \) delivering a \( \gamma \)’”[Hug00, page 11].

\(^5\)Again, represented as the non-negative real values.

\(^6\)The type \( \text{Maybe} \ \alpha \) is commonly used in Haskell for representing the possibility of failure: as of a function that usually would return \( \alpha \) values, now would return \( \text{Maybe} \ \alpha \), where \( \text{Nothing} \) represents the failure for computing an \( \alpha \) value [Hug00].
also allows for signal functions that are stateful, that is, they may depend not only on instant $t$, but on all instants in $[0, t]$. However, they provide no means of extending the current set of stateful functions, providing only for the stateful primitives of \texttt{integral} and \texttt{derivative}.

The authors also address an experimental feature of embedding subsystems, with separate control of time flow, as two different systems may have two logically distinct time frames.

Regarding the implementation of AFRP, the authors choose to adopt a \textit{continuation} based mechanism, instead of a stream based implementation. In this implementation, each signal transformer receives a time delta, indicating the time amount passed since the previous time step, and the current input signal, generating a \textit{continuation}, which will be applied to the next time step, and an output signal. The authors also present some optimizations for the continuation-based implementation.

### 3.1.6 Other Related Work and Summary

Several other developments relating FRP and AFRP were done, specially in the last decade. Essentially, FRP has evolved in two distinct directions.

One use of FRP is as a “glue language” for combining host language components in ways that have well defined resource usage: RT-FRP [WTH01] and E-FRP [WTH02] resulted from this research. In these cases, instead of being embedded languages, the code compiles directly to low level and function usage is somewhat restricted, allowing for the verification of operational resource bounds and termination.

The other use of FRP has expressiveness as dominating concern for different domains. This approach has been used to create DSELs embedded in different languages for a number of complex domains. Haskell already has different research embedded in, such as Frob [PHE99] or Yampa [HCNP02] (regarding robotics), FVision [PHRH01] (regarding visual tracking), and Fruit (an AFRP-based Graphical User Interface (GUI) toolkit, used in [NCP02] as the GUI for a proof-of-concept example of traffic surveillance by visual tracking). Scheme is also the host language for research regarding debugging: in [MCKR04] the authors present a language for scriptable debugging. In their work, they consider debugging “as a temporal activity with the running program generating a stream of events (entering and exiting methods, setting values, and so on)”.

As such, they build their debugging language atop FrTime [CK06], connecting it with the JVM for debugging. The authors also consider debugging as maintaining a set of invariants,
checked against the several events that the debugger generates.

Other contributions regarding FRP focused in the optimization, such as lowering [BCK07], a method for static optimization of graph-based implementations of FRP, as in FrTime and FlapJax. Basically, this method condenses the effects of several lifted functions in only one lifted function, reducing the amount of computation.

Now that we have seen different domain applications for the FRP, we summarise in table 3.1 the main differences between the different implementations already discussed earlier in this section.
<table>
<thead>
<tr>
<th>Feature</th>
<th>Fran</th>
<th>FrTime</th>
<th>Frappé</th>
<th>FlapJax</th>
<th>AFRP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation based on</td>
<td>Streams</td>
<td>Graphs</td>
<td>Graphs</td>
<td>Graphs</td>
<td>Continuations</td>
</tr>
<tr>
<td>Host Language</td>
<td>Haskell</td>
<td>Scheme</td>
<td>Java</td>
<td>JavaScript</td>
<td>Haskell</td>
</tr>
<tr>
<td>Evaluation model (section 3.1.2)</td>
<td>Push-driven</td>
<td>Push-driven</td>
<td>Push-driven</td>
<td>Push-driven</td>
<td>Push-driven</td>
</tr>
<tr>
<td>Evaluation through sampling or through changes</td>
<td>Sampling</td>
<td>Changes</td>
<td>Changes</td>
<td>Interchangeable</td>
<td>N.A.</td>
</tr>
<tr>
<td>Pros</td>
<td>Implicit treatment of time</td>
<td>Incremental program development</td>
<td>Interchangeable evaluation through sampling or changes</td>
<td>Avoids time-space leaks</td>
<td></td>
</tr>
<tr>
<td>Cons</td>
<td>At the implementation level, there is no way to identify signal functions that only react to changes in the input signal and doesn’t retain enough information for runtime optimization</td>
<td>Reaction only to changes takes some expressiveness.</td>
<td>No type-safety and excessive verbosity</td>
<td>Provide no means to extend the stateful signal functions and fewer expressiveness due to the inability to have collections of Signals</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: Summary of different advantages and disadvantages introduced by each implementation regarding FRP
3.2 Actors

The Actors model is a mathematical model for concurrent computation. As such, the actor model emphasises the communication occurring during computation [Cil81]. In the actor model, each communication is described as a message arriving at a computational agent called actor [Cil81]. Each arriving message to an actor is referred in the actor model as an event. As such, all events in the actor model are arrival events and there is no such thing as sending event [Cil81].

Formally, an actor is a pair consisting of a (unique) mail address and a current behavior [AH87]. Each mail address is associated with a mail queue for incoming messages [AH87]. The mail address allows actors to communicate by sending each other messages. The mail system provides asynchrony and buffering of communications between actors, although the order in which the communications are delivered is nondeterministic. In this system, the sender must specify a specific target to which the message is to be sent. The set of actors that each actor knows about is usually called the acquaintances of such actor. The current behavior indicates how the actor acts towards each message it receives. The actions it may perform are ([AH87, AH85]):

- **Send messages** to specific actors whose mail address it knows, in particular even to itself (although an actor does not necessarily need to know its own address).
- **Create new actors.** Initially, the mail address of the newly created actors may be known only to the creator, but may be later communicated to the other actors.
- **Specify the replacement behavior**, which will accept the next communication. The replacement behavior may process the next communication even as other actions occurring as a result of processing the previous communication are still being executed.

It is important to notice that, in this model, the only way to affect the behavior of an actor is to send it a communication and that all actors in a system carry out their actions concurrently [AH85].

In the rest of the section we present contributions regarding Actors in the Scala language.

3.2.1 Scala Actors

The work regarding Scala actors has been mainly developed by Philipp Haller and Martin Odersky. They describe their work in two papers: [HO08] and [HO06].

In [HO06] the authors state that “operating system threads and threads of virtual machines, such as the JVM [LY99], are usually too heavy-weight”. As such, the authors seek for lightweight Actor abstractions for the JVM. The focus in virtual machines is, according to the authors, due to its increasing importance for real-time control systems, and because the virtual machines are expected to become ubiquitous on mobile devices. Usually, these devices are exposed to severe resource constraints, where only a few hundred kilobytes of memory is available to a virtual machine and applications. Due to these restrictions programming abstractions used by applications have to be very lightweight to be useful, and thread-based concurrency abstractions are too heavyweight.

The authors describe that the common alternative to programming with threads is event-driven programming models, and most of these support event-driven programming only through inversion of control: “instead of calling blocking operations (e.g. for obtaining user input), a program merely registers its interest to be resumed on certain events (e.g. an event signaling a pressed button, or changed contents of a text field).” In the process, event handlers are installed in the execution environment, being called when certain events occur. When this happens, the execution environment dispatches the events to the installed handlers. As such, the program never calls these event handlers itself. Thus, “control over the execution of program logic is inverted.”
In [HO06] the authors make actors threadless, introducing event-based actors as an implementation technique to obtain very lightweight actor abstractions without inversion of control. As the authors put it, “the central idea is as follows: an actor that waits in a receive statement is not represented by a blocked thread but by a closure that captures the rest of the actor’s computation. The closure is executed once a message is sent to the actor that matches one of the message patterns specified in the receive [statement]”. The implementation details are beyond the scope of this work, but the interested reader may refer to [HO06] for more details.

Continuing the previous work, in [HO08] the same authors show how thread-based and event-based programming can be unified under a single actor abstraction. In their work, an actor can suspend with a full stack frame (receive) or it can suspend with just a continuation closure (react). The first form of suspension corresponds to thread-based, the second form to event-based programming. The new system combines the benefits of both models: on one hand, threads support blocking operations such as system Input/Output (I/O), and can be executed on multiple processor cores in parallel; on the other hand, event-based computation is more lightweight and scales to larger numbers of actors. Again, the implementation details are beyond the scope of this work, but its results are worthy of notice.

In [HO08], the authors compare the throughput of blocking operations in a queue-based application. The application is structured as a ring of $n$ producers/consumers (from now on called processes) with a shared queue between each of them. Initially, $k$ of these queues contain tokens and the others are empty. Each process loops removing an item from the queue on its right and placing it in the queue on its left. Their implementation is compared to pure Java threads, and SALSA (version 1.0.2), a Java-based actor language [VA01]. These results are shown in figure 3.2 (note that throughput is given on a logarithmic scale).

![Figure 3.2: Throughput (number of token passes per second) for a fixed number of 10 tokens](image)

From the figure 3.2, we can see that throughput of Scala actors remains basically constant (at about
30,000 tokens per second), regardless of the number of processes. In contrast, throughput of pure Java threads constantly decreases as the number of processes increases. The VM is unable to create a ring with 5500 threads as it runs out of heap memory. In contrast, using Scala actors the ring can be operated with as many as 600000 processes. Also, throughput of Scala actors is on average over 13 times higher than that of SALSA.

In their previous work, [HO06], the authors had already shown similar results, although event-based actors performed even better. All these results will influence the design of the final architecture of RTBScript, as will be shown in chapter 5.
Chapter 4

Development Roadmap

In the development of this work a wide range of topics were covered, including monitoring systems, programming languages design and concurrency models. In this section we present the ideas that led to the development of RTBScript.

4.1 The idea of RTBScript

As referred in chapter 1, as computer systems gained greater importance, so did the need to monitor them and to notify the interested parties. Many solutions already exist in this area although, sometimes, each solution focus in specific systems, using specific technologies.

As already presented in section 1.2.2, due to the RTB Platform capabilities to seamlessly integrate different technologies, it was discussed how and if a system could be developed allowing to present all the data in a central view of all the systems. It is also worth noticing that most of these systems work easily with discrete events, but such domains also contain elements that are conceptually continuous (e.g., CPU or memory usage are continuous values).

With these concepts in mind, monitoring and notification systems became only one more example of hybrid domains. Extrapolating from monitoring and notification systems, it was perceived that broader domains, including, for example, the sales of a company, the actions of a user through a website, or watching the values of companies’ stocks are also hybrid domains. However, the data referring to each domain is usually scattered in different systems. With all these concepts, the idea for RTBScript was born: designing a language able to deal with hybrid domains and also able to get its data from different sources.

To deal with the problem of modeling DES and hybrid domains, the research led to the FRP concept, already presented in section 3.1.

4.2 Integration in the RTB Platform: the host language problem

Due to the enterprise context of this project, the resulting work would have to integrate into the already existing platform. We have already seen in section 3.1 that most of the work relating FRP, is, as its name indicates, developed in functional languages, especially in Haskell. However, the RTB Platform is developed in Java, a language where the functional paradigm is mainly absent. Also, we have also seen an existing FRP implementation in Java, Frappé, that has some problems, specially the lack of type-checking and the verbosity of the solution. As such, only two options were possible: either to explore the
possibilities regarding languages that could easily interact with Java, or to pursue an option in Java that
could eliminate (or alleviate) the already mentioned problems. Due to the existing Frappé work, which
already explored FRP in Java, choosing the former allowed the exploration of broader possibilities.

Different languages can interact with Java code with full use of existing Java libraries or existing
application code, e.g. Groovy\(^1\) or Clojure\(^2\), but Scala presented as the best option due to different
aspects, already brought out in section 1.2.3. One aspect was the possibility to call Java from Scala,
allowing for the interaction between this project and the existing RTB Platform. Other aspects were that
Scala allows to add new language constructs in form of libraries (unlike, e.g., Groovy) or that Scala has
a Java-like syntax (unlike, e.g., Clojure, which is LISP-like).

4.3 Localised computing and performance

As seen in section 3.1, FRP is one model in the broader SPS model, characterized by filters where
some kind of computation is done and that communicate through a series of channels where streams are
passed. It is easy to see the similarity with the actors model of computing presented earlier. Even more,
the actors model is specially adequate for concurrent computation, allowing for different actors to be
spawned across cores or CPUs in the typical enterprise servers. With concurrent execution, performance,
a non-negligible attribute due to the enterprise interests of this work, also benefited.

4.4 External data sources

As seen in chapter 3, many contributions in FRP had a rather limited number of external event sources
(the mouse clicks in Fran [EH97], or timers, mouse clicks and keyboard presses in FrTime [CK06]). Again,
with the insight of SPSs, regarding sources where data is brought into the system, it was noted that the
different sources from the RTB Platform should map to sources in RTBScript. However, this is not a
trivial task, and must take into account the current representation of such sources in the existing RTB
Platform. We will detail this problem further in section 5.1.3

\(^1\)http://groovy.codehaus.org/
\(^2\)http://clojure.org/
Chapter 5

RTBScript

RTBScript is a high-level scripting language aiming to use the RTB Platform’s event collection capabilities, such as integration with different technologies and from distributed sources, as a means to create customised reports. This language will allow to get data from different sources and, through modeling of dataflows, create rules that, when met, trigger the execution of certain actions.

5.1 RTBScript Architecture

This section describes the RTBScript’s architecture. We start by presenting the architecture of RTBScript in the RTB Platform, and then, more specifically, RTBScript’s architecture itself.

5.1.1 Global Architecture

RTBScript is a new component in the RTB Platform. The components responsible for the event collection of the different sources of data are the RTB Data Collectors. These components, using the Kiron Middleware [Con06], are able to collect events from sources using different technologies, such as Java Servers, .Net, or even events generated in browsers with Javascript or Flash. All the events collected from the different sources pass through the RTB Data Collectors before being persisted to a database. The other main component of the RTB Platform is the RealTimeBiz Management Console, where standard reports are displayed to the business owners.

Since RTBScript is meant to deal with events as they are generated, the RTB Data Collectors are the recommended points to which RTBScript should be connected to. Another option could be the database, due to its central role in the architecture. However, databases are useful to store the data, not to propagate it: if RTBScript would use the database as its data source, it would have to use a pull model, constantly querying the database for changes in the different data sources, and introducing a delay at least equal to the update interval. Using RTB Data Collectors, it is possible to use a push model, since these components send each event directly to the RTBScript server, allowing for (near) real-time processing of events.

In figure 5.1 we can see the global architecture of the RTB Platform with RTBScript. All the communications between components are made using Kiron [Con06].

It is important to notice that, due to this architectural choice, RTBScript will deal only with events generated after being set up, that is, it won’t be collecting past events already registered in the database. We will come back to this aspect later in this chapter, to assess its importance.

In figure 5.1 it is possible to see that, if necessary, RTBScript would be able to communicate with the database, although not currently doing so. We will address this possibility further in this chapter.
In a RTBScript server there is an *engine*, responsible for the initial processing of each event. Due to the current specifications of the RTB Platform, each event source belongs to what is referred to as an *Application*, allowing for the partition of different sources by different Applications.

In figure 5.2 we can see a conceptual layout of RTBScript architecture.

### 5.1.2 The RTBScript Engine Architecture

As we have already seen, the events are delivered to the RTBScript Engine from the RTB Data Collectors. However, one of the first responsibilities of the Engine is to separate each event according to the Application it belongs.

To allow the events to be delivered to the RTBScript Engine, there is an object in the Engine that acts as point of delivery for all the events. This object is called the **RawSignalCollectorService** and, through the Kiron Remote Method Invocation (RMI) [Con06], it is invoked by the Data Collectors to deliver the raw signals. In program 5.1 we can see the simple definition (in Java) of this delivery point.

The **RawSignalReceiver** is a small object (written in Scala) that preprocesses the raw signals and then delivers it to the main engine. The definition of this object can be seen in program 5.2.

In these small examples we can see the interaction between Java and Scala code (in this case, calling the Scala **RawSignalReceiver** object code from Java’s **RawSignalCollectorService** implementation).

The line `Engine ! Process(rawSignal)` deserves a few explanations:

- **Engine** is called an *object* in Scala, and it is a singleton class. Obviously, each singleton class has one and only one instance.

- `!` seems like an infix operator, but in fact, due to the pure object-orientation of Scala, is the invocation of the method with the same name in the object **Engine**, that is, is equivalent to `Engine.(Process(rawSignal))`. 
Program 5.1: Point of delivery of the raw signals

```java
public class RawSignalCollectorService implements RemoteObject {
    // RemoteObject is from the Kiron Middleware
    private static RawSignalReceiver _receiver = new RawSignalReceiver();

    private RawSignalCollectorService(){}

    public static void registerRawSignal(RawSignal rawSignal) {
        _receiver.registerRawSignal(rawSignal);
    }
}
```

- `Process(rawSignal)` is in fact an instantiation of class `Process` (in Scala not all instantiations need the keyword `new` - please refer to [OSV08] for more details, as this is beyond the scope of this work), with `rawSignal` as its argument.

In reality, `Engine` is an Actor (presented in section 3.2), and `Process(rawSignal)` is the message to be delivered to that actor, so that `Engine ! Process(rawSignal)` only leaves the message in the actor’s mailbox and returns immediately.

The task of `Engine` is to distribute the different raw signals for the corresponding Applications for further processing, as each RTB Application can have its set of distinctive event sources. In figure 5.3 we can see a conceptual representation of the Engine architecture. It is worth noting that, unlike the usual method invocation, all communication between actors is done through message passing.
Program 5.2: Delivery of Raw Signals to the Engine

```scala
object RawSignalReceiver extends RawSignalCollector {
  // RawSignalCollector is an abstract class with an abstract
  // registerRawSignal method
  def registerRawSignal(rawSignal : RawSignal) {
    preprocess(rawSignal);
    Engine ! Process(rawSignal);
  }
  def preprocess(rawSignal: RawSignal) {
    ...
  }
}
```

Figure 5.3: RTBScript Engines architecture

5.1.3 Type Safety

To provide flexibility to the different domains, the RTB Platform allows for the definition of different types of events to be processed. This specification uses an XML schema to define, for each RTB Application, groups of events and the properties that each one has.

In figure 5.4 we present a small example of the resulting event types in the RTBMC. In this example, there are 3 groups of events: Teaser, Page and Login. In each group, there is the indication of which events the RTB Platform will collect for that group. In one last note, each event can have a set of properties (not shown in the figure): e.g., the event View of the group Page could have the property of which page was viewed.

However, in the RTB Data Collectors, these events are in a too general form, mainly mapping properties to values, with no way to provide type safety from different types of events. Since we want to be able to provide type safety in RTBScript, there is a need to, prior to the processing of the different events, translate them to instances of the corresponding types.

Part of the responsibility of the RTBScript Engine is exactly that one: transform the generic events, named raw signals, into typed ones based on meta-information in each raw signal indicating their type.
The other responsibility is to distribute each event to the corresponding RTB Application, as seen in figure 5.3.

As an example, consider the simple XML schema in listing 5.3 (the true XML schema used in the RTB Platform has more elements, but these are not relevant for this work).

Listing 5.3: Sample XML schema for a Sale type

```xml
<rtbevents>
  <eventtype name="Events">
    <event name="Sale">
      <column name="productId" type="number" />
      <column name="quantity" type="number" />
      <column name="price" type="number" />
    </event>
  </eventtype>
</rtbevents>
```

In program 5.4 we can see the corresponding class to the defined XML. As we can see, the event name in XML is the name for the class in Scala. Also, the different properties (corresponding to the tag `column` in the XML), translate to arguments in the constructor of the `Sale` class. One last aspect worthy of notice is that the type specified in the XML has to be matched to existing types in Scala. In the XML example we can see the type `number` (as defined in the RTB Platform, representing both integer and floating point numbers), translated to the type `Double` in the corresponding Scala class.

Program 5.4: Corresponding Scala class to the XML representation

```scala
case class Sale(productId: Double, quantity: Double, price: Double);
```

However, creating the corresponding classes to the XML schema is not enough: we have already seen that events arrive to the RTBScript Engine in a very crude way, as mappings of properties to values and some meta-information regarding types and to which Application each event belongs to. However, to integrate RTBScript into the RTB Platform, each element defined by the tag `event` in the XML is mapped to a corresponding `source of events` in RTB Script, where types are mandatory. As such, before being delivered to the `sources` for further processing by the user scripts, those mappings must be translated to instances of the corresponding classes.

In figure 5.5 we can see a small UML representation of the classes that are involved in the translation process.
When the engine for the corresponding application receives an event to process (from the main engine, already described), it passes it to a dispatcher built for each application that, through meta-information in the raw signals, dispatches it to a translator. This translator builds a new instance of the corresponding type based on the received raw signal, and returns this typed version, which is then dispatched to the corresponding signal source, described later in this chapter.

In program 5.5, we can see an example of a translator for the presented XML. Again the keyword `new` is not used to indicate a new instance of the class `Sale`, although a new instance is being created.

Program 5.5: Translator for the example class Sale

```scala
object EventsSaleTranslator extends Translator[Event[Sale]] {
  val _1 = "productId";
  val _2 = "quantity";
  val _3 = "price"

  def translate(rawSignal: RawSignal): Event[Sale] = {
    SomeEvent(
      Sale(
        rawSignal.getValue(_1),
        rawSignal.getValue(_2),
        rawSignal.getValue(_3)
      )
    )
  }
}
```

5.2 The RTBScript Language

In this section we present the foundations of the RTBScript language, its basic concepts and implementation.

Due to its importance in this work, type parameterization will be commonly used throughout the rest of the document. To ease the understanding of the used definitions, we present a short explanation of
Type parameterization allows to write generic classes. For example, arrays (in Scala) are generic and take a type parameter: in this document this will be represented as $\text{Array}[T]$, that is, the class being parameterized appears before the [], and the parameter type (or types, as there can be more than one, separated by ’,’) inside it. Types in italic would refer to “placeholders” to concrete types. Types represented in monospace will refer to actual types. As an example, $\text{Array[String]}$ would refer to an array that would contain instances of type $\text{String}$. When we want to refer to the type of determined object, we will use : to separate the object of its respective type: e.g., $x : \text{Int}$, to indicate an object $x$ of type $\text{Int}$.

Since Scala allows for the definition of first-class functions, and these are also important to this work, we present the definition that we will use in this document: functions will have its input type (the domain) separated from the output type (the codomain) by the symbol $=>$ As an example, the function $\text{successor}$ (given an $\text{Int}$ number, returns its successor), would be referred as of the type $\text{Int} \Rightarrow \text{Int}$. As another example, the simple integer sum could be defined of the type $\text{Int, Int} \Rightarrow \text{Int}$. However, in $\text{(Int, Int)} \Rightarrow \text{Int}$ there is only one parameter to the function, but of the type of a 2-tuple (commonly, a $\text{Pair}$) of $\text{Int}$’s. This detail will prove its importance later in the document. When referring to higher-order function, we will also use the () to disambiguate between representations as, e.g., $(\text{Int} \Rightarrow \text{Int}) \Rightarrow \text{Int}$ to represent a function that receives a function from $\text{Int}$ to $\text{Int}$ and returns an $\text{Int}$ as a result.

Now that we have explained the terminology, we proceed to introduce the basic concepts in the RTBS.

5.2.1 Basic Concepts and Core Primitives

The RTBS language has its foundations in the concepts presented in Arrowized Functional Reactive Programming. As already showed in section 3.1, AFRP descends from the original FRP work, Fran, where some space-time leaks could occur due to the first-class nature of behaviors and events. Due to the work of John Hughes in the Arrow framework [Hug00], “FRP was recast as an instance of Arrows, which directly gave [AFRP] firm theoretical underpinnings” [NCP02].

Remembering from section 3.1.5 in AFRP, there are two key concepts: Signals, which represent both behaviors and events in FRP, and the Signal Transformers (or Signal Functions, as we will call them in the rest of this document), which represent the functions that map signals to signals:

$$\text{SignalFunction}[\alpha,\beta] = \text{Signal}[\alpha] \Rightarrow \text{Signal}[\beta]$$

where

$$\text{Signal}[\alpha] = \text{Time} \Rightarrow \alpha$$

To avoid space-time leaks, signals are simply not allowed as first-class values. Instead, the programmer has access only to the signal functions. However, the actual representation of the type $\text{SignalFunction}$ in RTBS, as in AFRP, is hidden, so that signal functions cannot be built or applied directly to signals. Instead of allowing the user to define arbitrary signal functions from scratch (which would allow to introduce time-space leaks), RTBS, again as in AFRP, provides a set of primitive signal functions and a set of special composition operators (or “combinators”) with which more complex signal functions can be defined.

As such, a RTBS program expresses the composition of a possibly large number of signal functions into another composite signal function that is then run in the underlying system. However, the composition of signal functions does not have to be completely linear, as will be illustrated shortly. Indeed, because signal functions are abstract, we should guarantee that we have a sufficient set of combinators to compose our signal functions without loss of expressive power as we will see later.
Considering the functional paradigm history behind FRP and AFRP, it is expected that the already existing ordinary functions can be “lifted” to the corresponding signal functions. As such, the first operator we introduce is the lift operator, \( \text{arr} \) (named after \( \text{arrow} \)):

\[
\text{arr} : (a \Rightarrow b) \Rightarrow SF[a, b]
\]

As we can see, \( \text{arr} \) takes as its parameter a function from some type to another, and returns the corresponding signal function (for conciseness, it is used \( SF \) instead of \( \text{SignalFunction} \) as shown earlier) from and to the types specified by the function.

Another typical operation with functions is the function composition: given \( f: a \Rightarrow b \) and \( g: b \Rightarrow c \), then another function can be the composition of the earlier defined as \( h(x) = g(f(x)) \) of the type \( h: a \Rightarrow c \). To compose signal functions, there is also a primitive combinator:

\[
\gggg : SF[a, b], SF[b, c] \Rightarrow SF[a, c]
\]

As such, we could write

\[
SFh = \text{arr}(h)
\]

as equivalent to:

\[
SFh = \text{arr}(f) \gggg \text{arr}(g)
\]

Note that \( \gggg \) actually represents reverse function composition in its mathematical sense.

However, most programs are not simply linear compositions of functions, and it is often the case that more than one input and/or output is needed. For example, suppose now that we have \( f: a \Rightarrow b \) and \( g: a \Rightarrow c \) and we wish to define \( h(x) = (f(x),g(x)) \), that is, given its argument, return a pair of the application of \( f \) and the application of \( g \) to the same argument. To provide the same operation with signal functions there is another combinator:

\[
\&\& : SF[a, b], SF[a, c] \Rightarrow SF[a, (b, c)]
\]

Again, we could write

\[
SFh = \text{arr}(h)
\]

\[
= \text{arr}(f) \&\& \text{arr}(g)
\]

So far, we have looked only to \textit{stateless} signal functions: the value of the output signal at time \( t \) depends only on the value of the input signal at \( t \). All the functions that are converted to signal functions by \( \text{arr} \) are of this kind. As this would clearly be insufficient, RTBScript also provides the primitive \( \text{TIArr} \) to create signal functions that are stateful. Such primitives produce an output signal that may depend not just on the input signal at \( t \), but at all times in \([0; t]\). The \( \text{TIArr} \) (named after \textit{Time Input Arrow}) combinator is defined as:

\[
\text{TIArr} : (a, \text{DTime}, state \Rightarrow (b, state)), state \Rightarrow SF[a, b]
\]

Since this definition is a bit more complicated than those presented so far, we will explain it further: a stateful function maps an input value of type \( a \) at instant \( t \) to the corresponding output value of type \( b \). Since it is a function that depends on time itself, and since continuous representation can only be achieved through sampling, it is also provided the \textit{time delta} (represented by the type \( \text{DTime} \)) since the last sample was provided for evaluation. However, this is not enough since the function also might need information regarding all times in \([0; t]\). As such, some sort of state must be maintained by the function. In this case, it is provided the last state so that it can process the new output value along with the new state. This new state will be provided when the next sample of input is available. Due to this model, it is necessary to provide the initial state, so that, when the first sample is provided, the function may compute properly.

As presented in section 3.1, a basic stateful primitive is the integration. With the following definition
integralFun : Double, DTime, Double => (Double, Double)
integralFun(x, dt, accum) = { temp = accum + x * dt; (temp, temp) }

integral : SF[Double, Double]
integral = TIarr(integralFun, 0)

the integral primitive computes the time integral of its input signal (in this case, defined for the type Double). With this definition, integrating the constant 1 would yield the local time (represented by the type Double), that is, how much time passed since the execution started:

localTime : SF[a, Double]
localTime = const(1.0) >>> integral

We could go on and on in this manner, adding combinators as they are needed to solve particular “argument wiring” problems, but it would not be feasible, since, as we said earlier, signal functions cannot directly be built by the user, due to the guarantees we want to provide. As such, we need to find the minimal universal set of combinators that is sufficient to express all possible wirings.

In fact, three of the combinators we already introduced, arr, >>>, and &&& constitute a minimal universal set (as proven in [Hug00]). However, this is not the only minimal set: in the original work defining the Arrow class [Hug00] the set arr, >>>, and first were the minimal universal set chosen. first is defined as first(x,y)= (f(x),y) for some function f: a => b. As such, first is defined in RTBScript as

first : SF[a, b] => SF[(a, c), (b, c)]

As an example to see how this set is minimal, here are the definitions of second (defined as second(x,y)= (x,f(y)) ) and &&& in terms of the already defined combinators:

second : SF[a, b] => SF[(c, a), (c, b)]
second(f) = arr(swap) >>> first(f) >>> arr(swap)

&&& : SF[a, b], SF[a, c] => SF[a, (b, c)]
 f &&& g = arr(dup) >>> (first(f) >>> second(g))

where

swap : (a, b) => (b, a)
swap((x, y)) = (y, x)
dup : a => (a, a)
dup(x) = (x, x)

Another important feature of any language is to have some kind of conditional choice capability, and so RTBScript has it too. Indeed, this can be achieved with the already provided combinators: given the signal functions flag : SF[a, Bool] and sfx, sfy : SF[a, b], then the signal function:

SFIf : SF[a, b]
SFIf = ((sfx &&& sfy) &&& flag) >>> arr(funcIf)

where

funcIf((x, y), b) = if b then x else y

behaves like sfx whenever flag yields a true value, and like sfy whenever it yields false. However, this is not always enough, because there are situations where we would prefer that a signal function switch
into, or literally become, some other signal function, rather than continually alternate between two signal functions based on the value of a boolean. As an example, we could consider a succession of new signal functions to switch into as a succession of particular events occurs, much like state changes in a finite state automaton.

This functionality is achieved in RTBScript, as in AFRP [HCNP02], using events and switching combinators. Events in RTBScript are just abstract values that are analogous to the Maybe type presented in section 3.1.5. A signal of type Signal[Event[b]] is called an event stream, and is a signal that, at any point in time, yields either NoEvent, as indication for no value, or SomeEvent(b), as carrying a value of type b. A signal function of type SF[a,Event[b]] generates an event stream.

From AFRP, we can learn that “although event streams and continuous values are both represented as signals, there are important semantic differences between them. Semantically speaking, event streams should not be ‘infinitely dense’ in time; practically speaking, their frequency should not exceed the internal sampling rate.”

The existing switching combinator is called switch, whose type is given by:

\[
\text{switch} : \text{SF}[a,(b,\text{Event}[c])], (c=>\text{SF}[a,b]) \Rightarrow \text{SF}[a,b]
\]

Although it seems complex, the underlying idea is pretty simple: the switch combinator behaves as the provided signal function until the first event in the event stream occurs, at which point the event’s value is then delivered to the function to process it. The output of that function is the signal function that the switch combinator will switch into. Later in this chapter we will see a concrete example of such combinator.

So far, we have seen that several minimal sets of combinators exist to express all possible functions wiring, but in practice it is better to think in terms of a commonly used set of combinators rather than a minimal set. We present in listing 5.6 the set of common combinators in RTBScript, and, since an image is worth a thousand words, figure 5.6 shows the graphical representation of these (the orange rounded rectangles represent signal functions, the circles represent normal functions, and the rectangle a constant value).

Listing 5.6: Set of commonly used combinators in RTBScript

| arr   | (a=>b) => SF[a,b] |
| TIarr | (a,DTime, state => b), state => SF[a,b] |
| const | b => SF[a,b] |
| >>>  | SF[a,b], SF[b,c] => SF[a,c] |
| first | SF[a,b] => SF[(a,c),(b,c)] |
| second | SF[c,d] => SF[(a,c),(a,d)] |
| &&&   | SF[a,b], SF[a,c] => SF[a,(b,c)] |
| ***   | SF[a,b], SF[c,d] => SF[(a,c),(b,d)] |
| loop  | SF[(a,c),(b,c)], c => SF[a,b] |
| switch | SF[a,(b,Event[c])], (c=>SF[a,b]) => SF[a,b] |

One important feature to take into account is that when more than one input and/or output is needed, these are represented as n-tuples both at input or output, so that, in reality, each signal function maps one type to one other, even if those types are tuples of other types. We will return to this issue later in this chapter.
Figure 5.6: RTBScript combinators
5.2.2 Implementation of RTBScript

The different signal function types

As we have seen in section 3.1.1, Fran [EH97] had its implementation based in streams (through the use of lazy lists in Haskell). Although appropriate for many purposes, this stream-based implementation, according to [NCP02], did have some deficiencies:

1. At the implementation level, there is no way to identify signal functions that only react to changes in the input signal. As such, sampling must occur at every time step, even though the program will only react to specific input events. Identifying signal functions that only react to changes in input would enable the implementation to make a blocking call to the operating system until an appropriate event occurs, a substantial performance improvement.

2. The implementation does not retain enough information to do any runtime optimization of the dataflow graph.

As of the latter, we will address it later, in section 5.4.

Regarding the former, it is then obvious there should be, at the implementation level, some mechanism allowing to distinguish between the different types of signal functions. Basically, there are three different types of signal functions:

Generic (or time-input variable) the most general case of a signal function, where there is no particular “extra” information known about the transition function. As such, this signal function is considered time-input variable, that is, it may depend not only of the sample at instant $t$, but it may also depend of the time delta since the last sample;

Arrowized denotes a “pure” signal function, that is, a stateless function: at any time $t$, the output signal at $t$ depends only on the input sample at $t$, but not on the time delta since the last sample;

Constant denotes a signal function that has gone “constant”. In this case, the output value do not change from one sample to the next, regardless of input sample or time delta since last sample.

The formal definition of these types of signal functions goes beyond the scope of this work, but can be found in [NCP02].

From the definitions, it is easy to see that a signal function $sf$ that is constant, would also be arrowized and also generic, since the constant signal functions are a subset of the arrowized, that in turn are a subset of the generic signal functions. If we consider the predicates $isConst(sf), isArr(sf)$ and $isGen(sf)$ to indicate if a signal function $sf$ is constant, arrowized, or generic, respectively, we would have:

$$(isConst(sf) \Rightarrow isArr(sf)) \land (isArr(sf) \Rightarrow isGen(sf))$$

With these different types of signal functions it is now easy to see why the existence of the three different “lifters” in RTBScript: const, arr and TIarr correspond to the constant, arrowized and the generic types of signal functions, respectively. This way we can easily identify each type of signal function. The identification of each type will allow for some optimizations, discussed further in section 5.4.

The use of Actors

We have seen in section 3.1 that FRP is a synchronous dataflow language, or in other words, at every time step, a signal function will consume exactly one value from its input stream and produce exactly one value on its output stream.
In section 3.2 we have seen that the actors model resembles this synchronization scheme: each actor waits for messages and when receives a message, processes it (where it may send messages to others), and then awaits for the next message. If to each message the actor receives, it sends exactly one message to other actor, then it can be compared to the synchronous definition given above (however, note that message delivery between actors is still asynchronous).

As such, all the RTBScript combinators are in fact actors. To each combinator, corresponds an actor that knows how to process each signal according to its function.

Consider the following example in RTBScript language (where \( f: a \Rightarrow b \) and \( g: b \Rightarrow c \)):

```
Program 5.7: Simple RTBScript example
val SFh = arr(f) >>> arr(g)
```

(The `val` keyword actually belongs to the Scala language, and it is used to declare constant values. Please note that this only prevents `SFh` from being altered after its definition.)

In figure 5.7 we represent the corresponding actors scheme given the RTBScript in program 5.7.

```
val SFx = ((arr(x) >>> arr(y)) >>> arr(z)) >>> arr(w)
```

From figure 5.8, it is easy to notice that message forwarding introduces several “relay” messages between the different actors. As we have seen earlier, each actor is suspended awaiting for messages to process, being awakened when such messages arrive to its mail “inbox”. However, with this schema, the composition actors are constantly being awakened only to forward the messages received from the actors they are composing. Also, the number of messages being passed seems excessive for such simple composition.

With these problems in mind, a new solution should be found: instead of composition actors relay the messages between the composed actors, there should be some mean so that composed actors could communicate directly. In fact, this is what RTBScript implements, through channels.

Channels provide for typed versions of message passing between actors. We have already seen that actors receive messages to process, however, in its purest form, an actor puts no restriction to what types
of messages it receives: if the current behavior finds a match to a message in its mailbox, processes it and moves to next message; however, if no match is found, the message is kept in the mail inbox queue until the behavior of the actor allows it to process it (due to this case, an message could be indefinitely in the mailbox, if the behavior of the actor never matches the message – thus, it is a common case to include a match that captures “everything else”, when all the other expected messages have been matched).

With channels, all the messages that pass through it must be of determined type. This is used in RTBScript to provide type safety between the signals that are transmitted between actors. Each actor can create multiple channels that it can communicate to the others and, through it, the other actors send typed messages to the actor who created it. However, the channel can only be used to pass messages to the actor who created it. Also, the channel can only be used as “input” channel to the respective actor, that is, if an actor sends a message through a channel, and expects a message as an answer, that channel cannot be used to get it, since if the receiver would send a message to that channel, it would instead receive its own message.

To illustrate what in fact happens in the RTBScript system, we now explain how these two combinators, \( \text{arr} \) and \( \ggg \), are implemented in RTBScript. In program 5.9 we can see the (simplified) implementation of the compositor \( \text{arr, SFArr} \):

In line 1 we can see the definition of \( \text{SFArr} \) as presented in listing 5.6: \( \text{SFArr} \) is just a signal function from some type \( TIn \) to type \( TOut \), and receives a function from those types (\( \text{val func}: TIn \rightarrow TOut \)). Lines 3-5 show the creation of an actor which behavior is always the same (the \( \text{loop} \) construct), and processes the messages through \( \text{react} \) (the high performance lightweight implementation of Scala actors, already presented in section 3.2.1). Lines 6-14 define the behavior of this actor:

- in lines 6 and 7, the actor expects to receive a sample in input and produces the output through the application of the lifted function. We can see that, in fact, actors send signals to each other: each signal contains the information about the sample input (\( \text{value} \)) and also the time elapsed since the last sample (\( \text{dt} \));

- in lines 8 and 9, we can see the processing of a control message, \( \text{CreateChannel} \). This message tells the actor to create a new channel to communicate with it, and then replies to the sender with the created channel.

- in lines 10 and 11, there is a “catch all” rule for message processing, so that an erroneous message
Program 5.9: The implementation of the arr combinator

```scala
1 case class SFArr[TIn, TOut](private val func: TIn => TOut) extends SignalFunction[TIn, TOut] {
2
3 val processor: Actor = actor {
4   loop{
5     react {
6       case channel ! Signal(dt, value: TIn) => {
7         reply(Signal(dt, func(value)))
8       }
9       case CreateChannel(channel) => {
10          reply(new ForwarderChannel[Signal[TIn]](channel))
11       }
12       case _ => { // "catch all" rule
13          ...}
14     }
15   }
16 }
17 }
18 }
```

delivered to this actor won’t be indefinitely in the mailbox, as already described.

As of the composition combinator, >>>, we can see in program 5.10 that this actor does not receive signals to process, as it was expected from figure 5.8. In fact, all this actor expects is the control message to create channels. However, in this case, the actor does not create a channel to itself: it asks to its acquaintances to create channels themselves and returns the resulting channel chain to whoever asked for it.

In figure 5.9, we illustrate how the actors really communicate in the RTBScript system.

![Figure 5.9: Real implementation of program 5.8](image)

We can see that whichever actor connects to the signal function SFx, it will not really communicate with the composition actors but with the first simpler combinator in the chain of compositions. As such, the composition actors are only awakened to create the channel chains, greatly reducing the number of messages exchanged between actors.
Program 5.10: The implementation of the composition combinator

```scala
1  case class SFComposition[TIn, TTemp, TOut](val sf1: SignalFunction[TIn, TTemp], val sf2: SignalFunction[TTemp, TOut]) extends SignalFunction[TIn, TOut] {
2 3  val processor: Actor = actor {
4    loop{
5      react {
6        case CreateChannel(channel) => {
7          val chanTo2nd = sf2.processor !? CreateChannel(channel)
8          val chanTo1st = sf1.processor !? CreateChannel(chanTo2nd)
9          reply(chanTo1st)
10        }
11        case _ => {
12          // "catch all" rule
13          ...
14        }
15      }
16    }
17  }
18}
```

With this design, each combinator only has to know how to create the channels to provide its functionality. However, only the composition combinator is able to avoid relaying messages: if we consider the example of `first` (represented in figure 5.10), the actor must do some processing regarding types. From listing 5.6 we can see that it receives a pair `(a, c)` and outputs a pair `(b, c)`, but the enclosed signal function maps `a` to `b`. As such, `first` has to extract the corresponding `a` value, pass it to the enclosing signal function, wait for the output `b` value, and then create a pair to the output of `first`, of type `(b, c)`. Although these combinators (`arr`, `>>>` and `first`) are a minimal set, the other combinators are implemented natively, following a similar approach.

An important issue not brought out so far was the reuse of already defined signal functions. Due to the imperative style of the Scala language, the script in program 5.11 is allowed (please consider all the used functions valid).

In this example, we can see that the already defined signal function `SFmiddle` is used in the definition of `SF1` and `SF2`. One could argue that the signals from `arr(f)` (from `SF1`) would be delivered to `arr(g)` and also delivered to `arr(z)` (from `SF2`) since they traverse the same “middle” section. This would clearly not be what was intended (that effect could be obtained with `arr(f) >>> SFmiddle >>> (arr

Figure 5.10: Real implementation of combinator `first`
Program 5.11: A signal function reuse RTBScript example

```scala
val SFmiddle = arr(x) >>> arr(y)
val SF1 = arr(f) >>> SFmiddle >>> arr(g)
val SF2 = arr(w) >>> SFmiddle >>> arr(z)
```

(g)\\&\\& arr(z)), so the system has to be robust enough to handle this case. In fact, the RTBScript implementation again uses channels to solve this problem: to each definition of the signal functions, the RTBScript creates a separate channel chain to each, so that signals from one signal function are not delivered to the wrong signal function. This is represented in figure 5.11.

![Figure 5.11: Separate channel chains for common definitions of signal functions.](image)

Note that, without channels, with signals being delivered directly to each actor, each signal would have to have in its implementation the list of remaining combinators that would yet have to process it. This option would imply that when a signal entered the RTBScript system, it would be “filled” with the combinators that would process it (that is, all the dataflow for that signal) and then passed to the first combinator. The first combinator would then process the signal, remove itself from the top of the signal’s list, and finally forward it along to the next combinator in the top of the list, and so on. This would result in signals that would be more complex and bigger in memory usage, which we avoid with this approach.

As a final note regarding the use of actors in the implementation of RTBScript, we now describe some operational implications of such use:

**Exception handling:** as we have seen, the `arr(f)` combinator creates an actor to process the lifted `f`
function. So far, we have not addressed the issue of such function being unable to correctly produce the output and, instead, throwing an exception: in this case, the exception would be delivered to the corresponding actor. If the actor should not catch the exception, it would be itself terminated by the underlying system. When another actor would try to deliver a message to the terminated actor, an exception would be raised due to the unsuccessful message delivery, and this actor would also be terminated. It is easy to see that this would cause a chain termination of all the actors in a dataflow. As such, in reality, the actors are “shielded” from potential exceptions from lifted functions: when such case occurs, the sample is just ignored, and the exception caught by the processing actor.

**Blocking actions:** In RTBScript each lifted function can perform any action, including blocking calls to the underlying system. However, since each lifted function is in fact an actor, this implies that it will also be blocked, preventing it from processing the signals that can arrive to its mailbox in the meanwhile. If the blocking call is actually longer than the interval between signals, this will introduce delays in the dataflow. Although the RTBScript system guarantees that the dataflow will still be conceptually correct, this is still an undesired effect. We will return to this issue in section 5.4.

**The Signal Sources**

So far we have seen how to combine the different signal functions, but we have only focused on the signal functions that *transform* the different signals. We now focus on the signal functions that bring the different samples to the RTBScript’s dataflow processing system, the *signal sources*.

Signal sources are a special type of signal functions, defined as:

\[
\text{SignalSource}\[a\] = SF[\text{Nothing} , a]
\]

For brevity, from now on, when referring to the type `SignalSource[a]`, we’ll use the shorter term `SS[a]`.

The type `Nothing` is part of the Scala type system: it is at the very bottom of Scala’s class hierarchy, as it is a subtype of every other type. However, there are no values of this type. Usually, the type `Nothing` is used to denote abnormal termination using exceptions (please refer to [OSV08] for further explanations). However, this is not the reason it is defined as such in RTBScript: since there is no value of this class, it is also not possible to introduce it in a middle of a dataflow, that is, it is not possible to create a signal function such that `Nothing` is the output type (`SF[a, Nothing]`). As such, it is only possible to compose signal sources in the beginning of dataflows, but not in the middle of compositions. This is also conceptually consistent, since signals enter the dataflow from signal sources, and they are “created from nothing” in signal sources.

It is through the signal sources that the different samples are turned into signals and introduced to the dataflow for processing. It is also in the signal sources that the creation of the channel chains is initiated. As such, when a signal function is composed with a signal source, a channel chain is created to represent the signal function’s dataflow.

An important aspect about signal sources is that the user cannot create them at will. The signal sources are directly related to the distributed sources available in the RTB Platform. As such, each signal source is generated according to the XML model for each application in the RTB Platform. Considering the example already shown in listing 5.3 earlier in this chapter, the corresponding signal source would be generated: `SaleSignalSource`, that is, the name of the event source followed (by default) by “Signal-Source” (this name could be set to something else, such as “Stream”). Since each signal source belongs to a group of events, and each group to an RTB Application, we make use of the Scala packages system to
properly locate each signal source. Currently, the signal sources would be distributed by the packages in the following way: rtbscript.apps.<applicationName>.<groupName>.<signalSourceName>, where the italicised names are replaced by the actual names. The types created for each signal source (presented earlier in section 5.1.3) are also distributed by the packages, but instead of <signalSourceName>, it is used the <eventName>.

The other important aspect about signal sources is their relation to the continuous and discrete domains, since signal functions deal with both concepts. As such, signal sources for continuous values will be of type SS[a], while the discrete ones will be of type SS[Event[a]] (using the already described Event[a]).

Returning to the sales example, it is easy to see that SaleSignalSource will be of type SS[Event[Sale]], since these have been defined as discrete occurrences. Further in this chapter will discuss the relation between discrete and continuous values, and how to convert one into another.

Now that we have covered (almost) all the RTBScript language, we present the overall architecture of the RTBScript in figure 5.12.
Figure 5.12: The overall architecture of RTBScript regarding actors.
5.2.3 The RTBScript Library

So far we have seen the different combinators available to the user, and inclusively, we have shown in section 5.2.1 that, from those, is possible to find not one, but several minimal subsets. However, as these provide expressiveness to the language, they do not always ease the use of it.

As such, the RTBScript language also provides a library, where includes several functions that aid in writing scripts.

We now present some definitions of the existing library. Please note that this is not Scala code.

**Basic functions**

dup and swap are commonly used functions, due to the particularity of signal functions’ use of n-tuples. Although these were already defined in section 5.2.1, we re-introduce them again:

\[
\text{swap : } (a, b) \Rightarrow (b, a) \\
\text{swap((x,y)) = (y,x)}
\]

\[
\text{dup : } a \Rightarrow (a, a) \\
\text{dup}(x) = (x, x)
\]

**Changing between discrete and continuous representations**

Sometimes it is useful to change between representations from a continuous signal function to a discrete one, or vice-versa. The functions changes and hold provide for these operations, respectively. In figure 5.13 there is a visual representation of these functions.

\[
\text{hold : } a \Rightarrow SF[Event[a],a] \\
\text{hold(initial) = loop(arr(extractor), initial)}
\]

\[
\text{changes : } a \Rightarrow SF[a,Event[a]] \\
\text{changes(fromWhat) = loop(arr(stripDuplicates), fromWhat)}
\]

\[
\text{extractor : } (Event[a],a) \Rightarrow (a,a) \\
\text{extractor((NoEvent, x)) = (x,x)} \\
\text{extractor((SomeEvent(y),x)) = (y,y)}
\]

\[
\text{stripDuplicates : } (a,a) \Rightarrow (Event[a],a) \\
\text{stripDuplicates((x,y)) = if x==y then (NoEvent,y) else (SomeEvent(x), x)}
\]

It is important to notice that the use of these functions has its quirks: one could expect that hold(x) >>> changes(x) (creating a signal function of type SF[Event[a],Event[a]]) would generate the same number of events (that is, the composition of such signal functions would result in the identity function), but such is not true. In fact, this would filter the consecutive events that yielded the same value. As an example, consider the following set of consecutive events: \{SomeEvent(1), SomeEvent(2), SomeEvent(2), SomeEvent(3), SomeEvent(2)\}. After passing through the referred signal function, the set would be \{SomeEvent(1), SomeEvent(2), SomeEvent(3), SomeEvent(2)\}, “dropping” one event (the second SomeEvent(2)).
Event counting and filtering

Two common operations used in events relate with counting how many events were already detected, and filtering some of those events. As such, we now define countE and filterE (the last E stands for Events). Please note that filterE filters out those events that do not satisfy the predicate.

\[
\text{countE} : \text{SF}[\text{Event}[a], \text{Int}] \\
\text{countE} = \text{loop}(\text{arr}(\text{counter}), 0)
\]

\[
\text{counter} : (\text{Event}[a], \text{Int}) \Rightarrow (\text{Int}, \text{Int}) \\
\text{counter}((\text{NoEvent}, x)) = (x, x) \\
\text{counter}((\text{SomeEvent}(y), x)) = (x+1, x+1)
\]

\[
\text{filterE} : (a \Rightarrow \text{Bool}) \Rightarrow \text{SF}[\text{Event}[a], \text{Event}[a]] \\
\text{filterE}(f) = \\
\text{arr}({} \\
\text{case SomeEvent}(x) \Rightarrow \text{if } f(x) \text{ then SomeEvent}(x) \text{ else NoEvent} \; ; \\
\text{case NoEvent } \Rightarrow \text{NoEvent} \; ; \\
})
\]

Acting when events occur

As we have seen earlier, the RTBPlatform deals in its majority with discrete signal sources. As such, it is common that we want to react when such events occur. To ease this task, we define two new functions: on and onceOn. The former applies a function whenever an event occurs, and the latter applies a function only when the first event occurs in a dataflow.

\[
\text{on} : \text{SF}[a, \text{Event}[b]], (b \Rightarrow c) \Rightarrow \text{SF}[a, \text{Event}[b]] \\
\text{on}(sf, f) = sf >>\text{arr}({} \text{case SomeEvent}(x) \Rightarrow f(x); \text{SomeEvent}(x) \})
\]

\[
\text{onceOn} : \text{SF}[a, \text{Event}[b]], (b \Rightarrow c) \Rightarrow \text{SF}[a, \text{Event}[b]] \\
\text{onceOn}(sf, f) = \text{switch}(\text{on}(sf, f) >>\text{dup, selectorFunction})
\]
selectorFunction: \( c \Rightarrow SF[a,b] \)
selectorFunction(x) = arr( identity ) //It doesn’t matter the x value, its discarded

identity: \( a \Rightarrow a \)
identity(x) = x

5.3 The RTBScript Language in action

We have seen until so far the RTBScript language is defined and some library functions available in it. From all we have shown, it is now clear that the lifting of common functions plays an important role in both FRP and AFRP. However, we have not yet shown how common functions are declared in the RTBScript’s host language, Scala. As such, we feel that some introduction to how functions are declared in Scala is needed before moving to the RTBScript Language examples. We start by showing how to declare common functions in Scala, and only then proceed to the RTBScript examples.

5.3.1 Scala function literals

As we have referred in section 1.2.3, Scala has higher-order functions, that is, functions are first-class entities: not only can we define functions and call them, but we can also write functions as *unnamed literals* and then pass them around as *values*. The basic syntax for function literals is as follows:

\[
(\text{typedArguments}) \Rightarrow \text{body}
\]

where *typedArguments* is a comma-separated list of parameters, defined as

\[
\text{parameterName} : \text{parameterType}
\]

and *body* is the instruction to execute (if more than one instruction is provided, these must be enclosed in \{ \}). However, as we have seen, this creates *unnamed* function literals. In Scala, if we want to name a function, we can use

\[
\text{def functionName} = \text{functionLiteral}
\]

so that we can apply the named function as *functionName*(arguments). Type parameterization of functions is also possible using

\[
\text{def functionName[typeParameters]} = \text{functionLiteral}
\]

where *typeParameters* is a comma-separated list of the type names to be bound.

We illustrate all these definitions in program 5.12.

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(x: Int) =&gt; x+1 //function literal that adds one to an Int number.</td>
<td>The resulting function type is Int =&gt; Int</td>
</tr>
<tr>
<td></td>
<td>The resulting function type is Int =&gt; Int</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>((x: Int) =&gt; x*2)(2) //applies the unnamed function literal with</td>
<td>parameter 2, yielding 4 as a result</td>
</tr>
<tr>
<td></td>
<td>parameter 2, yielding 4 as a result</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>def add = (x: Int, y: Int) =&gt; x+y //names add to the defined function</td>
<td>literal</td>
</tr>
<tr>
<td></td>
<td>literal</td>
<td></td>
</tr>
</tbody>
</table>
add(1,2) // applies add with parameters 1 and 2, yielding 3 as a result

```scala
def swapInt = (x: (Int,Int)) => (x._2,x._1) // names swapInt to a function that receives a pair of Ints. x._1 and x._2 get the values inside the pair
```

```scala
swapInt((1,2)) // would output (2,1)  
swapInt(("hello","world")) // would fail because of type clashes
```

```scala
def dup[TIn] = (x: TIn) => (x,x) // names dup to a type parameterized function literal
```

```scala
dup(1) // outputs (1,1)  
dup("hello") // outputs ("hello","hello")
```

```scala
def swap[T1 , T2] = (x: (T1,T2)) => (x._2,x._1) // names swap to a type parameterized function literal receiving a pair of the respective types.
```

```scala
swap((1,2)) // outputs (2,1)  
swap(("hello","world")) // outputs ("world","hello")
```

As we can see, in this small example we have already defined in Scala the basic functions presented in the previous section (dup and swap). Lifting these functions would result in the corresponding signal functions.

Since Scala is a strongly typed language, which uses type inference, some shorter forms of function literals can be introduced. One way to make a function literal briefer is to leave off the parameter types, if the compiler has sufficient information to infer them. However, Scala goes even further to make function literals more concise. Using the placeholder syntax, we can use underscores as placeholders for one or more parameters, so long as each parameter appears only one time within the function literal. For example, _ > 0 is very short notation for a function that checks whether a value is greater than zero. In fact, the function literal _ > 0 is expanded to an equivalent of the slightly more verbose x => x > 0. It is important to notice that each underscore refers only to one parameter, and not to the repetition of the same parameter: for example, _ > _ would be expanded to an equivalent of (x, y) => x > y.

In program 5.13 we can see an example of shorter forms of function literals and the placeholder syntax.

There is only one more case of particular usefulness of function definitions to the RTBScript Language: the use of pattern matches in the creation of function literals. Pattern matching is widely used in the Scala language, but its applications goes beyond the scope of this work. The interested reader can refer to [OSV08] for further information regarding this topic. To RTBScript, the definition of function literals through patterns is particularly useful when dealing with tuples. Using the pattern matching syntax, each definition of function literals is expanded to a function with only one argument. This proves useful
Program 5.13: Examples of function literals shorter forms and the placeholder syntax

```scala
val someNumbers = List(-11, -10, -5, 0, 5, 10) // declares a variable
  someNumbers of type List[Int], since the compiler is able to infer
  the type of the list

someNumbers.filter((x: Int) => x > 0) // would output List(5, 10) of
  type List[Int]

someNumbers.filter((x) => x > 0) // outputs the same result as
  earlier: in this case, since the compiler knows what type the list
  contains, it can infer the type of the x parameter as Int,
  avoiding the need to write it explicitly

someNumbers.filter(x => x > 0) // in the particular cases where there
  is only one parameter, the () can even be omitted

someNumbers.filter(_ > 0) // placeholder syntax, equivalent to x => x
  > 0
```

since, as we have seen earlier, the different signal functions map one input to one output, using tuples to simulate multiple input or output. In the program 5.14 we show some examples of this syntax.

Program 5.14: Examples of function literals through pattern matching

```scala
def caseSwap = { case (x,y) => (y,x) } // the variables x and y will
  be bound to the values that are inside the pair (avoiding the use
  of ._1 and ._2 as earlier)

/*
  expanded version:
def caseSwap = (x) => x match { case (x,y) => (y,x) }
*/
```

The use of pattern matching brings yet another advantage when used with the case classes in Scala. If we remember from section 5.1.3, the Sale class was declared as a case class. This brings two advantages: all arguments in the parameter list of a case class are maintained as immutable fields, being accessed directly by its name; the second advantage is that allows for variables in case expressions to be bound directly to the field values. An example of such capabilities is shown in the program 5.15.

Program 5.15: Examples of case classes advantages

```scala
def fn1 = (x: Sale) => x.quantity * x.price // since x is of type
  Sale, its parameters can be easily accessed by their declared
  names. However, these cannot be modified.

def fn2 = { case Sale(_, qtd, pr) => qtd * pr } // the variables qtd
  and pr are bound to the values of quantity and price, respectively.
  In this context, the '_' means that we have no interest in that
  value, not the underscore used in placeholder syntax.
```
5.3.2 RTBScript Language examples

Now that we can properly understand how to declare function literals in the Scala language, we can move forward to the RTBScript language itself. In this section we will show several examples of scripts using both discrete and continuous signal sources. The definitions for the library functions that we use in the examples can be found in the appendixes of this work, so that nothing is missing. As a side note, for consistency, we will be using the Scala representation of function types instead of the one introduced in section 5.2. The only difference will be the enclosing () to indicate the input parameters: e.g., \( a \Rightarrow b \) becomes \((a) \Rightarrow b\), and \( a, b \Rightarrow c \) becomes \((a, b) \Rightarrow c\); in case of tuples, \((a, b) \Rightarrow c\) becomes \(((a, b)) \Rightarrow c\).

We start by picking up the Sale example that has been presented so far. As seen in section 5.2.2 there is a corresponding discrete signal source, SaleSignalSource. In program 5.16 we remember the definition of these.

Program 5.16: Remembering the Sale example definitions

```scala
case class Sale(productId: Double, quantity: Double, price: Double);

SaleSignalSource // of type SS[Event[Sale]] = SF[Nothing, Event[Sale]]
```

One expected use of such stream would be to know the total value of items sold so far. Knowing how many sales already occurred it is also not hard to imagine:

```scala
1 // SaleTotalValue: SF[Event[Sale], Event[Double]]
2 val SaleTotalValue =
3  arr( mapE( (sale: Sale) => sale.quantity * sale.price ) )
4
5 // SalesAccumulatedValue: SF[Event[Sale], Double]
6 val SalesAccumulatedValue =
7  SaleTotalValue >>> SFaccum
8
9 // TotalNumberSales: SF[Event[Sale], Int]
10 val TotalNumberSales = SFcountE
```

In this example we can see the use of the library function `mapE`, which only applies a function to the value of an event, and creates an event whose value is the one given as output of that function. From the definition of Sale, we can see that it gives information relative to how many items of determined product were sold, at a given unitary price. As such, to calculate the value of each sale, it is just a matter of multiplying the quantity for the price. This is what the signal function `SaleTotalValue` computes. However, this gives only the total relative to each sale: to find the total value of all sales, it is needed that these subtotals are accumulated, which is done through SFaccum. SFaccum is a library function that given discrete events carrying a number, accumulates all the values of the discrete occurrences. Note that SFaccum transforms a discrete event source into a continuous one. As of `TotalNumberSales`, its definition is even easier: since counting events is common, there is already a library signal function that counts any kind of events, SFcountE.

Now imagine that instead of the total number of sales, we want to find how many items were sold:

```scala
11 // SaleTotalUnits: SF[Event[Sale], Double]
12 val SaleTotalUnits =
13  arr( mapE( (sale: Sale) => sale.quantity ) ) >>> SFaccum
```

50
Again, the use of \texttt{SFaccum} suffices. But so far, we have not combined any of the information we were able to get from the use of the signal functions. As an example, let us try to find the average number of sold units by each sale:

```scala
// AverageUnitsPerSale: SF[Event[Sale], Double]
val AverageUnitsPerSale = (SaleTotalUnits &&& TotalNumberSales) >>> arr( {case (totalUnits, totalSales) => totalUnits / totalSales } )
```

Now we have all the signal functions we need to get the information we were looking for. However, if the script would be run as is, it would produce no results! Although all the signal functions are defined, none has been connected to a \textit{signal source}. Since signal sources introduce the different values to the different signal functions, all the signal functions that are not connected to a signal source contain the information of the dataflow, but no data is passing through it. As such, its only a matter of connecting the defined signal functions to the respective signal sources, and everything is set in motion:

```scala
// runningAverageUnitsPerSale: SF[Nothing, Double]
val runningAverageUnitsPerSale = SaleSignalSource >>> AverageUnitsPerSale
```

Let us now explore another language construct: \texttt{on}. Assume that the business manager of certain company wants to be warned when the value of all the sales reaches 10000. In this case we could write:

```scala
// SalesReached10000: SF[Event[Sale], Bool]
SalesReached10000 = SalesAccumulatedValue >>> arr( (x) => x > 10000 )
```

The next step seems to be

```scala
on(SalesReached10000) {
  // will fail
  // do something
}
```

however this would not work. On one hand, the language construct \texttt{on} actuates when \textit{discrete} events are triggered, and \texttt{SalesReached10000} is of type \texttt{SF[Event[Sale],Bool]}, that is, represents a continuous \texttt{Bool} value. This would imply that “something” would be run at sampling rate, which is conceptually wrong, since this would make it implementation dependent. On the other hand, since \texttt{SalesReached10000} is only a signal function, that was not connected to no signal source, this dataflow would have no values. The correct definition would be:

```scala
// SalesReached10000: SF[Event[Sale], Event[Bool]]
val SalesReached10000 = SalesAccumulatedValue >>>
  arr( (x) => x > 10000 ) >>>
  SFchanges >>>
  SFfilterE(_ == True)

on(SaleSignalSource >>> SalesReached10000) {
```
The use of SFchanges is as described in section 5.2.3. However, this signal function would detect both transitions from False to True and the reverse. Since we only want to know when the condition becomes True, we should filter out all the other transitions. If we assume that there are no sales of negative value, the SalesAccumulatedValue would never decrease, so a transition from True to False would never actually occur, but conceptually this is the correct way.

We can also see that this time the on construct will behave as expected, since there is a signal source composed to the rest of the dataflow, and SalesReached10000 generates discrete occurrences. As a note, Messenger does not belong to RTBScript, and is just an example of an actor for doing something specific: send emails. Another possible action to take in the on construct would be to interact with the database of the RTBPlatform (e.g., writing a new event in the database), as brought out in section 5.1.1.

In a final note before moving forward to other example, we highlight that we could define any other limit for the sales value to reach. We could also have defined SalesReached10000 as

```scala
val SalesReached10000 =
  (SalesAccumulatedValue &&& const(10000)) >>>
  arr( { case (totalValue, limit) => totalValue > limit } ) >>> ...
```

or, using some signal source to indicate the limit (e.g., SalesLimit), even as

```scala
val SalesReachedLimit =
  (SalesAccumulatedValue &&& SalesLimit) >>>
  arr( { case (totalValue, limit) => totalValue > limit } ) >>> ...
```

Now that we have seen a few examples regarding discrete signal sources, we will introduce an example application for conceptually continuous signal sources. First, consider the MachineStatus type presented in program 5.17.

Program 5.17: Definition of the MachineStatus type and MachineStatusSignalSource

```scala
case class MachineStatus(cpuUsage: Double, totalMemory: Double, memoryUsed: Double);

```

MachineStatus corresponds to a common set of properties usually found in monitoring systems: the CPU and memory usage of a machine. In this case, MachineStatusSignalSource is a conceptually continuous signal source. From examination of the MachineStatus class we can assume that the CPU usage already comes in a percentage form (that is, belonging to \([0, 1]\)), but this is not so with the memory related properties. Let's start by defining signal functions that give the MemoryUsage and CPUUsage separately:

```scala
// MemoryUsage: SF[MachineStatus, Double]
val MemoryUsage =
  arr( (ms: MachineStatus) => ms.memoryUsed / ms.totalMemory )
```
Imagine that the system administrator would like to be warned when the machine had its CPU usage higher than 90% for more than one hour or when the memory available was less than 20% for more than 30 minutes. As we can see, there is a clear notion of time involved in such requirements. To produce such signal functions we will use the already shown combinator `TIarr`:

```scala
// CPUUsage: SF[MachineStatus,Double]
val CPUUsage =
  arr( (ms: MachineStatus) => ms.cpuUsage )
```

```scala
// Imagine that the system administrator would like to be warned when the machine had its CPU usage higher than 90% for more than one hour or when the memory available was less than 20% for more than 30 minutes. As we can see, there is a clear notion of time involved in such requirements. To produce such signal functions we will use the already shown combinator TIarr:

```scala
def cpuHowLong =
  (usage : Double , dt: DTime , howLong : DTime ) => {
    val tmp = dt + howLong ;
    if( usage >0.9)
      (tmp , tmp )
    else
      (0,0)
  }

// memoryHowLong: (Double, DTime, DTime) => (DTime, DTime)
def memoryHowLong =
  (usage : Double , dt: DTime , howLong : DTime ) => {
    val tmp = dt + howLong ;
    if((1 - usage ) < 0.2)
      (tmp , tmp )
    else
      (0,0)
  }

// cpuAlarmCondition: SF[Double,Bool]
val cpuAlarmCondition =
  TIarr( cpuHowLong ) >>> arr( x => x > 3600)

// memoryAlarmCondition: SF[Double,Bool]
val memoryAlarmCondition =
  TIarr( memoryHowLong ) >>> arr( x => x > 1800)

// areLimitsReached: SF[(Double, Double),Bool]
val areLimitsReached =
  (cpuAlarmCondition *** memoryAlarmCondition) >>>
arr( { case (cpuReached, memReached) => cpuReached || memReached } )

// usages: SF[MachineStatus,(Double, Double)]
val usages = CPUUsage &&& MemoryUsage

// allTogether: SF[Nothing,Event[Bool]] (or SS[Event[Bool]])
val allTogether = MachineStatusSignalSource >>>
  usages >>>
  areLimitsReached >>>
  SFchanges >>>
  SFfilterE(_ == True)

on(allTogether) {
  Messenger ! SendEmail("Check the machine...")
}

We can see that we have to use SFchanges >>> SFfilterE(_ == True) in order to generate the discrete occurrences that trigger on. If instead of on, we used onceOn, the body (in this case, Messenger ! SendEmail(...)) would only be executed in the first generated event, no matter how many events would be generated later.

In all this section we have seen several examples of different applications of the RTBScript Language. We have seen both discrete and continuous signal sources, and how to interact with these. We have also shown different signal functions that allow to change between both conceptual discrete and continuous domains. With all this expressive power and the provided capabilities of the RTB Platform, we have seen examples of the application of RTBScript to different domains, ranging from a sales management example to a monitoring system example.

5.4 RTBScript Runtime Optimizations

Throughout this chapter we have discussed the implementation of RTBScript using actors. Since each actor consumes resources, it would be desirable that only the minimal set needed to process the different dataflows would be active during the execution of the RTBScript. As such, runtime optimizations to the dataflow would be desired, so that the system would, automatically, adjust itself for better performance. However, as we will see in the following cases, the optimization options are either non desirable or even impossible to implement due to different reasons.

- arr(f) >>> arr(g):
  One of the most common cases in RTBScript is the composition of lifted functions. As we have seen in section 5.2.2, to each of this lifted function, a new actor is created. One could argue that instead of creating 2 actors in this case, only one could be created with the definition of arr(h) such that h(x)= g(f(x)). Although this optimization is even capable of merge n composed signal function into only one (provided that all signal functions are arr), an important problem poses: since each lifted function can do blocking calls, when separated, these calls only affect a small portion of the computation, allowing the others to continue computing their values; if these functions were
all composed and then lifted, any blocking call would block the entire composition. As such, this optimization is intentionally left out, even because the used actors in the implementation are the lightweight version of the Scala actors.

- **sf >>> const(c)= const(c) for some sf: SF[a,b]:**

Due to the mathematical foundations of AFRP, it would be expected that the identity sf >>> const(c)= const(c) would also hold true for RTBScript, and would be possible to simply replace the entire dataflow of actors by the corresponding actor of the constant. However, this is not true in the reality of RTBScript. Since AFRP has been developed in purely functional environments, side effects were not possible in the lifted versions of functions, and the identity would hold true in these environments. However, in an environment with side effects this is no longer true. Considering the example of Tlarr(f)>>> arr(g)>>> const(c)>>> arr(h) it is easy to see that, although the output of arr(g) will be ignored due to const(c), the semantics of arr(g), as presented in section 5.2.2, consider the changes in the input signal and, as such, when connected to other signal function, should evaluate g to produce an output value. This evaluation, if g contains side-effects, cannot be ignored. Since, at the time, there is no way to know if g contains side-effects or not (as we would easily drift into the halting problem), no optimization can be done to such dataflow.

- **first(f)>> second(g)= second(g)>> first(f)= f *** g:**

Another identity that would be expected to hold true are those regarding the first, second and *** combinators due to their obvious relation, easily seen in figure 5.6. However, again due to the possible side effects of the evaluation of f or g, these identities do not hold, since the order of evaluation may prove important when processing each signal function. In the case first(f)>> second(g) it is guaranteed that f is evaluated first than g, while in second(g)>> first(f), it is exactly the opposite. However, and due to the asynchronous nature of the actors, no guarantee of order can be provided in the case of f *** g. One could argue that we might define f *** g as first(f)>> second(g), but this would impose a sequential evaluation, not taking advantage of the inherently concurrent model of actors. Even so, this optimization would only reduce the involved actors from 2 to 1.

- **const(c)>> const(d)= const(d):**

This easy to notice identity is the only one that can effectively be implemented in the RTBScript since it does not rely on evaluation of functions, but only on the values of c and d. However, this seems pretty uncommon for a user to actually write due to the well defined semantics of const and >>>. Yet again, a “chain” of n actors in this case could be reduced to only one, but these actors are extremely simple, so that the actual gain would not be perceived.

We finish this chapter with a reference to the work by Hughes regarding Arrows [Hug00]: in it, the author discusses the different composition of combinators rather thoroughly, and states that, in the presence of side-effects, most of the “Arrow Laws” will not hold. Since RTBScript’s host language allows for side effects, without other methods that guarantee for absence of side effects in the functions, no relevant optimizations can be performed.
Chapter 6

Results

In this chapter we will present and discuss the developed work, first by comparing the concluded work with the different goals of chapter 2, and second by comparing our solution to other related work.

6.1 Goals achievement

In chapter 2 we have defined a set of goals so that we could pursue an appropriate solution that would meet those same goals. In this section we present the features of RTBScript that provide for each of those goals.

Ease of use

As shown in section 5.2.3, we provide the RTBScript language with a library that eases the development of the different signal functions. These library functions prevents much of the “boilerplate” code that the user would have to write. Moreover, due to Scala higher-order functions, the library can easily be extended by the user. Nevertheless, further developments can easily be included later in the RTBScript library, if that seems appropriate.

Expressiveness

In section 5.2.1 we have seen the different combinators provided for the RTBScript. In the same section we have also seen that there is at least one minimal set of combinators that are sufficient to, through combination, define the others. With this minimal set, it is proved that it is possible to create any kind of dataflow. Since we provide more than the minimal set of combinators we guarantee that expressiveness is not impaired.

Modularity

Through the use of the different combinators, RTBScript is able to easily compose already defined signal functions into more complex ones. Since the implementation prevents “misdirecting” the different dataflows, even if signal functions are reused (as shown in 5.2.2), larger programs are easily composed from smaller ones, each of which is itself a reactive program.

Continuity and Event Modeling

In RTBScript, we provide means for defining both discrete and continuous sources of signals, along with the combinators to deal with such sources. We have also shown in section 5.2.3 how to change between representations, using changes and hold.

Causality

All the defined combinators in RTBScript are causal, that is, a signal function result at instant \( t \)
can be uniquely determined by the input signal on the interval \([0, t]\). The combinator \(\text{Tiarr}\) is used in such cases. However, we have also seen the case where the output is only determined from the input sample at time \(t\), with \(\text{arr}\), or, when output is not even dependent of the sample, with \(\text{const}\). We also guarantee that any composition of combinators is still causal, so that this goal is always met.

**Incremental Program Development**

Although we have not explicitly addressed this feature throughout this work, the reason is simple enough: the Scala language already supports an interactive console in a Read-Eval-Print loop. Due to its library nature in Scala, RTBScript is easily imported to such console and, since we use actors in our implementation, the reactive component of RTBScript continues to operate in the background, allowing the user to continue interacting with the console.

### 6.2 Comparison with other work

In this section we compare RTBScript with the related work described in chapter 3.

#### 6.2.1 Fran

Fran [EH97] was the first to introduce the FRP concept. It introduced implicit treatment of time, and allowed to model both continuous and discrete domains. Just like Fran, RTBScript also treats time implicitly in its engine. One of the advantages of RTBScript over Fran is that it won’t allow for time-space leaks, a feature that we borrowed from Arrows. Another advantage of RTBScript over Fran is that, at runtime, RTBScript is able to identify signal functions that only react to changes in the input and, as a result, it retains enough information for runtime optimization. However, as we have seen earlier, these optimizations are limited due to the possibility of side-effects. Nevertheless, this runtime information might prove useful in further developments (please refer to the chapter of Future Work for an example of such developments). A feature that exists only in Fran is the detection of predicates (described in section 3.1), through the use of interval analysis.

#### 6.2.2 FrTime

One of the goals of FrTime [CK06] was the incremental development of programs in a common Read-Eval-Print loop evaluation scheme. As we have seen earlier, RTBScript can also be used in such way. However, the expressiveness of RTBScript surpasses that of FrTime: since FrTime uses a graph evaluation through changes in the input sources, it is unable to represent stateful signal functions. For example, while RTBScript is able to represent, e.g., \(\int t\, dt\), FrTime is not.

#### 6.2.3 Frappé

Frappé [Cou01] is a FRP implementation in Java, a mainstream programming language. As stated earlier, RTBScript also tries to bring the concept of FRP closer to these mainstream languages. However, RTBScript, unlike Frappé, also provides type safety, a typical feature in several FRP implementations. Also, since Frappé uses an evaluation model based on changes, as FrTime, it suffers from the same lack of expressiveness.

However, type safety and greater expressiveness are not the only advantages of RTBScript over Frappé: RTBScript is also much less verbose than Frappé.

Please consider the example in 6.1. In it, several things are happening:
Program 6.1: A Frappé code example

```java
// Only the type is important:
Drawable ball = ...;

Behavior mouse = FRPUtilities.makeBehavior(sched, frame, "mouse");

Behavior anim =
    FRPUtilities.liftMethod(
        sched,
        new ConstB(ball),
        "move",
        new Behavior[] { mouse });
```

- Line 4 binds the variable `mouse` to a behavior that, when sampled, will return the value of `frame.getMouse()` at the time of sampling. The property name is passed as a string to `makeBehavior()` because the implementation uses Java reflection on the class instance to look up the appropriate accessor method and to register for notification when the property changes.

- Lines 6 to 12 show the use of `FRPUtilities.liftMethod()`, a static method for lifting. The second argument to `liftMethod()` is a Behavior whose sample values implement the given method. In this example, we pass `new ConstB(ball)` as the second argument. This is a new constant Behavior whose value at every sample point is `ball`. Since `ball` is an instance of `Drawable`, it supports `Drawable`’s `move()` method.

By using `liftMethod`, the `move()` method is lifted to operate on Behaviors rather than static values: the method is applied pointwise to the values of the target instance Behavior and argument Behaviors at every sample time. In this case, the target instance is a constant Behavior, and the argument is the Behavior variable `mouse` whose value at every sample point is a `Point2D`. The result will be a Behavior whose value at every sample time is a `Drawable` that renders a ball moved to the current mouse position.

- Lines 4 to 8 show a global scheduling context, `sched`, used by the implementation. The programmer obtains such a context once during initialization, and simply passes it to all constructors of Frappé classes.

The corresponding RTBScript example is shown in program 6.2.

Program 6.2: The equivalent RTBScript code

```rtb
/* From the Frappé example, we assume that the value ball, the function move and the signal source mouse with equivalent meanings are available: */
ball: Drawable
mouse: SS[Point2D] = SF[Nothing,Point2D]
move: ((Drawable, Point2D)) => Drawable
*/

//anim: SF[(Drawable, Point2D),Drawable]
val anim = (const(ball) *** mouse) >>> arr(move)
```
From the shown examples, we can see that RTBScript allows for more concise code: we can see that `new ConstB(ball)` is replaced by `const(ball)` and `FRPUtilities.liftMethod(..., "move", ...)` by `arr(move)`. Also, Frappé exposes part of his implementation, making a sort of scheduler `sched` as a mandatory argument to every constructor of Frappé classes. In RTBScript, all the implementation is hidden from the user.

Because of these issues, RTBScript presents itself as a good alternative to Frappé, bringing the FRP concept closer to the Java mainstream language.

6.2.4 Arrowized Functional Reactive Programming

RTBScript has many of its foundations in AFRP [NCP02]. As such, there are several similarities between both works: the set of provided combinators, the prevention of time-space leaks, and the inability to have collections of signals (a feature that Fran was able to provide). However, in [NCP02], although there were stateful signal functions, it would not be possible to define new stateful signal functions (except using compositions of the two provided, `integral` and `derivative`). In RTBScript we introduce a “lifter” for such use: `TIarr`, already presented in chapter 5.
Chapter 7

Future Work

The developed solution sets a working basis to extract the data from the RTB Platform, and to express different rules that can lead to automatic triggering of actions when the conditions are met. Due to its high-level characteristics and to the flexibility introduced by the RTB Platform itself, RTBScript can be used for a variety of domains. Nonetheless, there is always room for improvement. In the next sections we present some possible extensions of functionalities and optimizations.

7.1 Multiple RTB Data Collectors

In figure 7.1 we present a possible architecture for the RTB Platform. In this case, we can see that, instead of just one RTB Data Collector, there are several. This scheme avoids single points of failure regarding the Data Collector and avoids overloading it, as more and more data sources come into play.

As such, RTB Platform becomes a distributed architecture, introducing problems of their own. One common problem of these architectures is the ordering of information delivery, as no guarantees are usually given and reordering may happen, that is, an older event can reach the RTBScript Server after a newer one). However, as we have already seen, Event α represents a time-ordered sequence of values.

Another common problem in these architectures is the asynchrony of event sources, as they usually do not have information about each other and, as such, each one believes in its own time: e.g., source 1 believes now is 2:09, while, at the same time, source 2 believes it is 2:00. Notice that this is different from the reordering problem as information is delivered in the right order considering each source individually; only when the sources are interleaved, the problem arises: an event that occurred at 2:05 at source 1 would appear to happen after one that occurred at 2:00 at source 2, though, in reality, this is false.

Currently, RTBScript does not address these problems. However, some form of guarantee for total ordering in message delivery [CDK05] from Data Collectors to RTBScript Engine might be enough. In this scheme, however, it might be needed for Data Collectors to communicate among themselves, which is not currently supported by the RTB Platform. Other developments could be done by determining to which extent the precise ordering of information can be ignored in the domain it aims to, as many domains do not require it.

7.2 Distributed Architecture

As seen in chapter 5, RTBScript is a single-server component in the RTB Platform, with a central engine. In this case, a failure of this component does not affect the entire platform, but it is no longer able to provide the services it is intended to. Even more, as different sources can grow, so will the load in RTBScript’s Engine grow, and the usual performance problems may arise.
In its core, RTBScript uses the Actors model to process the different dataflows as intended. As there is no shared state, these actors, as we have seen in section 3.2, can work concurrently. But even better, they can work “separated” from each other. From [HO08], we can see that “message sends are location transparent and actors can be spawned on remote nodes”. In its work, the authors already present two working prototype implementations\(^1\) of the service layer based on TCP and JXTA\(^2\) peer-to-peer framework.

As such, work can be done in an effort to make the RTBScript Engine a truly distributed system. However, other problems may arise, such as how to divide the dataflows by different servers. Also, this distributed architecture with multiple RTB Data collectors scheme presented previously may bring problems into play. Surely, more research can be done in this.

7.3 Further expansion of the RTBScript Library

In the Yampa [HCNP02] implementation\(^3\) of AFRP, the authors provide a rich set of functions allowing to easily interact with the developed implementation. As examples, these functions include easier ways

---
\(^{1}\)The current Scala Actors implementation can be found at http://www.scala-lang.org/, as it is now integral part of the language common libraries

\(^{2}\)See http://www.jxta.org/

\(^{3}\)Available at http://www.haskell.org/yampa/
to provide event filtering or mapping functions to events.
In this project, although we already provide some library constructs, we feel that there is room for improvements in the set of library items provided.

7.4 Automatic XML model processing

As we saw in section 5.1.3, the RTB Platform uses an XML schema to introduce flexibility in the definition of different events and the grouping of those events. We have also seen that, due to the requirement for type-safety and to the crude representation of the events in the RTB Platform, it was needed to implement some mechanism of translation from the cruder representation to instances of the corresponding types. We have also seen some examples in section 5.1.3 of these translators.

Although users do not realise the existence of these translators and are not directly affected by them in the language elements, the developers cannot say the same.

The definition of the XML model for each RTB Application is something that, usually, happens once in the lifetime of the application, although it can be updated later. As such, after the definition of the model, the developers can easily write the corresponding translators. However, as easy as it is, it is still error prone since each translator is relatively similar to each other, and “copy-paste” errors are easy to come by in development.

However, this similarity in the translators can be used to our advantage: a not so complex program can be written that could process the existing model and, through it, generate the corresponding translators and dispatchers (of section 5.1.3).

Currently, no such program was developed, as it was not the focus of this work, but it could easily be implemented through XML parsing, generating Scala code.

7.5 Syntactic Sugaring

Consider the following simple example: imagine that we want to calculate, for some reason, the 10% of the average value of each sale. Consider that we have a signal source for all the sales that are performed, named SaleSignalSource. In this signal source, each event regarding a sale contains information of what product was sold, the quantity of the product sold, and the unitary product price (as we have seen earlier in chapter 5). Also consider the existing Signal Function SalesAccumulatedValue of type SF[Event[Sale],Double] and TotalNumberSales of type SF[Event[Sale],Int] (again, as described in chapter 5).

To achieve the desired value, the script in 7.1 would fulfill the requirement.

Program 7.1: RTBScript to find the one tenth of the average value of each sale

```
val tenPercent = SaleSignalSource >>>
(SalesAccumulatedValue &&& TotalNumberSales) >>>
arr( { case (total, numberSales) => (total / numberSales) } ) >>>
arr( _ * 0.1 )
```

We can see that most of the complexity in this script is due to the need to wire signal functions together using the various pairing/unpairing combinators such as &&& and *** and due to the repeated composition. Precisely to address this problem, Paterson in its work, A new notation for Arrows [Pat01], has suggested the use of special syntax to make arrow programming more readable. Using this special arrow syntax, the above RTBScript code could be rewritten as presented in program 7.2.

It is worth noticing some things to note about the structure of this code:
Program 7.2: Equivalent code in the suggested special syntax in *A new notation for Arrows*

```haskell
tenPercent = proc SaleSignalSource -> do
  total <- SalesAccumulatedValue -< SaleSignalSource
  numberSales <- TotalNumberSales -< SaleSignalSource
  returnA -< (total / numberSales) * 0.1
```

- Due to its ties to the Haskell community, the syntax `proc pat -> ...` is analogous to a Haskell lambda expression of the form `\ pat -> ...`, except that it defines a signal function rather than a normal Haskell function;

- In the syntax `pat <- SFexpr -< expr`, the expression `SFexpr` must be a signal function of type, e.g., `SF[TIn,TOut]`, in which case `expr` must have type `TIn` and `pat` must have type `TOut`;

- The overall syntax:
  ```haskell
  proc pat -> do
    pat_1 <- SFexpr_1 -< expr_1
    pat_2 <- SFexpr_2 -< expr_2
    ...
    returnA -< expr
  ```
  defines a signal function. If `pat` has type `TIn` and `expr` has type `TOut`, then the type of the signal function is `SF[TIn,TOut]`. In addition, any variable bound by one of the patterns `pat_i` can only be used in the expression `expr` or in an expression `expr_j` where `j > i`. In particular, it cannot be used in any of the signal function expressions `SFexpr_i`.

An uncommon feature of this syntax is that it seems “reversed” (in the sense that the flow goes from the right to the left), but this is usual in the Haskell community. Nevertheless, it is still a form of syntactic sugaring worth considering. In figure 7.2(a) we show a diagram for the sugared definition of `tenPercent`. Figure 7.2(b) shows the same diagram but overlaid with the combinators implied by the unsugared definition of `tenPercent` (for clarity, the lifting via `arr` of the primitive functions – i.e. those drawn with circles – is omitted).

In its work, Paterson contributed to ease the use of Arrows by writing a preprocessor that converts the syntactic sugar into conventional Haskell code. Although it cannot be directly applied to RTBScript due to the obvious Scala and Haskell differences, it could provide some insight to further develop the existing syntax of RTBScript and ease even more the development of scripts. However, the use of a preprocessor has its disadvantages: the preprocessor may create generally obfuscated source code, which decreases traceability to the original typed code. Also, when debugging or profiling code, it is harder, if not sometimes impossible⁴, to match it to the original, making the task much more difficult. Further investigation would be needed regarding this subject.

### 7.6 Optimizations through annotations

As we have seen in chapter 5, due to the possible existence of side-effects in the lifted functions, the runtime optimizations are no longer possible as the results may be different of those expected.

Nevertheless, the introduction of runtime optimizations could benefit the performance of the overall system. One possible solution could be the use of *annotations* (not only available in the Java language, ⁴As a extreme example, the C preprocessor is often used in *The International Obfuscated C Code Contest*
but also in the Scala language) to identify those functions that guarantee no side-effects. Based on these markings, a runtime optimization system could be implemented.

However, further investigation is needed regarding this subject. In an environment where the user takes responsibility in the optimization process, without being guaranteed that its markings are correct, the optimization process could lead, again, to the already referred problems.

Figure 7.2: Signal flow diagrams for tenPercent
Chapter 8

Conclusions

Many domains, such as communication networks or automated manufacturing systems, are sufficiently expressed by events, discrete occurrences happening through time. Several systems were developed to aid the collection of these events and trigger automatic notifications when some kind of condition was met and, along with these systems, several Event Subscription Languages were developed, each one introducing different operators of its own. However, many domains, namely relating to physical ones (e.g., dealing with positions and velocities), are also conceptually continuous. Functional Reactive Programming is a high-level programming paradigm that abstracts both concepts and is well suited to model these hybrid domains.

On the other hand, many of these application domains are reactive rather than transformational: the input to a reactive system is not known in advance, but arrives continuously as the program executes. In such a reactive system is expected that inputs are interleaved with outputs, computing outputs in response to the input stimuli as they arrive.

To implement such reactive systems, two elements are fundamental: the system that allows the collection of the input and the synchronous dataflow programming language. In this work we have shown one such language, RTBScript, in the context of an enterprise system providing for the collection of input stimuli, the RTB Platform. This platform provides for the collection of samples from different distributed systems, even if using different technologies. It has also other capabilities, like the generation of standardized reports based on the collected samples.

A common approach to implement synchronous dataflow programming languages is to provide a small set of primitive processing elements and a set of composition operators. In such languages, the programs are formed by using the combinator primitives to compose processing elements into a conceptual directed graph structure. We have presented the different combinators available in the RTBScript, inspired in AFRP, and shown that these provide for a minimal set for all the possible combinations. We have also shown how the dataflow programming model provides a natural form of modularity for many applications, since larger programs are composed from smaller components, each of which is itself a reactive program.

In this work we have detailed the implementation of RTBScript, having the same premisses in mind and several design goals what would allow RTBScript, in interaction with the RTBPlatform, to be used in a wide range of hybrid domains. Using Actors, a mathematical model for concurrent computation, we have also approached the dataflow language implementation in a novel way. Although this mathematical model is not used to enforce types, we have shown how to provide the expected type safety of signal functions, using channels. From an operational perspective, we have also described some optimizations to the message passing between actors. However, and since the host language of RTBScript allows side effects, we demonstrated that most runtime optimizations of the dataflow are not possible.

We have laid out some examples of domains that used both continuous and discrete representations,
using the combinators to model different conditions of interest to a potential user.

At last, we have detailed some possible enhancements, with special focus in the distributed architecture of the RTB Platform, and also in the possibility of distributing the RTBScript implementation itself. Another issue to develop further is the introduction of syntactic sugaring.

With the RTBScript language we provide an abstraction to develop complex applications, bringing FRP closer to mainstream languages without compromising conciseness of the code, its expressiveness or type safety. Using a novel approach based on lightweight actors and the underlying concurrent processing model, RTBScript is also adequate to the enterprise needs for performance.
Bibliography


Appendices

In these appendices we can find the definition in Scala of the different helper functions that were used in the section 5.3.

```scala
/*
Higher order library function
Avoids dealing with Events representation
mapE[TIn, TOut] : ((TIn) => TOut) => (Event[TIn] => Event[TOut])
*/
def mapE[TIn, TOut] = 
  (fn: TIn => TOut) => {
    (x: Event[TIn]) => 
    x match {
      case SomeEvent(y) => SomeEvent(fn(y))
      case NoEvent => NoEvent
    }
  }

/*
Higher-order library function
Receives a function that given the current sample, and the accumulated value, returns the new accumulated value
Avoids dealing with Events representation
accum[TIn, TAccum] : ((TIn, TAccum) => TOut) => (((Event[TIn], TAccum)) => (Event[TIn], TAccum))
*/
def accum[TIn, TAccum] = 
  (fn: (TIn, TAccum) => TAccum) => 
  (x:(Event[TIn], TAccum)) => 
  x match {
    case (SomeEvent(y), accum) => (SomeEvent(y), fn(y, accum))
    case (NoEvent, accum) => (NoEvent, accum)
  }

/*
Given discrete occurrences with some Double value, accumulate them.
SFaccum: SF[Event[Double], Double]
*/
def SFaccum =
```
```scala
loop(
    arr( accum[Double, Double]( (newValue, accum) => accum + newValue ) ) >>>
    arr( (x: (TIn, TAccum)) => (x._2, x._2) ),
    0.0)

/*
Given discrete occurrences of any type, counts the number of them.
SFaccum: SF[Event[Any], Int]
*/
def SFcountE =
loop(
    arr( accum[Any, Int]( (newValue, accum) => accum + 1 ) ) >>>
    arr( (x: (TIn, TAccum)) => (x._2, x._2) ),
    0)
```
RTBScript: A High-Level Language for Hybrid Domain Modeling using Actors

Filipe Miguel Duarte Cristóvão

Dissertação para obtenção do Grau de Mestre em Engenharia Informática e de Computadores

Outubro 2009