Systems Engineering

Colombian School of Engineering Julio Garavito

On the implementation of a distributed real-time reactive programming language

Architecture and operation of a reactive middleware for processing large volumes of data (Big Data) and its application on a message communication environment.

Daniel Eduardo Beltrán Franco

Bachelor's Thesis, Spring 2019

Abstract

Reactive applications demand detection of the changes that occur in a domain of interest and for timely reactions, examples rang from simple interactive applications to complex monitoring tasks involving distributed and heterogeneous systems. Over the last years, different programming paradigms and solutions have been proposed to support such applications. Reactive Programming has been the paradigm provided for long time solutions to the organizations in the Internet Era where data is produced in almost any machine, connection, application, etc. Taking into account that currently web applications require much more dynamism, it is vital that they update without using external or internal dependencies. It is expected then, that a system interacts with its environment, interspersing inputs and outputs temporarily, which means that within an adequate amount of time (depending on the application) the system returns a response depending on the received inputs and continues with the cycle of execution, that is, being an asynchronous client. Event-based programming (specifically Complex Event Processing) systems enable the definition of high level situations of interest from low level primitive events detected in the environment to big data production and consumption. In this paper, we focus on showing a distributed reactive focused language adopting reactive principles like: message-driven architecture, responsiveness, elasticity and resiliency. We also present the reactive programming state of the art and the form the reactive language supports time-changing values and their composition as dedicated language abstractions with recognition of certain patterns that behave accordingly to a given context.

The state of the art exposes the investigations by different communities, belonging to the databases, distributed systems, and programming languages areas. It is our belief that a deeper understanding of these research fields including their benefits and limitations their similarities and differences, could drive further developments in supporting reactive applications. Despite some differences between the implementations, we believe that such comparison can trigger an interesting discussion across the communities, favor knowledge sharing, and let new ideas emerge.

Contents

List of Figures

List of Tables

List of Codes

CHAPTER 1

Introduction

Internet and hardware have been evolving at a very fast phase bringing new advantages and new challenges to developers, costumers, and enterprises. As technology becomes more complicated, it is necessary to have more elaborated organizations since the organizational schema has been growing into a hierarchical one and the quantity of internet users have been growing on and on; it is important to deliver proper services that satisfy the needs of today's costumers. The applications should be more responsive, deliver a smooth experience without freezing and do not provide a poor performance to the users. Hence, the fact that developers as well as companies deal with contemporary investigations, on one hand to handle network traffic (especially for high peaks of traffic) for both users along with data in motion, and on the other, the constant demand for computing systems that make efficient use of memory and CPU that grants the ability to interact with different processes in different node clusters for the operation of an application.
Modern software is mainly focused on real-time systems, it went from data stored and being queried from time to time to a data change right away in the software. The subjects that we are resolving in nowadays world are simple, to improve user experience and to minimize the effort needed by systems to deliver software.

Stream processing is one way to help turn this data change of all sizes: big, medium, or small into simple event data as quickly as possible. As systems embrace data in motion, traditional architectures are being re-imagined as pure stream-based architectures. In those systems, real-time data is captured, processed, and used to modify its behaviour with response times of seconds or even less. There is a major business value in sub-second response times to changing information, rather than the hours, days or even weeks a system with a traditional batch-based architecture may take to respond and that has already been demonstrated (Joseph and Pandya, 1986). The bigger push and demand for reactive-based services came from the web pages, due to its modern nature where data constantly changes on the client's screen and the allowance of dynamic behavior unlike the traditional passive infrastructure. Specifications such as Reactive Streams, Reactive Systems, and stream processing libraries such as ReactiveX[1] or Akka Streams (Stivan, Peruffo, and Haller, 2015), provide

[1] *ReactiveX:* *http:// reactivex.io*

the standards, principles and patterns necessary to implement such systems effectively.

Nonetheless, there is a lack of languages and frameworks that are explicitly distributed and concurrent with a special focus on an asynchronous model alongside logical semantics which are regularly reinforced to control discrete flows of events and event detection including reaction rules for data representation and action (Paschke and Kozlenkov, 2009) explained under the Complex Event Processing (CEP) model (Luckham, 2002). Besides, the distributed languages are not generic and its support is very scarce, that is, it is software that is developed for a specific use case that limits its usefulness in other application scenarios, in consequence it is a software that is not extensible. On top of that, in massive scale systems whose node topology seem to dynamically change depending on the conditions (nodes joining or leaving at any moment), it is required formal proposals that face with frequent connections and disconnections (De Troyer, Nicolay, and De Meuter, 2018) or even with mutability of the data.

What is proposed in this degree work is an overall review of the elements that composes reactive programming set like: theory, frameworks, APIs and languages. We also examine a concrete chat example showing how the proposed language behaves during an event-based environment. Accurately, we present the following contributions:

- An updated state of the art of reactive programming and functional reactive programming with examples and concise analyses of the presented investigations.
- A reactive oriented event-based programming language with explicit support for distribution and time manipulation.
- A partial implementation of the compiler.
- Programming examples and execution of a chat scenario using the reactive event-based programming language.

1.1 Outline

This document is organized and composed as follows:

Chapter 2 shows the reactive framework used in the Netflix platform, its importance today's industry, expands the concepts from Chapter 1, and explains the IoT common topology.

Chapter 3 collects an amount of frameworks, languages, and different implementations in the development history of the reactive programming paradigm. This section identifies the past work of reactive distributed programming.

Chapter 4 describes the language based on reactive concepts such as: observables, LTL formulae and local networking. Furthermore, it exposes three sets of language approaches: ReactiveX statements, LTL declarations and Causality operators.

Chapter 5 features the syntax and semantics of the grammar language embracing the full infrastructure: parser, linker, typechecker, compiler and others.

Chapter 6 displays the execution of a chat program that exchanges messages between two nodes and analyzes the generated aspect code.

Chapter 7 concludes the paper and presents a future work.

CHAPTER 2

Motivation

2.1 Netflix

Netflix is a subscription service for T.V. and movie streaming over the internet, it has millions of costumers across hundreds of countries streaming thousands of titles over millions of devices across the globe. Just in North America, Netflix makes a third of internet traffic at night.

The API traffic has grown from ∼20 million API requests per day (in 2010) to >2 billion per day and those requests come from at least 800 different type of devices including: smartphones, tablets, smart TVs, PS3s, Wii, Xbox. In the procedure of communicating the devices to Netflix, the devices *'talk'* to a facade that contains the services of Netflix, this API layer also has hundreds of dependencies, basically links to external services. But, as the complexity of the service started to grow, the control over the data dropped off, the quantity of requests made at the same time was also an on-going issue, so the Netflix team investigated different solutions to this trouble, most of these brought implicit problems with it so they asked themselves, *"is there another way to administrate concurrency?" especially when the amount of data volume varies through the day...*

The way the Netflix team answered to this was by rearchitecting the API infrastructure and reinventing the client-server interaction model since a typical device performs at least dozen network calls (each one with a network latency) against the Restful API and each call will do about 4 or more service calls, yet as a company you don't really know: the performance of the client's device, the way the network is connected, the latency, among other things. Therefore, if we take all the network latencies produced from the multiple requests and sum them up, it starts to be a problem; so what it was thought was to collapse those calls in one single request, since the servers have much more power than the clients, the server then will be the one that perform the heavy computation. Soon, the Netflix team started to check different approaches but callbacks, threads and concurrency were part of the general concern. Microsoft's open-source Reactive Extensions library (ReactiveX aka Rx) was the key candidate for this situation.

The Netflix API takes advantage of the Reactive Extensions library by modelling each event as a collection of data rather than a series of callbacks and this modelling makes the entire service layer asynchronous returning an Observable<T> (a flux of data T emitted in a timeline) on all service methods using language implementations (like RxJava) or coding on ReactiveX (later explained in Chapter 3).

Making all return types Observable combined with a functional programming style on the backend, frees up the service layer implementation to safely use concurrency and asynchronous messaging. It also enables the service layer implementation to:

1. Conditionally return data immediately from a cache.
2. Use blocking or non-blocking I/O using more or less resources.
3. Make client code treat all interactions with the API as async.
4. Use multiple threads.
5. Migrate an underlying implementation from network based to in-memory cache.

This can all happen without ever changing how client code interacts with or composes responses. In short, client code treats all interactions with the API as asynchronous but the implementation chooses if something is blocking or non-blocking.

“ *Functional reactive programming with RxJava has enabled Netflix developers to leverage* ***server-side concurrency*** *without the typical thread-safety and synchronization concerns. The API service layer implementation has* ***control*** *over concurrency primitives, which enables us to pursue* ***system performance*** *improvements without fear of breaking client code.* ”

Ben Christensen

Maintaining a control and monitoring in the service layer is an architectural advantage because it optimizes and improves the functionality over time even during difficult connection setups (Maglie, 2016). The results were pretty good for Netflix, latency was reduced by increasing the number of threads used by Observables, CPU consumption reduced, and async processing with constant monitoring was greatly helpful to the organization.

RxJava

RxJava is the Java implementation of ReactiveX for the JVM and is available in the ReactiveX repository in GitHub[1] (prior to September 2014 was in the Netflix repo). The first languages supported (beyond Java itself) were Groovy, Clojure, Scala and JRuby. New language adapters can be created through the contributions of the community. Supports Java 6 (to include Android support) and higher with a target build for Java 8 with its lambda support (Davis, 2019).

[1] *RxJava:* https://github.com/ReactiveX/RxJava

RxJava uses the Observables to control the emission of events with a Java syntax on it. Additionally, it manipulates these events with a broad set of operators, the collection of operators provisioned by ReactiveX considers various categories:

- Creation
- Transformation
- Filter
- Combination
- Error Handling
- Utility
- Conditional and Boolean
- Mathematical and Aggregation
- Backpressure
- Connectable
- Conversion

The primary property of these operators is the operator chaining, which allows us to link these operators one to another as in a chain modifying the observable result of the previous operation. Each operator acts by means of taking an observable as a parameter and return another observable modified by said operator. Also, something to be highlighted is that ReactiveX allows the creation of custom operators expanding the set of operations as needed.

Still, it's useless to generate a stream of data without anyone listening to it, it's like the phrase: *"if a tree falls in a forest and no one is around to hear it, does it make a sound?"* that raises the question about the ineffectiveness of perceiving an event. These data emitting Observables relate to the Observers in consuming the events sent by the Observables as long as the Observers are subscribed to the Observables.

It is interesting to contemplate Observables and Observers respectively as a Producer and Consumer architecture. Since, the category of Backtracking Operators (Backpressure) allows Observers to report to the Observables the rate at which they should be transmitting data; because it may exist the case that an Observable emits data at a higher rate than an Observer can process them causing the internal buffers to fill and show a *OutOfMemory* error.

2.2 IoT and mesh networks

A mesh network is an infrastructure of nodes (each node representing a device) that are connected to each other. The main advantage of mesh topology is multiple paths to the destination node. If one path is down, we have another path to reach the destination.

These nodes transmits data to each other in order to extend the signal to route, relay, and proxy traffic to/from clients. Each node spreads the signal a little further than the last, minimizing the possibility of dead zones as shown in Figure 2.1. Where lowercase nodes ($a, b, c, \ldots, g$) represent devices that create and emit events, and uppercase letters ($A, B, C, \ldots, E$) represent routing nodes that transmit and communicate those events in the interest of reaching the desired destination.

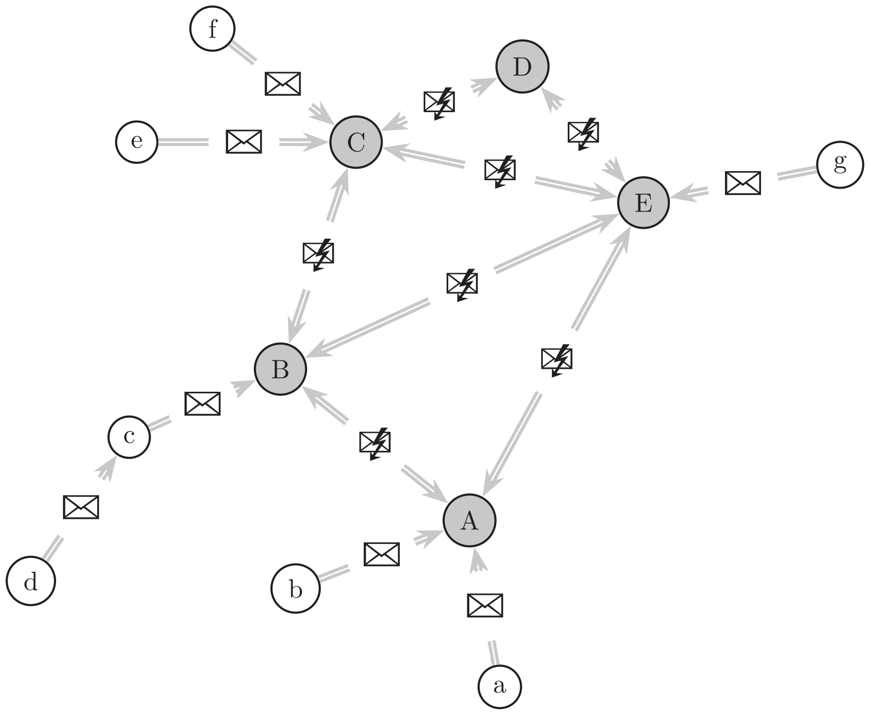


Figure 2.1: Mesh network topology

Traditional IoT devices that use Wi-Fi and cellular connectivity depend on the cloud to relay messages between devices. This works great when you're making a standalone product but sometimes you need more than that (Souryal, Wapf, and Moayeri, 2009). Mesh development kits are not just connected to the Internet, they're gateways to the Internet and create a local wireless mesh that other devices can join.

These devices work together to ensure that messages get where they are going, and power products that are not possible or economically feasible with Wi-Fi or cellular connectivity. Mesh gives every IoT device a local network to understand and connect with the world around it, ensuring products have the information they need at particular time.

Why use mesh networking for IoT?

While wireless mesh networking technologies has been around for some time, recently has the power of mesh reached a point of maturity alongside high availability from chip and silicon vendors. With newer approachable costs, wireless mesh networking has become ideal for IoT builders. And with the rise of connected homes and industry support on open source resources like Thread (network protocol for IPv6), Mesh is now truly accessible while being low-cost enough to scale for production. As such, wireless mesh networking is becoming a much more viable real choice for industrial and commercial IoT applications. It can provide additional services in a system where extending a connection between two nodes is limited.

The following examples are shown in order to understand the implementation of the mesh networking in different day-to-day systems/scenarios, especially IoT ones (Pu, 2011), like:

- **Smart cities—wireless** mesh networking is great for extending radio signals through parking garages, campus grounds, business parks, and other outdoor facilities. Parking garages that utilize space availability checkers benefit greatly from mesh networks because they can extend the signal throughout the whole space, and be able to communicate when a spot has been taken by other clients.
- **Healthcare equipment—wireless** mesh networks can help monitor and locate medical devices quickly. They can also act as a backup for medical equipment that always needs to remain online. If one node loses connectivity, another node can step in to keep the connection alive.
- **Farming—wireless** mesh networking is also great for tracking sun exposure and water levels across your crops. You can scale at a low cost with Mesh-enabled nodes across a whole acreage to create a cellular-connected IoT farm. Nowadays, has a lot of applications.
- **Industrial internet—wireless** mesh networking is also great for tracking pallets and monitoring large physical objects with a highly reliable wireless connectivity network. With wireless mesh networks, you can easily track key data across your factory floor, and across multiple locations to identify issues before they happen.
- **Smart home—wireless** mesh networks can help you track and manage temperatures across your house. Setup one powered gateway and use temperature sensors and Mesh-enabled nodes in each room to capture live data and adjust settings automatically.

CHAPTER 3

State of the art: Reactive programming

There has been a wide variety of implementations at a corporate level (the use of frameworks that generates value to the company) and at a research level (the implementation of new ways to build reliable systems through innovation); in these two cases, it is evident the extension of libraries and the thought-process when developing new multi-purpose frameworks and APIs.

The presence of reactive libraries (such as ReactiveX) impelled the interest in making use of Reactive streams in different programming languages and in different tools that would be adapted to those already used in the market.

3.1 Reactive programming á la ReactiveX

ReactiveX (Rx)

Reactive Xtensions was born around 2010, by the computer scientist Erik Meijer at Microsoft. It proposes a model that consists in Observables that emits data, a set of operators to modify the data, and Observers that consume or *'watch'* the data. However, it does not have explicit support for distribution. Other frameworks, despite of the fact that do not handle the concept of Observables (such as ReactJS, RxAda, etc), they have been inspired by these ideas (Salvaneschi, Margara, and Tamburrelli, 2015) for example Facebook's ReactJS, although it does not use the ReactiveX's syntax, it has the concepts of reactivity, where you can create components that observe other component's changes.

ReactiveX has been implemented in many languages (see Table 3.1) and there are other researches that try to do it in languages like Ada (Mosteo, 2017). There are even researches, though they do not refer properly to the syntax and semantics of a language, they refer to reactive frameworks that have been very popular, for instance in the JavaScript community ReactJS is very popular because it allows you to create components that observe changes in other components, mainly in web pages, and these become more interactive because you can see the changes in the screen of your web application

Equivalences	
Language	Implementation
Java	RxJava
JavaScript	RxJS
C#	Rx.NET
C#(Unity)	UniRx
Scala	RxScala
Clojure	RxClojure
C++	RxCpp
Ruby	Rx.rb
Python	RxPY
Groovy	RxGroovy
JRuby	RxJRuby
Kotlin	RxKotlin

Table 3.1: Languages supported by ReactiveX

In the interest of further explaining this framework, a couple of examples in RxJava are shown with the use of operators:

Example # 1 Here, we are going to simply take a string message and print it.

Observable**.just** creates an Observable that emits n event instances, in this first case only one, the string *"Hi"*.

```
    Observable.just("Hi");
```

Code 3.1: Observable .just operator

Since *Observable* is an object, we can assign it, initialize it and instantiate it. Notice how we specify the type of data that is going to be emitted when the Observable is declared. Also, lets add another string to the operator.

```
    Observable<String> hi = Observable.just("Hi", "there");
```

Code 3.2: Observable .just with assignment

Now, it is very important to put something to consume that emitted data, and here we brought up the concept of *Observer*, Observer in the form of a **.subscribe** method is the way to *'hear'* that data and doing something about it, otherwise is only going to be data flying around.

In this case we are going to print the strings emitted by the Observable using the syntax of Lambda expressions (we use lambda because it allow us to write in a very concise way to construct an expression).

```
   Observable<String> hi = Observable.just("Hi", "there");
   hi.subscribe(s -> System.out.println(s));
```

Code 3.3: Observable .just.subscribe with assignment

The variable s is going to go through the emitted values, one by one, and print them. Then the output will be the following.

```
Hi
there
```

Code 3.4: Observable result

Example # 2 Furthermore, these Observables can change the data (depending on the needs) before outputting the result by appending and linking operators in the declaration. For instance, we are going to read a sequence of integers from a list, filter them, sort them, and print them. All of that in one single declaration.

First, lets read the sequence.

```
List<Integer> integers = Arrays.asList(54,12,10,78,69,33,66,99,84);
```

Code 3.5: Integer sequence as list

But, we can't use **.just** as Example # 1 because it will print it as an array, and we need each integer in order to manipulate them. So we use **.fromIterable** with the objective of taking each number individually.

```
Observable<Integer> intNum = Observable.fromIterable(integers);
```

Code 3.6: Taking each integer individually

Secondly, we will filter the even numbers using **.filter**. Again, using the expressive power of a Lambda expression.

```
intNum.filter(i -> i % 2 == 0);
```

Code 3.7: Even number filtering

After that, lets sort it in ascending order adding a **.sorted** to the chain.

```
intNum.sorted();
```

Code 3.8: Number sorting

Finally, we subscribe an Observer that prints the resulted set of values (employing a Lambda expression) with the conditions we stated on each step, using **.subscribe** we attach the Observable to the Observer.

```
intNum.subscribe(i -> System.out.println(i));
```

Code 3.9: Subscribing to an Observer to print the result

Then, the output will be the following.

```
10
12
54
66
78
84
```

Code 3.10: Output

We also can write all of the previous steps in one single declaration thanks to the chaining nature of the Observable Operators. Removing even the assignation to the Observable.

```
    List<Integer> integers = Arrays.asList(54,12,10,78,69,33,66,99,84);
    Observable.fromIterable(integers)
            .filter(i -> i % 2 == 0)
            .sorted()
            .subscribe(i -> System.out.println(i));
```

notice the operator chaining

Code 3.11: One single declaration

Which produces the exact same result.

Project Reactor

Written in Kotlin, Project Reactor[1] is part of the Spring Framework compatible with reactive programming from version 5. Reactor is a reactive quartering library to create non-blocking applications in the JVM based on the Reactive Current Specification. Project Reactor is very similar to ReactiveX, uses *Flux* instead of *Observable* but it does the same thing, includes operators and also writes in a declarative way.

Apache Flink

Apache Flink[2] follows a paradigm that covers the process of data flow as the unifying model for real time. Data analysis, continuous flows, and batch processing both are incorporated in the programming model as in the execution engine.
The combination of durable message queues allows: arbitrary reproduction of data streams (such as Apache Kafka or Amazon Kinesis), stream processing programs do not distinguish between processing the latest events in real time, continually adding data in large files, or processing terabytes of historical information. Instead, these different types of calculations simply begin processing at different points in the lasting flow, and maintain different state forms during the calculation (Carbone, Katsifodimos, Ewen, Markl, Haridi, and Tzoumas, 2015).

Akka Actors

The Actor Model [3] provides a higher level of abstraction for writing concurrent and distributed systems. It reduces the developer's obstacle of dealing with data blocking and thread management, which facilitates the correct writing of parallel and concurrent systems. Akka Actors was defined in the 1973 paper by Carl Hewitt (Hewitt, Bishop, and Steiger, 1973), but was popularized by the Erlang language. One essential use case was at Ericsson with great success on building highly concurrent and reliable telecommunications systems.

[1] *Project Reactor:* *https://projectreactor.io/*
[2] *Apache Flink:* *https://flink.apache.org/*
[3] *Akka:* *https://doc.akka.io/docs/akka/current/guide/index.html*

Eclipse Vert.x

Vert.x[4] is very flexible, whether it is simple network utilities, sophisticated and modern web applications, HTTP / REST micro-services, large-volume event processing or a back-end full-message message application, Vert.x is devised for micro-services due to its low weight and processor demand.

In brief, many of these implementations are not distributed explicitly, they do not have formal semantics and even in our opinion, many of them have a complex syntax that was derived from immediate needs, as a result ReactJS has a complex syntax but it is practical for front-end developers. A lot of these frameworks have been developed and supported by the community, therefore many do not have a formal defined semantics and syntax.

3.2 Functional reactive programming

FRP (Functional Reactive Programming) is a framework to program systems that deal with events and continuous time, the core concepts are Behaviors and Events. A **Behavior** is a time-varying value like an On and Off switch, the weather, a mouse click, etc. An **Event** is a discrete value in time, in the example of the mouse click, a mouse click can *happen* but you do not know the current value of the click.

Several Functional Reactive Programming (FRP) languages have been investigated. Frappé (Courtney, 2001) for example, is a Java library that instantiates Java Beans by connecting them and implementing combinator classes to control the propagation of events. However, Frappé has limitations such as: **1)** it assumes that event processing is single-threaded and synchronous waiting for each event to be propagated through the FRP combiner graph before the next event is handled. ReactiveXD by its reactive-distributed nature, will handle events such as streams in a multi-threading model, **2)** is unable to detect predicates of instantaneous events; that is to say those that happen in a specific time; in ReactiveXD, thanks to the LTL (Linear Temporal Logic) formulas, we could have multiple events on several timelines (seen on a model of vector clocks) and **3)** it does not support time generalized transformations, while ReactiveXD will allow you to make changes to the temporary event flows through similar operators to Reactive Xtensions.

Fran (Functional Reactive Animation) (Elliott and Hudak, 1997), brings up an approach to animation modeling through the notions of *behaviors* and *events* at the moment of developing animation programs by its simplicity, when it comes from modification to execution, it expresses semantics in temporal terms.

Procera (Voellmy, Kim, and Feamster, 2012), is a controller architecture and high-level network control language providing a declarative, expressive, and compositional framework that allows operators to manage high-level complex dynamic control policies.

Flapjax (Meyerovich, Guha, Baskin, Cooper, Greenberg, Bromfield, and Krishnamurthi, 2009), is a language that implements JavaScript with the

[4] *Vert.x:* https://vertx.io/

principles of FRP making use of the reactive paradigm that solves problems related to the amount of callbacks done in an interactive application.

Frob (Functional Robotics) (Peterson, Hudak, and Elliott, 1999), is a domain-specic language embedded in Haskell for robot control, as well it uses a value of type *Behavior* which is a continuous quantities that vary over time seen as, for example, **Behavior velocity** that represents a time-varying robot velocity. And a value of type *Event*, a discrete domain of events that occur in a specific time in a particular robot scenario **Event moveToTheLeft** represents a complex condition that happens in a timeline.

We can see that the lack of support for distribution is also transferred to use cases. For example, RMPL (Reactive Model-Based Programming Language) (Williams et al., 2001) gives a first look at the design level regarding the existing challenge of creating highly competent systems in real-life environments such as robotic networks the coordination of multiple vehicles. However, RMPL is mostly focused on using shorter path algorithms for autonomous rovers, therefore, it does not make use of distribution in a set of rovers at a general level, but it rather only administrate each rover individually. ReactiveXD brings embedded features to attack this problem in environments with multiple devices interacting with each other. Although, this example is a candidate to have explicit distributed semantics at the implementation level, the reactive framework is used at a local level and distribution is done using traditional imperative means.

In contrast, these languages, frameworks and libraries are not explicitly distributed and they also have specific application surface domains like Elm (Czaplicki and Chong, 2013) for graphic interfaces.

3.3 Distributed reactive

Mars Exploration's article discusses the implementation of a reactive programming language based upon models for mission coordination on Mars that involve the cooperation of vehicles, through the consideration of a team of rovers that explore different sites of interest on unknown lands and, consequently, the constant generation of data of different sizes that must be communicated between rovers and transmitted to control stations (Williams et al., 2001). This is especially important since, the InSight lander reached Mars on Nov. 26, 2018, at 11:52:59 a.m. PT (2:52:59 p.m. ET).

The document (Kambona, Boix, and De Meuter, 2013), which shows practices to build collaborative web applications, begins with a view of the common problems faced by developers when creating collaborative web applications (such as the presence of Callback Hell), and brings several solutions that have been proposed throughout the years, such as the use of promises in environments that make use of JavaScript, thus providing a mechanism capable of assigning an advantage in the use of synchronous mechanisms.

CHAPTER 4

ReactiveXD: a language for distributed real-time reactive programming

This section describes the Extended Backus-Naur Form (EBNF) grammar for ReactiveXD, a grammar with support for distribution and concurrency implementing Reactive Extensions concept Observables and using the principles of ReactiveX.

For this ReactiveXD's EBNF (see Figure 4.1) , the words that are in a **bold** font are reserved by the language. The words that are in between a (‘) and a (’) are the tokens needed to delimit the reachability of a determined terminal rule (the provided scope for a declaration). The words in a *italic* font and between (⟨) and a (⟩) represent a non terminal. Finally the terminals that have a (//) at the start of the lexical element simply is a way to express that it uses super grammars that already have production rules defined to extend the lexical tools in the language; they can bring any alphanumeric word (e.g. *Ids* with concrete syntax constraints) or external resources that has its own language definition (e.g. *Aspect oriented programming*).

The language is defined as follows:

⟨*Decl*⟩	::=	⟨*ObsrvDecl*⟩ | ⟨*JVarD*⟩ | ⟨*MSig*⟩
⟨*ObsrvDecl*⟩	::=	`Observable` ‘`<`’⟨*EventType*⟩‘`>`’ `OId` ‘`=`’ ⟨*ObsrvAssig*⟩
⟨*ObsrvAssig*⟩	::=	‘new’ `Observable` ‘`(`’ ⟨*Ep*⟩ ‘`)`’ ‘`;`’
⟨*EventType*⟩	::=	`Event`
⟨*Ep*⟩	::=	**call** ‘`(`’⟨*ESig*⟩‘`)`’ | *EId*`({`⟨*Par*⟩`})`
	|	⟨*Location*⟩
	|	⟨*Ep*⟩||⟨*Ep*⟩ | ⟨*Ep*⟩&&⟨*Ep*⟩ | `!`⟨*Ep*⟩
⟨*Location*⟩	::=	`localhost` | “Ip:Port”
⟨*MSig, ESig*⟩	::=	`// method and field signatures (AspectJ-style)`
⟨*Par*⟩	::=	`// argument or parameter expression`

Figure 4.1: EBNF base grammar of ReactiveXD

ReactiveXD uses the concept of Observables similar to ReactiveX to interact with the nodes of a system by sending data. *Decl* defines the posible declarations within the code file, *ObsrvDecl* focuses explicitly on the instantiation and initialization of the Observable as we can see in *ObsrvAssig*, the *EventType* focuses in the type of data that is going to be treated, in this case events; but also grants the possibility of extending the language through the implementation of other event processing schemes. *Ep* is the event predicate, that calls a function in order to operate over it. The *Location* is a non-terminal that describes the location hosting the object that monitors events. The Observers are defined implicitly as the language allows the communication between devices.

In addition to understand the grammar graphically exposed above, a syntax diagram is provided as follows that points out the lexical path that an ReactiveXD's Observable will take when declared:

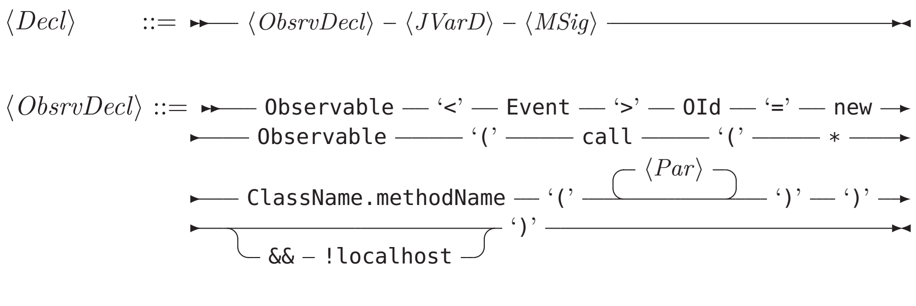

⟨Decl⟩ ::= ⟨ObsrvDecl⟩ – ⟨JVarD⟩ – ⟨MSig⟩
⟨ObsrvDecl⟩ ::= Observable — ‘<’ — Event — ‘>’ — OId — ‘=’ — new
Observable — ‘(’ — call — ‘(’ — *
ClassName.methodName — ‘(’ — ⟨Par⟩ — ‘)’ — ‘)’
&& – !localhost — ‘)’

Figure 4.2: Syntax grammar of an Observable

The previous syntax diagram provides the overall route that the declaration of an Observable will take when declared, it starts by using the object syntax similar to Java with the given type *Event*, after that, it states a *class* and a *method* related to that class with zero or more arguments, and last but not least it provides an optional field which is *!localhost*. One key note in this syntax diagram is that *!localhost* appears as an optional terminal but that is merely to present it as a boolean value that can be *true* or *false* depending on the context, on future development of the language it could be an optional parameter.

To explore the current potential state of ReactiveXD, a couple of use cases are proposed to understand the logic and concepts of the language specifically the design of it, that is the model which is later compiled to support distribution and the implication of events occurring between n nodes.

4.1 Collaborative figure example

In the process of building a ReactiveX-like language we gotta model the core concept of it, the **Observable**. This Observable as it was seen in Chapter 3, is going to emit values when an event happens at a time a specific type of data appears. In the State of the Art we showed up types of data like Integers and Strings, here we declare a new type called **Event**. This *"Event Observable"* is going to match a expression (called event predicate) that uses aspect oriented language's syntax similar to AspectJ. We also add the word **"new"** to have a representation of the Observable as an object rather than a single type of data.

For instance, let's think about a theoretical collaborative application that draws and moves figures in a canvas. The Observable that represents this scenario with ReactiveXD's grammar could be like.

```
Observable<Event> figEl = new Observable (call(* FigureElement.getXY())
                              && !localhost);
```

Code 4.1: Collaborative figure example

So, if one of the nodes that participates in the application moves a figure then an event predicate will play its role by emitting an event each time a figure's position is changed and that result should be noticeable in all the nodes that are executing the distributed application. Then, it will match all the method calls from *getXY()* to objects of type *FigureElement* on any host. Also, the localization of events is restricted (**!localhost**) that means the Observable will not emit events happening in the host were the Observable is deployed.

4.2 Chat example

Taking this into a more day-to-day scenario, an implementation of the language can be seen for example in a chat architecture. The model is pretty simple.

We have two nodes *A* and *B*.

Figure 4.3: Nodes instances

Each one has the chat application running inside and establishes a communication session with each other.

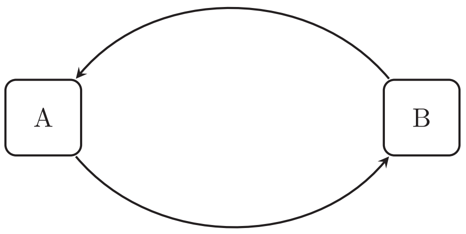


Figure 4.4: Node communication

The conversation operates by sending and receiving messages asynchronously (not necessarily at a constant phase or at the same time).

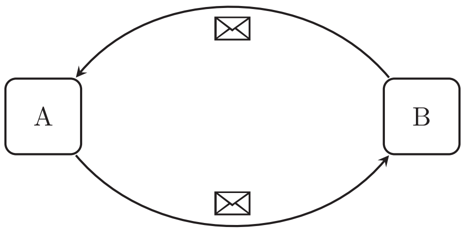


Figure 4.5: Node messaging transmission

In the process of the exchanging messages, node *A* will have an Observable capturing the messages sent by *B* and similarly, *B* will have an Observable capturing the messages sent by *A*.

```
Observable<Event> friendA = new Observable (call(* GroupChat.main())
                            && !localhost);

Observable<Event> friendB = new Observable (call(* GroupChat.main())
                            && !localhost);
```

Code 4.2: Chat observables

When the Observable instantiating operation starts; the compiler realizes that a distributed instance session has begun. And every time a client sends a message to other, an event will occur in the application's samples.

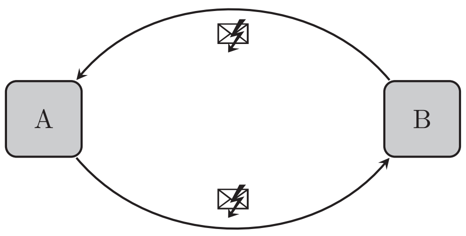


Figure 4.6: Node events

After the emission of events (messages sent between the nodes A and B) in the application, an aspect will be created displaying the point of interest where the application distributed messages between the clients.

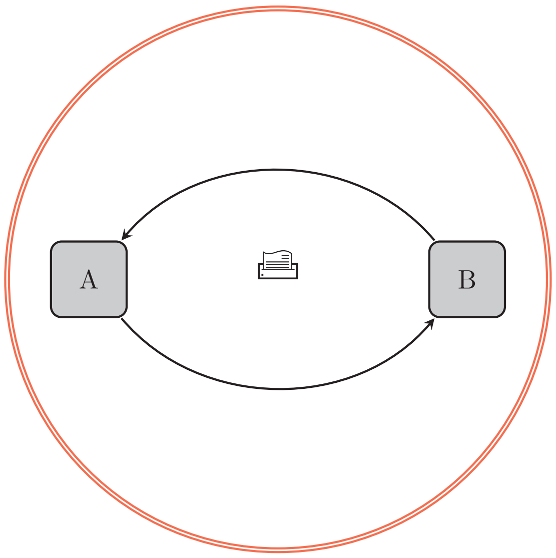


Figure 4.7: Node aspect

In there, it recognizes a sequence of events that happened in a chat session under a distributed context as a result of exposing the execution and compilation of the program.

4.3 Time management architecture

Causality

The goal of finding causality relations in the events consists on defining predicates on complex time dependencies. We ensure that the system has a correlation between a stream of events where one of them affects the nearest to it.
Therefore, we argue that such systems could be enriched with real-time detection of intricate patterns of distributed events with sophisticated time dependencies.

To give an idea the syntax could be defined like.

```
Observable<Event> causFigEl = new Observable (call(* FigureElement.getXY())
                              && !localhost && !causal);
```

Code 4.3: Causal figure example

Where, the causal relation defines the occurrence of an event in a synchronous type of schema. In other words, it is stated to be concurrent when the hole expression is evaluated by the language.

LTL formulas

The validity of LTL (Linear-time Temporal Logic) formulas are defined over paths and an LTS (Linear Temporal Statement) satisfies an LTL formula if and only if it satisfies the formula on all paths. To show that an LTS violates a property, it is sufficient to exhibit a counter-example, that is, a path for which the property is not true. So with that in mind, we can build up an entire data system where an event predicate expression depends on satisfying or not the path of a given property. Defining small approaches to a discrete-time stateful model that investigates the implementation on a reactive program (Jeffrey, 2012).

Case in point.

```
Observable<Event> ltlFigEl = new Observable (always(figEl next(causFigEl)));
```

Code 4.4: LTL figure example

The Observable will emit an event each time the property is violated. This property indicates for ***ltlFigEl*** it should be always true that: *"immediately after the Observable **figEl** emits an event the Observable **causFigEl** emits an event"*. These semantics in the language help us, in some sort of way, to model an *'order'* on the events and to express desired properties to occur in the system by coding it in a predicate using linear temporal logic (LTL) expressions.

CHAPTER 5

Implementation

With the DSL (Domain Specific Language), we are building an abstraction like taking LEGO bricks in order to make something. We use a parser generator to read the grammar then we create an AST (Abstract Syntax Tree). After that, we write the code generator, passing this AST into a code.

We then validate the model, since the end-user should have a robust language with instant feedback. In sense, the structure that is used are graphs (e.g. a method call in Java is a pointing declaration of a method). We should use Type Checking to make a more sophisticated language. And we use Type Inference Engine to explain to business people what is this language.

For implementation it its used Xtext, an open source project that provides a set of APIs and DSLs to develop the language for support to eclipse, intellij, and web browsers. It uses Google Guice for injecting components. And one important asset is that the compiler's components of the language are independent of Eclipse or OSGi so it can be used in any Java environment.

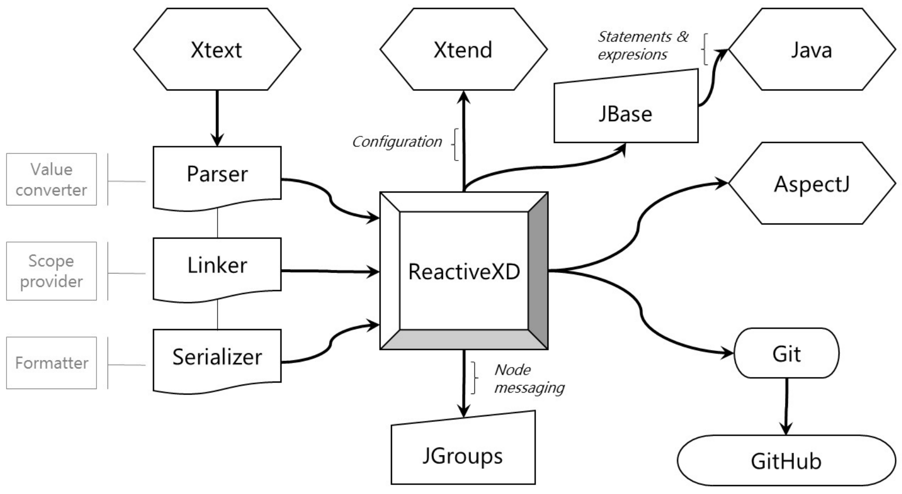


Figure 5.1: Architecture of ReactiveXD

Xtext will generate our Java Bean classes (EMF classes) which are initially empty but we implement the source code in a later time. Xtend is a programming language that translates to Java source code directly with a lot of enhancements. JGroups is the software that allow us to communicate node instances. JBase presents syntax to make the language close to Java coding. AspectJ will be the language that is going to be generated. And Git is our version control system that uses GitHub as external repository.

5.1 Decentralized arquitectures

We start by introducing a model of distributed systems as groundwork for defining the objective as there is no central server, when we do distribution, there is no need for a server to communicate the distribution. Therefore, between the devices there is events communication. The nodes, are producers and consumers of events, and there is a near broadcast, also listening. And they build what we propose in reactive programming and Mesh networking.

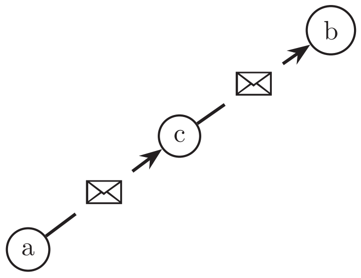


Figure 5.2: Decentralized node messaging

So as in the previous figure Figure 5.2, if the node *a* wants to communicate with the node *b* it can do it through *c*, which is also a node with the same nature as *a* and *b*, exposing the unprofitable use of a central server that redirects the messages. This helps on distributed infrastructures since the nodes has low impact on the overall performance of the system, especially if the set of nodes is big and the entire node grouping system is complex with any number of connections. All of this is accomplished thanks to JGroups, a technology that focuses on node messaging with support for distributed applications and language implementations.

5.2 Compiler

The compiler works with Xtext to implement a compiler for the language. JBase it is used to generate pure java expressions and statements using the Xtext framework and the compiler generates AspectJ and Java code to instrument monitoring in distributed applications.

The development workflow of programming in Xtext encompasses on: translating, generating, validating, customizing, and implementing a Domain-Specific Language (DSL) under a continuous integration configuration (Bettini, 2016).

Translation

So in the case of ReactiveXD, after we have created the grammar as an Extended Backus-Naur Form (EBNF), the next step is to translate it into the Xtext semantics. In here, a collection of syntax rules are specified and may increase the number due to the Xtext way to define a grammar. The following piece code grammar defined on Xtext shows the ReactiveXD design, *OrEvent* (line 23) will iterate over the different kind of logical operators supported by the language as: and (&&), or (||), and not (!).

```
1  grammar co.edu.escuelaing.reactivexd.ReactiveXD with jbase.Jbase
2
3  import "http://www.eclipse.org/xtext/xbase/Xbase"
4  import "http://www.eclipse.org/xtext/common/JavaVMTypes" as jvmTypes
5
6  generate reactiveXD "http://www.escuelaing.edu.co/reactivexd/ReactiveXD"
7
8  Model:
9    (’package’ name = QualifiedName ->’;’?)?
10   importSection = XImportSection?
11   typeDeclaration += Decl*
12 ;
13
14 Decl:
15   ObsrvDecl | JVarD | MSig
16 ;
17
18 ObsrvDecl:
19   ’Observable’ ’<’ type=EventType ’>’ name=ID ’=’ assignment=ObsrvAssig
20 ;
21
22 ObsrvAssig:
23   {ObsrvAssig} ’new’ ’Observable’ ’(’ props+=OrEvent* ’)’ ’;’
24 ;
25
26 enum EventType:
27   Event
28 ;
29
30 // more rules are defined here
```

Code 5.1: ReactiveXD's piece of Xtext grammar file

Generation

Xtext configures its artifacts by generating stub classes in Java and Xtend. Xtend is a less *"syntactic noisy"* Java language that uses a Java-like syntax, including Java generics and Java annotations. Xtend has a lot of characteristics, for instance, it provides multi-line expressions to address all the code that is going to be generated on a chosen language (e.g. AspectJ), we define the limits with triple single quotes (’’’) and specify variable expressions using guillemets («»). Consider then the following code written in Xtend.

```
package co.edu.escuelaing.ReactiveXD.generator

import org.eclipse.xtext.generator.IFileSystemAccess
import org.eclipse.xtext.generator.IGenerator
import co.edu.escuelaing.ReactiveXD.ReactiveXD.ObsrvDecl
import co.edu.escuelaing.ReactiveXD.ReactiveXD.EventClass
import co.edu.escuelaing.ReactiveXD.jvmmodel.ReactiveXDJvmModelInferrer
import java.util.Set
import java.util.TreeSet
import java.util.HashMap

.
.
.

class ReactiveXDGenerator implements IGenerator {

    .
    .
    .

    def CharSequence generateObservables(EventClass modelo, String packageName
         , Set<String> libraries) {

    aspectClass = modelo.name
    var packageDefinition = '''package «packageName»;

    '''
    var observables = modelo.declarations.filter(Observables)
    var Set<String> importedLibraries = new TreeSet()
    importedLibraries+=libraries
    var pointcuts = new TreeSet<String>

    var after = new HashMap<String, String>()

    if(modelo.declarations.containsObservable)
    importedLibraries+="co.edu.escuelaing.reactivexd.groupsimpl.*"
    importedLibraries+="co.edu.escuelaing.reactivexd.handlercontrol.*"
    importedLibraries+="co.edu.escuelaing.reactivexd.core.ObsrvDecl"
    importedLibraries+="co.edu.escuelaing.reactivexd.core.NamedEvent"
    importedLibraries+="co.edu.escuelaing.reactivexd.core.Event"
    importedLibraries+="java.util.Map"
    importedLibraries+="java.util.HashMap"

    var aspect = '''
    public aspect «aspectClass.toFirstUpper»{

      «FOR event:modelo.declarations»
        «IF event instanceof ObsrvDecl
          pointcut «event.name.toFirstLower»():
            «createPointCut(event as ObsrvDecl, pointcuts)»;
          after(): «event.name.toFirstLower»(){
            «IF !observables.isEmpty»
              Event event = new NamedEvent("«event.name»");
              «ReactiveXDJvmModelInferrer.handlerClassName» distribuidor = «
                   ReactiveXDJvmModelInferrer.handlerClassName».getInstance();
              event.setLocalization(distribuidor.getAsyncAddress());
              Map map = new HashMap<String, Object>();
              distribuidor.multicast(event, map);
              System.out.println("[Aspectj] After: Recognized an event in
                   ObservableConstructor");
            «ENDIF»
          }
        «ENDIF»
      «ENDFOR»

      «FOR pointcut:pointcuts»
```

```
        «pointcut»;
      «ENDFOR»
    }
    '''
    var imports = '''
    «FOR tipo:importedLibraries»
      import «tipo»;
    «ENDFOR»

    '''
    return packageDefinition+imports+aspect

  }

  .
  .
  .

}
```

Code 5.2: ReactiveXD's piece of Xtend code generator

The AspectJ code generated focuses on naming aspects (line 45) and pointcuts (line 49) depending on the executed sample (in this document a group chat), defines the necessary join point (line 50) to be called on an instance of the execution of an event, of course it would not be complete if an advice is allocated, this advice (lines 51-58) works as an event distributor after the execution of the called method from a class finished. It also contains, the package definition (lines 25-27) that gives the path of the file and a set of imports (line 30).

Validation

The correctness of a program rely on the communication of errors or warnings in the code this validation operates while the program is being coded and provides feedback to the programmer, by default there is a validator with MWE2 (Modeling Workflow Engine 2), one simple validation is the uniqueness of the variables names (line 12) as you can see in the following piece of code.

```
1    .
2    .
3    .
4    language = StandardLanguage {
5        name = "co.edu.escuelaing.reactivexd.ReactiveXD"
6        fileExtensions = "rexd"
7
8        serializer = {
9          generateStub = true
10       }
11       validator = {
12         composedCheck = "org.eclipse.xtext.validation.NamesAreUniqueValidator"
13         // Generates checks for @Deprecated grammar annotations, an
               IssueProvider and a corresponding PropertyPage
14         generateDeprecationValidation = true
15       }
16       junitSupport = {
17         junitVersion = "5"
18       }
19     }
20   .
```

```
    .
    .
```

Code 5.3: ReactiveXD's piece of MWE code checker

Customization

The customization of the language can be seen in many ways, writing less syntactic rules in the Xtext grammar, refactoring code in Xtend code generation, extending validations in the MWE2 file, including loggers, importing google guice injectors, and so on. Though, for this academic exercise we keep it as simple as it is; with default validators and lack of use of external libraries/components.

Implementation

Ergo, the overall language should implement a code like this:

```
  Observable<Event> obs = new Observable (call(* Class.method(parameters))
                          && !localhost);
```

Code 5.4: Distributed Observable code

Where it defines the type of data **Observable** to distribute the data, a **call** which is aspect-like typing and a **localhost** to determine the set of data received under a set of nodes excluding the one where the Observable is declared. After that it should produce an AspectJ file containing a pointcut with the name of the executed class and an aspect with an **after():** tag that shows the execution of the program and the recognition of the interested event pattern. That is, the state where the language is right now, as shown it has a communication model through a message-driven architecture illustrating the data flow in the form of events simulating the stream event theory that reactive languages have implemented in the past.

5.3 Real-time infrastructure

The real-time infrastructure is based upon the consumer and producer model that defines the behaviour of the language and suits to the functionality of it. Especially that is important an infrastructure like this on web technologies that are leading the way to improve user experience. A handful of common use cases contains: real-time stats the technology that was first used to show live information highly beneficial to sports, betting and analytics, notifications when something interesting to a user is shown it then becomes available, chat the technology that helps in delivery instant interaction between people and multiplayer games giving the ability to instantly deliver player moves, game state changes, and scoring updates.

CHAPTER 6

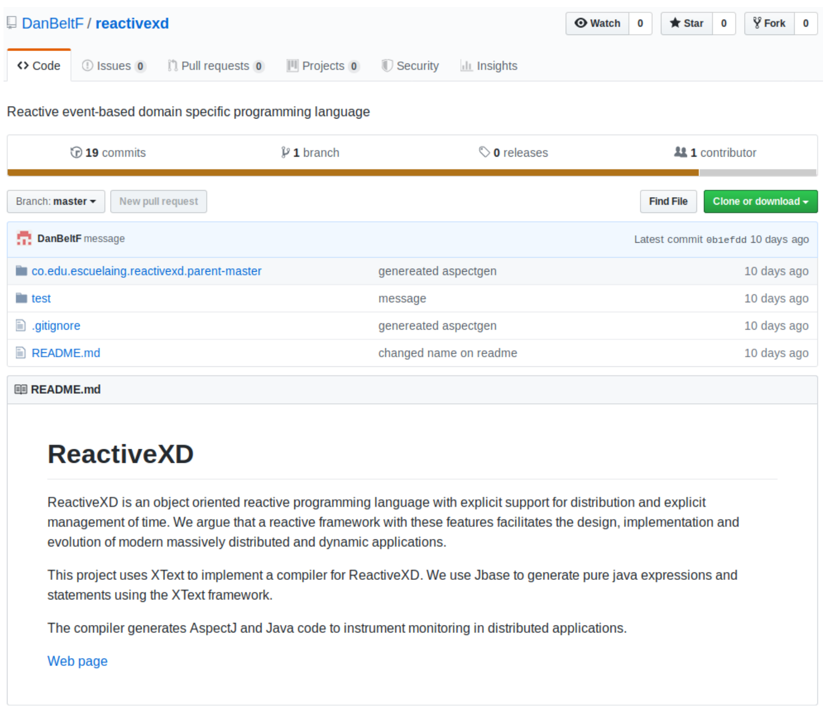
Evaluation

In the use cases shown before, the basic Collaborative figure example and the Chat example, we were able to analyze the dynamic behavior of a distributed reactive language that checks an event predicate expression and then parses it into a distributed environment with an aspect oriented expression that evaluates the execution of the data.

6.1 Chat example

The GroupChat example consists on a service that allows one node (terminal console/computer/server) sending messages to another and then generates an aspect for the session.

To run the example, firstly we go to the repository.


DanBeltF / reactivexd

Watch 0 Star 0 Fork 0

Code Issues 0 Pull requests 0 Projects 0 Security Insights

Reactive event-based domain specific programming language

19 commits 1 branch 0 releases 1 contributor

Branch: master New pull request Find File Clone or download

DanBeltF message Latest commit eb1efdd 10 days ago

co.edu.escuelaing.reactivexd.parent-master	genereated aspectgen	10 days ago
test	message	10 days ago
.gitignore	genereated aspectgen	10 days ago
README.md	changed name on readme	10 days ago

README.md

ReactiveXD

ReactiveXD is an object oriented reactive programming language with explicit support for distribution and explicit management of time. We argue that a reactive framework with these features facilitates the design, implementation and evolution of modern massively distributed and dynamic applications.

This project uses XText to implement a compiler for ReactiveXD. We use Jbase to generate pure java expressions and statements using the XText framework.

The compiler generates AspectJ and Java code to instrument monitoring in distributed applications.

Web page

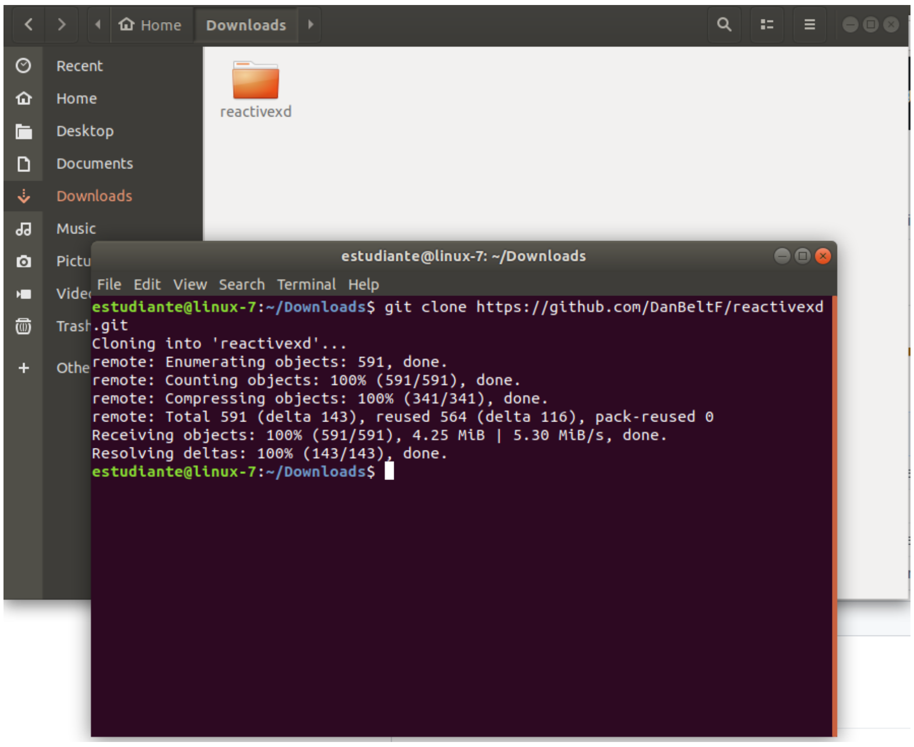
Figure 6.1: Project's GitHub repository web site

Now, we will download it by either a ZIP file or by cloning it with Git with the command:

```
$ git clone https://github.com/DanBeltF/reactivexd.git
```

Code 6.1: Cloning the repo

And we place it on a directory, here we use the directory *Downloads/*.



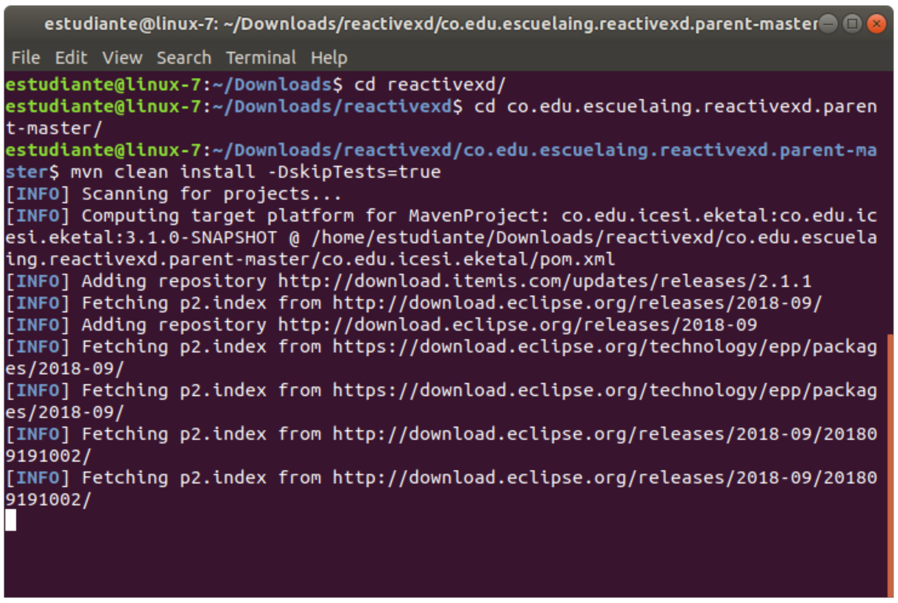
```
estudiante@linux-7:~/Downloads$ git clone https://github.com/DanBeltF/reactivexd.git
Cloning into 'reactivexd'...
remote: Enumerating objects: 591, done.
remote: Counting objects: 100% (591/591), done.
remote: Compressing objects: 100% (341/341), done.
remote: Total 591 (delta 143), reused 564 (delta 116), pack-reused 0
Receiving objects: 100% (591/591), 4.25 MiB | 5.30 MiB/s, done.
Resolving deltas: 100% (143/143), done.
estudiante@linux-7:~/Downloads$
```

Figure 6.2: Cloning the remote repository in our machine

After that, we access the folders named *reactivexd/* and *co.edu.escuelaing.reactivexd.parent-master/*, and we build the language with the command:

```
$ mvn clean install -DskipTests=true
```

Code 6.2: Installing dependencies


estudiante@linux-7: ~/Downloads/reactivexd/co.edu.escuelaing.reactivexd.parent-master
File Edit View Search Terminal Help

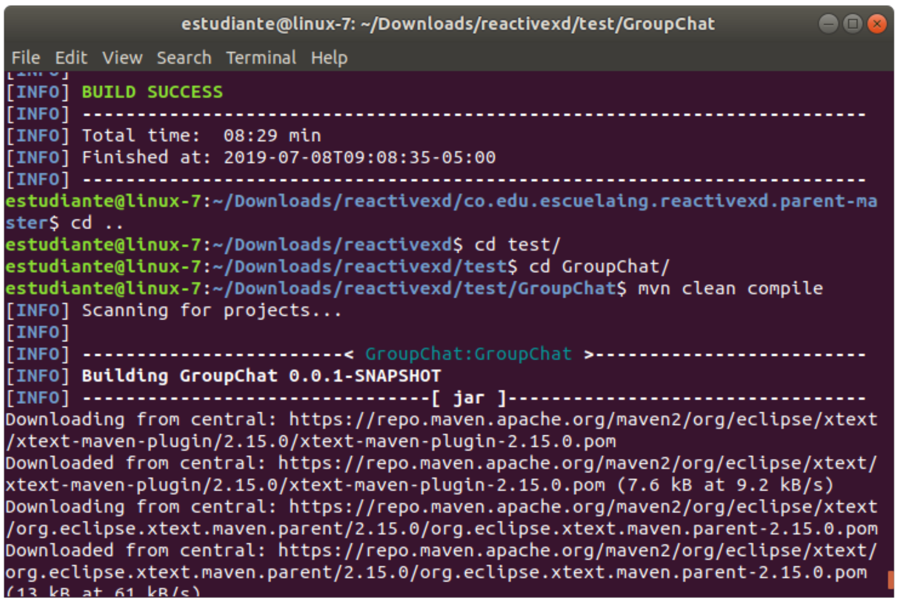
```
estudiante@linux-7:~/Downloads$ cd reactivexd/
estudiante@linux-7:~/Downloads/reactivexd$ cd co.edu.escuelaing.reactivexd.paren
t-master/
estudiante@linux-7:~/Downloads/reactivexd/co.edu.escuelaing.reactivexd.parent-ma
ster$ mvn clean install -DskipTests=true
[INFO] Scanning for projects...
[INFO] Computing target platform for MavenProject: co.edu.icesi.eketal:co.edu.ic
esi.eketal:3.1.0-SNAPSHOT @ /home/estudiante/Downloads/reactivexd/co.edu.escuela
ing.reactivexd.parent-master/co.edu.icesi.eketal/pom.xml
[INFO] Adding repository http://download.itemis.com/updates/releases/2.1.1
[INFO] Fetching p2.index from http://download.eclipse.org/releases/2018-09/
[INFO] Adding repository http://download.eclipse.org/releases/2018-09
[INFO] Fetching p2.index from https://download.eclipse.org/technology/epp/packag
es/2018-09/
[INFO] Fetching p2.index from https://download.eclipse.org/technology/epp/packag
es/2018-09/
[INFO] Fetching p2.index from http://download.eclipse.org/releases/2018-09/20180
9191002/
[INFO] Fetching p2.index from http://download.eclipse.org/releases/2018-09/20180
9191002/
```

Figure 6.3: Installation of the external project dependencies needed to use the language

The next stage consists on compiling the example, for that, we go back and then enter in the directory *test/GroupChat/* which contains an example that simulates a temporal chat session between two or more nodes.
For the compilation we use the command:

```
$ mvn clean compile
```

Code 6.3: Compilation code


estudiante@linux-7: ~/Downloads/reactivexd/test/GroupChat
File Edit View Search Terminal Help

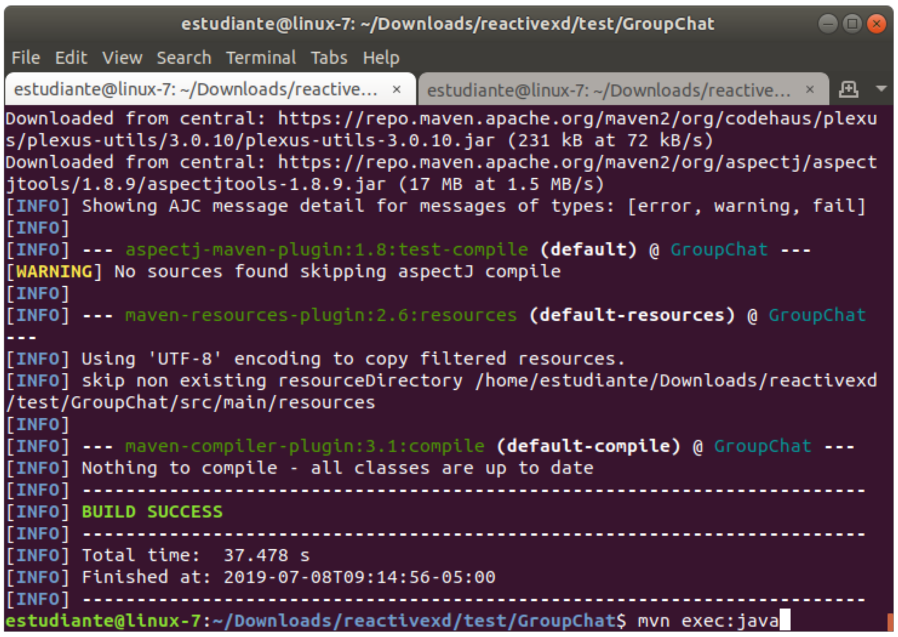
```
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  08:29 min
[INFO] Finished at: 2019-07-08T09:08:35-05:00
[INFO] ------------------------------------------------------------------------
estudiante@linux-7:~/Downloads/reactivexd/co.edu.escuelaing.reactivexd.parent-ma
ster$ cd ..
estudiante@linux-7:~/Downloads/reactivexd$ cd test/
estudiante@linux-7:~/Downloads/reactivexd/test$ cd GroupChat/
estudiante@linux-7:~/Downloads/reactivexd/test/GroupChat$ mvn clean compile
[INFO] Scanning for projects...
[INFO]
[INFO] -----------------------< GroupChat:GroupChat >------------------------
[INFO] Building GroupChat 0.0.1-SNAPSHOT
[INFO] --------------------------------[ jar ]--------------------------------
Downloading from central: https://repo.maven.apache.org/maven2/org/eclipse/xtext
/xtext-maven-plugin/2.15.0/xtext-maven-plugin-2.15.0.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/eclipse/xtext/
xtext-maven-plugin/2.15.0/xtext-maven-plugin-2.15.0.pom (7.6 kB at 9.2 kB/s)
Downloading from central: https://repo.maven.apache.org/maven2/org/eclipse/xtext
/org.eclipse.xtext.maven.parent/2.15.0/org.eclipse.xtext.maven.parent-2.15.0.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/eclipse/xtext/
org.eclipse.xtext.maven.parent/2.15.0/org.eclipse.xtext.maven.parent-2.15.0.pom
(13 kB at 61 kB/s)
```

Figure 6.4: Execution of a build lifecycle goal to compile the project's code with Maven

Consequently, execute the command to run the chat application:

```
$ mvn exec:java
```

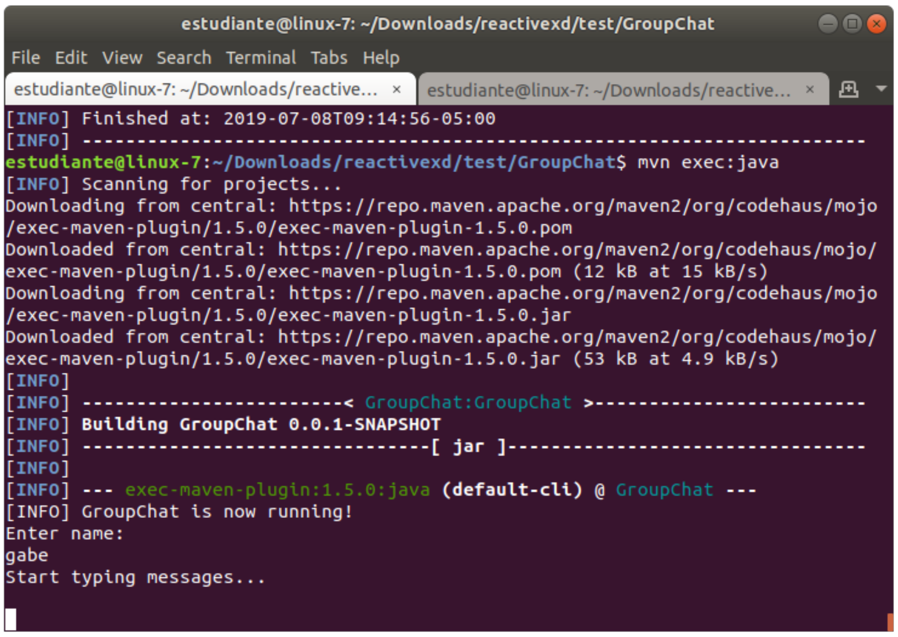
Code 6.4: Execution of Java sources



```
Downloaded from central: https://repo.maven.apache.org/maven2/org/codehaus/plexu
s/plexus-utils/3.0.10/plexus-utils-3.0.10.jar (231 kB at 72 kB/s)
Downloaded from central: https://repo.maven.apache.org/maven2/org/aspectj/aspect
jtools/1.8.9/aspectjtools-1.8.9.jar (17 MB at 1.5 MB/s)
[INFO] Showing AJC message detail for messages of types: [error, warning, fail]
[INFO]
[INFO] --- aspectj-maven-plugin:1.8:test-compile (default) @ GroupChat ---
[WARNING] No sources found skipping aspectJ compile
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ GroupChat
---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory /home/estudiante/Downloads/reactivexd
/test/GroupChat/src/main/resources
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ GroupChat ---
[INFO] Nothing to compile - all classes are up to date
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  37.478 s
[INFO] Finished at: 2019-07-08T09:14:56-05:00
[INFO] ------------------------------------------------------------------------
estudiante@linux-7:~/Downloads/reactivexd/test/GroupChat$ mvn exec:java
```

Figure 6.5: Running the Java program within the same VM as Maven

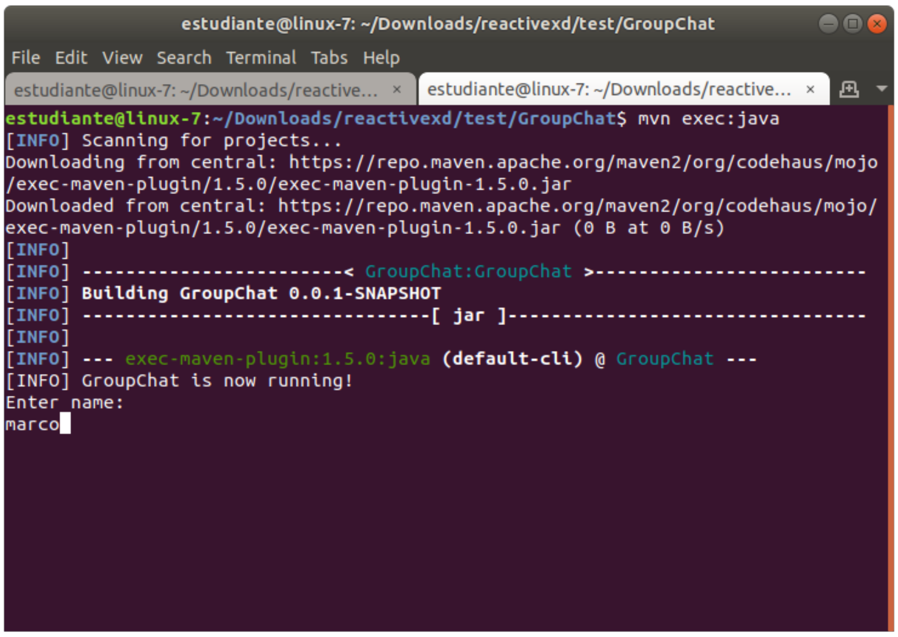
Add a name for temporarily identify your node in the session.



```
[INFO] Finished at: 2019-07-08T09:14:56-05:00
[INFO] ------------------------------------------------------------------------
estudiante@linux-7:~/Downloads/reactivexd/test/GroupChat$ mvn exec:java
[INFO] Scanning for projects...
Downloading from central: https://repo.maven.apache.org/maven2/org/codehaus/mojo
/exec-maven-plugin/1.5.0/exec-maven-plugin-1.5.0.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/codehaus/mojo/
exec-maven-plugin/1.5.0/exec-maven-plugin-1.5.0.pom (12 kB at 15 kB/s)
Downloading from central: https://repo.maven.apache.org/maven2/org/codehaus/mojo
/exec-maven-plugin/1.5.0/exec-maven-plugin-1.5.0.jar
Downloaded from central: https://repo.maven.apache.org/maven2/org/codehaus/mojo/
exec-maven-plugin/1.5.0/exec-maven-plugin-1.5.0.jar (53 kB at 4.9 kB/s)
[INFO]
[INFO] ----------------------< GroupChat:GroupChat >-----------------------
[INFO] Building GroupChat 0.0.1-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- exec-maven-plugin:1.5.0:java (default-cli) @ GroupChat ---
[INFO] GroupChat is now running!
Enter name:
gabe
Start typing messages...
```

Figure 6.6: Assigning a name to first terminal's instance

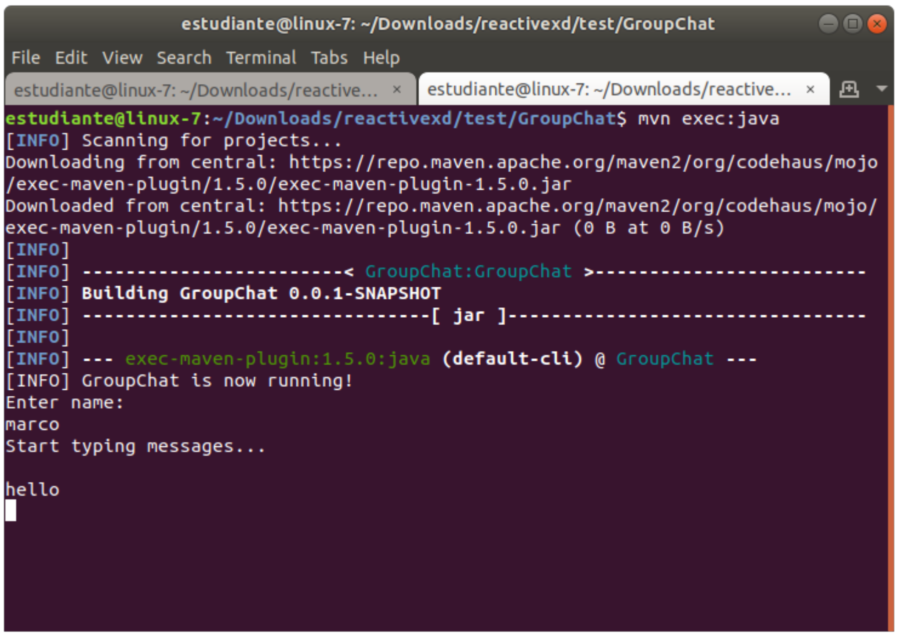
In the same way, we open another Terminal instance (another tab in this case) and write the same command, assigning as well a temporarily name to identify this node.


estudiante@linux-7: ~/Downloads/reactivexd/test/GroupChat
File Edit View Search Terminal Tabs Help
estudiante@linux-7: ~/Downloads/reactive...
estudiante@linux-7: ~/Downloads/reactive...

```
estudiante@linux-7:~/Downloads/reactivexd/test/GroupChat$ mvn exec:java
[INFO] Scanning for projects...
Downloading from central: https://repo.maven.apache.org/maven2/org/codehaus/mojo
/exec-maven-plugin/1.5.0/exec-maven-plugin-1.5.0.jar
Downloaded from central: https://repo.maven.apache.org/maven2/org/codehaus/mojo/
exec-maven-plugin/1.5.0/exec-maven-plugin-1.5.0.jar (0 B at 0 B/s)
[INFO]
[INFO] ------------------------< GroupChat:GroupChat >------------------------
[INFO] Building GroupChat 0.0.1-SNAPSHOT
[INFO] --------------------------------[ jar ]--------------------------------
[INFO]
[INFO] --- exec-maven-plugin:1.5.0:java (default-cli) @ GroupChat ---
[INFO] GroupChat is now running!
Enter name:
marco
```

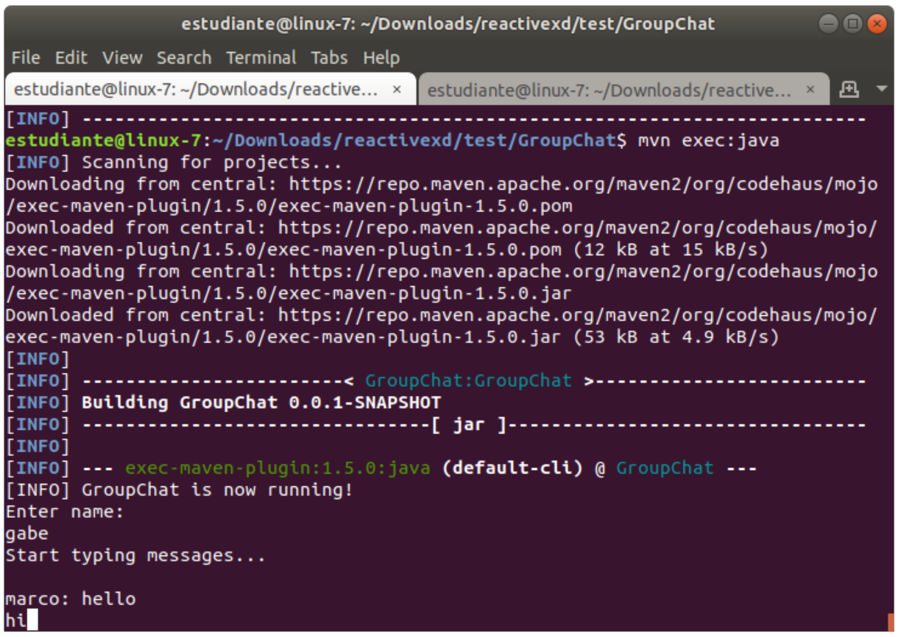
Figure 6.7: Running the Java program as Maven in other terminal

Subsequently, start to exchange messages between the node's instances, checking how they appear on each terminal.


estudiante@linux-7: ~/Downloads/reactivexd/test/GroupChat
File Edit View Search Terminal Tabs Help
estudiante@linux-7: ~/Downloads/reactive...
estudiante@linux-7: ~/Downloads/reactive...

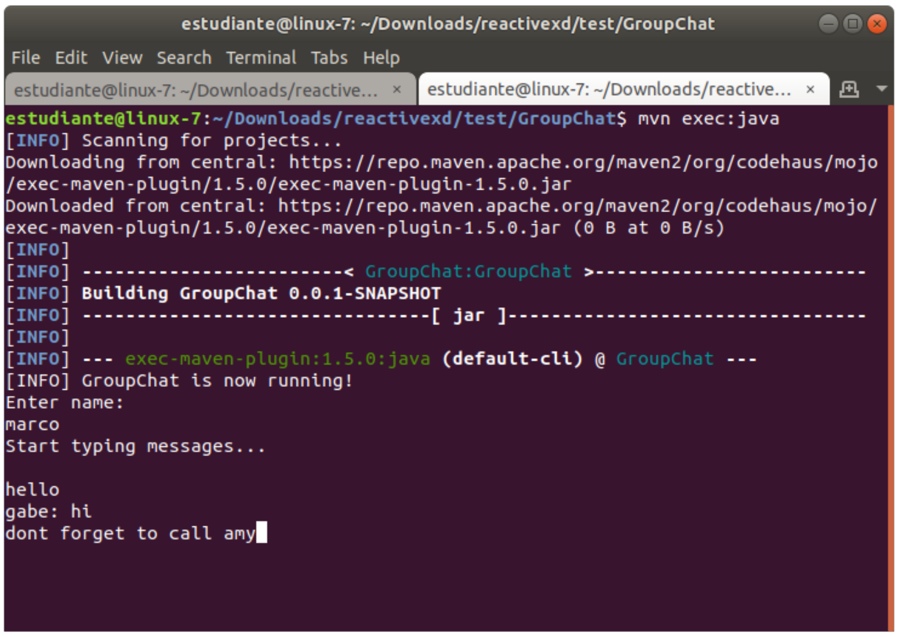
```
estudiante@linux-7:~/Downloads/reactivexd/test/GroupChat$ mvn exec:java
[INFO] Scanning for projects...
Downloading from central: https://repo.maven.apache.org/maven2/org/codehaus/mojo
/exec-maven-plugin/1.5.0/exec-maven-plugin-1.5.0.jar
Downloaded from central: https://repo.maven.apache.org/maven2/org/codehaus/mojo/
exec-maven-plugin/1.5.0/exec-maven-plugin-1.5.0.jar (0 B at 0 B/s)
[INFO]
[INFO] -----------------------< GroupChat:GroupChat >------------------------
[INFO] Building GroupChat 0.0.1-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- exec-maven-plugin:1.5.0:java (default-cli) @ GroupChat ---
[INFO] GroupChat is now running!
Enter name:
marco
Start typing messages...

hello
```

Figure 6.8: Message #1, from: marco to: gabe


estudiante@linux-7: ~/Downloads/reactivexd/test/GroupChat
File Edit View Search Terminal Tabs Help
estudiante@linux-7: ~/Downloads/reactive...
estudiante@linux-7: ~/Downloads/reactive...

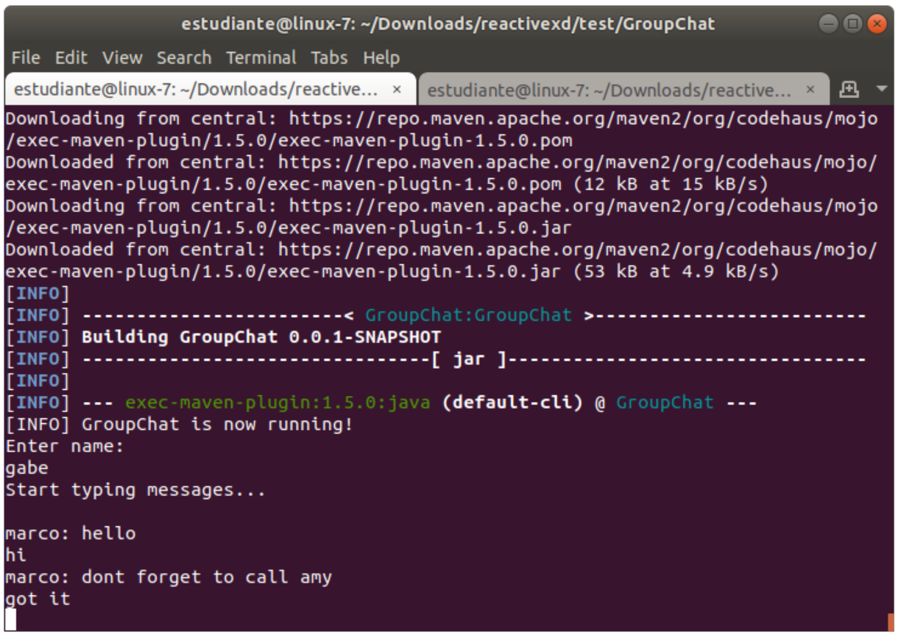
```
[INFO] ------------------------------------------------------------------------
estudiante@linux-7:~/Downloads/reactivexd/test/GroupChat$ mvn exec:java
[INFO] Scanning for projects...
Downloading from central: https://repo.maven.apache.org/maven2/org/codehaus/mojo
/exec-maven-plugin/1.5.0/exec-maven-plugin-1.5.0.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/codehaus/mojo/
exec-maven-plugin/1.5.0/exec-maven-plugin-1.5.0.pom (12 kB at 15 kB/s)
Downloading from central: https://repo.maven.apache.org/maven2/org/codehaus/mojo
/exec-maven-plugin/1.5.0/exec-maven-plugin-1.5.0.jar
Downloaded from central: https://repo.maven.apache.org/maven2/org/codehaus/mojo/
exec-maven-plugin/1.5.0/exec-maven-plugin-1.5.0.jar (53 kB at 4.9 kB/s)
[INFO]
[INFO] -----------------------< GroupChat:GroupChat >------------------------
[INFO] Building GroupChat 0.0.1-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- exec-maven-plugin:1.5.0:java (default-cli) @ GroupChat ---
[INFO] GroupChat is now running!
Enter name:
gabe
Start typing messages...

marco: hello
hi
```

Figure 6.9: Message #2, from: gabe to: marco


estudiante@linux-7: ~/Downloads/reactivexd/test/GroupChat
File Edit View Search Terminal Tabs Help
estudiante@linux-7: ~/Downloads/reactive...
estudiante@linux-7: ~/Downloads/reactive...

```
estudiante@linux-7:~/Downloads/reactivexd/test/GroupChat$ mvn exec:java
[INFO] Scanning for projects...
Downloading from central: https://repo.maven.apache.org/maven2/org/codehaus/mojo
/exec-maven-plugin/1.5.0/exec-maven-plugin-1.5.0.jar
Downloaded from central: https://repo.maven.apache.org/maven2/org/codehaus/mojo/
exec-maven-plugin/1.5.0/exec-maven-plugin-1.5.0.jar (0 B at 0 B/s)
[INFO]
[INFO] ------------------------< GroupChat:GroupChat >------------------------
[INFO] Building GroupChat 0.0.1-SNAPSHOT
[INFO] --------------------------------[ jar ]--------------------------------
[INFO]
[INFO] --- exec-maven-plugin:1.5.0:java (default-cli) @ GroupChat ---
[INFO] GroupChat is now running!
Enter name:
marco
Start typing messages...

hello
gabe: hi
dont forget to call amy
```

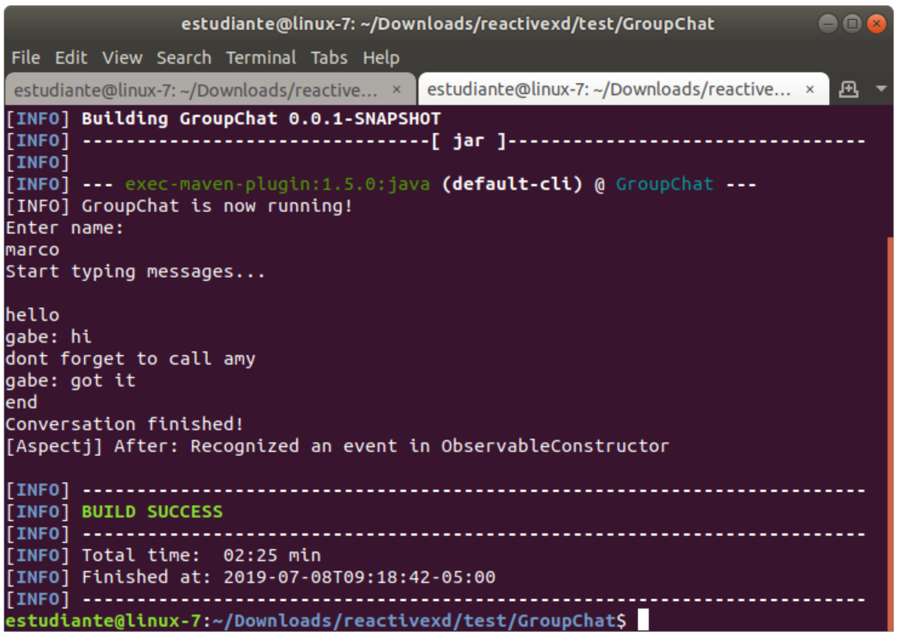
Figure 6.10: Message #3, from: marco to: gabe


estudiante@linux-7: ~/Downloads/reactivexd/test/GroupChat
File Edit View Search Terminal Tabs Help
estudiante@linux-7: ~/Downloads/reactive...
estudiante@linux-7: ~/Downloads/reactive...

```
Downloading from central: https://repo.maven.apache.org/maven2/org/codehaus/mojo
/exec-maven-plugin/1.5.0/exec-maven-plugin-1.5.0.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/codehaus/mojo/
exec-maven-plugin/1.5.0/exec-maven-plugin-1.5.0.pom (12 kB at 15 kB/s)
Downloading from central: https://repo.maven.apache.org/maven2/org/codehaus/mojo
/exec-maven-plugin/1.5.0/exec-maven-plugin-1.5.0.jar
Downloaded from central: https://repo.maven.apache.org/maven2/org/codehaus/mojo/
exec-maven-plugin/1.5.0/exec-maven-plugin-1.5.0.jar (53 kB at 4.9 kB/s)
[INFO]
[INFO] ------------------------< GroupChat:GroupChat >------------------------
[INFO] Building GroupChat 0.0.1-SNAPSHOT
[INFO] --------------------------------[ jar ]--------------------------------
[INFO]
[INFO] --- exec-maven-plugin:1.5.0:java (default-cli) @ GroupChat ---
[INFO] GroupChat is now running!
Enter name:
gabe
Start typing messages...

marco: hello
hi
marco: dont forget to call amy
got it
```

Figure 6.11: Message #4, from: gabe to: marco

Moreover, type **end** on a terminal to finish the chat session.

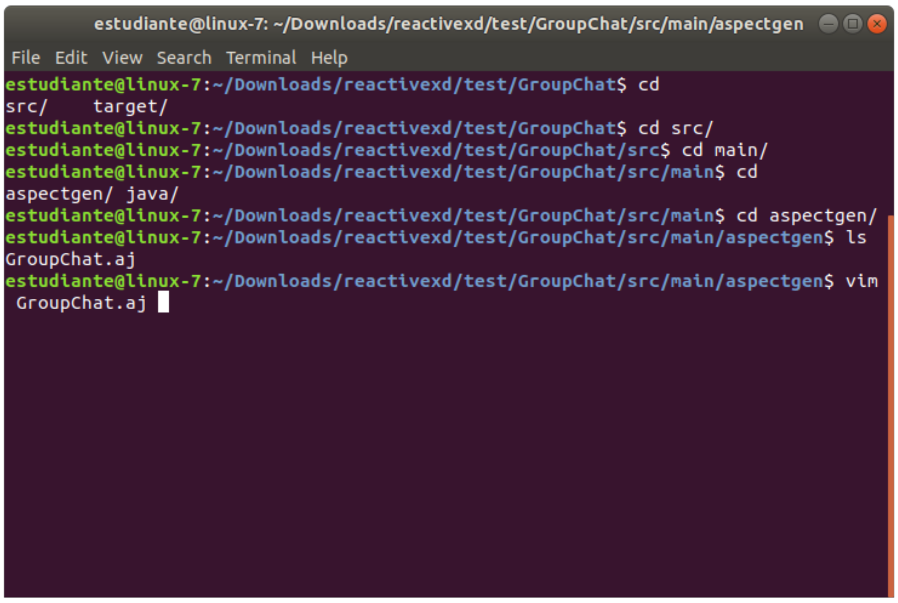

estudiante@linux-7: ~/Downloads/reactivexd/test/GroupChat
File Edit View Search Terminal Tabs Help
estudiante@linux-7: ~/Downloads/reactive...
estudiante@linux-7: ~/Downloads/reactive...

```
[INFO] Building GroupChat 0.0.1-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- exec-maven-plugin:1.5.0:java (default-cli) @ GroupChat ---
[INFO] GroupChat is now running!
Enter name:
marco
Start typing messages...

hello
gabe: hi
dont forget to call amy
gabe: got it
end
Conversation finished!
[Aspectj] After: Recognized an event in ObservableConstructor

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  02:25 min
[INFO] Finished at: 2019-07-08T09:18:42-05:00
[INFO] ------------------------------------------------------------------------
estudiante@linux-7:~/Downloads/reactivexd/test/GroupChat$
```

Figure 6.12: Message #5, from: marco to: gabe

As you may have noticed, after the **end** word is typed and entered, the message *"Conversation finished!"* appears but also another message appears.

What this message means, is that an AspectJ file has been created and of course an aspect representing the group chat session that just happened, this aspect is located on the following path.

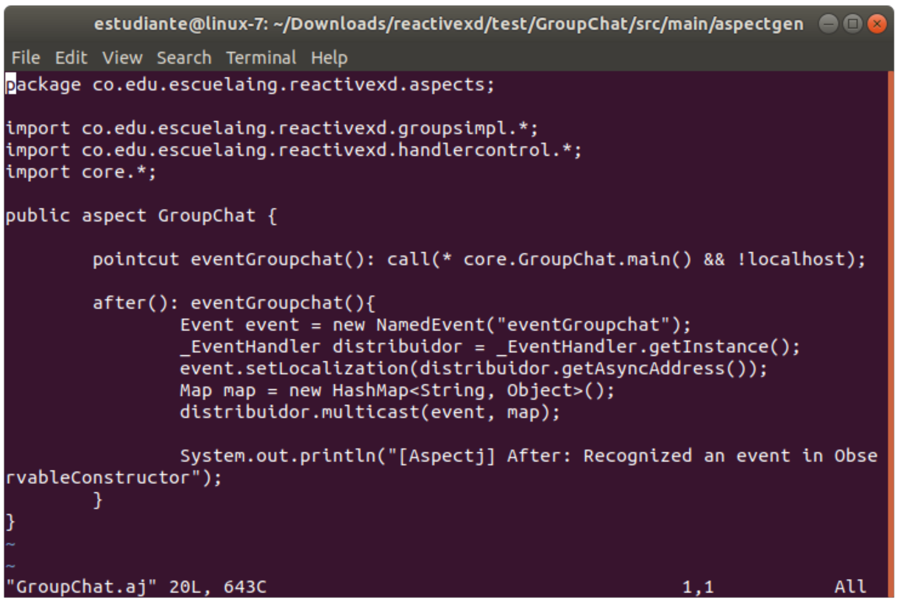


estudiante@linux-7: ~/Downloads/reactivexd/test/GroupChat/src/main/aspectgen

```
estudiante@linux-7:~/Downloads/reactivexd/test/GroupChat$ cd
src/    target/
estudiante@linux-7:~/Downloads/reactivexd/test/GroupChat$ cd src/
estudiante@linux-7:~/Downloads/reactivexd/test/GroupChat/src$ cd main/
estudiante@linux-7:~/Downloads/reactivexd/test/GroupChat/src/main$ cd
aspectgen/ java/
estudiante@linux-7:~/Downloads/reactivexd/test/GroupChat/src/main$ cd aspectgen/
estudiante@linux-7:~/Downloads/reactivexd/test/GroupChat/src/main/aspectgen$ ls
GroupChat.aj
estudiante@linux-7:~/Downloads/reactivexd/test/GroupChat/src/main/aspectgen$ vim
 GroupChat.aj
```

Figure 6.13: Path for the generated AspectJ file

By opening it with a text editor like Vim, we can take a look at its contents.



estudiante@linux-7: ~/Downloads/reactivexd/test/GroupChat/src/main/aspectgen

```
package co.edu.escuelaing.reactivexd.aspects;

import co.edu.escuelaing.reactivexd.groupsimpl.*;
import co.edu.escuelaing.reactivexd.handlercontrol.*;
import core.*;

public aspect GroupChat {

        pointcut eventGroupchat(): call(* core.GroupChat.main() && !localhost);

        after(): eventGroupchat(){
                Event event = new NamedEvent("eventGroupchat");
                _EventHandler distribuidor = _EventHandler.getInstance();
                event.setLocalization(distribuidor.getAsyncAddress());
                Map map = new HashMap<String, Object>();
                distribuidor.multicast(event, map);

                System.out.println("[Aspectj] After: Recognized an event in Obse
rvableConstructor");
        }
}
~
~
"GroupChat.aj" 20L, 643C                                      1,1           All
```

Figure 6.14: AspectJ generated code for chat

As previously explained, an aspect and a pointcut are created with the name of the class with only the first letter to uppercase. The aspect works as a *‘wrapper’* for the entire code and the pointcut is simply a point of interest where we want the code react before or after the execution of a specific method. The code inside the **after():** is concretely named an advice, that is, what the code is going to do “after” the pointcut occurs.

So, we were presented with simulation of a chat client for two nodes that exchanges messages in a temporal session, and that exchanging is translated into a generation and communication of events happening from one host to another on the same context where they could communicate and react to each other.

CHAPTER 7

Conclusion

In this document, it was shown the general implementation for ReactiveXD, a DSL built for distribution and reactiveness that focus on event transmission and helps the potential users of the language to be more productive in their real-time architecture and to have an immediate feedback on possible errors in a program or, in this case, a distributed application. Abstractions as Observables can drive the DSL characteristics such as distribution, concurrency, or data liveliness in order to propose sensible event constructs in a given program context. In this paper, we focused on a distributed asynchronous reactive language to test concurrency adopting some form of type interface. We presented a few general ideas from the reactive paradigm for implementing efficient systems along with focusing on data change over time. In this point of view, the system is able to build as many parts of the program as possible, keeping a good user experience. Observable messages were placed on important parts of the example programs, avoiding delays errors that can confuse the user. It was given a chat case study that: applies the presented patterns to implement a statically typed DSL and its origin from a simple expression reactive reduced Java-like language with AOP features. It was used Xtext as the language workbench for implementing the compiler and the IDE support for ReactiveXD and Xtend as a DSL extension for implementing type systems using a syntax that mimics formal systems. The patterns shown in the paper can be reused also for implementing languages with other language frameworks.

As future work, it is proposed to improve the language by including Observers in a explicit way (defined as a rule in the grammar and coded in the language semantics) so the overall compilation will be much more complete, sturdy and loyal to ReactiveX's principles. With the ambition to bring a new syntactic improvement to the language, it is also proposed to support Lambda Expressions in the event predicates. We will use Lambda Expressions because it increases the expressive power of Java by using functional interfaces which have only one abstract method that offers a simple and easily readable syntax reducing the number of lines and boilerplate code presented in a code artifact, also in compiling time will not create a .class file. Inevitably, we focus our language to be minimum but the main implementation is not altered, so the Lambda Expression is some sort of syntactic sugar to the language definition. It is also proposed to investigate better practices to design a DSL with code generation, robust validation and improved examples that show the full capability of the final software.

Bibliography

Bettini, L. (2016). *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd.

Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., and Tzoumas, K. (2015). “Apache flink: Stream and batch processing in a single engine”. In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* vol. 36, no. 4.

Courtney, A. (2001). “Frappé: Functional reactive programming in Java”. In: *International Symposium on Practical Aspects of Declarative Languages*. Springer, pp. 29–44.

Czaplicki, E. and Chong, S. N. (2013). “Asynchronous functional reactive programming for GUIs”. In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation-PLDI’13*. ACM Press.

Davis, A. L. (2019). “RxJava”. In: *Reactive Streams in Java*. Springer, pp. 25–39.

De Troyer, C., Nicolay, J., and De Meuter, W. (2018). “Building IoT Systems Using Distributed First-Class Reactive Programming”. In: *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, pp. 185–192.

Elliott, C. and Hudak, P. (1997). “Functional Reactive Animation”. In: *International Conference on Functional Programming*.

Hewitt, C., Bishop, P., and Steiger, R. (1973). “A universal modular actor formalism for artificial intelligence”. In: *Proceedings of the 3rd international joint conference on Artificial intelligence*. Morgan Kaufmann Publishers Inc., pp. 235–245.

Jeffrey, A. (2012). “LTL types FRP: linear-time temporal logic propositions as types, proofs as functional reactive programs”. In: *Proceedings of the sixth workshop on Programming languages meets program verification*. ACM, pp. 49–60.

Joseph, M. and Pandya, P. (1986). “Finding response times in a real-time system”. In: *The Computer Journal* vol. 29, no. 5, pp. 390–395.

Kambona, K., Boix, E. G., and De Meuter, W. (2013). “An evaluation of reactive programming and promises for structuring collaborative web applications”. In: *Proceedings of the 7th Workshop on Dynamic Languages and Applications*. ACM, p. 3.

Luckham, D. (2002). *The power of events*. Vol. 204. Addison-Wesley Reading.

Maglie, A. (2016). “ReactiveX and RxJava”. In: *Reactive Java Programming.* Springer, pp. 1–9.

Meyerovich, L. A., Guha, A., Baskin, J., Cooper, G. H., Greenberg, M., Bromfield, A., and Krishnamurthi, S. (2009). “Flapjax: a programming language for Ajax applications”. In: *ACM SIGPLAN Notices.* Vol. 44. 10. ACM, pp. 1–20.

Mosteo, A. (2017). “RxAda: An Ada implementation of the ReactiveX API”. In: pp. 153–166.

Paschke, A. and Kozlenkov, A. (2009). “Rule-based event processing and reaction rules”. In: *International Workshop on Rules and Rule Markup Languages for the Semantic Web.* Springer, pp. 53–66.

Peterson, J., Hudak, P., and Elliott, C. (1999). “Lambda in motion: Controlling robots with Haskell”. In: *International Symposium on Practical Aspects of Declarative Languages.* Springer, pp. 91–105.

Pu, C. (2011). “A world of opportunities: CPS, IOT, and beyond”. In: *Proceedings of the 5th ACM international conference on Distributed event-based system.* ACM, pp. 229–230.

Salvaneschi, G., Margara, A., and Tamburrelli, G. (2015). “Reactive programming: A walkthrough”. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering.* Vol. 2. IEEE, pp. 953–954.

Souryal, M. R., Wapf, A., and Moayeri, N. (2009). “Rapidly-deployable mesh network testbed”. In: *GLOBECOM 2009-2009 IEEE Global Telecommunications Conference.* IEEE, pp. 1–6.

Stivan, G., Peruffo, A., and Haller, P. (2015). “Akka. js: towards a portable actor runtime environment”. In: *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control.* ACM, pp. 57–64.

Voellmy, A., Kim, H., and Feamster, N. (2012). “Procera: a language for high-level reactive network control”. In: *Proceedings of the first workshop on Hot topics in software defined networks.* ACM, pp. 43–48.

Williams, B. C., Kim, P., Hofbaur, M., How, J., Kennell, J., Loy, J., Ragno, R., Stedl, J., and Walcott, A. (2001). “Model-based reactive programming of cooperative vehicles for mars exploration”. In: *Int. Symp. Artif. Intell., Robotics, Automation Space (ISAIRAS-01), Montreal, QB, Canada.* Vol. 2001. Citeseer.