Reactive Programming: a Walkthrough

Guido Salvaneschi
Technische Universität Darmstadt
salvaneschi@cs.tu-darmstadt.de

Alessandro Margara
Università della Svizzera italiana (USI)
alessandro.margara@usi.ch

Giordano Tamburrelli
Vrije University
g.tamburrelli@vu.nl

***Abstract*—Over the last few years, Reactive Programming has emerged as the trend to support the development of reactive software through dedicated programming abstractions. Reactive Programming has been increasingly investigated in the programming languages community and it is now gaining the interest of practitioners. Conversely, it has received so far less attention from the software engineering community.**

This technical briefing bridges this gap through an accurate overview of Reactive Programming, discussing the available frameworks and outlining open research challenges with an emphasis on cross-field research opportunities.

I. Motivation

Many modern software systems are *reactive*: they respond to the occurrence of events of interest by performing some computation, which may in turn trigger new events. Examples range from graphical user interfaces, which react to the input of the users, to embedded systems, which react to the signals coming from the hardware, to monitoring and control applications, which react to the changes in the external environment.

Designing, implementing, and maintaining reactive software is arguably difficult. Indeed, reactive code is asynchronously triggered by event occurrences. Because of this, it is hard to trace and understand the control flow of the entire system [1].

Reactive Programming (RP) [2] is a recent programming paradigm that supports the development of reactive applications through dedicated language abstractions. It is based on concepts like time-varying values (a.k.a. *signals* or *behaviors*), events streams to model discrete updates, automatic tracking of dependencies, and automated propagation of change.

It is our belief that effective and well engineered reactive systems require the combined efforts from practitioners as well as researchers on programming language design and software engineering.

Practitioners drive the requirements that RP languages and framework should fulfill. Researchers on programming languages can integrate appropriate RP abstractions into languages to ease the development of reactive applications. Finally, the software engineering community can conceive frameworks and methodologies to successfully integrate RP into the software development lifecycle.

RP has been increasingly investigated in the programming languages community (e.g., [3]). Recently, initiatives like the *Reactive Manifesto* [4] – yet considering RP in a significantly broader sense compared to the research community – witness the gaining interest of practitioners. Conversely, RP has received so far less attention from the software engineering community.

This technical briefing moves from the above premises with a twofold purpose: (*i*) present and promote RP concepts, challenges, and implementations to the software engineering community; (*ii*) stimulate innovative contributions in the design and development of RP frameworks and reactive applications.

II. RP in a Nutshell

The traditional approach to implement reactive applications is the *Observer* design pattern that decouples event consumers (observers) from event producers (observables). However, such solution has been criticized for a long time (e.g., [1], [3]). beacuse of lack of composability, inversion of the logical relation among reactive entities and limited readability.

RP has been proposed to address these issues. To make the discussion more concrete, we show an example in of how RP models time-varying values, tracks dependencies, and automates change propagation. The following pseudocode snippet receives the mouse.clicked event and the current mouse.position time-changing value.

```
1 val clicked: Event[Unit] = mouse.clicked
2 val scaledPosition: Signal[(Int,Int)] = mouse.position ∗ 0.5
3 val lastClick: Signal[(Int,Int)] = clicked.snapshot(scaledPosition)
```

The position of the mouse is scaled by a 0.5 factor (Line 2) and a snapshot of the position if taken every time the clicked event occurs (Line 3). In conventional imperative programming, any future change to the value of mouse.position does not impact the value of scaledPosition. In RP, instead, Line 2 defines a *constraint* in the form y=f(x) rather than a statement. The runtime identifies the dependency between scaledPosition and mouse.position in Line 2 and ensures that scaledPosition gets constantly updated to reflect the latest value of mouse.position. Similarly, Line 3 defines a time-changing value containing the scaled mouse position at the time of the last click. Also, Line 3 demonstrates the practical need to interface time-changing values like scaledPosition with discrete events like clicked.

The potential of this mechanism is surprising. For example, from an architectural viewpoint, the y and x entities in a constraint can be, respectively, (part of) the View and the Model in a MVC architecture. With RP, the View is automatically updated by the runtime to reflect changes in the Model. As a result, there are no bugs because of forgotten updates, no redundant computations in case programmers code defensively and update *too much* regardless it is necessary, and applications are easily extensible as constraints can be *composed*, i.e., built on top of other constraints.

Historically, RP was first proposed in the context of strictly functional languages. Specifically, functional reactive programming was first introduced in Haskell to support interactive animations [5]. After the first experiences in Haskell, RP has reached a wider audience being implemented in Scheme [6], Javascript [3] and Scala [7], [8]. Since then, RP has become increasingly popular. Concepts inspired by RP have been applied to Microsoft Reactive Extensions (Rx) [9] and stimulated a significant number of novel popular front-end libraries such as React.js, Bacon.js, Knockout, Meteor, and Reactive.coffee.

III. Outline

The technical briefing starts with a description of the requirements coming from modern reactive applications. It introduces RP providing first an overview of its building blocks and then diving into more advanced topics. Next, the briefing shows some practical examples of the most relevant off-the-shelf solutions for RP, pointing out their potentials and differences. Finally, it discusses open challenges in the RP domain from a software engineering perspective and outlines synergies with related fields such as Cloud Computing and Big Data [10]. The detailed outline of the technical briefing is discussed hereafter.

1. *Reactive applications: why and where?* We show the pervasiveness of reactive applications in modern software systems. Such applications range from Graphical User Interfaces to distributed monitoring and control systems, from specific Internet of Things scenarios to generic embedded systems.

2. *Traditional approaches for reactivity.* Before the advent of RP, the demand for reactivity drove to the development of dedicated approaches. In this part of the briefing, we present traditional approaches for reactivity and we highlight their limitations and how they motivated the development of RP.

3. *RP for beginners.* We present the core abstractions offered by RP, namely time-varying variables, automated detection of dependencies and propagation of updates. We discuss the main benefits and the guarantees offered by this new programming paradigm through comparative examples. We show how new abstractions interface with existing solutions (e.g., events).

4. *Advanced RP.* In this part of the briefing, we explore more advanced concepts. For example, we present the main algorithms to support automated propagation of changes in RP. We discuss the integration of RP with modern programming languages, with an emphasis on the Object Oriented paradigm. We present the benefits and challenges of exploiting RP for distributed systems and mobile enviroments.

5. *RP frameworks overview.* This part of the briefing demos three state of the art frameworks for RP: REScala [7], Rx [9], and DREAM [11]. We discuss and compare the expressiveness of these frameworks relying on simple but realistic reactive applications.

6. *Open challenges* We conclude the briefing with an interactive discussion on the open research and development challenges in the area of RP. In particular, the discussion will cover the integration of RP abstractions within programming languages, the development of RP frameworks to support distributed systems with heterogeneous requirements, and the integration of RP within the software development lifecycle. Finally, we will introduce the similarities with data-flow based computational models currently adopted in Cloud Computing and Big Data analytics and discuss the perspective on the convergence of these fields.

IV. About the Organizers

Guido Salvaneschi is a postdoctoral researcher at Technical University of Darmstadt. He works on language design, especially for reactive applications. He developed the reactive language REScala and (co-)authored papers on RP published in OOPSLA, FSE, MODULARIY and DEBS.

Alessandro Margara is a postdoctoral researcher at Università della Svizzera italiana (USI). He works in the area of distributed systems focusing on middleware for distributed event-based systems. He developed the distributed RP framework DREAM and (co-)authored papers on event-based and reactive systems published in the IEEE Transactions of Parallel and Distributed Systems, ICDCS, Middleware and DEBS.

Giordano Tamburrelli is assistant professor at Vrije University in Amsterdam (VU), previously he has been Marie Curie Fellow at Università della Svizzera italiana (USI). He works in the area of software engineering with a focus on self-adaptive systems and service oriented architectures. He (co-)authored papers on interactive and mobile systems published in ICSE, FSE, and other relevant conferences of the sector.

Acknowledgments

This work is supported by the German Federal Ministry of Education and Research (BMBF), grant No. 01IC12S01V.

References

[1] I. Maier, T. Rompf, and M. Odersky, "Deprecating the Observer Pattern," Tech. Rep., 2010.

[2] E. Bainomugisha, A. L. Carreton, T. Van Cutsem, S. Mostinckx, and W. De Meuter, "A survey on reactive programming," *ACM Comput. Surv.*, 2012.

[3] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi, "Flapjax: a programming language for Ajax applications," ser. OOPSLA '09. New York, NY, USA: ACM, 2009, pp. 1–20.

[4] Http://www.reactivemanifesto.org/.

[5] C. Elliott and P. Hudak, "Functional reactive animation," ser. ICFP '97. New York, NY, USA: ACM, 1997, pp. 263–273.

[6] G. H. Cooper and S. Krishnamurthi, "Embedding dynamic dataflow in a call-by-value language," ser. ESOP'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 294–308.

[7] G. Salvaneschi, G. Hintz, and M. Mezini, "Rescala: Bridging between object-oriented and functional style in reactive applications," ser. MODULARITY '14. New York, NY, USA: ACM, 2014, pp. 25–36.

[8] I. Maier and M. Odersky, "Higher-order reactive programming with incremental lists," ser. ECOOP'13. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 707–731.

[9] J. Liberty and P. Betts, *Programming Reactive Extensions and LINQ*, 1st ed. Berkely, CA, USA: Apress, 2011.

[10] G. Salvaneschi, P. Eugster, and M. Mezini, "Programming with implicit flows," *Software, IEEE*, vol. 31, no. 5, pp. 52–59, Sept 2014.

[11] A. Margara and G. Salvaneschi, "We have a DREAM: Distributed reactive programming with consistency guarantees," ser. DEBS '14. New York, NY, USA: ACM, 2014, pp. 142–153.