REScala: Bridging Between Object-oriented and Functional Style in Reactive Applications

Guido Salvanesci  
Software Technology Group  
Technische Universität Darmstadt  
salvanesci@informatik.tu-darmstadt.de

Gerold Hintz  
Technische Universität Darmstadt  
gerold.hintz@stud.tu-darmstadt.de

Mira Mezini  
Software Technology Group  
Technische Universität Darmstadt  
mezini@informatik.tu-darmstadt.de

Abstract
Traditionally, object-oriented software adopts the Observer pattern to implement reactive behavior. Its drawbacks are well-documented and two families of alternative approaches have been proposed, extending object-oriented languages with concepts from functional reactive and dataflow programming, respectively event-driven programming. The former hardly escape the functional setting; the latter do not achieve the declarativeness of more functional approaches.

In this paper, we present REScala, a reactive language which integrates concepts from event-based and functional-reactive programming into the object-oriented world. REScala supports the development of reactive applications by fostering a functional declarative style which complements the advantages of object-oriented design.

Categories and Subject Descriptors D.1.5 [Software]: Programming Techniques—Object-oriented Programming; D.3.3 [Programming Languages]: Language Constructs and Features

General Terms Languages, Design

Keywords Functional-reactive Programming; Scala; Event-driven Programming

1. Introduction
Reactive applications are an important class of software systems. In these applications, events or state changes, e.g., user interaction, data changes in a Model-View-Controller design, network messages, value acquisition from sensors, etc., trigger computations, which may in turn update the state of the system, eventually triggering new events and/or computations. Even if reactive systems have been studied for a long time, they are still difficult to design and maintain. At the code organization level, proper modularization is hard to achieve because reactions involve cross-module entities and must be triggered in several places in code. At runtime, the normal control flow is interleaved with reactions to events, leading to interactions that are hard to foresee.

Object-oriented (OO) reactive applications traditionally adopt the Observer pattern [13], which relies on the concept of inversion of control [14] to decouple the observers from observables. Other than that, the pattern does not contribute much to managing the complexity of reactive systems and has been criticized for cluttering code and hindering composability of reactions [19].

Two classes of alternative approaches have emerged to address the complexity of reactive applications. The first class includes languages that support event-driven programming at the language level. Examples are C# [8], Ptolemy [29], EventJava [11], ESCala [15], DominOJ [35]. These languages provide first-class representation for events; some of them support expressive event models with advanced features like quantification, implicit events and event correlation. We refer to this class as event-based languages. The second class includes languages with direct representation of reactive values and means to compose computations based on them through dedicated abstractions. The ideas around reactive values were originally explored by synchronous dataflow languages [3, 28] and functional-reactive programming (FRP) [10]. More recently, these concepts have been proposed in a more modern flavor in reactive languages like Scala.React [19], FrTime [6], and FlapJax [25]. We refer to this class as reactive languages.

Both classes have their tradeoffs, which calls for an integration of their concepts. Event-based languages nicely integrate with OO design, support OO modularity, encapsulation, late binding and fine-grained updates of object state, but do not achieve the declarative style and the level of expressiveness of reactive languages. With reactive languages, dependencies are defined in a more declarative way and updates are automatically performed by the runtime. But these languages do not fit well into the OO setting. Reactive abstractions do not support fine-grained changes to objects: Objects must be recomputed from scratch, a constraint that enforces immutability and does not integrate with OO modifiable state. In addition, events are still desirable, since they model certain phenomena in a direct and intuitive way.

In this paper, we present a language design that seamlessly integrated reactive values with an advanced event system. Thanks to this solution, it is possible to exploit the benefits of reactive abstractions without losing the advantages of OO design. In our design, both events and reactive values are object attributes in addition to fields and methods and exposed as part of the object interface. Crucially, the design comes with a rich library of operations (API) for bridging the gap between the worlds of events and reactive values making them composable to support a mixed OO and functional style.

We implemented these ideas in REScala, a reactive language based on Scala. Building upon existing approaches for event-driven and reactive programming, the key new contribution of REScala, its added value, is the unification of imperative, modular events and
reactive values making them composable to support a mixed OO and functional style in designing reactive systems. To the best of our knowledge such a unification has not been proposed before.

To summarize, in this paper, we make the following contributions:

- We provide an analysis of language-level support for reactive applications focusing on event systems and reactive values. We investigate their tradeoffs and how these abstractions relate to the OO and to the functional paradigms.

- We present the design of RESCALA, a language which combines signals and events and supports a mixed functional and imperative style. Thanks to the fluid integration of events and signals, RESCALA raises the level of abstraction in reactive applications, and promotes a gradual migration to a more declarative style.

- We provide a usable implementation of the language and apply RESCALA in several case studies. We demonstrate the crucial role of RESCALA’s conversion functions by refactoring four OO reactive applications. We introduce more than 90 signals and show the improvement of the resulting design.

The paper is organized as follows. Section 2 motivates the work analyzing the limitations of signals and events taken singularly. Section 3 presents the design of RESCALA. Section 4 describes our implementation. Section 5 validates our contribution with case studies. Section 6 presents related work. Section 7 concludes and outlines areas of future research.

2. Problem Statement

Traditionally, OO applications implement reactivity by using the Observer pattern. The limitations of this approach have been analyzed elsewhere [19, 25]. For convenience, we briefly summarize them. First, dependencies are not directly specified but rather established by inversion of control – this reverses the intuitive flow of the applications and makes code harder to understand and analyze. Additionally, a lot of boilerplate code is required to implement even elementary functionalities, which further complicates program comprehension. More importantly, separation of concerns is hard to achieve because reactive functionalities are mixed with the application logic. Since callbacks do not return a value, they are not composable, limiting extensibility and reuse and program comprehension cannot be guided by types. Finally, callbacks enforce exclusively an imperative programming style, since reaction is performed via side effects.

Event-based languages have emerged to address these limitations providing abstractions for event-based programming [11, 15, 29]. In this section, we review these approaches with their limitations and motivate the need for complementing event-based languages with abstractions for reactive values, in the spirit of FRP and dataflow programming [6, 7, 10, 19, 22, 25].

2.1 Event-based Languages

Languages in this class, like C#, EventJava [11], Ptolemy [29] and EScala [15] provide dedicated abstractions for events and event-driven interactions. Since RESCALA extends EScala, we take the latter as representatives of event-based languages to investigate their limitations.

Event abstractions and their advantages. EScala [15] combines concepts from OO and AOP. Beside imperative events, EScala supports implicit events. In the style of AOP, implicit events allow one to capture points in the execution of the program by the after(method) and before(method) pointcuts without having to explicitly trigger events at the boundaries of method executions, which is tedious and error-prone.

EScala also supports declarative events, which are defined as a combination of other events. For this purpose it offers operators like e1 | e2 (occurrence of one among e1 or e2), e1 & k e2 (e1 occurs and the predicate p is satisfied), e1 . map( f ) (the event obtained by applying f to e1 ). Event composition allows one to express the application logic in a clear and declarative way. Also, the update logic is better localized because a single expression models all the sources and the transformations that define an event occurrence. Compared to EventJava and Ptolemy, EScala takes a more object-centric view. Events are part of the interface of a class, so event-driven behavior nicely integrates with OO data abstraction, inheritance, and subtype polymorphism.

In Figure 1, we show a slice of a drawing application in EScala. The Figure class defines an implicit event after(moveBy), automatically triggered at the end of the execution of the moveBy method. The declarative event changed is triggered when one of the events resized, moved, or after(setColor) is triggered. The declarative event invalidated is defined as a transformation of the event changed. Handlers are registered and unregistered to events with the += and -= notation (cf. Line 12 in Figure 1). Events are explicitly triggered by the event() notation. EScala events integrate with objects in several ways. Events support visibility modifiers (Line 2), abstract events can be refined in subclasses (Line 19). Events can be overridden in subclasses (Line 20) and the inherited definitions can be accessed by super. Finally events are late-bound: For example in Line 12 if start refers to a RectangleFigure, the definition of changed in RectangleFigure is chosen.

Limitations of event abstractions. While event-based languages address several issues of reactive software, several drawbacks are still in place. The application control flow is still inverted, since updates are performed only indirectly by event handlers that return void and do not support composition. The definition of the events and of the reactions to them (the update logic) are separated, making dependencies hard to grasp in code. More generally, event handlers update the object state in an imperative way. Thus, side effects are inherent to those event models, which limits the migration to a more functional style.

Triggering an event in every point in the code where a variable on which other variables depend on is updated leads to code scatter-
2.2 Reactive Languages

Some of the issues with event-based languages are addressed by abstractions for reactive values provided by reactive languages. In the following, we briefly present the concept as it is supported by some contemporary languages and discuss its advantages. Subsequently, we focus on the limitations of this concept compared to events, which motivates the need for improving event-based languages with reactive abstractions instead of abandoning events.

A reactive value, a.k.a. **behavior** in FrTime [6] and Flapjax [25], or **signal** in Scala.React [19], is a language concept for expressing functional dependencies among values in a declarative way. Intuitively, a reactive value can depend on variables – sources of change without further dependencies – or on other reactive values. When any of the dependency sources changes, the expression defining the reactive value is automatically recomputed by the language runtime to keep the reactive value up-to-date. In this paper, we focus on signals, an abstraction for reactive values introduced by Scala.React [19], a library implementing reactive abstractions for Scala.

To give an intuition of signals and their advantages over events, we use code extracts from a program that simulates a 2D environment, called the Universe application, which we used as a case study for EScala [15]. The environment is populated by animals and plants; the simulation involves growing of animals and plants, movements of animals, and planning for food search. A tick represents a simulation step equivalent to an hour in the simulation time; elapsed hours, days, and weeks must be updated accordingly.

In Figure 2, we show side-by-side two code fragments that use EScala events (a) and Scala.React signals (b) to model the elapsed time.

### Figure 2: Simulation of Elapsed Time (a) with Events and (b) with Signals.

```scala
val hour: Int = 0
val day: Int = 0
val week: Int = 0

tick += nextHour

def nextHour() {
  hour = (hour + 1) % 24
  
  if (hour == 0) {
    val newDay = new Day()
    nextDay += newDay
  }
}

def nextDay() {
  newDay += newWeek
  week += nextWeek

val newWeek = nextWeek
val hour = Signal(tick()) % 24
val week = Signal(tick()/24) % 7 + 1
}

def nextWeek() {
  val new = new Week
  week += new
}

// (a)

// (b)
```

Signals (Figure 2b) enable the programmer to specify only the entities that are really part of the application logic: The tick, the hour, the day, and the week values. Each of them is declared together with its definition in terms of the other entities (for example, hour is defined in terms of ticks, Figure 2b, Line 2). The Scala.React library transparently performs all the necessary updates along the dependency chain of values declared as signals, c.g., to update the value of hour, when the value of tick changes. No additional programming logic is needed for these updates.

On the contrary, modeling dependent time-changing values by using events (Figure 2a) requires to introduce **artificial** entities (like the newDay event, the newWeek event, and the nextDay and the nextWeek callbacks). As a result, the code is much more complex. In addition, boilerplate code is introduced to register events (Lines 11 and 16), the definition of each entity is separated from declaration, and the application logic is spread among event definitions and callbacks. For example, the logic of day, declared at Line 3, is spread between Line 10, and Line 13.

Generally, by using signals, functional dependencies are expressed in a direct and declarative way. In contrast to the event-based reactivity, dependencies are not inverted. Since each reactive element is defined on the basis of its depending values, signals capture the design intention of the programmer; dependencies among reactive entities are automatically tracked and the runtime is in charge of keeping depending values updated. Another advantage, compared to inversion of control, is that the definition of the reactive behavior is not separated from the source of the change. As a result, reactive code is clearer and easier to read. Furthermore, since signals are reactive values themselves, new signals can be defined as dependents on existing ones. Signals composition fosters rapid implementation of new reactive functionalities and code reuse. Finally, signals identify dependencies which can be used to transparently cache the computed values.

2.3 Need for Complementing Events with Signals

While reactive values can model a computation in a simple and elegant way, they are not enough alone.

First, events are a well-established programming model in the OO community, they properly integrate with OO [15] and OO programmers are unlikely to refrain from using them.

Second, most of existing OO reactive applications are event-based – graphical libraries being probably the most widespread example. Rewriting all the existing event-based software to use signals is probably unfeasible.

Third, events are **conceptually** the correct way of modeling phenomena that happen at a point in time. For example, the reception of a network packet could be modeled by a signal that has an **option** type. The signal evaluates to **Some** when no packet is available and to **Some[Packet]** when a packet arrives (Figure 3, Line 1). It is clear, however, that a programmer would be only interested in the change of such a signal, making the use of an event much more suitable for this case (Figure 3, Line 3).

```scala
val packet: Signal[Option[Packet]] = Signal(...)

ev packetReceived[Packet] = ...
```

Finally, reactive values have been designed in functional languages where they are applied to immutable (typically primitive) values. As such, they conflict with mutable state and incremental computation. For example, a signal of a complex value such as `Signal[aList.filter(_ > 10): List[T]]` recomputes the filter function for all the elements of the list `aList`. Every time an element is added to `aList` – with a clear loss of performance. While there are attempts to incrementalize such computations they only work for certain operations and are limited the specific domain of data structures [20]. Instead, events are applicable in general and can be generated by partial modifications of objects (like the insertion of an element into a list). On the receiver part, objects can
1 val age = 0
2 val size = 1
3
4 def canLive: Boolean =
5   (age <= maxAge) && (size <= 3000) && (size >= 1)
6 }
7 def disease() = { age += 2 }
8
9 evt shouldDie[Unit] =
10  (after(getOlder) || after(grow) || after(disease)) &&
11  (!canLive.changed && !canLive() && killed }

Figure 4: Dependencies with Implicit Method Events (a). Dependencies as Signals (b).
value exposed by the most recent occurrence of \( e \). For illustration, consider the code snippet in Figure 5, where a signal `click.bind()` is built to represent the last position in which the mouse was clicked. Once defined, this signal encapsulates the imperative event and can be composed with other signals and mutable values into more complex signals. In Line 5, the mouse position is combined with a circle that changes its position on the screen — modeled as a var (Line 3) — to detect if the last click was on the circle.

\[
\text{val circle: Var[(Int, Int), Int] = Var((1,1),10)~}
\]

\[
\text{val lastClick: Signal[(Int, Int)] = mouse.lastClick~}
\]

\[
\text{val lastClickOnCircle: Signal[Boolean] = Signal(over(circle.bind(), circle))~}
\]

\[
\text{val nClick: Event[(Int, Int)] = mouse.click~}
\]

The conversion of events to signals by `bind` is stateless in the sense that at any point in time the value of the resulting signal is independent of that signal’s previous history. For example the signal `click.bind()` in Figure 5 (Line 5), does not remember previous positions of the mouse. To model situations when the value of a signal needs to depend on its previous values, RESCALA provides functions for stateful conversion of events to signals — in the following, we discuss three such functions: `fold`, `list`, and `last`. For illustration, suppose that we want to create a reactive value to keep track of the number of mouse clicks. A possible encoding based on events and reactions to events is shown in Figure 6a. The variable `nClick` records the number of observed mouse clicks; it is imperatively updated on any occurrence of the event `click` by the reaction attached to that event (line 4). A signal can then rely on `nClick` to react to the cumulative value. This solution has a number of drawbacks. First, the design is unnecessarily complex because it requires to register an imperative callback when a functional definition is possible. Second, it exposes the state in the `nClick` variable, so the programmer can accidentally modify its value.

\[
\text{val click: Event[(Int, Int)] = mouse.click~}
\]

\[
\text{val nClick: Var(0)~}
\]

\[
\text{click += 1~}
\]

\[
\text{Figure 5: hold at Work.}
\]

The conversion of events to signals by `fold` is stateless in the sense that at any point in time the value of the resulting signal is independent of that signal’s previous history. For example the signal `click.bind()` in Figure 5 (Line 5), does not remember previous positions of the mouse. To model situations when the value of a signal needs to depend on its previous values, RESCALA provides functions for stateful conversion of events to signals — in the following, we discuss three such functions: `fold`, `list`, and `last`. For illustration, suppose that we want to create a reactive value to keep track of the number of mouse clicks. A possible encoding based on events and reactions to events is shown in Figure 6a. The variable `nClick` records the number of observed mouse clicks; it is imperatively updated on any occurrence of the event `click` by the reaction attached to that event (line 4). A signal can then rely on `nClick` to react to the cumulative value. This solution has a number of drawbacks. First, the design is unnecessarily complex because it requires to register an imperative callback when a functional definition is possible. Second, it exposes the state in the `nClick` variable, so the programmer can accidentally modify its value.

\[
\text{val click: Event[(Int, Int)] = mouse.click~}
\]

\[
\text{val nClick: Var(0)~}
\]

\[
\text{click += 1~}
\]

\[
\text{Figure 5: hold at Work.}
\]

\[
\text{val circle: Var[(Int, Int), Int] = Var((1,1),10)~}
\]

\[
\text{val lastClick: Signal[(Int, Int)] = mouse.lastClick~}
\]

\[
\text{val lastClickOnCircle: Signal[Boolean] = Signal(over(circle.bind(), circle))~}
\]

\[
\text{val nClick: Event[(Int, Int)] = mouse.click~}
\]

\[
\text{in Figure 6a in a more concise declarative way. nClick is now encoded by accumulating converting the event click to a signal. The initial value for the accumulation is 0, while the accumulation is encoded in the lambda passed as the second parameter to fold.}
\]

Unlike `fold` that composes the values in a signal’s history, functions `list` and `last` just collect them into lists. Given an event `e`, the call `e.list()` returns a signal modeling the whole list of values produced by occurrences of `e`, while the `e.last(n)` returns a signal modeling a sliding window over the last `n` values exposed by occurrences of `e`. In Figure 7, `list` and `last` are used to reify into signals the complete history of the positions of mouse clicks (Line 2), respectively a sliding window over the last 5 values (Line 3). The definition of the `mean` signal (Line 5) illustrates how signals defined by `list` and `last` over the click event can be used in the definition of more complex signals; `mean` computes the average position over the last 5 clicks.

\[
\text{val click: Event[(Int, Int)] = mouse.click~}
\]

\[
\text{val nClick: Var(0)~}
\]

\[
\text{click += 1~}
\]

\[
\text{Figure 6: Tracking State with Events (a) and Stateful Signals with fold (b).}
\]

The initial value for the accumulation is 0, while the accumulation is encoded in the lambda passed as the second parameter to `fold`. Unlike `fold` that composes the values in a signal’s history, functions `list` and `last` just collect them into lists. Given an event `e`, the call `e.list()` returns a signal modeling the whole list of values produced by occurrences of `e`, while the `e.last(n)` returns a signal modeling a sliding window over the last `n` values exposed by occurrences of `e`. In Figure 7, `list` and `last` are used to reify into signals the complete history of the positions of mouse clicks (Line 2), respectively a sliding window over the last 5 values (Line 3). The definition of the `mean` signal (Line 5) illustrates how signals defined by `list` and `last` over the click event can be used in the definition of more complex signals; `mean` computes the average position over the last 5 clicks.

\[
\text{val click: Event[(Int, Int)] = mouse.click~}
\]

\[
\text{val nClick: Var(0)~}
\]

\[
\text{click += 1~}
\]

\[
\text{Figure 6: Tracking State with Events (a) and Stateful Signals with fold (b).}
\]

What is actually needed is a way to bridge between events and signals in a stateful way, i.e., an operation that turns events into signals whose actual values depend on their past values. This is what the `fold` function in RESCALA’s conversion API offers. With the `fold` function the programmer directly specifies how the value of a signal, that captures occurrences of an event, functionally depends on its past values. An initial value can be assigned, otherwise at the beginning the `fold` function evaluates to null. For illustration, Figure 6b shows a code snippet that uses `fold` to encode the logic in Figure 6a in a more concise declarative way. `nClick` is now encoded by accumulating converting the event `click` to a signal. The initial value for the accumulation is 0, while the accumulation is encoded in the lambda passed as the second parameter to `fold`.

1 In real programming practice, one would probably encapsulate this feature in a signal tracking the position of the last click, directly available in the mouse interface (Line 6).

2 The `unzip` function takes a list of pairs and returns a pair of lists. Given the input list \( [(l_i, r_i), i \in (0..n)] \), `unzip` returns the lists \( [l_i, i \in (0..n)] \) and \( [r_i, i \in (0..n)] \).
time the value of the signal is updated, enabling a to engage in composite event expressions. For illustration, consider the code snippet in Figure 4b. The refactored definition of canLive as a signal must be integrated with the rest of the application, which is in an event-driven style. Specifically, the canLive signal and the killed event need to be composed in the definition of the complex event shouldDie. This is achieved by using canLive.changed() to bridge the worlds of signals and of the events (Line 10).

In addition to changed(), RESCALA provides functions for more sophisticated integration of signals into event-driven computations. In the following, we discuss two such operations: snapshot and toggle. The snapshot function takes the instant value of a signal whenever an event occurs. The toggle function switches back and forth between two expressions of a signal when an event is raised. In the following, we motivate and illustrate these functions by examples.

To show the use of the snapshot function we further decompose the interface of the mouse object. Like in the previous examples, the signal mouse.position models a cursor’s current position, but now the event mouse.clicked carries no value and only models clicks form the user (Figure 9a). The snapshot function is applied to the signal mouse.position (Line 3) to sample the position of the mouse whenever the user clicks the button\(^3\). For comparison, Figure 9b, shows the same functionality implemented without the snapshot function.

\[
\begin{align*}
\text{def clicked: Event[Unit] = mouse.clicked} \\
\text{val position: Signal[[Int,Int]] = mouse.position} \\
\text{val lastClick: Signal[[Int,Int]] = position snapshot clicked} \\
\end{align*}
\]

(a)

\[
\begin{align*}
\text{def clicked: Event[Unit] = mouse.clicked} \\
\text{val position: Signal[[Int,Int]] = mouse.position} \\
\text{val lastClickPos = Var(0,0)} \\
\text{val lastClick: Signal[[Int,Int]] = Signal{ lastClickPos() }} \\
\text{clicked = () => lastClickPos().position()} \\
\end{align*}
\]

(b)

Figure 9: snapshot at Work (a). Tracking the Position of Last Click without snapshot (b).

The situation becomes worse when more reactive values are involved. For illustration, consider an application that in reaction to an event occurrence does not simply take a static snapshot of a reactive value, but needs to switch between two reactive values \(a\) and \(b\) returning alternatively one of them. This is the case e.g., with a graphical application that models a bouncing ball. When the ball reaches a border, the xBounce or the yBounce event occur and the moving direction of the ball needs to be inverted. Compared to the simple snapshotting discussed above, without proper support (Figure 10a), the developer would have even more complex callback logic (Lines 14-16). The information about the currently active reactive value (e.g. postSpeedX or negSpeedX) needs to be explicitly tracked (Lines 11-12); an update of this information would also be needed every time the event fires. Finally, the programmer has to implement the switching logic (Lines 8-9). In summary, interfacing events and signals by such a low-level programming activity basically would annihilate the advantages of reactive values.

This accidental complexity can be avoided by using RESCALA’s toggle function. For illustration, consider the code snippet in Figure 10b, where toggle is used in the context of the graphical application that models a bouncing ball. The inversion of the moving direction is encoded by switching the expression of the speedX and speedY signals (Lines 3-4), from speed.x to -speed.x, respectively from speed.y to -speed.y, whenever the events xBounce, respectively yBounce are raised.

3.2.3 Lifting Functions on Ordinary Values to Functions on Signals

To support gradual refactoring of applications to a more declarative style, it is fundamental that existing code can be reused with the abstractions introduced by RESCALA. To enforce compatibility of reactive abstractions with existing components, RESCALA provides conversions that lift a value to the reactive counterpart. The Signal.lift(f) function converts a function \(f: A\Rightarrow B\) to a function operating on a reactive value Reactive\([A]\) (either a signal or a var) and returning a Signal\([B]\). As a result, computations expressed by traditional functions that operate on traditional values can be turned into reactive computations operating on reactive values. In Figure 11, we show how the mean over the last mouse click positions – presented in Figure 7 – can be encoded by leveraging a regular mean function working on non-reactive values. The function is lifted (Line 6) and then applied on the reactive values (Line 9).

While we expect that most of the conversions required by programmers are meant to use existing non reactive functions with reactive values, RESCALA also supports the conversion in the opposite direction. When a function expecting a reactive value is applied to a traditional value, the value is automatically promoted – by using Scala’s implicit conversions – to guarantee type compatibility.

4. Implementation

RESCALA is implemented as a completely new Scala library. The user API of RESCALA provides both signals and events and subsumes the event-based ESCala interface. To explain why a complete reimplementation is needed we briefly summarize the mechanism behind ESCala events. The ESCala event system is based on an event graph connecting dependent events. Imperative events and implicit events are the nodes without a predecessor, declarative events form the rest of the graph. For example, if the e3 declarative event is defined by \(e^3 = e_1 \lor e_2\), e1 and e2 are connected to e3 in the graph. Each node maintains a list of the callbacks to execute in case the event associated to the node fires. When a leaf event fires, the graph is traversed in depth-first order starting from the firing event following the connections among events. The callbacks attached to each traversed event are collected. Finally, all handlers are executed in non-deterministic order [15]. Unfortunately, this mechanism is not suitable for signals. If signals are added, intermediate nodes represent signal expressions that depend on each other and must be executed during the traversal – not only at the end, like event handlers. In such a system, glitch freedom requires to control the order of update propagation – as we explain shortly. For this reason, we reimplemented the propagation system from scratch and used the same interface of ESCala for events. As a result, ESCala programs, that correctly do not rely on the order of handlers executions originated by the same change, are also RESCALA valid programs.

The RESCALA signal system is conceptually similar to existing implementations of other reactive languages [6, 19]. It is based on a directed graph to track dependencies between values and to keep them up-to-date. Dependencies are established in conjunction with the evaluation of signal expressions. To enforce the correct update order, the graph is topologically sorted and change propagation proceeds in order from changed values to the values depending on them. Topological sorting ensures glitch freedom [6], the property of avoiding temporary violations of the constraints expressed by

\(^3\text{snapshot is a method of Signal. Since Scala supports infix notation for methods, in Figure 9a, snapshot is invoked on the position signal passing clicked as a parameter.}\)
The main hypothesis that motivated RESCALA’s design is that the fluid integration of events and signals by conversion functions contributes to improved design quality of reactive object-oriented applications. To validate this hypothesis we performed a side-by-side comparison of alternative designs of four reactive object-oriented applications – designs using events only versus designs using the combination of events and signals via conversion functions.

5.1 Experimental Set Up

Case Studies. Our validation benchmark suite consists of four reactive OO applications (Figure 12), which were initially implemented based on events only and afterwards refactored to introduce signals integrated with events via conversion functions.

The Universe simulation [15] has been already presented in the paper. The simulation evolves in rounds and the state of each element at a given step is a function of the other elements and of the state of the simulation in the previous step. This structure allows one to express several aspects of the computation functionally. However, the elements of the simulation are mutable objects that encapsulate state, so the OO and the functional style must be properly combined. A screenshot of this application is shown in Figure 12a. ReactEdit is a minimal text editor implementing functionalities like text selection, line counting, and cutting-and-pasting of text. In previous work [32], we analyzed a text editor provided as a widget in the SWT graphic library, which is used, among other ap-
functions play a key role in this respect. To answer these questions, we followed a three-step process.

First, each case study was implemented with events and callbacks. Second, the case studies were refactored to introduce signals and compositions thereof with events via the conversion functions. Typically a refactoring concerned the reactivity for a certain concern of the application, e.g., time management in the Universe synchronous simulation or the palette to select the shape to draw in the GUI of the ReactShapes application. The decision about which concerns to refactor was made by looking at concerns involving functionally dependent values. Those values are good candidates for being expressed by signals. An example is time management in the Universe application, as shown in Figure 2. On the contrary, a criterion for rejecting a refactoring candidate was when a change is conceptually modeled in a proper way by events. For example, Figure 19 shows the select-all, copy and paste functionalities in the ReactEdit application which are activated by pressing a button in the UI (i.e., an event) and do not require composition. However, computations that depend on events can still be good candidates for refactoring. For example, building on top of events, we refactored a signal the fetching state of the ReactRSS application, as shown in Figure 15 and discussed shortly. Finally, in a separate step, various metrics related to answering our research questions were calculated for both versions. The first two steps were performed by students not involved in the third step, which was performed by the first author.

The calculated metrics are presented in Figures 13 and 14. Figure 13 reports, for each version of each application, the non-comment-non-space lines of code (LOCs) measured with CLOC\(^4\), the number of callbacks, the number of observers/events, and the number of signals. For each refactoring, we report more detailed data in Figure 14 (there is a row in the table for each identified refactoring; the concerns are listed in the last column of Figure 14). Column Conv Funs shows the number of conversion functions used in each refactoring, further discriminated in the number of conversions from signals to events (column S→E) and conversions from events to signals (column E→S). Data in the other columns characterize the effect of each refactoring. Column Callb. shows the number of callbacks that were removed after refactoring. When counting signals and events we consider also the signals/event created in intermediate computations (e.g., by a conversion function) if not already counted elsewhere. Column Signals shows the number of signals that are introduced in each refactoring; column Events shows the number of removed/added events.

### 5.2 Improved Design

We measure the improvement of composability by calculating two metrics. First, we observe that the number of non-composable abstractions (callbacks) is reduced. Second, we observe that the number of composable reactive abstractions (signals and events) is increased by the refactorings.

**Removed Callbacks.** Figure 14, Callbacks column, shows the number of callbacks that were removed due to the introduction of signals and the associated conversion functions. We observe a systematic reduction of callbacks in the events+signals version of each application by 44\% on average.

Since callbacks do not return a value, they are not composable, limiting extensibility and reuse. Conversion functions help reducing the amount of callbacks that are required in each application. With events, a handler is necessary to perform the action associated to the event, which typically imperatively updates some values. Instead, by turning events into signals, we turn their exposed values into reactive values that can freely be composed with other

\[^4\]Since the SWT widget amounts to \(\sim\)10K LOCs of Java it was not feasible for us to work on the original version.

\[^5\]http://cloc.sourceforge.net
signal expressions. This enables dependencies of computations on the occurrence of events and their exposed values to be expressed declaratively and new event values to be automatically propagated to those dependent computations.

**Increased Number of Composable Abstractions.** The results of the analysis of the refactorings shown in Figure 14 demonstrate that the refactorings enabled by interface functions increase the number of composable abstractions (Figure 14, Comp. column). Not surprisingly, signals largely contribute to increased composability (Figure 14, Signals column).

Overall, events increase in the refactorings (Figure 14, Events column). This is due to two causes. First, in some cases signals are not directly defined on top of existing events, but over a combination thereof. For example, in Figure 15 the before(fetch) and the after(fetch) are combined and it is the composed event that is converted to a signal. Second, in some refactorings, the signals added by the refactorings need to interface with the existing event-based part of the application, hence, events must be generated from signals – as in the case discussed for Figure 16. However, we also experienced cases in which events can be simply removed and replaced with signals.

### 5.3 Use of Conversion Functions

In this sub-section, we elaborate on the role of the conversion functions in the improved design composability. Indeed, in all refactorings, conversion functions are used in almost all the cases (Figure 14, second column). Exceptions are discussed at the end of this section. To give an intuition of how conversion functions are used, we graphically depict event-based applications as a graph (Figure 18a), in which the nodes without a predecessor denote directly triggered events on which other events (indirectly) depend (inner nodes of the graph).

From events to signals. Functions converting from events to signals are used to refactor some reactive functionality to signals, in cases when reactivity originates from events, graphically depicted in Figure 18c. For example, ReactRSS needs to fetch possible updates from the monitored websites. Since the operation is time-consuming, the application displays a message to the user. Figure 15 shows how a signal is used to express the "fetching state". The source of reactivity are the implicit events before(fetch) and the after(fetch) that express the begin and the end of the fetching phase. After composing these events, the hold conversion function is used to capture the state of the RSS fetcher.

```scala
lazy val fetcherState: Signal[String] =
  ((before(fetch) map { _ => "Started fetching" }) ||
  (after(fetch) map { _ => "Finished fetching" })) hold ""
```

From signals to events. Functions converting from a signal to an event are used when some piece of reactive functionality that is refactored to use signals still needs to interface to events, graphically depicted in Figure 18b. For illustration, we briefly discuss a refactoring in the universe case study. The example refactoring is about the time management concern, which was refactored to use signals (Figure 2) with the advantages already discussed in Section 2. However, the board on which the creatures move in the simulation is mutable and updated imperatively – a design typical of OO style.

This solution allows each creature in the simulation to access the board and change its state without carrying the board as a parameter in each computation. Due to the imperative design of the board, the signal-based time management must be converted to events before interfacing with the board. In Figure 16, an event is obtained from
the signal holding the current week through the `changed` function (Line 1) that fires every new week. A handler attached to the event imperatively removes the dead creatures form the board (Line 5) and makes those evolve that are still alive (Line 6).

**Exceptions.** There are two refactorings in Figure 15 – namely “Statistics tracker” and “State of the canvas” – that introduce signals without using conversion functions. We explain the reason for this for the “Statistics tracker” refactoring of ReactEdit. The other case in the ReactShapes case study is analogous, thus not further discussed. The “Statistics tracker” refactoring focuses on the part of the application concerned with displaying information on the text currently edited, e.g., the number of characters and the number of lines in the text. These values, however, are already available as signals, since the other refactoring of ReactEdit already introduced signals in the model of the application (e.g., text storage and caret position). For this reason, conversion functions are not needed. Note however, that conversion functions are still required in the second refactoring for the events that come from user interaction, so they are indirectly required to enable the “Statistics tracker” refactoring. In terms of Figure 18, the scenario discussed in this paragraph corresponds to performing a (b) refactoring followed by a (c) refactoring.

Still to answer is the question why, in the refactorings under consideration (“Statistics tracker” and “State of the canvas”), the conversions S → E are not needed either. Since the overall design of the case studies is OO, the result of a signal-based computation typically produces a side effect at some point. This is usually achieved by converting a signal to an event and binding a callback to the latter – hence the expected use of S → E conversions. When the information is displayed in the GUI, we also need to convert from signals to events, since the Swing library [33] we use (and OO graphic libraries in general) is based on events. Nevertheless, S → E are not needed because we wrapped the classes of the Swing library to directly support signals. For illustration, Figure 17 shows an example of a `Label`, a widget that displays text. The widget is directly attached to a signal when it is created and automatically updates the text according to the changes of the signal. Internally, this requires a conversion from signals to events, but this is encapsulated into the `ReLabel` class (Line 2) and does not appear in the counting of conversion functions in Figure 15.

**Discussion.** The classification of refactorings discussed above – signal to events vs. events to signals (i.e. Figure 18c vs. Figure 18b) – is useful to capture the role of interface functions in the refactorings. However, in practice, those cases are often mixed, and a single refactoring comprises both. This circumstance can be inferred from Figure 14 where in several refactorings both E → S and S → E conversions appear. The reason is that, in many cases, the refactoring to signals is surrounded by the event based systems. As a result the signal based computation introduced by the refactoring needs to interface to events at some point – hence conversion functions are needed in both directions.

One may wonder if the effect of conversion functions is simply to turn each event into a signal and then back to an event, artificially increasing the number of composable abstractions in Figure 14. This is, however, not the case. In the events+signals implementations, there is significantly higher number of signals than conversion functions (Figure 14, cf. column E → S and column Signals). This means that conversion functions not only introduce signals by turning events into signals, but enable more advanced refactorings towards more declarative style, where signals can be further defined as a composition of the existing ones.

6. Related work

Approaches closely related to RESCALA [6, 15, 17, 19, 25] were already discussed in Section 2. In this section, we focus on the approaches that are related to our research in a broader scope.

Functional-reactive programming was originally designed by Elliott in Haskell [10]. FRP focuses on the abstract representation of continuous time in functional programs. More generally, the term refers to language abstractions to support time-changing values, like signals or event streams. Frappe [7] ports to Java the ideas originally implemented in FRP and Haskell.

Constraint programming supports declarative relations among program entities and automatically enforces their consistency. For example the Kaleidoscope [12] and the graphical toolkits Garnet [26] and Amulet [27] allow the user to introduce constraints
that are automatically satisfied by the framework. Compared to the work in this paper, these languages only focus on updating dependencies and do not include an advanced event system like the one of RESCALA. SuperGlue [22] is a statically typed language that allows one to specify constraints among Java components. To reduce coupling, SuperGlue supports quantification over component types. The runtime is in charge of keeping constraints satisfied in a way that resembles reactive programming. Unlike RESCALA and other reactive programming approaches, SuperGlue focuses on constraints over components and not on composition of time-changing computations.

Data-flow languages provide abstractions to manipulate streams of values. Conceptually, these languages define nets of operators connected with wires. Examples include Esterel [2] and Lucid [28]. They have a synchronous notion of time which resembles the design of our synchronous timers with contemporary events. Unlike RESCALA, these languages focus on real-time requirements, providing boundaries to memory consumption and propagation time at the cost of sacrificing language expressiveness.

Event-based languages support events as language abstractions. EventJava [11] is a Java extension which borrows ideas from complex event processing and composite event detection. It supports event matching, predicate guarding, reaction to event combination and event correlation. As a consequence, complex reactive behavior can be expressed in a declarative way. Another event-based language is Ptolemy [29]. Whereas the Observer pattern decouples observables from observers, the latter still need to explicitly reference observables. Ptolemy specifically addresses this issue: an object can register to events by referring to the event type instead of referencing the subject that announces the event. Due to the quantification over the event types, observers are decoupled from observables. Finally, Rx [23] is a library originally developed for .NET and ported to other platforms. Rx has received great attention because it provides uniform abstractions, based on LINQ [24], for event composition over heterogeneous sources.

Complex event processing is about performing queries to detect patterns on event streams. For example, TelegraphCQ [5], and Cayuga [9] provide SQL-like queries over time-changing event streams. These systems share with reactive programming the concept of reacting to time-changing values and the declarative style of functional relations [21]. However, they are based on SQL-like query languages rather than integrating dedicated abstractions into a general-purpose language.

Self-adjusting computation (e.g. [1]) is a programming technique that automatically derives an incremental version of a given program. In self-adjusting computation, the program is initially executed to compute the result, then a mutator performs the updates when the input changes. The focus of self-adjusting computation is on efficient derivation of incremental algorithms, and not on raising the level of abstraction via proper linguistic constructs. For example, in self-adjusting computation, the programmer explicitly interacts with the runtime to initiate the change propagation across the dependencies.

Incremental and automatic update has been successfully applied to data structures and queries. Due to this restricted domain, these approaches can take advantage of techniques developed by research in databases to keep views synchronized with the underlying tables [4]. Willis et al. [34] studied queries incrementalization over mutable objects. Object fields are manually annotated and made observable by using AspectJ. Finally, the framework is in charge of tracking the updates and propagating the change to the query result. Rothamel and Liu [30] propose a similar approach based on code generation. While the general problem of incrementalizing and automatically updating generic computations is still a research challenge, incremental update and synchronization of data structures is currently implemented in libraries like Livelinq [18] and GlazedList [16].

7. Summary and Future Work

In this work, we presented RESCALA, a language that seamlessly integrates concepts from event-based programming and reactive languages into object-oriented design. We analyzed the limitations of both approaches and argued that their integration is fundamental to support a mixed functional and OO paradigm. We showed that RESCALA can effectively ameliorate the implementation of reactive applications by fostering a declarative and functional style without relinquishing the advantages of OO design. Finally we provided an evaluation of the language.

In the future, we plan to continue the development of RESCALA. We envisage several research directions. First, we plan to investigate a more direct support of reactive behavior over mutable data by integrating reactive data structures. Second, we want to introduce abstractions from complex event processing like joins and elaborate on matching over event patterns. Finally, we want to apply concepts from reactive programming to the distributed setting. This direction is promising since a huge amount of callbacks, commonly used to react to events in publish-subscribe systems, can be potentially replaced by signals. A more detailed discussion – including the challenge of enforcing glitch-freedom in a distributed setting – can be found in [31].

Acknowledgments

This work has been supported by the German Federal Ministry of Education and Research (BMBF) under grant No. 01IC12S01V SINFONIUM and by the European Research Council, grant No. 321217.

References


In this appendix, we show the RESCALA interface between signals and events. When a signal and an event can be the receiver and an argument, interchangeably, we show the function with the signal as a receiver, i.e., exposed by the Signal trait.

- Creates a signal by folding events with a given function.
  \[
  \text{fold}(e: \text{Event}, \text{init}: A)(f: (A,T) \to A): \text{Signal}\[A]\]
  - Returns a value computed by `f` on the occurrence of an event.
  \[
  \text{iterate}(e: \text{Event}, \text{init}: A)(f: A \to A): \text{Signal}\[A]\]
  - Returns a signal holding the latest value of the event `e`.
  \[
  \text{hold}(T)(e: \text{Event}, \text{init}: T): \text{Signal}[T]\]
  - Holds the latest value of an event as \text{Some(val)} or \text{None}.
  \[
  \text{holdOption}(T)(e: \text{Event}): \text{Signal[Option[T]]}\]
  - Returns a signal which holds the last `n` events.
  \[
  \text{last}(T)(e: \text{Event}, n: \text{Int}): \text{Signal[Seq[T]]}\]
  - Collects the event values in a reactive list.
  \[
  \text{list}(T)(e: \text{Event}): \text{Signal[Seq[T]]}\]
  - Delays a signal by a change occurrences.
  \[
  \text{delay}(T)(n: \text{Int}): \text{Signal[T]}\]
  - Counts the occurrences of an event.
  \[
  \text{count}(e: \text{Event}): \text{Signal[Int]}\]
  - On the event, sets the signal to one generated by the factory.
  \[
  \text{reset}(T, A)(e: \text{Event}, \text{init}: T)(f: (T) \to \text{Signal}[A]): \text{Signal[A]}\]
  - Switches the value of the signal on the occurrence of an event.
  \[
  \text{switchTo}(U)(e: \text{Event}): \text{Signal[U]}\]
  - Switches to a new signal once, on the occurrence of an event.
  \[
  \text{switchOnce}(T)(e: \text{Event}, \text{newSignal}: \text{Signal[T]}): \text{Signal[T]}\]
  - Switches between signals on the event `e`.
  \[
  \text{toggle}(T)(e: \text{Event}, \text{other}: \text{Signal[T]}): \text{Signal[T]}\]
  - Returns a signal updated only when `e` fires.
  \[
  \text{snapshot}(T)(e: \text{Event}): \text{Signal[T]}\]