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

Guido Salvaneschi
Software Technology Group
Technische Universität Darmstadt
salvaneschi@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.

MODULARITY '14, April 22–26, 2014, Lugano, Switzerland.

http://dx.doi.org/10.1145/10.1145/2577080.2577083

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 langauges 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 langauges 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.

```
1  abstract class Figure { ...
2    protected evt moved[Unit] = after(moveBy)
3    evt resized[Unit]
4    evt changed[Unit] = resized || moved || after(setColor)
5    evt invalidated[Rectangle] = changed.map(() => getBounds())
6   ...
7    def moveBy(dx: Int, dy: Int) { position.move(dx, dy) }
8    def setColor(col: Color) { color = col }
9    def getBounds(): Rectangle   ...
10 }
11 class Connector(val start: Figure, val end: Figure) {
12   start.changed += updateStart
13   end.changed += updateEnd
14  ...
15   val updateStart = { _ => ... }
16   val updateEnd = { _ => ... }  ...
17 }
18 class RectangleFigure extends Figure {
19   evt resized[Unit] = after(resize) || after(setBounds)
20   override evt moved[Unit] = super.moved || after(setBounds)
21  ...
22   def resize(size: Size) { this.size = size }
23   def setBounds(x1: Int, y1: Int, x2: Int, y2: Int) {
24     position.set(x1, y1); size.set(x2 − x1, y2 − y1)
25   } ...
26 }
```

Figure 1: EScala Events.

EScala also supports declarative events, which are defined as a combination of other events. For this purpose it offers operators like $e_1||e_2$ (occurrence of one among e_1 or e_2), $e_1\&\&p$ (e_1 occurs and the predicate p is satisfied), $e_1.map(f)$ (the event obtained by applying f to e_1). 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-

```
1  imperative evt tick[Unit]
2  var hour: Int = 0
3  var day: Int = 0
4  var week: Int = 0
5
6  tick += nextHour _
7  def nextHour() {
8    hour = (hour + 1) % 24
9  }
10 evt newDay [Unit] = tick && (() => hour == 0)
11 newDay += nextDay _
12 def nextDay () {
13   day = (day + 1) % 7
14 }
15 evt newWeek [Unit] = ...
16 newWeek += nextWeek _
17 def nextWeek() {
18   ...
19 }
```

(a)

```
1 val tick = new Var(0)
2 val hour = Signal{ tick() % 24 }
3 val day = Signal{ (tick()/24)%7 + 1 }
4 val week = Signal{ ... }
5
6
```

(b)

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

ing and tangling [32]. In addition, new events cannot be introduced transparently by clients: the original codebase must be modified by converting fields into observables and by adding event triggering. Therefore, event definitions are hardly extensible and require careful preplanning.

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 langauges 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.

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, e.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 form using them.

Second, most of existing OO reactive applications are event-based – graphic 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 `None` 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).

```
1 val packet: Signal[Option[Packet]] = Signal{ ... }
2
3 evt packetReceived[Packet] = ...
```

Figure 3: Packet reception with Events and Signals.

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)}: Signal[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()) || killed )
```

(a)

```
1  val age = new Var(0)
2  val size = new Var(1)
3   ...
4  val canLive: Signal[Boolean] = Signal {
5    (age() <= maxAge ) && (size() <= 3000) && (size() >= 1)
6  }
7  def disease() = { age *= 2 }
8
9  evt shouldDie =
10     canLive.changed && !canLive() || killed
11
```

(b)

Figure 4: Dependencies with Implicit Method Events (a). Dependencies as Signals (b).

be updated imperatively minimizing the update by performing fine grained changes.

The discussion so far shows that events and signals have their advantages and disadvantages and event-based applications cannot be refactored to use only signals without loss of desired properties. We derive that there is a need for a language design that supports a fluid transition between the two worlds and seamlessly integrates them into the OO setting. This was the goal driving the design of RESCALA, which we present next.

3. REScala

In this section, we present RESCALA, a reactive language that provides a powerful event system – recapitulated in Section 2 – with seamlessly integrated support for reactive values. Some details on the implementation of RESCALA are in Section 4. Reactive values are called signals in Scala.React [19]; we adopt the same terminology in RESCALA.

As argued in the previous section, to properly support reactive applications, language designs are needed that offer both imperative and functional styles of programming reactivity. But, just having them side-by-side is only half-way to a coherent language design; in addition, imperative and functional abstractions to reactivity should be made composable. To achieve this goal, the key innovation of RESCALA, consists in mechanisms to seamlessly bridge between the imperative and functional styles of reactive behaviors to make them composable.

3.1 Signals

In RESCALA, the general form of a signal *s* is *Signal*{*expr*}, where *expr* is a standard Scala expression. When *expr* is evaluated, all `Signal` and `Var` values it refers to are registered as dependents of *s*; any subsequent change of them triggers a reevaluation of *s*. RESCALA signals integrate seamlessly with OO design. They are class attributes like fields and methods. They too can have different visibilities. Public signals are part of the class interface: Clients can refer to them to build composite reactive values. Conversely, private signals are only for object-internal use.

RESCALA signals cannot be re-assigned new expressions once they are initialized. At first sight, it may seem intuitive to treat signals like object fields, which can be reassigned as needed. However, this makes applications harder to understand; signal values would depend not only on the control flow inside their expression, but also on the control flow of the application, which can assign a different signal expression. Hence, the definition of the dependencies is separated from the declaration of the signal; making signals reassignable comes at the risk of vanishing the motivations that lead to their introduction. Fortunately, our experience suggests that this need does not arise in practice. This design decision enforces uniformity across signals and methods: Method bodies cannot be assigned dynamically, signals expressions cannot be assigned after creation. In RESCALA, this design is technically achieved by declaring them with Scala's `val` modifier.

As expected, RESCALA signals will replace the use of (implicit) events in EScala for encoding functional dependencies between values. For example, consider the snippet from the Universe application in Figure 4a that uses events to express the functional dependency between the `canLive` attribute of an animal on its `age` and `size`. The logic of the simulation is the following: An animal may become ready to die whenever it gets older (as time elapses), whenever it grows (as it eats food), or whenever it has a disease (a disease is implemented by simply doubling the age); the `canLive` method determines at any of these points whether the animal can still be alive. Finally, the animal can die because it is killed by other animals.

The dependency between the `canLive` attribute of an animal on its `age` and `size` can be expressed more declaratively by refactoring the `canLive` method to a signal and turning `age` and `size` into Scala `vars`, as shown in Figure 4b, Lines 1-2. This design comes with a reduced number of events, simplifying the application. In particular, *technical events*, which are not part of the application logic disappear because changes of `age` and `size` are captured directly in the definition of `canLive`. The `change` operator in Line 10 converts a signal into an event and will be explained in details in Section 3.2.2.

3.2 From Events to Signals and Back

RESCALA provides a rich API of functions for converting events to signals and the other way around. The goal is to ensure that the same abstraction/composition mechanisms uniformly apply over them. Conversion functions also facilitate refactoring of code fragments from one style to the other. The complete list of functions supported by RESCALA is shown in the Appendix A. Due to lack of space, in the following, we discuss only a subset of them that is representative enough to give an intuition about the role and expressiveness of the API. As we discuss at the end of this section, RESCALA also supports functions lifting to improve compatibility of existing code with signals.

3.2.1 Integrating Events into Signals

Since RESCALA promotes a mixed OO and functional style, it is important to manage state at the boundary between imperative and functional fragments of applications. For this purpose, RESCALA provides a set of functions for converting events into signals, so that event-based imperative sub-computations can be wrapped up and abstracted over in functional computations.

The *basic function for converting events to signals* is `hold`: given an event `e`, the call `e.hold()` returns a signal representing the

value exposed by the most recent occurrence of `e`. For illustration, consider the code snippet in Figure 5, where a signal `click.hold` 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[1].

```
1 evt click: Event[(Int, Int)] = mouse.click
2 // circle = ((centerX, centerY), radius)
3 val circle: Var[(Int, Int),Int] = Var((1,1),10)
4 val lastClickOnCircle: Signal[Boolean] =
5                        Signal{ over(click.hold(), circle()) }
6 val lastClick: Signal[(Int, Int)] = mouse.lastClick
```

Figure 5: `hold` at Work.

The conversion of events to signals by `hold` 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.hold` 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's provides *functions for stateful conversion of events to signals* – in the following, we discuss three such functions: `fold`, `list`, and `last(Int)`.

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.

```
1 evt click: Event[(Int, Int)] = mouse.click
2 val nClick = Var(0)
3
4 click += { _ => nClick() += 1 }
```

(a)

```
1 evt click: Event[(Int, Int)] = mouse.click
2 val nClick: Signal[Int] = click.fold(0)( (x,_) => x+1 )
3
```

(b)

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 accumulatively 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[2].

```
1 evt click: Event[(Int, Int)] = mouse.click
2 val history: Signal[Seq[(Int,Int)]] = click.list()
3 val history5: Signal[Seq[(Int,Int)]] = click.last(5)
4
5 val mean = Signal {
6   val (x,y) = history5().unzip
7   val n = history5().length + 1
8   (x.sum/n, y.sum/n)
9 }
```

Figure 7: Abstracting over State with `list`/`last(Int)`.

Figure 8 shows the same functionality implemented without the support of conversion functions. The programmer needs to introduce a var (Line 2) and a callback (Line 3). The callback updates the var when the event occurs, so the depending signals (Lines 6-9) are updated. The callback and the var are not part of the application logic and serve the sole purpose of bridging events and signals. The logic of the application is now spread among the callback (adding the element to the list) and the definition of each signal (slicing the last 5 elements). Even worse, the relation between the `click` event and the `history`/`history5` signal is not explicit any more from the definition of those signals and must be harvested from the control flow.

```
1 evt click: Event[(Int, Int)] = mouse.click
2 val historyV: Var[List[(Int,Int)]] = Var(List())
3 click += { clickPosition =>
4   historyV()= clickPosition :: historyV()
5 }
6 val history: Signal[List[(Int,Int)]] = Signal{ historyV() }
7 val history5: Signal[List[(Int,Int)]] =
8                                  Signal{ history.slice(0,5) }
9 val mean = Signal{
10   val (x,y) = history5().unzip
11   val n = history5().length + 1
12   (x.sum/n, y.sum/n)
13 }
```

Figure 8: Abstracting over State without Interface Functions Support.

3.2.2 Integrating Signals into Event-Driven Computations

RESCALA also provides a set of operations that enable to seamlessly integrate signals into event-driven computations.

The most *basic operation for converting signals to events* is `changed()`: Given a signal `s`, `s.changed()` triggers an event every

[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 `s` 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.

```
1 evt clicked: Event[Unit] = mouse.clicked
2 val position: Signal[(Int,Int)] = mouse.position
3 val lastClick: Signal[(Int,Int)] = position snapshot clicked
4
```

(a)

```
1 evt clicked: Event[Unit] = mouse.clicked
2 val position: Signal[(Int,Int)] = mouse.position
3 val lastClickPos = Var(0,0)
4 val lastClick: Signal[(Int,Int)] = Signal{ lastClickPos() }
5 clicked += { _ =>
6   lastClickPos()= position()
7 }
```

(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. `posSpeedX` 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=>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 `evt e3 = e1 || e2`, `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] `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.

```
val speed = new Point(10,8)
evt xBounce,yBounce = ... // Events

val posSpeedX = Signal{ speed.x }
val posSpeedY = Signal{ speed.y }
val negSpeedX = Signal{ -speed.x }
val negSpeedY = Signal{ -speed.y }
val speedX = Signal{ if switchedX() posSpeedX() else negSpeedX() }
val speedY = Signal{ if switchedY() posSpeedY() else negSpeedY() }

val switchedX = Var(false)
val switchedY = Var(false)
xBounce += {_ =>
  switchedX() = !switchedX()}
yBounce += {_ =>
  switchedY() = !switchedY()}
```

(a)

```
val speed = new Point(10,8)
evt xBounce, yBounce = ... // Events
val speedX = Signal{speed.x}.toggle(xBounce){ -speed.x }
val speedY = Signal{speed.y}.toggle(yBounce){ -speed.y }
```

(b)

Figure 10: Bouncing Ball without `toggle` (a). `Toggle` Function at Work (b).

```
def mean(list: Seq[(Int,Int)]): (Double,Double) = {
  val (x, y) = list.unzip
  val n = list.length + 1.0
  (x.sum / n, y.sum / n)
}
val meanR = Signal.lift(mean)
evt click = new ImperativeEvent[(Int, Int)]
val history = click.last(5)
val meanS = meanR(history)
```

Figure 11: Lifting of Traditional Functions.

signal expressions. For example, consider the dependencies established by the following configuration of reactive values and signal expressions:

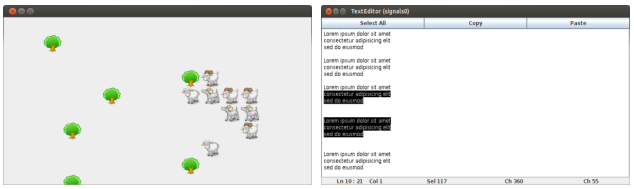
```
val x = Var(1)
val y = Signal{ x() * 2 }
val z = Signal{ x() * 3 }
val t = Signal{ y() + z() }
```

Suppose that the value of `x` is updated to 2. Then `y` must be updated to 4. If at this point `t` is updated, it evaluates to 5, which is clearly wrong, since after updating `z`, the correct – and stable – value of `t` is 10. To prevent such temporary values (i.e. *glitches*), nodes must be updated in the correct order, in this case `x-y-z-t`. Compared to EScala, a breadth-first traversal is needed *(i)*. Signal expressions must be evaluated inside signal nodes and the propagation must be stopped in case a signal expression does not change its value *(ii)*. As node dependencies are discovered at runtime, topological sorting cannot be guaranteed in advance and the graph must be restructured when the topological order is violated *(iii)*. Further details on this technique can be found in the technical report [19] and references therein.

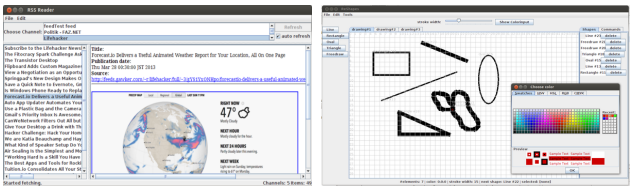
RESCALA preserves glitch freedom inside signal-based dependent computations and conversions between signals and events. When the computation escapes the reactive system and involves imperative events and callbacks, side effects can be performed. In that case, like with other event-based languages, side effects can establish data dependencies that are not under the control of the reactive system and the user is responsible of performing the updates in the correct order.

5. Validation

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.



(a) (b)



(c) (d)

Figure 12: The Case Study Applications: Universe (a), ReactEdit (b), ReactRSS (c), ReactShapes (d).

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-

Case Study	LOC	Callb.	Events	Signals
Universe				
Events	466	17	8	0
Events+Signals	442	6	12	30
ReactEdit				
Events	644	14	34	0
Events+Signals	632	10	36	22
ReactRSS				
Events	599	16	41	0
Events+Signal	595	14	38	9
ReactShapes				
Events	1161	17	30	0
Events+Signals	1160	5	56	30

Figure 13: Main Metrics for the Case Studies.

plications, in the Eclipse IDE. The analysis showed that a lot of complexity in the code is due to a design of reactivity that favors efficiency, requiring caching of intermediate values and incremental computations. ReactEdit is a minimal version of the SWT widget, which is malleable to investigating various design alternatives based on reactive abstractions[4]. A screenshot of this application is shown in Figure 12b.

ReactRSS is a RSS feed reader displaying a list of channels, which are periodically checked for updates. Fetched items are immediately displayed to the user in a side bar. When the user selects one of them, the HTML content is rendered in the main view. A screenshot of this application is shown in Figure 12c.

ReactShapes is a small drawing program. The user can drag and drop different shapes on a canvas, connect them with lines and change the stroke width and the color of each shape. The application supports an history and an undo function. Finally, the drawing canvas can be shared with other clients that participate in the same task from remote. A screenshot of this application is shown in Figure 12d.

We selected the case studies to cover different kinds of reactive behavior in common OO applications. In most cases, in desktop software, reactivity originates from user interaction, e.g., mouse movements or hitting a button on the keyboard. ReactEdit, ReactRSS and ReactShapes cover this class of applications. Another source of reactivity are asynchronous external events, like messages from the network. The ReactShapes and the ReactRSS applications implement this kind of functionality. Another common example of reactive applications are synchronous simulations, where at each round a change is propagated to all the entities in the application. The Universe case study covers this case.

Research Questions and Methodology. The main question for the validation was: Are designs based on the combinations of events and signals better? We look at improved composability, i.e., increased number of composable abstractions in code as an indication for better design. The secondary question is: If the designs are improved, in what extent are the conversion functions involved in this improvement. In the following, we demonstrate that the studied refactorings do indeed improve composability and that conversion functions play a key role in this respect. To answer this 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 to a signal the *fetching* state of the React RSS 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[5], 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 $S \rightarrow E$) and conversions from events to signals (column $S \rightarrow E$). Data in the other columns characterize the effect of each refactoring. Column *Callb.* shows the number of callbacks that where 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 where 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 ∼10K LOCs of Java it was not feasible for us to work on the original version.

[5] httpc://cloc.sourceforge.net

Case Study (Signals+Events)	Conv Funs	S → E	E → S	Signals	Callb.	Events	Comp.	Refactored Concern
Universe	2	1	1	+11	-2	-1	+10	Activity of the creatures
	10	5	5	+11	0	0	+11	Statistics
	2	0	2	+3	-2	0	+3	Evolution and reproduction
	9	8	1	+5	-7	+5	+10	Time management
ReactEdit	0	0	0	+7	-3	-3	+4	Statistics tracker
	9	7	2	+15	-1	+5	+20	Caret position and selection
ReactRSS	5	4	1	+1	-2	0	+1	Network fetcher
	2	0	2	+2	0	-2	0	RSS feeds store
	6	5	1	+6	0	-1	+5	UI for items channels and status
ReactShapes	0	0	0	+6	-6	0	+6	State of the canvas
	8	6	2	+7	0	+8	+15	Display information
	2	1	1	+8	-1	+6	+14	UI for menus
	1	1	0	+4	-2	+1	+5	History of executed commands
	1	1	0	+4	-2	+2	+6	Panel for drawing shapes
	1	0	1	+1	-1	+9	+10	Palette for shape selection
Total	+58	+39	+19	+91	-29	+29	+120	

Figure 14: Conversion Functions and their Effect in the Case Studies.

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.

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

Figure 15: Converting Events to Signals in a Refactoring.

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

```
1 time.hour.changed += {x  =>
2   board.elements.foreach { _ match {
3       case (pos, be) =>
4         if(be.isDead.getVal)
5           board.clear(pos)
6         else be.doStep(pos)
7 } }   }
```

Figure 16: Converting Signals to Events in a Refactoring.

```
1 val charCountLabel =
2   ReLabel(Signal { "Ch " + textArea.charCount() })
```

Figure 17: Use of Signals in the Graphic Interface.

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

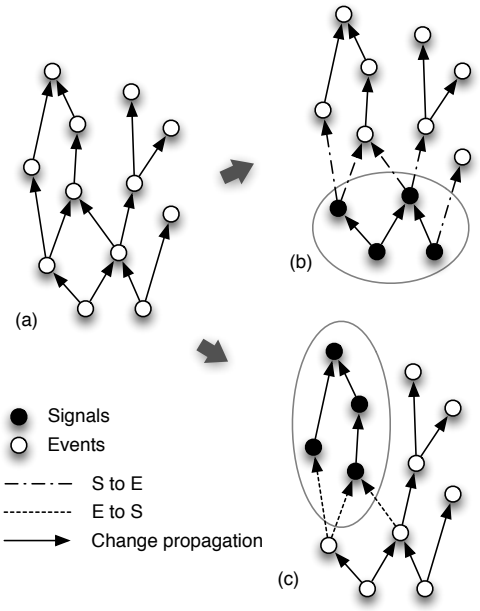


Figure 18: Reafactoring Event-based Applications to use Signals.

```
1 selectAllButton.clicked +=
2   {_=> textArea.selectAll; textArea.requestFocus}
3 copyButton.clicked +={_=> textArea.copy; textArea.requestFocus }
4 pasteButton.clicked +={_=> textArea.paste; textArea.requestFocus }
```

Figure 19: User Interaction in the ReactEdit Case Study.

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 patters 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 SINNODIUM and by the European Research Council, grant No. 321217.

References

[1] U. A. Acar, A. Ahmed, and M. Blume. Imperative self-adjusting computation. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 309–322, New York, NY, USA, 2008. ACM.

[2] G. Berry and G. Gonthier. The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87 – 152, 1992.

[3] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: a declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '87, pages 178–188, New York, NY, USA, 1987. ACM.

[4] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the 17th International Conference on Very Large Data Bases*, VLDB '91, pages 577–589, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[5] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, pages 668–668. ACM, 2003.

[6] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *ESOP, 15th European conference on Programming*, pages 294–308, 2006.

[7] A. Courtney. Frappe: Functional reactive programming in Java. In *Proceedings of the Third International Symposium on Practical Aspects of Declarative Languages*, PADL '01, pages 29–44, London, UK, 2001. Springer-Verlag.

[8] Microsoft corporation. C# language specification. v.3.0. http://msdn.microsoft.com/en-us/vcsharp/aa336809.aspx.

[9] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. Towards expressive publish/subscribe systems. In *Proceedings of the*

10th international conference on Advances in Database Technology, EDBT'06, pages 627–644, Berlin, Heidelberg, 2006. Springer-Verlag.

[10] C. Elliott and P. Hudak. Functional reactive animation. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, ICFP '97, pages 263–273. ACM, 1997.

[11] P. Eugster and K. R. Jayaram. EventJava: An extension of Java for event correlation. In *Proceedings of the 23rd European Conference on Object-Oriented Programming, ECOOP 2009*, Genoa, pages 570–594, Berlin, Heidelberg, 2009. Springer-Verlag.

[12] B. N. Freeman-Benson. Kaleidoscope: mixing objects, constraints, and imperative programming. In *Proceedings of the European conference on object-oriented programming systems, languages, and applications*, OOPSLA/ECOOP '90, pages 77–88, New York, NY, USA, 1990. ACM.

[13] Gamma, Helm, Johnson, and Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, 2000.

[14] D. Garlan and D. Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *Proceedings of the 4th International Symposium of VDM Europe on Formal Software Development: Conference Contributions - Volume I*, VDM '91, pages 31–44, London, UK, 1991. Springer-Verlag.

[15] V. Gasiunas, L. Satabin, M. Mezini, A. Núñez, and J. Noyé. EScala: modular event-driven object interactions in Scala. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD '11, pages 227–240. ACM, 2011.

[16] GlazedLists site. http://www.glazedlists.com/.

[17] D. Ignatoff, G. H. Cooper, and S. Krishnamurthi. Crossing state lines: Adapting object-oriented frameworks to functional reactive languages. In *FLOPS*, pages 259–276, 2006.

[18] LiveLINQ Site. http://www.componentone.com/SuperProducts/LiveLinq/.

[19] I. Maier and M. Odersky. Deprecating the Observer Pattern with Scala.react. Technical report, 2012.

[20] I. Maier and M. Odersky. Higher-order reactive programming with incremental lists. In G. Castagna, editor, *ECOOP 2013 Object-Oriented Programming*, volume 7920 of *Lecture Notes in Computer Science*, pages 707–731. Springer Berlin Heidelberg, 2013.

[21] A. Margara and G. Salvaneschi. Ways to react: Comparing reactive languages and complex event processing. In *REM*, 2013.

[22] S. McDirmid and W. C. Hsieh. SuperGlue: Component programming with object-oriented signals. In D. Thomas, editor, *Object-Oriented Programming, 20th European Conference, Nantes, France*, volume 4067 of *LNCS*, pages 206–229. Springer, 2006.

[23] E. Meijer. Your mouse is a database. *Commun. ACM*, 55(5):66–73, 2012.

[24] E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 706–706. ACM, 2006.

[25] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: a programming language for Ajax applications. Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications, OOPSLA '09, pages 1–20. ACM, 2009.

[26] B. A. Myers, D. A. Giuse, R. B. Dannenberg, D. S. Kosbie, E. Pervin, A. Mickish, B. V. Zanden, and P. Marchal. Garnet: Comprehensive support for graphical, highly interactive user interfaces. *Computer*, 23(11):71–85, Nov. 1990.

[27] B. A. Myers, R. G. McDaniel, R. C. Miller, A. S. Ferrency, A. Faulring, B. D. Kyle, A. Mickish, A. Klimovitski, and P. Doane. The Amulet Environment: New models for effective user interface software development. *IEEE Trans. Softw. Eng.*, 23(6):347–365, June 1997.

[28] M. Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud, LRI, April 2006.

[29] H. Rajan and G. T. Leavens. Ptolemy: A language with quantified, typed events. In J. Vitek, editor, *ECOOP 2008*, volume 5142 of *LNCS*, pages 155–179, Berlin, July 2008. Springer-Verlag.

[30] T. Rothamel and Y. A. Liu. Generating incremental implementations of object-set queries. In *Proceedings of the 7th international conference on Generative programming and component engineering*, GPCE '08, pages 55–66. ACM, 2008.

[31] G. Salvaneschi, J. Drechsler, and M. Mezini. Towards distributed reactive programming. In R. Nicola and C. Julien, editors, *Coordination Models and Languages*, volume 7890 of *Lecture Notes in Computer Science*, pages 226–235. Springer Berlin Heidelberg, 2013.

[32] G. Salvaneschi and M. Mezini. Reactive behavior in object-oriented applications: An analysis and a research roadmap. In *Proceedings of the 12th annual international conference on Aspect-oriented software development, AOSD'13*, 2013.

[33] Scala Swing library. http://www.scala-lang.org/api/current/index.html#scala.swing.package.

[34] D. Willis, D. J. Pearce, and J. Noble. Caching and incrementalisation in the Java query language. *SIGPLAN Not.*, 43(10):1–18, Oct. 2008.

[35] Y. Zhuang and S. Chiba. Method slots: Supporting methods, events, and advices by a single language construct. In *Proceedings of the 12th Annual International Conference on Aspect-oriented Software Development*, AOSD '13, pages 197–208, New York, NY, USA, 2013. ACM.

A. Signals-events API

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.
`fold[T,A](e: Event[T], init: A)(f :(A,T)=>A): Signal[A]`

– Returns a value computed by `f` on the occurrence of an event.
`iterate[A](e: Event[_], init: A)(f: A=>A) :Signal[A]`

– Returns a signal holding the latest value of the event `e`.
`hold[T](e: Event[T], init: T): Signal[T]`

– Holds the latest value of an event as `Some(val)` or `None`.
`holdOption[T](e: Event[T]): Signal[Option[T]]`

– Returns a signal which holds the last `n` events.
`last[T](e: Event[T], n: Int): Signal[Seq[T]]`

– Collects the event values in a reactive list.
`list[T](e: Event[T]): Signal[Seq[T]]`

– Delays a signal by `n` change occurrences.
`delay[T](n: Int): Signal[T]`

– Counts the occurrences of an event.
`count(e: Event[_]): Signal[Int]`

– On the event, sets the signal to one generated by the factory.
`reset[T,A](e: Event[T], init: T)(f: (T)=>Signal[A]): Signal[A]`

– Switches the value of the signal on the occurrence of `e`.
`switchTo[U](e: Event[U]): Signal[U]`

– Switches to a new signal once, on the occurrence of `e`.
`switchOnce[T](e: Event[_], newSignal: Signal[T]): Signal[T]`

– Switches between signals on the event `e`.
`toggle[T](e: Event[_], other: Signal[T]): Signal[T]`

– Returns a signal updated only when `e` fires.
`snapshot[T](e: Event[_]): Signal[T]`