Object-oriented Reactive Programming is *Not* Reactive Object-oriented Programming

Elisa Gonzalez Boix
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussels, Belgium
egonzale@vub.ac.be

Kevin Pinte
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussels, Belgium
kpinte@vub.ac.be

Simon Van de Water
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussels, Belgium
svdewate@vub.ac.be

Wolfgang De Meuter
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussels, Belgium
wdmeuter@vub.ac.be

ABSTRACT

According to chapter 3 of Abelson & Sussman [1], there are two fundamentally different ways to organise large systems: according to the objects that live in the system, or according to the streams of values that flow through the system. Even though the notions of "object" and "stream" have meanwhile taken many incarnations, the dichotomy still exists in modern programming languages. Marrying reactive programming and OOP is a research endeavour to come up with a unified model that embraces both styles of thinking. We identify two opposing research tracks towards the marriage. Existing work focuses on OO reactive programming, i.e., it uses object technology to compose reactions. Our work explores the converse: in the paper, we present the ROAM (Reactive Objects in AmbientTalk) model which is an experimental framework that explores objects as streams of reactive state.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-Oriented Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features

Keywords

object-oriented programming, reactive programming, language abstraction

1. INTRODUCTION

Modern applications are becoming ever more driven by externally produced events and streams of values. These originate in network interactions, rich graphical user interfaces, and a wide variety of on-board sensors (e.g GPS receiver, accelerometer, physical activity trackers etc). Keeping such applications responsive requires one to design them in an event-based style all the way down. However, event-based programming suffers from *inversion of control* [5], i.e., the application logic is scattered across numerous small event handlers (e.g., callbacks) that are triggered by a source external to the application. As a result, the control flow of the application no longer follows the textual order specified by the programmer, making it extremely hard to keep track of what is happening.

In this context, early work on Functional Reactive Programming (FRP), see e.g., [4], has been identified as a promising research avenue because it allows event-driven code to be written in a declarative style, thus avoiding the inversion of control. Technically, FRP introduces the notion of *event streams* and *time-varying values* and relies on the underlying execution technology to automatically manage event dependencies between expressions and their subexpressions. However, unfortunately today most software systems are still built in an imperative object-oriented programming style and this is something that will probably not change anywhere near soon. The fundamental question that needs to be answered is thus how to reconcile reactive programming with encapsulated mutable data structures, a.k.a. objects.

There are two fundamentally different ways in which object-orientation can be reconciled with reactive programming into one coherent programming model. First, the programming model can be designed around the *streams* of events and values that flow through a system. The idea is to design language features that allow programmers to create compositions of streams in order to manage the complexity of events streams and time-varying values. In other words, one "objectifies" reactive streams and one offers object technology to build and compose streams. This is what we call *object-oriented reactive programming*. Alternatively, the programming model can be more centred around the *objects* that live in a system, and offer language features that allow programmers to evoke and manage state changes of object fields by means of reactive computation. In other words, one "reactifies" objects and their constituent fields. This is what

we call *reactive object-oriented programming.*

In recent work, reactive programming *has* affected imperative languages. A famous example is Flapjax [7] for JavaScript, but others exist as well (see [2] for an overview). Moreover, a number of dedicated reactive language dialects have been proposed including FrTime [3] for Scheme and Scala.React [6] and REScala [8] for Scala. These language designs basically allow developers to define reactive streams of composite values (e.g., streams of cons cells and even objects) the constituents of which are immutable. As a consequence, interactions of reactive programming and *stateful* object-oriented programming remain largely unexplored. In some of the aforementioned work, ideas have been explored to connect stateful objects (e.g., in a GUI framework) to reactive streams or — conversely — to feed the values of reactive streams into the fields of stateful objects. In our work, we attempt to go further and consider *objects as entities encapsulating behaviour and streams of mutable state.* As a result, a reactive stream can consist of objects whose identity does not change over time but whose constituent state does. Moreover, even though the identities (i.e., the object seen as a value) do not change, dependencies installed on such objects are reevaluated if one of their constituent fields changes. We present ROAM (Reactive Objects in AmbientTalk) as a first experimental reactive object-oriented framework. ROAM was implemented as a reflective extension to AmbientTalk [9], our own homegrown (distributed) object-oriented programming language.

2. MOTIVATION

Reactive programming languages facilitate the development of event-driven and interactive applications by introducing events and signals. Signals (also known as behaviours) represent continuous time-varying values whereas events refer to streams of value changes that represent discrete values. In the past there has been a lot of research on functional reactive programming. Recently, languages like Scala.React and REScala attempt to bridge the gap between OO and functional reactive programming style of thinking. In this section, we argue that those solutions do not provide true reactive object-oriented programming.

2.1 Object-oriented Reactive Programming

Scala.React [6] is a representative scion of what we dub object-oriented reactive programming. As in many other reactive programming languages, Scala.React allows to transparently lift traditional function calls so that such calls automatically trigger reactive computations. Events in Scala.React are modeled in the form of event streams which are the discrete counterpart of signals.

In Scala.React, a signal s is initialized with the `Signal{expr}` operation where `expr` is an arbitrary Scala expression. After the evaluation of `expr`, s is said to depend on all other signals that are referred to from within `expr`. This means that a reevaluation of s will be triggered if any of them is updated. An important constraint is that these dependencies cannot be modified after the signal has been created. According to [6], this constraint stems from the observation that reassignment of signals would make applications harder to understand as the signal values would then not only depend on the control flow inside their expression, but also on the control flow of the rest of the application.

Consider the example in Listing 1 taken from [6]. This example denotes a signal that is conceived as an object with a single state variable initialized with the `new Path` expression. Every time the signal receives external event notifications (e.g., `mouseDown` or `MouseMove`), the signal's state variable is updated by the `self()= ...` expression. This will generate update notifications to all signals that depend on this signal in their turn. Hence, the signal can be considered as an object with one single mutable slot that accumulates changes as it receives notifications.

Listing 1: A path signal in Scala.React

```
1 val path: Signal[Path] = Signal.flow(new Path) { self =>
2   val down = self await mouseDown
3   self()= self.previous.moveTo(down.position)
4   self.loopUntil(mouseUp) {
5     val movement = self awaitNext mouseMove
6     self()= self.previous.lineTo(movement.position)
7   }
8   self()= self.previous.close()
9 }
```

REScala is a follow-up model of Scala.React. In REScala, signals are a third kind of class attributes next to the traditional fields and methods. This allows programmers to define composite signals based on signals from different classes. A signal is initialized using an arbitrary Scala expression, which — as in Scala.React — can also depend on one or more other signals. From [8], it is unclear how a signal's implicit dependency on other lexically scoped signals as in Scala.React differs from the explicit dependencies enumerated on the class declaration level in REScala. It is our conjecture that having signals declared as class attributes creates the dependencies every time a new instance of the class is created whereas creating these dependencies happens only once for a signal that solely depends on lexically visible signals. Nevertheless, the essence of the REScala model remains the same: signals are objects that can update local fields. However, expressions depending on those objects are not automatically reevaluated by updating those fields.

Apart from this difference, REScala adds many features to Scala.React. In order to promote the integration of functional code into an object-oriented setting, REScala allows developers to convert signals to events and vice versa. Applying the `hold` function to an event returns a signal that exposes the most recent occurrence of that event at any point in time. Signals that are created by using `hold` are thus stateless in the sense that they only expose the last occurrence of an event. On the other hand, the `changed` function can be applied on signals and returns an event that fires every time the value of the signal is updated. REScala offers functions to accumulate or keep track of the history of occurrences of the event (i.e., `fold`, `list` and `last(Int)`). Additionally, developers can use `snapshot` to get the current value of a signal every time a certain event occurs. A feature that is less common in previous approaches to reactive programming is that REScala offers signal-enabled (i.e., reactive) data structures, which expose some of their attributes as signals.

Even though REScala goes much further than Scala.React

in its object-oriented abstraction facilities, the dominant model of the language still consists of wiring together streams of values by means of object technology (i.e., class declarations).

2.2 Reactive Object-oriented Programming

Our work explores a programming model that marries object-oriented programming and reactive programming by altering the traditional semantics of objects, fields and methods. Entire objects become the unit of reactivity, rather than reactive values that can be defined inside objects. This results in what we call *reactive objects*.

Before explaining a particular language design experiment in the following section, we first explain the model intuitively by highlighting the difference with the work presented above. To this end, consider the code snippet below. Lines 1 - 8 show the definition of a coordinate in a two-dimensional grid. A coordinate is initialized with an x- and y-value and understands one method that computes the addition of two coordinates.

```
1  def Coordinate := object: {
2    def x;
3    def y;
4    def move(c) {
5      Coordinate.new(x + c.x, y + c.y);
6    }
7  }
8  def c1 := Coordinate.new(1,2);
9  def c2 := Coordinate.new(3,4);
10 def c3 := c1.move(c2);
11 c2.x := 42;
```

Lines 8-10 initialize two coordinate objects and compute their addition which is stored in `c3` variable. As a result, there is now a dependency of `c3` to `c1` and `c2`. In line 11 we made a state change (here in the form of an assignment) to one of the reactive values (i.e., a field of object stored in `c2`). This change is then propagated to the reactive value `c3`.

In brief, our vision considers the objects themselves to be the reactive values. In other words, the objects are streams of local states that can change and yet maintain their identity. The table 1 explains the essence of our model. Even though the `Coordinate` identifier is not updated by means of an assignment, the fact that it refers to an object that *is* modified triggers a reevaluation of all its dependent expressions (i.e., `Coordinate.move(c)` gets reevaluated).

3. REACTIVE OBJECTS

The reactive objects model consists of three key concepts: reactive objects, reactive fields and reactive methods. Reactive objects are objects that unify the concept of a reactive value into an imperative object-oriented setting. To fit well in such a setting, reactive objects require special semantics for the assignment of a field and method invocation. The semantics of reactive fields and reactive methods are the essential building blocks that take care of fundamental concepts of reactive programming such as lifting and automatically initiate the propagation of change.

In our model, objects are the unit of reactivity and propagation of change is triggered by performing assignments on reactive fields. When a field of a reactive object is changed, the local state of the reactive object is modified and the object propagates this change. An assignment thus results in a reevaluation of all the reactive objects that depend on the changed reactive object.

The semantics of a method invocation on a reactive object are defined by the argument list. If the argument list contains at least one reactive object, the method will be lifted implicitly. Lifting a method over an argument means recording a dependency from the method's result to the argument. Invoking methods with reactive objects as arguments effectively builds a dependency graph. The model tracks dependencies to make sure that the change of a reactive object can be propagated throughout the reactive application.

The semantics of lifting are defined in such a way that the invocation of a lifted method always returns a reactive object that encapsulates the return value of the evaluation of the method body. The value that is returned by such a lifted method is either already a reactive object, or it is a native value that is wrapped in a reactive object. The local state of the resulting reactive object is updated every time the method is reevaluated to reflect state changes up in the chain of dependencies. This allows us to define other reactive abstractions in terms of such a reactive object that is returned by a lifted method.

We distinguish between methods that do not perform side-effects, called *accessors*, and methods that do perform side effects, called *mutators*. Both types of methods have different semantics which are described below;

Accessor Methods. Accessor methods do not only depend on the reactive arguments with which they are invoked, but also on state changes of the object on which the method is defined. The object on which the reactive field is defined thus not only depends on the reactive objects that were passed as an argument to invoke the reactive method, but it also depends on `self` (i.e., `this` in Java). This is useful in situations where a method exposes information that is calculated based on a number of fields of a reactive object. These semantics are there to make sure that if a field points to the return value of such an accessor-method, it will always be in a consistent state with the state of the object that encapsulates said accessor method.

Mutator Methods. Mutator methods will not be reevaluated when its encapsulating field changes. Such methods are thus not allowed to place dependencies on the object on which they are defined as such a dependency could result in a cycle in the dependency graph. The consequence of having a cycle in the dependency graph is that an invocation of a mutator method would trigger a side-effect in the object, which in turn will cause another reevaluation (and thus invocation) of the method. Another problem that would occur when a mutator method is reevaluated whenever `self` changes, is that the reactive object may end up in an inconsistent state.

Table 1: Reactive Object-Oriented Programming Explained

	expression *using* `Coordinate` (e.g., `Coordinate.move(c)`)	expression *accessing* `Coordinate` (e.g., `Coordinate.x`)
Suppose the `Coordinate` variable is assigned	`Coordinate.move(c)` is reevaluated	`Coordinate.x` is reevaluated
Suppose `Coordinate`'s field x is assigned	`Coordinate.move(c)` is usually not reevaluated but is reevaluated in our model	`Coordinate.x` is reevaluated

3.1 Integration of Reactive Values with Object-Oriented Constructs

We now discuss how reactive objects behave with respect to traditional object-oriented constructs such as inheritance, delegation, constructors and object identity.

Inheritance Although we have not fully investigated inheritance yet, we propose semantics for two kinds of objects present in our model, namely reactive and traditional objects. If a reactive object (the child) inherits from a reactive object (the parent), a reactive object is returned. The returned object will be a reactive object encapsulating all of the fields and methods defined in the child and parent. Since it is a reactive object, it will also keep track of dependencies and trigger propagation of change if necessary.

If a traditional object inherits from another traditional object, the return value will be a traditional object as well with the fields and methods of the child as well as the parent encapsulated in the returned object. The syntax of the inheritance rules in an object-oriented programming language thus do not alter in this case.

Constructors The lifting semantics of our model do not apply to constructors. This means that even though such a constructor is invoked with one or more reactive objects in its argument list, it will not be lifted. The instantiation of a method will thus not be reevaluated due to the change of a field of any reactive object. Lifting such constructors would render it impossible to instantiate reactive objects that encapsulate other reactive objects.

Object Identity The lifting semantics of our model do not apply to functions that check for equality between objects. To preserve object identity, it is important that the contents of reactive objects (that wrap the return value of a lifted method) are changed whenever a method is reevaluated rather than creating a new reactive object every time change is propagated. When a reactive method is invoked, the method is lifted and the contents of the reactive object (instead of the reactive object itself) is used throughout the rest of the evaluation. When we compare object identity, these semantics are not applied because we want to check for the identity between the objects themselves and not between the objects they wrap.

Delegation Our model does not provide any special semantics in terms of delegation. Methods that are defined inside reactive objects can invoke methods on other reactive objects as well as on traditional objects. The semantics of the other way around do not change either, i.e., methods that are defined inside traditional objects can invoke methods on other traditional objects as well as invoke methods on reactive objects.

3.2 Reactive Objects in AMbientTalk (ROAM)

In this section, we introduce the language features offered in ROAM and we will see how they adhere to the model that is explained above. We will do so by implementing *"Reactive Circles"*, a small application that allows users to draw circles and ovals. Changes to the characteristics of the shapes will automatically update in the GUI. For example, one can define circles that follow the movement of the mouse. The implementation of this functionality is shown in listing 2.

Listing 2: Implementing reactive circles in ROAM

```
1  def mouse := reactiveObject: {
2    def x;
3    def y;
4    def clicked;
5  };
6  def ReactiveCircle := reactiveObject: {
7    def [jCircle, x, y, width, height, color];
8    def init(newx, newy, w,h,c) {
9      [x,y,width,height,color] := [newx, newy, w, h, c];
10     jCircle := JCircle.new(...);
11   };
12   def move(coordinates) {
13     x := coordinates.x;
14     y := coordinates.y;
15   };
16 };
17 def canvas := reactiveObject: {
18   def draw(circle) {
19     circle.jCircle.update(circle.x, circle.y, circle.
         width, circle.height, circle.color);
20   };
21 };
```

The code above shows three reactive objects. The first reactive object (defined in lines 1-5) represents the mouse. It has three fields: an x- and y-coordinate and a boolean indicating whether the mouse button is being clicked. At each point in time these fields hold the current state of the mouse.

The second reactive object (defined in lines 6-16) represents a circle shape in the application. A circle first holds a reference to a Java `RBall` object[1] in the `jball` field. Additionally, a circle has fields for the x- and y-coordinate of the center, width, height and color. `ReactiveCircle` also defines a `move` method that accepts an object that holds coordinates as an argument and changes the x- and y-coordinate of the reactive circle accordingly. It is important to note that

[1]AmbientTalk features linguistic symbiosis with Java. Developers can use Java classes and objects from within AmbientTalk programs and vice versa.

this method, since it is defined inside a reactive object, is a reactive (mutator) method. This means that the method is automatically lifted when it is invoked with a reactive object as its argument and that it will be reevaluated whenever a reactive field of this reactive object (that was passed as an argument) is reassigned.

Finally, the `canvas` object (defined in lines 17- 21) represents the canvas on which shapes are drawn. The canvas has one reactive method `draw` that updates the circles in the Java GUI. Note that the glue-code interfacing with Java is only necessary because we use the Java Swing and AWT frameworks to draw the GUI. In a full-fledged version of ROAM we would have a native reactive GUI-toolkit that eliminates the need for this glue-code.

The following code snippet creates two circles and draw them on the canvas.

```
22 def c1 := ReactiveCircle.new(5,150,20,20,'green);
23 def c2 := ReactiveCircle.new(20,20,110,110,'yellow);
24 canvas.draw(c1);
25 canvas.draw(c2);
```

The reactive method `draw` now depends on its arguments: the reactive objects representing the circles.

Below, in lines 26 and 27 we assign the width and color fields of respectively `c1` and `c2`. Because `c1` & `c2` are reactive objects, these assignments trigger a propagation of change and therefore reevaluation of the calls of the `draw` method.

```
26 c1.width := 40;
27 c2.color := 'blue;
28 c1.move(mouse);
```

Line 28 invokes the method `move` on `c1` with the reactive object `mouse` as its argument. This means that whenever the coordinates of `mouse` change (i.e., every time the mouse is moved), the `move` method is invoked on `c1`. The reevaluation of `move` assigns the `x` and `y` field of `c1`. This evokes the redrawing of `c1` because the reactive `draw` method was invoked with the reactive object `c1` (on line 24).

3.3 Limitations

As discussed before, objects are the unit of reactivity in ROAM. This coarse grained granularity renders it impossible to distinguish between different kinds of events occurring within one object. If we revisit the mouse example in listing 2 on lines 1 to 5, we see that an event source (i.e., the mouse) is implemented as a reactive object. The mouse exposes two kinds of events: the clicking of the mouse button (the field `clicked` is assigned) and the moving of the mouse (the fields `x` and `y` are reassigned). In the current implementation of ROAM it is however not possible to distinguish between these two events. The `move` method for example that is invoked on line 28 in listing 2 is reevaluated whenever one of the fields of `mouse` happens to change. This means that it will be reevaluated whenever the mouse moves but also whenever the mouse is clicked. We believe that it would be useful to introduce additional abstractions that allow to encapsulate multiple events of one concept (i.e., the mouse) inside one reactive object along with the functionality to distinguish which event was triggered.

The current implementation of ROAM enables embedding a reactive object `ro2` in another reactive object `ro1`. If `ro2` changes, only the dependents of `ro2` are reevaluated and not those of `ro1`. This means, that object composition has no effect on change propagation. We feel some use cases would benefit from a propagation strategy in which the change of the embedded `ro2` reevaluates the dependents of `ro2` as well as those of `ro1`. We are looking to further explore these semantics in the future.

4. CONCLUSION

In this paper we have identified two research tracks that target the marriage of reactive programming and object-oriented programming. In contrast to traditional approaches, which start from functional reactive abstractions and add object-oriented mechanisms to compose reactions, we explore *reactive object-oriented programming*. We have introduced a model for reactive objects in which objects are the unit of reactivity. Finally, we presented ROAM, an experimental reactive framework that implements the reactive objects model.

5. REFERENCES

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition, 1996.

[2] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter. A survey on reactive programming. *ACM Comput. Surv.*, 45(4):52:1–52:34, Aug. 2013.

[3] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *Proceedings of the 15th European conference on Programming Languages and Systems*, ESOP'06, pages 294–308, Berlin, Heidelberg, 2006. Springer-Verlag.

[4] C. Elliott and P. Hudak. Functional reactive animation. *SIGPLAN Not.*, 32(8):263–273, Aug. 1997.

[5] P. Haller and M. Odersky. Event-based programming without inversion of control. In *Proc. Joint Modular Languages Conference*, volume 4228 of *Lecture Notes in Computer Science*, pages 4–22. Springer, 2006.

[6] I. Maier and M. Odersky. Deprecating the observer pattern with Scala.React. Technical report, May 2012.

[7] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: a programming language for ajax applications. In *ACM SIGPLAN Notices*, volume 44, pages 1–20. ACM, 2009.

[8] G. Salvaneschi, G. Hintz, and M. Mezini. REScala: Bridging between object-oriented and functional style in reactive applications. Technical report, 2013.

[9] T. Van Cutsem, S. Mostinckx, E. Gonzalez Boix, J. Dedecker, and W. De Meuter. Ambienttalk: object-oriented event-driven programming in mobile ad hoc networks. In *Inter. Conf. of the Chilean Computer Science Society (SCCC)*, pages 3–12. IEEE Computer Society, 2007.