REScala Reference Manual

Guido Salvaneschi
Technical University of Darmstadt
salvaneschi@informatik.tu-darmstadt.de

November 2013

Contents

1 Introduction

This manual covers the main features of the REScala programming language. The manual introduces the concepts related to functional reactive programming and event-based programming from a practical perspective. The readers interested in a more general presentation of these topics can refer to Section 7 for the essential references.

Intended audience and prerequisites This manuscript is mainly intended for students who approach reactive programming in Scala for the first time. The manual assumes a basic knowledge of the Scala [8] language. No previous knowledge of reactive programming is assumed.

While a major aspect of REScala's design is the integration of events and signals, these two features can be used separately. A likely scenario is the use of REScala events only to design application that do not need time-changing values.

2 Signals and Vars

A signal is 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.

Consider the following example:

```
1 var a = 2
2 var b = 3
3 var c = a + b
4 println(a,b,c) //-> (2,3,5)
5 a = 4
6 println(a,b,c) //-> (2,4,5)
7 c = a + b
8 println(a,b,c) //-> (4,3,7)
```

Line 3 specifies the value of `c` as a function of `a` and `b`. Since Line 3 defines a statement, the relation $c = a + b$ is valid after the execution of Line 3. Clearly, when the value of `a` is updated, the relation $c = a + b$ is not valid anymore (Line 6). To make sure that the relation still holds, the programmer needs to recompute the expression and reassign `c`, like in line 7.

Reactive programming and REScala provide abstraction to express *constraints* in addition to statements. In REScala the programmer can specify that the constraint $c := a + b$ *always* holds during the execution of a program, and every time `a` o `b` change, the value of `c` is automatically recomputed.

For example:

```
1 val a = Var(2)
2 val b = Var(3)
3 val c = Signal{ a() + b() }
4 println(a.getVal,b.getVal,c.getVal) //-> (2,3,5)
5 a()= 4
6 println(a,b,c) //-> (2,4,5)
7 println(a.getVal,b.getVal,c.getVal) //-> (4,3,7)
```

In the code above, the `Signal` in Line 3 defines the constraint $c := a + b$. When one of the variables involved in the constraint is updated (Line 6), the expression in the constraint is recomputed behind the scenes, and the value of `a` is automatically updated.

As the reader may have noticed, expressing constraints in REScala requires to conform some syntactic conventions.

Defining Vars Programmers express reactive computations starting from vars. Vars wrap normal Scala values. For example `Var(2)` create a var with an [Int] value and initializes it to the value 2. Vars are parametric types. A var that carries integers has the type `Var[Int]`. The following are all valid vars declarations.

```
val a = Var(0)
val b = Var("Hello World")
val c = Var(false)
val d: Var[Int] = Var(30)
val e: Var[String] = Var("REScala")
val f: Var[Boolean] = Var(false)
```

Assigning Vars Vars can be directly modified with the `()=` operator. For example `v()=3` replaces the current value of the `v` var with `3`.

Defining Signals Signals are defined by the syntax `Signal`{*sigexpr*}, where *sigexpr* is a side effect-free expression. Signals are parametric types. A signal that carries integers has the type `Signal[Int]`.

Signal expressions When, inside a signal expression defining a signal `s`, a var or a signal is called with the `()` operator, the var of the signal are added to the values `s` depends on. In that case `s` *is a dependency* of the vars and the signals in the signal expression. All the following code snippets define valid signal declarations.

```
val a = Var(0)
val s: Signal[Int] = Signal{ a() + 1 }
```

```
val a = Var(0)
val b = Var(0)
val s = Signal{ a() + b() } // Multiple vars is a signal expression
```

```
val a = Var(0)
val b = Var(0)
val c = Var(0)
val s = Signal{ a() + b() }
val t = Signal{ s() * c() + 10 } // Mix signals and vars in signal expressions
```

```
val a = Var(0)
val b = Var(0)
val c = Var(0)
val s = Signal{ a() + b() }
val t = Signal{ s() * c() + 10 }
val u = Signal{ s() * t() } // A signal that depends on other signals
```

```
val a = Var(0)
val b = Var(2)
val c = Var(true)
val s = Signal{ if (c()) a() else b() }
```

```
def factorial(n: Int) = ...
val a = Var(0)
val s: Signal[Int] = Signal{ // A signal expression can be any code block
  val tmp = a() * 2
  val k = factorial(tmp)
  k + 2  // Returns an Int
}
```

Reading reactive values The current value of a signal or a var can be accessed using the `getVal` method. For example:

```
val a = Var(0)
val b = Var(2)
val c = Var(true)
val s: Signal[Int] = Signal{ a() + b() }
val t: Signal[Boolean] = Signal{ !c() }

val x: Int = a.getVal
val y: Int = s.getVal
val x: Boolean = t.getVal
```

2.0.1 Example: speed

The following example computes the displacement `space` of a particle that is moving at constant speed `SPEED`. The application prints all the values of the displacement.

```
1  val SPEED = 10
2  val time = Var(0)
3  val space = Signal{ SPEED * time() }
4
5  space.changed += ((x: Int) => println(x))
6
7  while (true) {
8    Thread sleep 20
9    time() = time.getVal + 1
10 }
11
12 -- output --
13 10
14 20
15 30
16 40
17  ...
```

The application behaves as follows. Every 20 milliseconds, the value of the `time` var is increased by 1 (Line 9). When the value of the `time` var changes, the signal expression at Line 3 is executed and the value of `space` is updated. Finally, the current value of the `space` signal every time the value of the signal changes. Technically, this is achieved by converting the `space` signal to an event that is fired every time the signal changes its value (Line 5). The conversion is performed by the `changed` operator. The `+=` operator attaches an handler to the event returned by the `changed` operator. When the event fires, the handler is executed.

Line 5 is equivalent to the following code:

```
val e: Event[Int] = space.changed
val handler:  (Int => Unit) =  ((x: Int) => println(x))
e += handler
```

3 Events

RESCALA supports different kind of events. Imperative events are directly triggers from the user, declarative events are triggered when the events they depend on are triggered. In reactive applications, events are typically used to model changes that happen at discrete points in time. For example a mouse click from the user. Some generalities about RESCALA events.

- Events carry a value. The value is associated to the event when the event is fired and received by all the registered handlers.
- Events are generic, like `Event[T]` and `ImpertiveEvent[T]` where `T` is the value carried by the event.
- Both imperative events and declarative events are subtypes of `Event[T]` and can referred to generically.

3.1 Imperative events

RESCALA imperative events are triggered imperatively by the programmer. One can think to imperative events as a generalization of a method call which supports (multiple) bodies that can be registered and unregistered dynamically.

3.1.1 Defining Events

Imperative events are defined by the `ImperativeEvent[T]` type. The value of the parameter `T` defines the value that is attached to the event. An event with no parameter attached has signature `ImpertiveEvent[Unit]`. The following code snippet show valid events definitions:

```
val e1 = new ImperativeEvent[Unit]()
val e1 = new ImperativeEvent[Int]()
val e2 = new ImperativeEvent[String]()
val e3 = new ImperativeEvent[Boolean]()
val e5: ImperativeEvent[Int] = new ImperativeEvent[Int]()
class Foo
val e4 = new ImperativeEvent[Foo]()
```

It is possible to attach more than one value to the same event. This is easily accomplished by using a tuple as a generic parameter type. For example:

```
val e1 = new ImperativeEvent[(Int,Int)]()
val e2 = new ImperativeEvent[(String,String)]()
val e2 = new ImperativeEvent[(String,Int)]()
val e3 = new ImperativeEvent[(Boolean,String,Int)]()
val e5: ImperativeEvent[(Int,Int)] = new ImperativeEvent[(Int,Int)]()
```

Note that since an imperative event is also an event. Therefore the following declaration is also valid:

```
val e1: Event[Int] = new ImperativeEvent[Int]()
```

3.1.2 Registering Handlers

Handlers are code blocks that are executed when the event fires. The += operator attaches the handler to the event. The handler is a first class function that receives the attached value as a parameter. The following are valid ways to attach a handler to an event:

```
var state = 0
val e1 = new ImperativeEvent[Int]()
e1 += { println(_) }
e1 += { println(_) }
e1 += (x => println(x))
e1 += ((x: Int) => println(x))
e1 += (x => {  // Multiple statements in the handler
  state = x
  println(x)
})
```

The signature of the handler must conform the signature of the event, since the handler is supposed to process the attached value and perform side effects:

```
val e2 = new ImperativeEvent[(Int,String)]()
e2 += (x => {
  println(x._1)
  println(x._2)
})
e2 += (x: (Int,String) => {
  println(x)
})
```

Note that events without arguments still need a `Unit` argument in the handler.

```
val e1 = new ImperativeEvent[Int]()
e1 += { x => println() }
e1 += { (x: Unit) => println() }
```

3.1.3 Firing Events

Events can be fired with the same syntax of a method call. When an event is fired, a proper value must be attached that conforms the signature of the event. For example:

```
val e1 = new ImperativeEvent[Int]()
val e1 = new ImperativeEvent[Boolean]()
val e1 = new ImperativeEvent[(Int,String)]()
e1(10)
e1(false)
e1((10,"Hallo"))
```

When a handler is registered to an event, the handler is executed every time the event is fired. The actual parameter is provided to the handler.

```
val e = new ImperativeEvent[Int]()
e += { x => println(x) }
e(10)
e(10)
-- output ----
10
10
```

If multiple handlers are registered, all of them are executed when the event is fired. Applications should not rely on any specific execution order for handler execution.

```
val e = new ImperativeEvent[Int]()
e += { x => println(x) }
e += { x => println("n: " + x)}
e(10)
e(10)
-- output ----
10
n: 10
10
n: 10
```

3.1.4 Unegistering Handlers

Handlers can be unregistered from events by the -= operator.

```
val e = new ImperativeEvent[Int]()
val handler1 = { x: Int => println(x) }
val handler2 = { x: Int => println("n: " + x) }

e += handler1
e += handler2
e(10)
e -= handler2
e(10)
e -= handler1
e(10)

-- output ----
10
n: 10
10
```

3.2 Declarative Events

RESCALA supports declarative events, which are defined as a combination of other events. For this purpose it offers operators like $e_1||e_2$, $e_1\&\&p$, $e_1.map(f)$. Event composition allows 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.

3.2.1 Defining Declarative Events

Declarative events are defined by composing other events. The following code snippet shows valid definitions for declarative events.

```
val e1 = new ImperativeEvent[Int]()
val e2 = new ImperativeEvent[Int]()

val e3 = e1 || e2
val e4 = e1 && ((x: Int)=> x>10)
val e5 = e1 map ((x: Int)=> x.toString)
```

3.2.2 OR Events

The event $e_1||e_2$ is fired upon the occurrence of one among e_1 or e_2. Note that the events that appear in the event expression must have the same parameter type (`Int` in the next example).

```
val e1 = new ImperativeEvent[Int]()
val e2 = new ImperativeEvent[Int]()
val e1_OR_e2 = e1 || e2
e1_OR_e2 += ((x: Int) => println(x))
e1(10)
e2(10)
–– output –––
10
10
```

3.2.3 Predicate Events

The event $e_1 \&\& p$ is fired if e_1 occurs and the predicate p is satisfied. The predicate is a function that accepts the event parameter as a formal parameter and returns `Boolean`. In other words the $\&\&$ operator filter the events according to their parameter and a predicate.

```
val e = new ImperativeEvent[Int]()
val e_AND: Event[Int] = e && ((x: Int) => x>10)
e_AND += ((x: Int) => println(x))
e(5)
e(15)
–– output –––
15
```

3.2.4 Map Events

The event $e_1 map f$ is obtained by applying f to the value carried by e_1. The map function must take the event parameter as a formal parameter. The return type of the map function is the type parameter value of the resulting event.

```
val e = new ImperativeEvent[Int]()
val e_MAP: Event[String] = e map ((x: Int) => x.toString)
e_MAP += ((x: String) => println("Here: " + x))
e(5)
e(15)
–– output –––
Here: 5
Here: 15
```

3.2.5 dropParam

The $dropParam$ operator transforms an event into an event with `Unit` parameter. In the following example the $dropParam$ operator transforms an `Event[Into]` into an `Event[Unit]`.

```
val e = new ImperativeEvent[Int]()
val e_drop: Event[Unit] = e.dropParam
e_drop += (_ => println("*"))
e(10)
e(10)
–– output –––
*
*
```

The typical use case for the $dropParam$ operator is to enforce compatibility among events with different types. For example the following snippet is rejected by the compiler since it attempts to combine two events of different types with the $||$ operator.

```
val e1 = new ImperativeEvent[Int]()
val e2 = new ImperativeEvent[Unit]()
val e1_OR_e2 = e1 || e2  // Compiler error
```

The following example is correct.

```
val e1 = new ImperativeEvent[Int]()
val e2 = new ImperativeEvent[Unit]()
val e1_OR_e2: Event[Unit] = e1.dropParam || e2
```

4 Conversion Functions

RESCALA provides functions that interface signals and events. Some of those functions can be called on a signal providing an event as the first parameter or can be called on an event providing signal as the first parameter. While the behavior is the same, the signature of the function is obviously different. For example, the function `snapshot` can returns a signal that is updated on an event occurrence. Hence, the function can be exposed both on the `Signal` and the `Event` interface. For example:

```
val e  = ... // An event
val s = ... // A signal

e.snapshot[V](s: Signal[V]): Signal[V]
s.snapshot[V](e : Event[_]): Signal[V]
```

For simplicity, in those cases, we document the signature of the function will all the interested objects in the parameters. For example:

```
def snapshot[V](e : Event[_], s: Signal[V]): Signal[V]
```

Fold

The `fold` function creates a signal by folding events with a given function. Initially the signal holds the `init` value. Every time a new event arrives, the function `f` is applied to the previous previous value of the signal and to the value associated to the event. The result is the new value of the signal.

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

Example:

```
val e = new ImperativeEvent[Int]()
val f = (x:Int,y:Int)=>(x+y)
val s: Signal[Int] = e.fold(10)(f)
e(1)
e(2)
assert(s.getValue == 13)
```

Iterate

Returns a value computed by `f` on the occurrence of an event. Differently from `fold`, there is no accumulator, i.e. the value of the signal does not depend on the previous values but only on the value carried by the event.

```
iterate[A](e: Event[_], init: A)(f: A=>A) :Signal[A]
```

Example:

```
var test: Int = 0
val e = new ImperativeEvent[Int]()
val f = (x:Int)=>{test=x; x+1}
val s: Signal[Int] = e.iterate(10)(f)
e(1)
assert(test == 10)
assert(s.getVal == 10)
e(2)
```

```
assert(test == 11)
assert(s.getVal == 10)
e(1)
assert(test == 12)
assert(s.getVal == 10)
```

Latest/latestOption

Returns a signal holding the latest value of the event `e`. The initial value of the signal is set to `init`.

```
latest[T](e: Event[T], init: T): Signal[T]
```

Example:

```
val e = new ImperativeEvent[Int]()
val s: Signal[Int] = e.latest(10)
assert(s.getVal == 10)
e(1)
assert(s.getVal == 1)
e(2)
assert(s.getVal == 2)
e(1)
assert(s.getVal == 1)
```

The `latestOption` function is a variant of the `latest` function which uses the `Option` type to distinguish the case in which the event did not fire yet. Holds the latest value of an event as `Some(val)` or `None`.

```
latestOption[T](e: Event[T]): Signal[Option[T]]
```

Example:

```
val e = new ImperativeEvent[Int]()
val s: Signal[Option[Int]] = e.latestOption(e)
assert(s.getVal == None)
e(1)
assert(s.getVal == Option(1))
e(2)
assert(s.getVal == Option(2))
e(1)
assert(s.getVal == Option(1))
```

Last

The `last` function generalizes the `latest` function and returns a signal which holds the last `n` events.

```
last[T](e: Event[T], n: Int): Signal[List[T]]
```

Initially, an empty list is returned. Then the values are progressively filled up to the size specified by the programmer. Example:

```
val e = new ImperativeEvent[Int]()
val s: Signal[List[Int]] = e.last(5)

assert(s.getVal == List())
```

```
e(1)
assert(s.getVal == List(1))
e(2)
assert(s.getVal == List(2,1))

e(3);e(4);e(5)
assert(s.getVal == List(5,4,3,2,1))
e(6)
assert(s.getVal == List(6,5,4,3,2))
```

List

Collects the event values in a (growing) list. This function should be used carefully. Since the entire history of events is maintained, the function can potentially introduce a memory overflow.

```
list[T](e: Event[T]): Signal[List[T]]
```

Snapshot

Returns a signal updated only when `e` fires. If `s` in the meanwhile changes its value, the change is ignored. When the event `e` fires, the resulting signal is updated ti the current value of `s`.

```
def snapshot[V](e : Event[_], s: Signal[V]): Signal[V]
```

Example:

```
val e = new ImperativeEvent[Int]()
val v1 =  Var(1)
val s1 = Signal{ v1() + 1 }
val s = e.snapshot(s1)
e(1)
assert(s.getValue == 2)
v1.setVal(2)
assert(s.getValue == 2)
e(1)
assert(s.getValue == 3)
```

Changed/change

The `changed` applies to a signal and returns an event that is fired every time the signal changes its value.

```
def change[U >: T]: Event[U]
```

Example:

```
var test = 0
val v =  Var(1)
val s = Signal{ v() + 1 }
val e: Event[Int] = s1.changed
e += ((x:Int)=>{test+=1})
v.setVal 2
assert(test == 1)
v.setVal 3
```

```
assert(test == 2)
```

The `change` function is similar to `changed`, but it provides both the old and the new value of the signal in a tuple.

```
change[U >: T]: Event[(U, U)]
```

Example:

```
val s = Signal{ ... }
val e: Event[(Int,Int)] = s.change
e += ((x:(Int,Int))=>{ ... })
```

Reset

When the `reset` function is called for the first time, the `init` value is used by the factory to determine the signal returned by the `reset` function. When the event occurs the factory is applied to the event value to determine the new signal.

```
reset[T,A](e: Event[T], init: T)(factory: (T)=>Signal[A]): Signal[A]
```

Example:

```
val e = new ImperativeEvent[Int]()
val v1 =  Var(0)
val v2 =  Var(10)
val s1 = Signal{ v1() + 1 }
val s2 = Signal{ v2() + 1 }

def factory(x: Int) = x%2 match {
  case 0 => s1
  case 1 => s2
}
val s3 = e.reset(100)(factory)

assert(s3.getVal == 1)
v1.setVal(1)
assert(s3.getVal == 2)
e(101)
assert(s3.getVal == 11)
v2.setVal(11)
assert(s3.getVal == 12)
```

Switch/toggle

The `toggle` function switches alternatively between the given signals on the occurrence of an event `e`. The value attached to the event is simply discarded.

```
toggle[T](e : Event[_], a: Signal[T], b: Signal[T]): Signal[T]
```

The `switchTo` function switches the value of the signal on the occurrence of the event `e`. The resulting signal is a constant signal whose value is the value carried by the event `e`.

```
switchTo[T](e : Event[T], original: Signal[T]): Signal[T]
```

The `switchOnce` function switches to a new signal provided as a parameter, once, on the occurrence of the event `e`.

```
switchOnce[T](e: Event[_], original: Signal[T], newSignal: Signal[T]):
Signal[T]
```

5 Common Pitfalls

In this section we collect some mistakes that are common to users that are new to reactive programming and REScala.

5.1 Accessing values in signal expressions

The `()` operator used on a signal or a var, inside a signal expression, returns the signal/var value *and* creates a dependency. The `getVal` operator returns the current value but does *not* create a dependency. For example the following signal declaration creates a dependency between `a` and `s`, and a dependency between `b` and `s`.

```
val s = Signal{ a() + b() }
```

The following code instead establishes only a dependency between `b` and `s`.

```
val s = Signal{ a.getVal + b() }
```

In other words, if `a` is updated, `s` is not automatically updated. With the exception of rare cases in which this behavior is desirable, this is almost certainly a mistake. As a rule of dumb, signals and vars appear in signal expressions with the `()` operator.

5.2 Attempting to assign a signal

Signals are not assignable. Signal depends on other signals and vars, the dependency is expressed by the signal expression. The value of the signal is automatically updated when one of the values it depends on changes. Any attempt to set the value of a signal manually is a mistake.

5.3 Side effects in signal expressions

Signal expressions should be pure. i.e. they should not modify external variables. For example the following code is conceptually wrong because the variable `c` is imperatively assigned form inside the signal expression (Line 4).

```
1 var c = 0                 /* WRONG - DON'T DO IT */
2 val s = Signal{
3   val sum = a() + b();
4   c = sum * 2
5 }
6  ...
7 foo(c)
```

A possible solution is to refactor the code above to a more functional style. For example by removing the variable `c` and replacing it directly with the signal.

```
val c = Signal{
  val sum = a() + b();
  sum * 2
}
 ...
foo(c.getVal)
```

5.4 Cyclic dependencies

When a signal `s` is defined, a dependency is establishes with each of the signals or vars that appear in the signal expression of `s`. Cyclic dependencies produce a runtime error and must be avoided. For example the following code:

```
val a = Var(0)              /* WRONG - DON'T DO IT */
val s = Signal{ a() + t() }
val t = Signal{ a() + s() + 1 }
```

creates a mutual dependencies between `s` and `t`. Similarly indirect cyclic dependencies must be avoided.

5.5 Objects and mutability

Vars and signals may behave unexpectedly with mutable objects. Consider the following example.

```
1  class Foo(init: Int){            /* WRONG - DON'T DO IT */
2    var x = init
3  }
4  val foo = new Foo(1)
5
6  val varFoo = Var(foo)
7  val s = Signal{ varFoo().x + 10 }
8  // s.getVal == 11
9  foo.x = 2
10 // s.getVal == 11
```

One may expect that after increasing the value of `foo.x` in Line 9, the signal expression is evaluated again and updated to 12. The reason why the application behaves differently is that signals and vars hold *references* to objects, not the objects themselves. When the statement in line Line 9 is executed, the value of the `x` field changes, but the reference hold by the `varFoo` var is the same. For this reason, no change is detected by the var, the var does not propagate the change to the signal and the signal is not reevaluated.

A solution to this problem is to use immutable objects. Since the objects cannot be modified, the only way to change a filed is to create an entirely new object and assign it to the var. As a result the var is reevaluated.

```
class Foo(x: Int){}
val foo = new Foo(1)

val varFoo = Var(foo)
val s = Signal{ varFoo().x + 10 }
// s.getVal == 11
varFoo()= newFoo(2)
// s.getVal == 12
```

Alternatively one can still use mutable objects but assign again the var to force the reevaluation. However this style of programming is confusing for the reader and should be avoided when possible.

```
class Foo(init: Int){
  var x = init
}
val foo = new Foo(1)

val varFoo = Var(foo)
val s = Signal{ varFoo().x + 10 }
```

```
// s.getVal == 11
foo.x = 2
varFoo()=foo
// s.getVal == 11
```

5.6 Functions of reactive values

Functions that operate on traditional values are not automatically transformed to operate on signals. For example consider the following functions:

```
def increment(x: Int): (Int=>Int) = x + 1
```

The following code does not compile because the compiler expects an integer, not a var as a parameter of the `increment` function. In addition, since the `increment` function returns an integer, `b` has type `Int`, and the call `b()` in the signal expression is also rejected by the compiler.

```
val a = Var(1)           /* WRONG - DON'T DO IT */
val b = increment(a)
val s = Signal{ b() + 1 }
```

The following code snippet is syntactically correct, but the signal has a constant value 2 and is not updated when the var changes.

```
val a = Var(1)
val b = increment(a.getVal)
val s = Signal{ b + 1 }
```

The following solution is syntactically correct and the signal `s` is updated every time the var `a` is updated.

```
val a = Var(1)
val s = Signal{ increment(a()) + 1 }
```

6 Technicalities

This section is meant to cover the implementation details of RESCALA that are necessary to correctly run the current the library.

6.1 Imports and dependencies

To work with REScala you need to properly import the reactive abstractions offered by the language. The following imports are normally sufficient for all REScala functionalities:

```
1 import react._
2 import react.events._
3 import macro.SignalMacro.{SignalM => Signal}
```

Note that signal expressions are currently implemented as macros, i.e. the body of a signal expression is macroexpanded. To use macros for signal expressions, the macro `SignalM` is imported and renamed to `Signal` (Line 3).

7 Essential Related Work

A complete bibliography on reactive programming is beyond the scope of this work. The interested reader can refer to [1] for an overview of reactive programming and to [7] for the issues concerning the integration of RP with object-oriented programming.

REScala builds on ideas originally developed in EScala [3] – which supports event combination and implicit events. Other reactive languages directly represent time-changing values and remove inversion of control. Among the others, we mention FrTime [2] (Scheme), FlapJax [6] (Javascript), AmbientTalk/R [4] and Scala.React [5] (Scala).

8 Acknowledgments

Several people contributed to this manual with their comments. Among the others Gerold Hintz and Pascal Weisenburger.

References

[1] E. Bainomugisha, A. Lombide Carreton, T. Van Cutsem, S. Mostinckx, and W. De Meuter. A survey on reactive programming. *ACM Comput. Surv. (To appear)*, 2013.

[2] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *ESOP*, pages 294–308, 2006.

[3] V. Gasiunas, L. Satabin, M. Mezini, A. Núñez, and J. Noyé. EScala: modular event-driven object interactions in Scala. AOSD '11, pages 227–240. ACM, 2011.

[4] A. Lombide Carreton, S. Mostinckx, T. Cutsem, and W. Meuter. Loosely-coupled distributed reactive programming in mobile ad hoc networks. In J. Vitek, editor, *Objects, Models, Components, Patterns*, volume 6141 of *Lecture Notes in Computer Science*, pages 41–60. Springer Berlin Heidelberg, 2010.

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

[6] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: a programming language for ajax applications. OOPSLA '09, pages 1–20. ACM, 2009.

[7] 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, pages 37–48, New York, NY, USA, 2013. ACM.

[8] Scala site. http://www.scala-lang.org/.