Reactive Programming Experience with REScala

Ragnar Mogk
Technische Universität Darmstadt
Germany

Guido Salvaneschi
Technische Universität Darmstadt
Germany

Mira Mezini
Technische Universität Darmstadt
Germany

ABSTRACT

Reactive programming is a recent programming paradigm that specifically targets reactive applications. Over the years, a number of reactive languages have been proposed, with different combinations of features, and various target domains.

Unfortunately, there is a lack of knowledge about the experience of developing software applications with reactive languages. As a result, a number of design choices in reactive programming languages remain disconnected from experience and the applicability of reactive programming to various domains remains unclear.

To bridge this gap, we report on our experience of developing reactive applications as well as teaching reactive programming in REScala, which we collected over several years of research and practice.

CCS CONCEPTS

• **Software and its engineering** → *Data flow languages*;

KEYWORDS

Reactive Programming, Programming Experience, Programming Paradigms, Language Design, Case Studies

ACM Reference Format:
Ragnar Mogk, Guido Salvaneschi, and Mira Mezini. 2018. Reactive Programming Experience with REScala. In *Proceedings of 2nd International Conference on the Art, Science, and Engineering of Programming (<Programming'18> Companion).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3191697.3214337

1 INTRODUCTION

Reactive programming (RP) is a programming paradigm that aims to expressing time-changing, interactive applications in purely functional languages. RP has been studied intensively [6, 8–11, 15, 28, 29, 33, 36, 38, 40, 41] and has spread from the original purely functional setting into imperative and object-oriented languages [3, 7, 21, 34].

However, little literature exists on developing applications in the reactive paradigm beyond the small case studies used to exemplify or evaluate each new RP language. As a result, not much is known about the programming experience with the reactive paradigm. For example, it is hard to evaluate if a RP language is a good fit for a specific domain, to compare different RP language designs, or even to build knowledge about recurring problems when developing with RP.

In this paper, we discuss our experience of developing reactive applications with REScala [34], a RP language embedded into Scala. Over the years, we used REScala to implement a number of case studies and libraries, gaining insights regarding several aspects of RP applications and languages including (1) the emergence of common idioms and patterns, (2) the role of RP in various domains, and (3) the design and the implementation of RP languages. For (1), we investigate usage experience with RP regarding various patterns that over the years emerged when using RP both in a purely functional settings as well as when including RP as part of imperative applications. Our case studies have been maintained over a period of multiple years and by different people. For (2), we consider RP libraries as well as standalone software and apply RP to several application domains. This experience is particularly important to gain insights about the applicability of RP to other domains than simple GUIs and animations – the traditional target of RP. For (3), we report on our experience with REScala specifically, both as users and developers of the language, to provide insights on how choices in the RP language design influence its use in reactive applications.

We believe that sharing our experience helps researchers to understand how RP is used effectively. It also guides developers to decide whether RP is applicable in their domain, and it supports designers of RP languages considering design tradeoffs.

The paper is structured as follows. Section 2 provides background on RP required for the rest of the paper. Section 3 contains our research questions, which share our experience in the different aspects of RP. The questions are grouped according to the three points above, and each questions can be read individually. Section 5 provides an overview of recent work on RP. Section 4 concludes.

2 REACTIVE PROGRAMMING BACKGROUND

Today's RP is derived from functional reactive programming (FRP). FRP was first applied to functional modelling of animations [11]. Early FRP systems compute individual frames of an animation given timestamp t of the frame. FRP languages provide composable operators for the computation of frames. In these systems, computation is *synchronous* and *pull based*, i.e., when the frame at time t is requested all operators are evaluated at that time and the resulting values are pulled into one final result.

More recently, reactive programming has been introduced into a number of mainstream languages mostly targeting interactive applications in the GUI domain [6, 21]. In the interactive setting, pull based evaluation of continuous time operators is found to be inefficient, because all inputs events have to be stored and reprocessed every time a frame is requested. To solve the inefficiency, push-based evaluation is introduced [10]. Input events are eagerly

<Programming'18> Companion, April 9–12, 2018, Nice, France

ACM ISBN 978-1-4503-5513-1/18/04…$15.00
https://doi.org/10.1145/3191697.3214337

pushed into the *reactive graph*, the graph of connected operators. For each input, operators recompute and cache their current value. Evaluation is still synchronous, i.e., all *reactions* (recomputations of operators) to an input event happen at the same time. The system produces the current state *now*, and it is impossible to query past or future timestamps t.

RP languages with synchronous semantics such as Flapjax [21] or FrTime [6] are different from languages such as reactive extensions [18], or stream processing systems [1, 42], where operators are asynchronous, i.e., there is no concept of *now*. This gap inspired researchers to combine asynchronous evaluation with synchronous RP languages [8].

2.1 REScala

REScala [34] adopts the synchronous, push-based propagation model, and supports two kinds of reactive abstractions, *events* and *signals* for discrete or continuous time changing values. Signals and events expose to the programmer a set of operators and conversions operators between the two.

A signal always holds a value, and represents state in the program, e.g., the current text of an input field. An event only has a value when the event *fires*, and it represents actions that occur in the program, e.g., when a button is clicked. The following example demonstrates part of the REScala API:

```
1  // Vars are signals, that take imperative input
2  val text = Var("Textfield")
3  // Evts are events, which are fired imperatively
4  val click = Evt[Unit]
5  // derives a signal, that counts the button clicks
6  val derived: Signal[Int] = click.count()
7  // combines two signals (+ is string concatenation)
8  val label = Signal { text() + derived() }
9  // read the current value of a signal
10 label.now // "Textfield 0"
11 //set the Var, every derived value updates
12 text.set("Button click") // "Button click 0"
13 //fire the evt, also automatic updates
14 click.fire() // "Button click 1"
15 //update multiple events and signals
16 transaction {
17   text.set("Transaction text")
18   click.fire()
19 } // label text is "Transaction text 2"
20 //add an imperative callback
21 click.observe(_ => println("the button was clicked"))
```

`Var` and `Evt` (Lines 2, 4) are created as inputs of changes from imperative code. Events and signals are derived from other events and/or signals (Lines 6, 8), as long as no cyclic dependencies are formed. Derived events or signals are automatically updated whenever an input changes (Lines 12, 14). The change is propagated synchronously, i.e., when multiple inputs change as one update (Line 16) then all derived events and signals are recomputed exactly once for that update. Intermediate values are prevented – a property called glitch freedom. Side effects are forbidden as part of normal operators, however, operators such as `count` in Line 6 have internal state managed by the language, and user defined imperative reactions (Line 21) are executed after a change has been fully processed. We strongly recommend the REScala manual[1] and the API documentation linked from the manual for details of specific operators or functionality.

Table 1: REScala case studies

Case study	Type	LoC	Domain
CRDTs	Library	706	Distributed
Datastructures	Library	688	Datastructures
Dividi	Application	254	Distributed
Editor	Application	1960	Text editing
Examples	Application	1713	Misc
Mill game	Application	428	Turn based
Paroli chat	Application	100	Chat
Pong game	Application	706	Realtime, distributed
Reactive streams	Library	93	Asynchrony
REScalafx	Library	70	GUI
REScalatags	Library	87	GUI
Swing	Library	1541	GUI
RSS	Application	1338	Content reader
Shapes	Application	1232	Drawing
Tests	Tests	587	Mics
Todolist	Application	177	Note taking
Universe	Application	475	Simulation

2.2 A Note on Scala Syntax

In the rest, we assume that Scala syntax is familiar to readers, or can be inferred form similarities to other languages. To help readers unfamiliar with Scala, some subtleties are clarified below:

```
1  // define a function
2  { (x: Int, y: Int) => x + y }
3  // type annotations can be inferred, and braces and
       parenthesis are often optional
4  stringEvent.map( someString => someString.length )
5  // unused parameters are named _
6  event.observe(_ => println("observed"))
7  // parenthesis or braces are used interchangeably
8  event.observe { _ => println("observed") }
```

Functions in Scala have very flexible syntax regarding the placement of parenthesis, braces, and type annotations. In general, on the left of `=>` there is a list of parameters, and on the right there is a sequence of statements for the function body (Line 2). The type of function parameters and the scope of the function is often provided by the surrounding context, e.g., in Line 4 the `map` operator of the `Event[String]` receives a function accepting a `String`. Underscore indicates unused parameters (Line 6) and parenthesis are interchangeable with braces (Line 8).

3 PROGRAMMING EXPERIENCE

Since the early development of REScala we implemented applications to verify the viability of the design of REScala. Since then,

[1] www.rescala-lang.com/manual

several people have been contributing to the project developing a number of case studies. The most important case studies are shown in Table 1.

The case studies cover different types of applications. Standalone applications provide self-contained functionalities. Reactive libraries are reusable libraries that expose an API based on RP. Since they heavily use reactive features, we also include the REScala test cases. The size of the case studies is very diverse, ranging from few lines of code to thousands lines of code (LoC column). Finally, the case studies cover different applicative domains which include text editing, simulation, graphics, and gaming (Domain column).

3.1 Idioms and Patterns

Certain idioms and patterns seem to reoccur in reactive programs, and we elaborate how they help with maintainability and testing of RP applications.

3.1.1 Idiomatic Code. As every programming paradigm, RP requires a specific coding style that takes time to learn and refine. Developers with an imperative background perform side effects on shared state rather than opting for functional processing of event streams. For example, to count how often an event is fired, a mutable variable is updated inside of a `map` expression, and the value of the variable is propagated:

```
var count = 0
val mapped: Event[Int] = event.map { _ =>
  count += 1 // bad: mutation inside operator
  count
}
```

However, mutations of `count` outside of the `map` are ignored by the language, and are hard to reason about. The idiomatic REScala solution is to use `fold`, an operator with internal state managed by the reactive language, that applies a user defined function to generate the new state from the old state:

```
val counted: Signal[Int] =
  event.fold(0) { (count, _) => count + 1 }
// or just `event.count`
```

In our experience, fold is adequate to model any stateful computations, but it requires programmers to understand purely functional state management (e.g., using the accumulator and return value to manage state). In some cases, it is necessary to handle multiple events in the same `fold` to make the state accessible in all cases. Examples are user interfaces, where a user modifies the same value by entering text, or dragging a slider. Our case studies include an example where an elevator computes the time it has spent waiting on the current floor, depending on a `reachedFloor` event which resets the value and a `tick` event increasing the value:

```
val waitingTime: Signal[Int] =
  reachedFloor.reset(0) { _ =>
    tick.iterate(0) { acc =>
      if (isWaiting()) acc + 1 else acc
  }}
```

The code above is hard to understand even for seasoned RP developers and requires a nested call of `iterate` inside a `reset` both of which have semantics derived from fold. Fortunately, there is a more idiomatic alternative when folding over multiple events. REScala supports an extended syntax for folds, which takes a list of events and associated update functions, to compute the next state from the current one:

```
val waitingTime: Signal[Int] =
  Events.fold(0)( acc => Seq(
    // The -> operator creates a pair
    reachedFloor -> {_ => 0},
    tick -> {_ => if (isWaiting()) acc + 1 else acc }
  ))
```

The extended fold states the intention more clearly. Each event is paired with a handler function (Line 4) describing the behavior of the fold when the corresponding event occurs. The `reachedFloor` resets the state to one, and `tick` increments the state by one if the elevator is currently waiting. When multiple events occur at the same time, the handlers are executed from top to bottom.

The correct use of fold is one of the common problems we see when developers with an imperative background write applications with RP. However, developers seem to know that mutable state is problematic and they try to use `fold`, but they often end up with complex and hard to understand `fold` expressions. We believe the issue to be a lack of available examples and references for writing readable complex fold expressions.

3.1.2 Maintenance. For maintainability, we want to highlight one crucial part of RP: statically known dependency relations with automatic update propagation. As a contrived example – for the sake of brevity – consider a door system that imperatively turns the lights on or off, when the doors open or close:

```
var light = On
object DoorSystem {
  def onClose() = { light = Off }
  def onOpen() = { light = On }
}
```

Other objects rely on the state of `light` and must be notified when it changes, but the API does not specify how notifications happen. With RP dependencies are made explicit, and notifications happen automatically:

```
object DoorSystem {
  val closed: Signal[Boolean]
}
val light = Signal {
  if (DoorSystem.closed()) Off else On
}
```

In the updated example, the `closed` state of the door is a proper part of the public API of the `DoorSystem` object, including the fact that the state changes over time. The example generalizes to bigger projects. RP operators integrate seamlessly into existing APIs, and work with existing tools: the type system, IDE based refactoring, linters, etc.

It is possible to achieve a similar result by manually encoding time changing values and observers into the API of an objects, however maintaining the encoding requires discipline. In our experience with the REScala case studies, the better API design is used in most cases by all developers.

3.1.3 Testing with Reactive Programming. Our experience with testing in the cases studies is limited to unit tests. We observed no particular difficulties: Testing RP is not different than testing code in the OO paradigm. RP operators are tested at a similar granularity as method calls, and tests exercise the operators and observe the resulting values, using imperative accessors. For events, it is often useful to use the `list` operator (Lines 5, 7) to make all occurrences of the event accessible as a list:

```
1  val input = Evt[String]
2  val greeting: Event[String] =
3    input.map(name => s"Hello $name")
4  val inputLog: Signal[List[String]] =
5    input.list()
6  val greetingLog: Event[List[String]] =
7    greeting.list()
8
9  assert(inputLog.now() == Nil)
10 assert(greetingLog.now() == Nil)
11
12 input.fire("world")
13
14 assert(inputLog.now() == List("world"))
15 assert(greetingLog.now() == List("Hello world"))
```

RP programs are written in a more functional style composing behavior based on small individual functions, such as the `map` in Line 3.

To test complex reactive graphs the language provides tools similar to mock based solutions in object oriented languages. It is possible to simulate the value of individual events or signals to test derived operators on specific inputs. Assume that the value of some `input` signal is not under control of the tests, but a derived signal `testMe` is tested:

```
1  val input: Signal[String] = ...
2  val testMe = Signal { input().toLowerCase }
3
4  reevaluate(testMe).assuming(input -> "TEST String")
5  assert(testMe.now() == "test string")
```

The code in Line 4 forces a reevaluation of the signal, and simulates the values of the inputs of the signal. Using this technique allows very fine granular tests without modifying the code to be tested, and we believe the strategy supports testing arbitrarily complex applications, much bigger that the examples in our case studies.

3.2 Applicability of Reactive Programming

Since its early days, RP has been applied to more domains outside of purely functional animations. We highlight some of them, and report how RP has adapted and grown to support the new use cases, and how RP compares existing similar paradigms.

3.2.1 Application Domains. The REScala case studies in Table 1, consist of interactive application domains, such as text editing, note taking, games, and networking, and one non-interactive simulation (Universe).

Regardless of the application domain, our case studies demonstrate how interactive tasks are modeled in RP. We discuss examples and highlight why RP simplifies the development of interactive tasks. RP supports detecting gestures and other complex user input using stateful event processing, as shown by Shapes – a drawing application. Automatic management of state in RP allows one to write applications with persistent state such as Todolist[2], a web application which stores a list of tasks (surviving browser restarts). A design similar to interactive software was also valid of the Dividi distributed ledger that merges local and remote updates using eventual consistent semantics. Handling network messages is similar to handling user input, and automatic change propagation lends itself very well to ensure the whole system becomes eventually consistent, when all network messages are received.

On the other hand, RP has less advantages over the imperative paradigm, in the case of non interactive applications such as the Universe simulation. The simulation requires an update loop to compute the next state given the current one, without the need to process new input. The cyclic structure of the update loop is not handled by RP. In this case, the application uses imperative code. We believe that better integration of cyclic processes into RP is an area that would benefits from further research.

Besides our direct experience in the case studies, it is interesting to note that special purpose RP languages have been designed for domains such as robotics [15], network switches [13], and wireless sensor networks [27]. In these works, RP is used to handle the interaction between multiple computer systems. Our conclusion is that the concepts of RP are demonstrated to be useful in every interactive domain, either with a general purpose or special purpose language implementation.

3.2.2 Purity in (Functional) Reactive Programming. RP has been initially developed in the context of pure functional languages, and operators in RP are still free of side effects. However, imperative languages allow to change the graph during execution. Consider creating a button for a UI, deriving a label signal that states how often the button was clicked, and displaying the label text on the button itself:

```
val button = new Button
val label: Signal[String] = button.click.count
  .map(c => s"This button was clicked $c times")
label.observe(txt => button.text = txt)
```

The last line sets the text of the button when the label changes – an imperative interaction not allowed in a pure language. In pure languages, the label text of the button has to be provided when the button is created, which leads to a cyclic definition (button depends on label, and label on button). A solution is shown in the following code example where the handler is registered as part of the button creation, and no imperative change is required:

[2]Todolist is an implementation of http://todomvc.com/.

```
def pipeline(click: Event[Unit]): Signal[String] =
  click.count
  .map(c => s"This button was clicked $c times")
val button = new Button(pipeline)
```

We provide a function to the button, and the function describes the pipeline to process the button clicks. With this approach the created pipeline is private to the button. To share events and signals between multiple UI components (buttons, labels, etc.), they all have to be initialized together with a combined and predefined pipeline of operators. As a result, purely functional RP languages often only limited use cases, like one animation, or a single UI window. Elm[3] applied FRP to larger web applications. However, to keep purity the RP abstractions were removed from Elm, as they suffered from the problems we explained. In REScala we make the opposite choice and sacrificed purity to keep the abstractions of RP and integrate them with imperative applications. The integration allows programmers to add and reconnect events and signals as required.

3.2.3 REScala Libraries. Motivated by the need to develop case studies for REScala in various domains, we have been working on a number of libraries, including UI libraries, collections, libraries for distribution, concurrency, and fault tolerance.

Many such libraries are implemented on top of REScala, using the API. Examples include the integrations with Swing and JavaFX, both of which use the APIs for imperative handlers to bridge between the Java UI libraries, and the RP language.

For a number of libraries we took an approach that we call *shallow reactivity* where a wrapper exposing the reactive abstractions is added to the original library API. As an example, consider Swing, the Java graphic library. The REScala RESwing library provides Swing component with reactive operators, e.g., an input field is a signal holding the current text inserted by the user. The reactive library is a wrapper around Java Swing components, and uses them internally. Wrapping the internals of Swing allows RP developers to directly use values and components provided by RESwing, without the necessity to use imperative code to manage updates. RP is well suited to express interactions with UI libraries, and modern libraries such as JavaFX and the HTML DOM have interfaces that are designed on RP concepts.

However, designing reactive collections as a wrapper results in operators having wildly unexpected runtime behavior. For example, consider our collection library. In this case, adding an element to a list is expected to be a cheap operator prepending a new node to the list. When adding to the wrapped list, however, derived properties such as the size of the list are recomputed. The recomputation traverses the full list and counts all elements, because the reactive wrapper library does not use knowledge about the internal behavior of collections.

In this case, a better approach is to integrate RP with the internal code of collections, to enable incremental behavior for derived operators, as shown by the reactive lists of Maier [19]. The resulting library is used similar to the wrapper library, but with efficient incremental updates.

[3]Elm is the full version of FElm [8]

3.2.4 Reactive Programming vs. Stream Processing. Stream processing systems such as Spark [42] or Flink [1] and to a lesser extend also reactive extensions [18] provide an API that looks similar to RP[4] on the surface, but with different semantics and use cases.

As we discussed in Section 2, RP is concerned with the current state of an application *now*, and provides a fundamentally synchronous view on changes at each point in time. Stream processing, on the other hand, is concerned with individual asynchronous data elements combined in streams. Multiple changes that occur at the same time are a priori unrelated in streaming systems, and the use of special operators is required to correlate inputs. Streaming operators are allowed to rearrange, group, and aggregate input streams, because there is no inherent relation among inputs that occur at the same time.

As a price for the flexibility when processing individual inputs, stream processing systems do not offer signals and provide no glitch freedom. The lack of a consistent view on state of multiple inputs, makes common interactive tasks hard to express. Recurring tasks in our case studies, such as "*when* the mouse is dragged, draw the *selected* shape in the *current* color and line width", are straightforward to express in RP but have no simple equivalent in stream processing.

3.3 Reactive Languages

As with any language RP evolves with its usage. In this section we discuss considerations when designing an RP language, and show how our case studies guide the design of the language.

3.3.1 API Size. We discuss multiple RP languages and the size of their APIs, and how API design relates to the intended use cases and concepts.

FElm [8] is designed for asynchronous composition of GUIs, and requires three operators for this purpose, combining signals, folding over past values, and making a computation asynchronous. Fran [11] is designed for animations, a similar use case to FElm. Fran does not include asynchronous computations, but has operators for transforming time, and dynamically recombining signals, required to change the speed and behavior of animations. In total, the Fran API lists around 10 to 20 operators in the API, including some minor variations of common operators.

In contrast, reactive extensions [18], includes operators specific to collections, asynchronous execution, and mathematical aggregation. The result is a library with over 450 operators, 80 of which are considered *core operators*[5].

REScala does not target a specific application domain, and includes operators for combining events, signals, and converting between the two, as well as operators which integrate with imperative code. On the other hand, REScala refrains from including operators for a specific domain, such as mathematical aggregations. In total REScala has an API size of around 40 operators. About half are convenience operators for events, which form a small event processing sub language. The inclusion of operators into the REScala

[4]Sometimes stream processing is also called RP. Here we talk about the kind of RP as implemented by REScala.

[5]Core operators are those that exist not only because of multiple overloads, such as `averageInteger` and `averageDouble`, see http://reactivex.io/documentation/operators.html

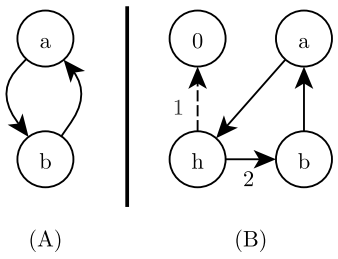

Figure 1: Creating a cycle

API is driven by our case studies. Most of the REScala operators are generic and used in multiple case studies – their usage is not limited to specific domains. Other operators such as `list` (c.f., Section 3.1.3) are mainly used during testing, but are so useful there to warrant their inclusion. The case studies often derive more complex or specific operators from the basic ones, however none of them are common enough to warrant inclusion into REScala. Some commonly useful operators for a specific domain are offered in libraries, (c.f., Section 3.2.3).

3.3.2 API Design. RP languages minimize the number of concepts necessary to use their API, exposing only events and signals abstractions to programmers. As we discuss in Section 3.3.1, RP languages offer a rich set of operators in their API, building on top of events and signals. The derived operators have a semantics that is expressible as (a combination of) basic concepts and such semantics in REScala directly corresponds to the operator implementation.

For example, consider the `||` operator in REScala, which is derived from the following event expression:

```
def || [A](a: Event[A], b: Event[A]): Event[A] =
  Event { if (a().nonEmpty) a() else b() }
```

Accessing the value of an event inside an event expression, yields a `Option` representing the fired value. The event expression above defines, that the result of `a || b` is a new event `e` with the following behavior: If a fires a value (is `nonEmpty`), then e fires the value of a, otherwise e fires the value of b (if b is also empty, then e does not fire). Specifying operators using a small core set of concepts, makes the language much more accessible for novices and allows one to easily explain the semantics of complex programs.

3.3.3 Cycles. RP languages require that the dependency graph is acyclic to ensure that propagation terminates[6], but most RP languages do not statically enforce the graph to be acyclic. However, problems with cycles only occur rarely in practice. Consider the following code to create a cycle between signals a and b:

```
val a = Signal { b() + 1 }
val b = Signal { a() - 1 }
```

Figure 1(A) shows the cycle the code is supposed to create to create. However, the code does not compile for a RP language like REScala, because the embedding language prevents cyclic definitions of variables. To create a cyclic graph, imperative code is used[7]:

```
val h = Var(Signal(0))
val a = Signal { h()() + 1 }
val b = Signal { a() - 1 }
h.set(b)
```

Figure 1(B) illustrates this approach. The first line creates a helper `Var` (h) containing the constant signal with value 0. The signal a depends on h and the inner value of h – `h()` accesses the value of h and `h()()` additionally accesses the inner value. and the last line changes h to point to b.

Code that explicitly sets vars to other reactives is rare in practice. In our case studies, we find that only the universe simulation and the pong game have the potential for cycles. In both cases, the cycle is related to the update steps of the simulation, i.e., the position of the pong ball depends on the speed, and the speed depends on the position (the speed changes when the ball bounces of a wall). Cycles are part of the domain model of simulations, and in the case studies it is explicitly avoided to create cyclic graphs.

4 CONCLUSION

In recent years, RP has gained popularity with several proposals for languages embracing the reactive paradigm both in the academia and in industry. This paper addresses the lack of information available on the programming experience with reactive languages, providing insights on the design of the languages themselves, on the problems that developers encounter when using this paradigm, as well as on the application domains for which RP is effective.

Our experience with RP indicates efficient applicability in the domain of interactive applications and properly supports library modularization, testing and static typing. These observations show the potential of RP for supporting larger applications where maintenance is a challenge – a research line which we are actively investigating.

5 RELATED WORK

In the area of the implementation of RP languages, Ramson and Hirschfeld recently proposed Active Expressions [31] as a fundamental primitive for different RP implementations.

Tool support has lead to debuggers that specifically target RP [2, 35]. These debuggers adopt the dependency graph or a visualization of event streams as the model provided to developers to debug RP programs.

The use of RP has been investigated in specific application domains, e.g., the simulation of autonomous vehicles [12]. In some cases, the peculiarities of the application domain led to RP implementations tailored to embedded systems [37], IoT [4], robotics [15], network switches [14], and wireless sensor networks [27]. RP has also received formal treatment [16, 17].

[6] Limited and controlled cycles are permissible, if the cyclic computation reaches a stable state and terminates.

[7] Lazy initialization is an alternative to using explicit side effects.

A recent focus of RP is distribution [5, 9, 20, 23, 25, 32, 39]. Research includes new consistency models, such as eventual consistency via CRDTs for replicated signals [24] and notions of relaxed glitch freedom that account for an error margin [30]. Also, in the area of distributed RP, researchers proposed mechanisms to achieve fault tolerance. Leased signals [26] enable reacting to a partial failure when a remote host does not produce a value after a timeout. Recent investigations of fault tolerance in REScala support to store and recover program state upon failures, automatically update and share distributed parts of the state providing eventual consistency, and handle errors when programmer control is necessary [22].

ACKNOWLEDGMENTS

We thank all contributors to REScala and related projects. This work has been supported by the LOEWE initiative (Hessen, Germany) within the NICER project, by the European Research Council, advanced grant No. 321217, by the German Research Foundation (DFG) as part of projects A2 and C2 within the Collaborative Research Center (CRC) 1053 – MAKI, and by the DFG project SA 2918/2-1.

REFERENCES

[1] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. 2014. The Stratosphere platform for big data analytics. *VLDB* 23 (2014).

[2] Herman Banken, Erik Meijer, and Georgios Gousios. 2018. Debugging Data-Flows in Reactive Programs. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA.

[3] Elisa Gonzalez Boix, Kevin Pinte, and Wolfgang De Meuter. 2013. Object-oriented Reactive Programming is Not Reactive Object-oriented Programming. (2013). http://soft.vub.ac.be/Publications/2013/vub-soft-tr-13-16.pdf

[4] Ben Calus, Bob Reynders, Dominique Devriese, Job Noorman, and Frank Piessens. 2017. FRP IoT Modules as a Scala DSL. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS 2017)*. ACM, New York, NY, USA, 15–20.

[5] Andoni Lombide Carreton, Stijn Mostinckx, Tom Van Cutsem, and Wolfgang De Meuter. 2010. Loosely-coupled Distributed Reactive Programming in Mobile Ad Hoc Networks. In *Proceedings of the 48th International Conference on Objects, Models, Components, Patterns (TOOLS'10)*. Springer-Verlag, Berlin, Heidelberg, 41–60. http://dl.acm.org/citation.cfm?id=1894386.1894389

[6] Gregory H. Cooper and Shriram Krishnamurthi. 2006. Embedding Dynamic Dataflow in a Call-by-Value Language. In *Lecture Notes in Computer Science*. https://doi.org/10.1007/11693024_20

[7] Antony Courtney. 2001. Frappé: Functional Reactive Programming in Java. In *PADL '01 Proc. Third Int. Symp. Pract. Asp. Declar. Lang.* 29–44. https://doi.org/10.1007/3-540-45241-9_3

[8] Evan Czaplicki and Stephen Chong. 2013. Asynchronous Functional Reactive Programming for GUIs *(PLDI '13)*. ACM. https://doi.org/10.1145/2491956.2462161

[9] Joscha Drechsler, Guido Salvaneschi, Ragnar Mogk, and Mira Mezini. 2014. Distributed REScala: An Update Algorithm for Distributed Reactive Programming *(OOPSLA '14)*. ACM, New York, NY, USA. https://doi.org/10.1145/2660193.2660240

[10] Conal Elliott. 2009. Push-pull functional reactive programming. *Proc. 2nd ACM SIGPLAN Symp. Haskell - Haskell '09* (2009), 25. https://doi.org/10.1145/1596638.1596643

[11] Conal Elliott and Paul Hudak. 1997. Functional reactive animation, Vol. 32. ACM. https://doi.org/10.1145/258948.258973

[12] Bernd Finkbeiner, Felix Klein, Ruzica Piskac, and Mark Santolucito. 2017. Vehicle Platooning Simulations with Functional Reactive Programming. In *Proceedings of the 1st International Workshop on Safe Control of Connected and Autonomous Vehicles (SCAV'17)*. ACM, New York, NY, USA, 43–47.

[13] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. 2011. Frenetic: A Network Programming Language. *SIGPLAN Not.* 46, 9 (Sept. 2011). https://doi.org/10.1145/2034574.2034812

[14] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. 2011. Frenetic: a network programming language. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming (ICFP '11)*. ACM, New York, NY, USA, 279–291. https://doi.org/10.1145/2034773.2034812

[15] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. 2003. Arrows, Robots, and Functional Reactive Programming. In *Summer School on Advanced Functional Programming 2002, Oxford University (Lecture Notes in Computer Science)*, Vol. 2638. Springer-Verlag, 159–187.

[16] Alan Jeffrey. 2012. LTL Types FRP: Linear-time Temporal Logic Propositions As Types, Proofs As Functional Reactive Programs. In *Proceedings of the Sixth Workshop on Programming Languages Meets Program Verification (PLPV '12)*. ACM, New York, NY, USA, 49–60. https://doi.org/10.1145/2103776.2103783

[17] Tetsuo Kamina and Tomoyuki Aotani. 2018. Harmonizing Signals and Events with a Lightweight Extension to Java. The Art, Science, and Engineering of Programming, 2018, Vol. 2, Issue 3, Article 5. (2018). https://doi.org/10.22152/programming-journal.org/2018/2/5

[18] Jesse Liberty and Paul Betts. 2011. *Programming Reactive Extensions and LINQ*. Apress.

[19] Ingo Maier and Martin Odersky. 2013. Higher-Order Reactive Programming with Incremental Lists *(ECOOP'13)*. https://doi.org/10.1007/978-3-642-39038-8_29

[20] Alessandro Margara and Guido Salvaneschi. 2014. We Have a DREAM: Distributed Reactive Programming with Consistency Guarantees *(DEBS '14)*. ACM, New York, NY, USA. https://doi.org/10.1145/2611286.2611290

[21] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. 2009. Flapjax: A Programming Language for Ajax Applications, Vol. 44. ACM Press. https://doi.org/10.1145/1640089.1640091

[22] Ragnar Mogk, Lars Baumgärtner, Guido Salvaneschi, Bernd Freisleben, and Mira Mezini. 2018. Fault Tolerance in Interactive Applications. In *European Conference on Object-Oriented Programming (ECOOP 2018)*. Dagstuhl, Germany.

[23] Florian Myter, Tim Coppieters, Christophe Scholliers, and Wolfgang De Meuter. 2016. I now pronounce you reactive and consistent: handling distributed and replicated state in reactive programming. New York, New York, USA. https://doi.org/10.1145/3001929.3001930

[24] Florian Myter, Tim Coppieters, Christophe Scholliers, and Wolfgang De Meuter. 2016. I Now Pronounce You Reactive and Consistent: Handling Distributed and Replicated State in Reactive Programming. In *Proceedings of the 3rd International Workshop on Reactive and Event-Based Languages and Systems (REBLS 2016)*. ACM, New York, NY, USA, 1–8.

[25] Florian Myter, Christophe Scholliers, and Wolfgang De Meuter. 2017. Handling Partial Failures in Distributed Reactive Programming. *4th Workshop on Reactive and Event-based Languages & Systems* (2017).

[26] Florian Myter, Christophe Scholliers, and Wolfgang De Meuter. 2017. Handling Partial Failures in Distributed Reactive Programming. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS 2017)*. ACM, New York, NY, USA, 1–7.

[27] Ryan Newton, Greg Morrisett, and Matt Welsh. 2007. The Regiment Macroprogramming System. In *2007 6th International Symposium on Information Processing in Sensor Networks*. IEEE. https://doi.org/10.1109/IPSN.2007.4379709

[28] Henrik Nilsson, Antony Courtney, and John Peterson. 2002. Functional reactive programming, continued. In *Proc. ACM SIGPLAN Work. Haskell - Haskell '02*. ACM Press, 51–64. https://doi.org/10.1145/581690.581695

[29] John Peterson, Valery Trifonov, and Andrei Serjantov. 2000. Parallel functional reactive programming. 16–31. http://link.springer.com/chapter/10.1007/3-540-46584-7_2

[30] José Proença and Carlos Baquero. 2017. Quality-Aware Reactive Programming for the Internet of Things. In *Fundamentals of Software Engineering - 7th International Conference, FSEN 2017, Tehran, Iran, April 26-28, 2017, Revised Selected Papers (Lecture Notes in Computer Science)*, Mehdi Dastani and Marjan Sirjani (Eds.), Vol. 10522. Springer, 180–195.

[31] Stefan Ramson and Robert Hirschfeld. 2017. Active Expressions: Basic Building Blocks for Reactive Programming. *CoRR* abs/1703.10859 (2017). arXiv:1703.10859

[32] Bob Reynders, Dominique Devriese, and Frank Piessens. 2014. Multi-Tier Functional Reactive Programming for the Web. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2014)*. ACM, New York, NY, USA, 55–68. https://doi.org/10.1145/2661136.2661140

[33] Guido Salvaneschi, Sven Amann, Sebastian Proksch, and Mira Mezini. 2014. An Empirical Study on Program Comprehension with Reactive Programming *(FSE 2014)*. ACM, 564–575. https://doi.org/10.1145/2635868.2635895

[34] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. 2014. REScala: Bridging Between Object-oriented and Functional Style in Reactive Applications *(MODULARITY '14)*. ACM. https://doi.org/10.1145/2577080.2577083

[35] Guido Salvaneschi and Mira Mezini. 2016. Debugging for Reactive Programming. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 796–807.

[36] Guido Salvaneschi, Sebastian Proksch, Sven Amann, Sarah Nadi, and Mira Mezini. 2017. On the Positive Effect of Reactive Programming on Software Comprehension: An Empirical Study. *IEEE Trans. Softw. Eng.* 43, 12 (Dec. 2017), 1125–1143.

[37] Kensuke Sawada and Takuo Watanabe. 2016. Emfrp: A Functional Reactive Programming Language for Small-scale Embedded Systems. In *Companion Proceedings of the 15th International Conference on Modularity (MODULARITY Companion 2016)*. ACM, New York, NY, USA, 36–44.

[38] Neil Sculthorpe and Henrik Nilsson. 2009. Safe functional reactive programming through dependent types. *ACM Sigplan Not.* 44, 9 (2009), 23–34. http://dl.acm.org/citation.cfm?id=1596558

[39] Tom Van Cutsem, Elisa Gonzalez Boix, Christophe Scholliers, Andoni Lombide Carreton, Dries Harnie, Kevin Pinte, and Wolfgang De Meuter. 2014. AmbientTalk: programming responsive mobile peer-to-peer applications with actors. *Computer Languages, Systems & Structures* 40, 3-4 (Oct. 2014). https://doi.org/10.1016/j.cl.2014.05.002

[40] Atze van der Ploeg and Koen Claessen. 2015. Practical principled FRP: forget the past, change the future, FRPNow! ACM, 302–314. http://dl.acm.org/citation.cfm?id=2784752

[41] Zhanyong Wan and Paul Hudak. 2000. Functional reactive programming from first principles. *ACM SIGPLAN Notices* 35 (2000). https://doi.org/10.1145/358438.349331

[42] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, Berkeley, CA, USA. https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf