Debugging for Reactive Programming

Guido Salvaneschi, Mira Mezini
Technical University of Darmstadt, Germany
{salvaneschi,mezini}@cs.tu-darmstadt.de

ABSTRACT

Reactive programming is a recent programming technique that provides dedicated language abstractions for reactive software. Reactive programming relieves developers from manually updating outputs when the inputs of a computation change, it overcomes a number of well-know issues of the Observer design pattern, and it makes programs more comprehensible. Unfortunately, complementing the new paradigm with proper tools is a vastly unexplored area. Hence, as of now, developers can embrace reactive programming only at the cost of a more challenging development process.

In this paper, we investigate a primary issue in the field: debugging programs in the reactive style. We analyze the problem of debugging reactive programs, show that the reactive style requires a paradigm shift in the concepts needed for debugging, and propose RP Debugging, a methodology for effectively debugging reactive programs. These ideas are implemented in Reactive Inspector, a debugger for reactive programs integrated with the Eclipse Scala IDE. Evaluation based on a controlled experiment shows that RP Debugging outperforms traditional debugging techniques.

Categories and Subject Descriptors

D.2.6 [**Programming Environments**]: Interactive environments

Keywords

Functional-reactive Programming, Debugging

1. INTRODUCTION

Reactive programming (RP) has been proposed as a viable alternative to the Observer design pattern in developing reactive applications such as graphic user interfaces, animations and event-based systems. The idea behind RP is to support language-level abstractions for signals – time-changing values that are automatically updated by the language runtime. In RP, programmers specify the functional dependency of a signal on other values in the application and changes are automatically propagated when it is required. This way, programmers do not risk to forget updating dependent values and benefit from a programming style that is easier to read – thanks the declarative approach of RP – and supports composition – as signals can be composed to define other signals. It has been shown that applications based on RP are more composable [33]. Our earlier studies also suggest that RP is less error prone [48] and easier to understand [46] than the Observer design pattern.

RP witnessed a long incubation in experimental research projects, which clarified the semantics foundations [12] and investigated the issue of providing a sound [19] and efficient [13] implementation. Since then, researchers proposed several RP implementations such as FrTime [7] (Scheme), Flapjax [33] (Javascript), Scala.react [27] (Scala) and DREAM [29] (Java) – just to mention a few. Recently, concepts from RP have been adopted in a number of front-end Javascript libraries like AngularJS.js, Razor.js, React, Web frameworks like Scala Lift, and by Microsoft's Reactive Extensions (Rx), popularized by the Netflix success story.

RP abstractions are now well understood and properly supported in a variety of languages. Yet, programmers that want to embrace RP have to face a number of challenges due to the immaturity of the field. A primary issue concerns supporting RP in the entirety of the development process through a proper tool ecosystem. In particular, novice RP developers struggle for the lack of proper debuggers – essential instruments to fix errors and understand programs since the early age of computing.

Of course, modern IDEs provide support for debugging high-level languages, but, unfortunately, existing debuggers are hardly useful for RP applications. The issue is a *conceptual* one. Existing debuggers are inherently based on the imperative programming model for which they have been designed and they are unsuitable for the declarative and data flow-oriented model of RP. Concepts like stepping-over statements, breakpointing or inspecting memory changes assume an imperative model where statements execute one after the other and modify memory state. Designing a debugger for RP requires *a paradigm shift*.

In this paper, we propose a novel debugging technique, RP Debugging, which addresses the urgent needs of developers when debugging applications in the RP style. The key contribution is to adopt the dependency graph among signals – the same model developers adopt to reason about reactive applications – as a primary runtime representation of the program during the debugging process. We show that well-known concepts from traditional debugging find a natural mapping into this model. RP Debugging is implemented in Reactive Inspector, a plugin for the Eclipse IDE. A controlled experiment confirms the benefits of RP Debugging. In summary, this paper makes the following contributions.

- We propose RP Debugging, a debugging technique that specifically addresses applications written in the RP style. The design of RP Debugging is motivated by our experience in RP on several projects, requirements observed on independent projects and

ICSE '16, May 14-22, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-3900-1/16/05...$15.00
DOI: http://dx.doi.org/10.1145/2884781.2884815

a preliminary study based on 89 subjects. We show that traditional debugging concepts have an elegant counterpart in RP Debugging.

- We analyze the debugging process of reactive applications and identify common error patterns. We show that in contrast to traditional debugging techniques, which hardly help for such applications, RP Debugging is effective.
- We provide Reactive Inspector, a reference implementation for RP Debugging, in the form of an Eclipse plugin. We evaluate Reactive Inspector with a controlled experiment involving 18 subjects which confirms its advantages over traditional debugging for reactive applications.

Reactive Inspector and the data collected in the experiments are publicly available.[1]

2. BACKGROUND

Reactive applications are traditionally developed in OO programming by using the Observer design pattern. This solution has been criticized for a long time. For example, as handlers typically return void and perform their action via side effects, reactions cannot be composed. Also, the execution of the application depends on both control and data flow, which complicates program comprehension. Finally, inversion of control makes automated analysis of reactive applications hard. The interested reader can refer to [33] for a complete overview.

RP solves the aforementioned problems by introducing time-changing values, often referred to as behaviors or signals (in the rest, we use the latter, conforming to most of Scala literature). Signals are essentially constraints that are automatically enforced by the language runtime by recalculating their value when an inconsistency is detected.

For illustration, consider the code snippet in Figure 1. It defines two *vars* `a` and `b` (Lines 1-2), i.e., reactive values that, in contrast to signals, can be imperatively updated. Line 3 defines a signal `s` which depends on `a` and `b` according to the signal expression `a()+b()`. The `()` notation inside signal expressions establishes a dependency among reactive variables. Signals can depend on vars and on other signals, like in Line 4. When a var is updated (Line 8), the signals that depend on the var are automatically updated without programmer intervention (Lines 9-10). It is easy to see that the relation between signals and vars can be described by a directed graph where edges model dataflow dependencies. Operationally, a change in a node of the graph triggers a reevaluation of the signal expressions in the dependent nodes. This is actually the implementation technique adopted by most RP frameworks. Figure 2 shows the evolution of the dependency graph for the code in Figure 1. Lines 1-4 in Figure 1 correspond to node creations (steps 1-2-3-6 in Figure 2) and dependency construction (steps 4-5-7). A var update (Line 8) triggers a reevaluation in the graph (steps 8-10).

Simple as it is, this programming paradigm proved effective in managing the complexity of reactive applications in a number of fields, including Web applications [25], interactive GUIs [33], animations [8], wireless sensor networks [34] and robotics [19]. For example, the application in Figure 3 shows a validation form in a Web application. The `pwd` signal contains the current input of the user in the password field. Consistently with existing RP frameworks, we assume that the graphic library in use is signal-aware and directly provides such value as a signal, e.g., via the `Window.PwdField` field (Line 1). The application requires that the password is at least 6 characters long and contains at least a digit.

[1]http://guidosalva.github.io/reactive-inspector/

```
1  val a = Var(1)
2  val b = Var(2)
3  val s = Signal{ a() + b() }
4  val t = Signal{ s() + 1 }
5  println(s.get()) // 3
6  println(t.get()) // 4
7
8  a()=4
9  println(s.get()) // 6
10 println(t.get()) // 7
```

Figure 1: Signals and vars in RP.

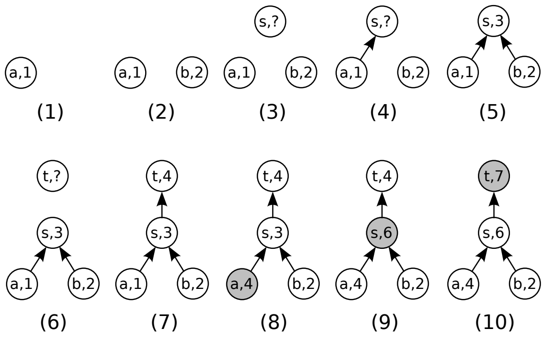


Figure 2: Evolution of the dependency graph.

These conditions are expressed by the `hasEnoughChars` signal and by the `containsDigit` signal (Lines 2-3). The form color, expressed by the signal in Line 5, is green only if the password inserted by the user is valid, red otherwise. Similarly, a proper warning message is displayed depending on which condition is not satisfied (Line 11). Crucially, the entire application is written in a declarative style where each entity is determined by its definition and no callbacks are required.

3. DESIGNING RP DEBUGGING

RP Debugging is a new debugging paradigm which provides support to inspect and reason about the flow of changes through a reactive application. When an application is debugged with RP Debugging, the user can visualize the dependency graph and use it as the basic model for reasoning about the application execution.

In the rest of this section, we will first motivate the need for a new debugging approach for reactive applications by a discussion of the conceptual mismatch between the concepts underlying RP and traditional debuggers followed by some empirical evidence collected in a preliminary study and by experience with reactive application development. Subsequently, we provide a high-level overview of the features of the RP Debugging approach proposed in this paper.

3.1 Motivation

Limitations of Traditional Debuggers.

Debuggers can provide different functionalities [37, 39, 45, 4, 8], but the fundamental feature is to interrupt the execution of the program regularly (stepping) or in certain points of the execution (breakpoints) to better understand the program flow and inspect the intermediate states of the memory. This model, developed in the old times of assembly languages, is still adopted for modern high level OO languages. Such approach is inherently designed for an imperative, shared memory based, *control*-flow driven programming model. It is however unsuitable for RP – functional, based on immutable data and *data*-flow driven. In the rest, we analyze this mismatch in detail:

```
val pwd = Window.PwdField    // Input signal
val hasEnoughChars = Signal{ pwd().size() > 6 }
val containsDigit = Signal{ pwd().contains("0..9") }

val formColor = Signal{
  if (hasEnoughChars() && containsDigit())
    Color.GREEN
  else
    Color.RED
}
val warningLabel = Signal{(
    if (!hasEnoughChars()) "Password too short!"
    else " " ) + (
    if (!containsDigit()) "A digit is required!"
    else ""
)}
```

Figure 3: A validation form implemented with RP.

- **Imperative vs. declarative**. In imperative programming, developers explicitly define control flow by means of control structures. This way, they can reason about memory states and changes, and instruction reachability [23]. Instead, in RP, programmers do not explicitly define control: RP is declarative and the execution flow among the evaluation of signal expressions is implicit and data driven. Hence, a debugger that is only designed for explicit control structures ignores a fundamental part of the execution flow of a RP application.
- **Lack of abstractions**. Traditional debuggers are not aware of RP abstractions like *dependencies* among signal expressions or *change propagation* in the dependency graph. Hence, programmers are forced to reason in terms of the low level concepts that, in the specific RP framework, are used to implement RP. This state of the affairs is similar to a debugger for low level languages which – ignoring exceptions – steps through the GOTO statements that implement exceptions in that specific runtime.
- **Mismatch in the mental model**. The runtime model adopted by traditional debuggers is the *execution stack*. In RP, instead, developers reason in terms of dependency graph, are interested in knowing which dependencies are active at a certain point in time and use the dependency graph as the runtime model to understand the evolution of reactive applications (Section 3.1).

Collecting Empirical Evidence.

The validity of the dependency graph as the reference model to reason about RP programs is suggested by a number of independent experiences. Over the years, we observed that programmers spontaneously develop the solution of an explicit representation of the dependency graph. Besides our personal experience as RP developers [47, 48, 10], students involved in classes on RP [46] independently developed small applications to visualize the dependency graph. These systems adopt various technologies (e.g., Graphviz, Flash) and offer different levels of refinement, but all of them focus on displaying the evolution of the graph over time. Also, practitioners developed similar representations [41, 21, 42]. While none of these approaches offers all features of RP Debugging, these attempts strengthen our confidence in the solution we propose.

The design of RP Debugging has been conceived during the development of a number of reactive applications publicly available. These include software we developed from scratch, like a distributed drawing application, an RSS feed reader and a text editor. Also, we refactored existing applications to use RP, such as the Swing Scala library, the Twitter API and a minimal reactive version of the Scala collections framework [2]. The bugs presented in Section 4 are drawn from this experience and the functionalities we selected for RP Debugging are those we needed during development.

Q1	I tried to use a debugger with a RP program at least once	Y/N
Q2	Traditional debuggers (stepping over statements, breakpoints, variables inspection) are *not* suitable for RP because these operations are hardly applicable to the declarative RP model.	Likert
Q3	Which features would you like to see in a debugger that specifically targets RP?	Open
Q4	Visualizing the dependency graph would help debugging RP programs.	Likert
Q5	Visualizing the propagation of the values in the dependency graph would help debugging RP programs.	Likert

Figure 4: Questions in the preliminary study.

To achieve a more objective view of the limitations of traditional debugging in the RP context and to collect guidelines for the design of RP Debugging, we organized a preliminary study involving 89 subjects. All of them are $3^{rd}/4^{th}$ year students from a CS program attending an advanced Software Engineering class which includes two lectures (2h + 2h) and homework on RP. The problem of debugging RP programs was never discussed during the lectures. Subjects were given the questionnaire in Figure 4. Order avoids that certain questions influence the answer to others, especially Q4 and Q5 can bias Q2. Y/N indicates a Yes/No answer, Likert indicates a 1-5 Likert scale (*Strongly agree - Agree - Neither agree nor disagree - Disagree - Strongly disagree*), Open is for an open text question – in Q3 we let the answer optional to collect only genuine suggestions.

According to the answers of Q1, 21 subjects attempted to use a (traditional) debugger with a RP program. Given that students were not suggested to use a debugger during the lectures/homework, we consider this low number not surprising.

The answers to Q2 show that traditional debuggers are considered unsuitable for RP by the majority of subjects (Figure 5 top). This trend seems to be stronger if we consider subjects who actually tried to debug RP programs (Figure 5 bottom left) than subjects who didn't (Figure 5 bottom right). However, statistical tests are only close to significance in showing a difference between populations (Mann-Whitney U test, $p = 0.066 > 0.05$).

In total 36 subjects answered Q3. Among them, 14 explicitly mentioned the visualization of the dependency graph (e.g., *"Navigating the dependecy graph"*). A separate set of 10 subject did not explicitly mention the dependency graph but proposed features that also inspired the design of RP Debugging, e.g., *"See how the computation values flow through, when one value is changed"*, *"Live monitoring of the dependencies and snapshots of the whole environment at the time of a certain execution"*, *"That it shows the current and the last value of a Signal"* and *"Possibility to step to next point in program execution when a specific event fires"*. This result strengths our confidence that visualizing the dependency graph and its evolution is a real need for developers.

Finally, the answers to Q4 and Q5 (Figure 6) also suggest that visualizing the dependency graph and change propagation through the graph is a desirable feature. In Q4 and Q5, 96.62% – respectively 87.64% – of the subjects answered *strongly agree* or *agree*.

3.2 RP Debugging in a Nutshell

Main Features of RP Debugging.

RP Debugging consists of adopting the dependency graph as *the* model to reason about RP code. It includes the following features:

- At the definition site of the signals, the user can step through the construction of the graph, visualizing the creation of new nodes and of new dependencies among reactive values as soon as they are established. We demonstrate this feature in Sections 4.1, 4.4.

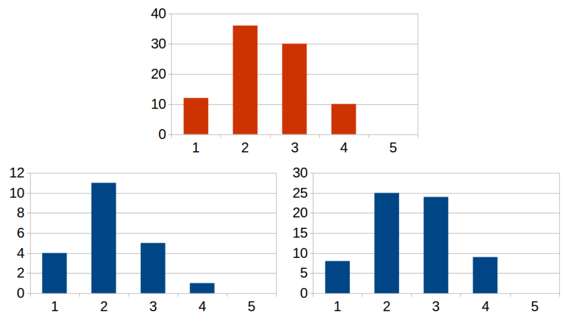


Figure 5: Answers to Q2 (top) and their breakdown for Yes in Q1 (bottom left) and No in Q1 (bottom right).

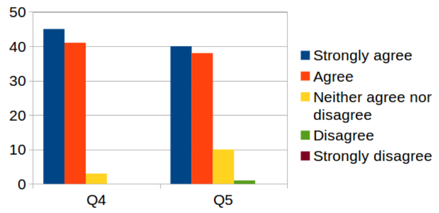


Figure 6: Answers to Q2 (left) and Q5 (right).

- When the execution reaches the assignment of a var, the content in the nodes of the dependency graph starts changing. Similar to what developers would do with lines of code in imperative programming, they can step through the update of values in the dependency graph, and control the potentially changing shape of the graph to make sure it reflects their intentions. This feature is demonstrated in Section 4.3.
- Programmers can set (conditional) breakpoints on the update of a node. The execution continues to traverse imperative and reactive code in the application until the node update is hit. At this point the reactive debugger stops and returns the control to the developer. This feature is demonstrated in Section 4.2.
- Programmers can inspect the performance of an application on a *per node* basis (absolute performance). Also, in RP Debugging developers can observe the number of times a node outputs a different value as a percentage of reevaluation times (relative performance). This information is particularly useful to detect performance bugs related to erroneous graph configurations. This feature is demonstrated in Section 4.5.

Paradigm Shift.

Figure 7 shows how the main concepts of traditional debugging find a counterpart ($\Longrightarrow$ in the rest) in RP Debugging.

Stepping Users step over code to execute a statement at a time. As the execution is slowed down, the user can check the actual control flow of the application and stop the execution at interesting points. $\Longrightarrow$ Stepping over statements makes little sense for declarative languages. The user can step through the node update propagation in the dependency graph.

Breakpoints Stepping until a certain point in the execution may be tedious. Users can ask the debugger to stop the execution when an instruction in the flow is hit. $\Longrightarrow$ Users can stop the execution when a node in the dependency graph is evaluated and the result of its expression is updated.

Inspect memory During stepping, programmers can inspect the content of the memory, i.e., the active variables in the stack frame and the visible objects on the heap. $\Longrightarrow$ Programmers can inspect

Traditional Debugging	RP Debugging
Stepping over statements	Stepping over the dep. graph
Breakpoint on line X	Breakpoint on node X
Inspect memory	Inspect values in the dep. graph
Navigate object references	Navigate signals in the graph
Per-function absolute performance	Per-node relative performance

Figure 7: Traditional debugging vs. RP Debugging.

```
1  val inputPacket = Eth0.lastPacket()
2  val toggleButtonOn = GUI.button1.content
3
4  val lastPacket = Signal{
5    if (toggleButtonOn.get)
6      inputPacket().getAllContent()
7    else
8      inputPacket().getHeader()
9  }
10 GUI.lastPacketSlot(lastPacket)
```

Figure 8: Missing dependencies.

the current value of the reactive variables in the graph and inspect the *dependency relations among them.*

Navigate objects structure In OO debuggers, programmers can navigate object fields to inspect the objects structure. $\Longrightarrow$ Programmers can access vars and signals declared in the code to inspect the dependency graph they originate.

Performance In traditional debuggers, performance is analyzed on *per function* basis and it is *absolute* (time spent in each function). $\Longrightarrow$ Programmers analyze *per node* absolute performance and can inspect *relative* performance as fraction of node reevaluations that issued a new value.

4. RP DEBUGGING AT WORK

In this section, we discuss common issues in RP programs and the use of RP Debugging to solve them.

4.1 Missing dependencies

Designing a reactive application requires to express the correct dependencies among reactive entities to make sure that changes are properly propagated. A common source of bugs is to erroneously specify dependencies, for example forgetting to establish one. Unfortunately, this results in bugs that are hard to detect as they do not make themselves catastrophically apparent (e.g., with exceptions). Conversely, dependent values are simply not updated and the application becomes less reactive.

Missing dependencies are hard to detect with traditional debuggers. Consider the code in Figure 8, a snippet from a network packet sniffer, which displays the header of the detected packets. We focus on a specific functionality: toggling a button displays not only the header, but the whole content of the last sniffed packet – stored in the `inputPacket` signal. The current display format for the packet is modeled by the `lastPacket` signal (Line 4). Depending on the state of the toggle button (Line 5), `lastPacket` can be the complete packet content or the header only (Lines 4-9). Finally, the last received packet is displayed in the desired format (Line 10).

Unfortunately, when the programmer executes the application, she realizes that toggling the button has no effect, i.e., the last packet remains in the same format. More surprisingly, the issue is temporary: The next packet is displayed in the right format. The bug is subtle: Inside the signal expression in Lines 4-9, the dependency on the `toggleButtonOn` variable is expressed using the `get` call[2] (Line 5). In contrast to the `()` method, which creates a dependency on the signal in the immediately outer scope *and* returns

[2]Parameterless methods do not require parentheses in Scala.

```
1  val time = App.time // Global time signal
2
3  class Square(center: (Signal[Int],Signal[Int]),
4               side: Signal[Int]){...}
5  class Circle(center: (Signal[Int],Signal[Int]),
6               radius: Signal[Int]){...}   ...
7  class Picture(original: Image, scale: Signal[Int],
8                center: (Signal[Int],Signal[Int])){
9    val size = Signal{ original.size / scale() } ...
10 }
11 val slowIncreasing = Signal{ time() / 100 }
12 val fastIncreasing = Signal{ time() / 10 }
13 val pulsing = Signal{ (time() % 10) / 50 }  ...
14 val quadratic = Signal{ time() * time() }
15
16 new Circle((fastIncreasing,fastIncreasing),pulsing)
17 // ...More combinations!
```

Figure 9: Fragment of an animation in RP.

the current value, `get` *only* returns the current value of the signal. As a result, when `toggleButtonOn` is updated, `lastPacket` is not consequently updated. When the next packet arrives, the signal expression is reevaluated, `get()` reads the most recent value of the button, and the packet is displayed in the correct format.

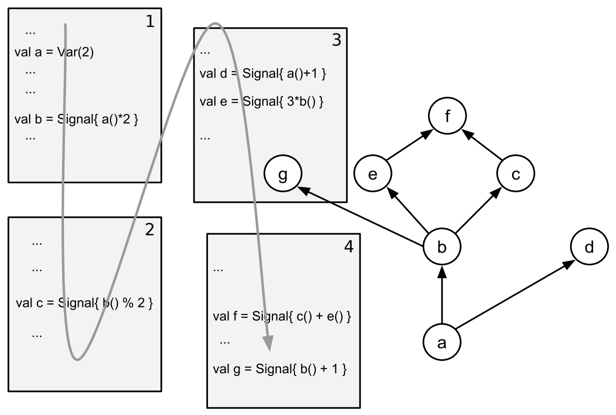
A traditional debugger provides no hint on why, upon toggling the button, the behavior of the application does not change. Instead, thanks to RP Debugging, developers can easily spot that the dependency between `toggleButtonOn` and `lastPacket` is missing.

Interestingly, the *dual* of the bug above, i.e., the presence of a dependency that is unintentionally established, is also a common issue in RP. One can incur in such bug when dependencies are collected in the entire control flow of a signal expression – like e.g., in Scala.react [27]. In this case, a sequence of deeply nested calls originating from the expression can result in an unintended dependency being established. Similar to the previous case, RP Debugging would help developers detecting this spurious dependency.

4.2 Bugs in Signal Expressions

A common design for animations in the RP style (Figure 9) is to use a `time` signal (Line 1) as a source of change for the whole animation. Lines 3-10 show graphic elements that can be added to the animation and receive signals as input. This way, position and dimension of each graphic elements depend on time through certain effects (Lines 11-14), e.g., the `quadratic` signal models a size that increases quadratically over time while the animation is progressing. In the animation, graphic elements (Lines 3-10) and visual effects (Lines 11-14) are freely combined. For example, Line 16 adds to the animation a circle whose `x` and `y` center coordinates increase indefinitely (i.e., the circle is moving over the diagonal of the canvas) and whose size is pulsing regularly.

Sadly, the execution incurs in a `DivisionByZeroException`. In a traditional debugger, a programmer would proceed roughly as follows. From the stacktrace, she knows that the exception originates in Line 9, which computes the new (time-changing) size of a custom picture in the animation. However, as the combination of graphic elements and visual effects depends on control flow, (e.g., based on user interaction in a CASE environment), the root cause of the error is hard to detect. The attempt to add a breakpoint on the evaluation of the signal expression in Line 9 will only stop the execution upon every change of the `time` signal. Also, this approach will consider all objects of class `Picture`, even if different `Picture` instances own an independently updated `size` signal – only one of them faulty. In summary, this process does not help finding the bug unless the programmer reconstructs the overall shape of the dependency graph and realizes that there is a dependency between a `pulsing` signal and a `size` signal. The former reaching zero causes the exception.



```
...
val a = Var(2)
...
...
val b = Signal{ a()*2 }
...
```

```
...
...
val c = Signal{ b() % 2 }
...
```

```
...
val d = Signal{ a()+1 }
val e = Signal{ 3*b() }
...
```

```
...
val f = Signal{ c() + e() }
...
val g = Signal{ b() + 1 }
```

Figure 10: RP Debugging **for software comprehension.**

In RP Debugging, the graph is the fundamental model and the tedious process of reconstructing the graph is not necessary. Also, the programmer can set a breakpoint on the node that throws the exception to stop the execution before the exception occurs (after an exception, the graph can be used to set a breakpoint the next time). Next, the programmer can inspect the shape of the graph, its current values, and she can detect incorrect dependencies. Conditional breakpoints can further speed up this process (Section 4.6).

4.3 Understanding RP programs

We now consider the use of RP Debugging not related to a specific bug. We refer to a common scenario where developers use a debugger for software comprehension. Often developers step through an application to figure out what is the control flow and which are the (dynamic) relations among runtime entities. In this process, developers traverse the control flow across functions and mentally connect functions calls to their declarations [23].

This technique works well for imperative programming, but RP abstractions make it rather ineffective. Figure 10 (left), shows a control flow that spawns over four modules (the gray arrow defines a possible execution order). Several reactive values `a`, `b`, ..., `g` are defined along the flow. Finally, the value of `a`, `a()=3`, is updated.

We consider the use of a traditional debugger to inspect the behavior after the update. As RP is declarative, there are several possible evaluation orders of signal expressions after an update. For example, a developer may witness the debugger jumping across[3]:

(g, mod 4)(b, mod 1)(d, mod 3)(e, mod 3)(c, mod 2)(f, mod 4)

where (`x`, mod m) refers to the evaluation of `x`'s signal expression in module m. Unfortunately, such execution trace would puzzle any developer and can hardly help program comprehension. There are two aspects. (1) **Erratic behavior**: Because of the *declarative* nature of RP, signal declarations (and updates), conceptually, do not have any order. This model creates a tension with the operation mode of debuggers for *imperative* programming which instead step through ordered statements. In RP, the only constraint on order of signal expressions evaluation is that dependencies are satisfied, hence the key to understand program execution is data flow across the dependency graph (Figure 10, right). (2) **Many-to-many relations** Upon a signal/var change, the debugger jumps to another location of the code, where a dependent signal is updated – a case similar to a function call, at first sight. However, a function call is a may-to-one relation: There are (potentially) several calls for each function definition. This is not the case for signals which are

[3]We assume that the debugger automatically skips non relevant classes of the RP library – a feature known as *step filters* available in many debuggers

```
1  class ReactiveList[T](list: Signal[List[T]]) {
2    val size = Signal { list().size }
3    def filter(p: T => Boolean) =
4      new ReactiveList[T](Signal { list().filter(p) })
5   ...
6  }
7  class SourceList[T](list: Var[List[T]]) extends
8          ReactiveList[T](list) {
9    def this(set: List[T]) = this(Var(set))
10   def this(vals: T*) = this(List(vals: _*))
11   def add(x: T) = list() = x :: list.get()
12  ...
13 }
14 val list = new SourceList[Int]
15 var prevList: ReactiveList[Int] = list
16 for (i <- 1 to 100) {
17   val filteredList = prevList.filter(_ > i)
18   prevList = filteredList
19 }
20 list.add(10)
```

Figure 11: RP program with a memory and responsiveness bug.

many-to-many relations: A change can be fired everywhere a var is assigned and (potentially) *multiple* other signals that depend on the change are updated in non-deterministic order. As a result, a traditional debugger, after a var assignment, would jump to code locations that are related only because they depend on the same data flow – an aspect that traditional debuggers cannot capture. Noticeably, a graph structure with a high number of signals depending on a few vars – the configuration that magnifies the effect of many-to-many relations – is common in RP. For example, in animations, changes often uniquely depend on time, as shown in Section 4.2. Similarly, in most GUIs, reactions solely depend on mouse clicks, mouse moves, and key insertions.

RP Debugging is based on the observation that the execution model that helps programmers' comprehension is not the (non deterministic) flow across signal expressions, but rather the graph where dependencies are expressed explicitly (Figure 10, right). This way, the execution logic of the program can be given a meaning based on explicit data flow.

4.4 Memory and Time Leaks

When RP frameworks are implemented as libraries/embedded DSLs [33, 48, 7, 27], the interaction between imperative code and RP code can lead to surprising behavior.

Consider the example in Figure 11, which shows an implementation of a reactive data structure in Scala.[4] Reactive data structures expose their attributes as signals [26]. For example, the `size` attribute (Line 2) is a signal that always contains the updated size of the list and can be composed with other signals. In Figure 11, the implementation is based on an internal list (Line 7). When a modifier method is called, the internal list is updated and change propagates to dependent values. Methods that return a list, such as `filter`, allocate a new list which depends on the current internal list. Such solution is correct according to the semantics of reactive data structures: The `filter` method returns a list automatically updated upon insertion into the original list (Line 17). Unfortunately, every time the `filter` method is called, a new signal is created to keep the new list updated (Line 3). As programmers can pass the list around in the code, a function may apply the filter operator to the list and, at a later time, pass the result to another function which also applies filtering. In a large code base, this process may easily repeat in an undisciplined way – for simplicity we show the effect with a loop, Line 17. The dependency graph grows in an uncontrolled manner with two consequences: (1) out of memory errors due to the

[4]For simplicity we consider a basic implementation not integrated with the Scala Collections library.

```
1  val finalImage = Signal{
2    if (mousePosition() overlaps button)
3      computeShadowImage(image())
4    else
5      image()
6  }
7  // –– Refactoring ––
8  val overlaps = Signal{ mousePosition() overlaps button }
9  val finalImage = Signal{
10   if (overlaps())
11     computeShadowImage(image())
12   else
13     image()
14 }
```

Figure 12: Performance bottleneck.

allocated signals, and (2) reduced responsiveness due to the long propagation chain.

A traditional debugger would not detect (1) except if equipped with some functionality to display the growing heap structure. Even in this case, a programmer would be forced to reason at a very low level, in terms of objects and references. (2) would remain definitely unexplained unless the programmer manually reconstructs the shape of the dependency graph.

With RP Debugging, visualizing the dependency graph shows how dependencies grows for every `filter` call. This way, it is possible to immediately identify (1) memory consumption – as excessive creation of new nodes – and (2) decrease of responsiveness – due to chains that are too long in the graph.

4.5 Performance bugs

We now consider performance of reactive applications. In imperative programming, update of dependent computations is explicit and programmers easily control performance of the update process. In RP, instead, updates are transparent, and different performance behaviors can emerge for functionally equivalent dependency graphs.

Consider an application where an image is shadowed every time the mouse cursor moves over a button (Figure 12). The image to display is modeled by the `finalImage` signal which depends on the `mousePosition` signal and the `image` signal. Unfortunately, once the conditional becomes true (Line 2), the `finalImage` signal is evaluated every time `mousePosition` changes its value. Each evaluation also unnecessarily recomputes the shadowed image with `computeShadowImage`.

After noticing poor performance, in a traditional setting, a programmer may check the profiling information, which shows the relative time spent in each method call. The result reveals that a lot of time is spent inside the `computeShadowImage` call. At this point, the `finalImage` signal becomes suspicious: The developer might add a breakpoint to the signal expression associated to `finalImage`. The following debugging session will continuously hit against the breakpoint, indirectly suggesting that there is an issue with the update mechanism. The programmer will finally identify the `mousePosition()` signal as the source of the problem.

With RP Debugging, the programmer can simply inspect the structure of the dependency graph. She will notice (1) that the `finalImage` node is computationally intensive. (2) that the node depends on `mousePosition` and `image`. (3) that the node output changes a minor fraction of times compared to the amount of reevaluations – thanks to *relative* performance estimation in RP Debugging. She can easily conclude what the problem is and refactor the signal as the bottom part of Figure 12 shows. The refactored version introduces an intermediate `overlaps` signal, which does not change when the condition remains true. This way, the signal expression of `finalImage` is evaluated only when necessary.

Query	Description
`nodeCreated(node)`	Node creation
`nodeEvaluated(node)`	Node evaluation
`nodeValueSet(node)`	Value of a node is set
`dependencyCreated(node1, node2)`	Dependency creation
`evaluationYielded(node, value)`	Evaluation yields value
`evaluationException(node)`	Evaluation throws exception

Figure 13: The Query Language

4.6 Advanced RP Debugging

Besides the features described above, RP Debugging provides advanced features that help developers finding the root cause of a bug. We devote less space to these functionalities as they are essentially an application of existing techniques to RP Debugging. Yet, they greatly simplify debugging RP software in certain scenarios.

Inspecting History.
Inspired by omniscent debugging and back-in-time debugging (Section 8), RP Debugging supports a back-in-time mode where changes to the dependency graph over time can be inspected by navigating history back and forth. This way, programmers observe the evolution of the graph in terms of changing dependencies, node creations and node updates.

This technique is particularly useful to reconstruct the cause of a failure because it avoids reexecuting the entire program when the error occurred somewhere in the past execution. For example, when the reevaluation of a signal expression leads to an exception, one can go back in time and find the var update that caused the failure.

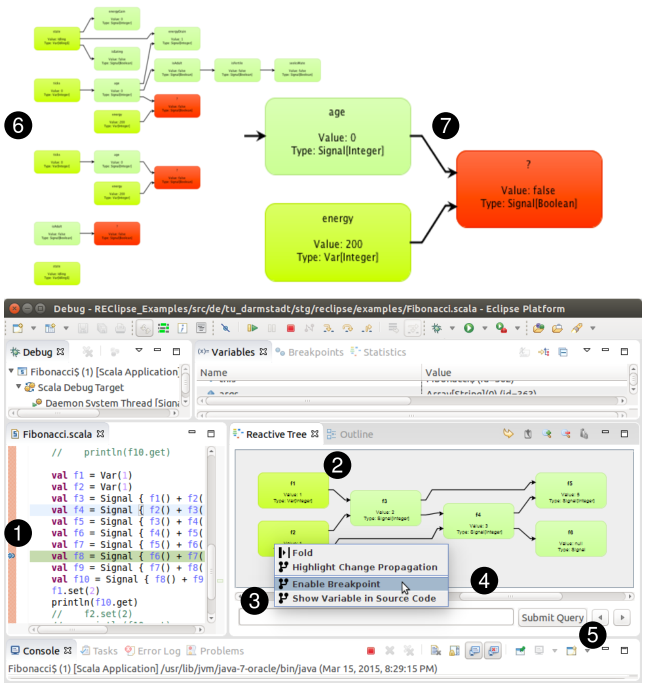
Conditional Breakpoints and Queries.
A query language helps programmers identifying relevant points in the program execution. The purpose is twofold. First, if a query is entered before the execution in debugging mode, the debugger suspends the program once a matching event occurs. This use is similar to conditional breakpoints. Second, in back-in-time mode, in case history is too long for manual inspection, programmers can specify a query to find the relevant point in the graph evolution.

Figure 13 shows the commands in the query language. These are the most common cases according to our experience (Section 6) but since queries are parsed with ANTLR [1] and trivially translated to Esper queries (Section 5) the language can be easily extended. The use of queries as conditional breakpoints is a powerful tool to shorten the stepping process and stop the execution exactly where necessary. For example in the "Bugs in Signal Expressions" example (Section 4), one can set an `evaluationException` breakpoint on the `size` signal and directly inspect the dependencies in the graph when the exception is thrown.

5. IMPLEMENTATION

Reactive Inspector, our reference implementation of RP Debugging, is an Eclipse plugin integrated with the debugger of the Scala Eclipse IDE [49]. Reactive Inspector supports the REScala [47] reactive language and is made of about 8000 LOC. In Reactive Inspector (Figure 14), when the user is debugging a reactive application (1), the dependency graph is displayed in the GUI (2). Users can set a breakpoint on a node (3). A sliding bar provides access to the previous states in the history of the graph (4) and an input field allows one to specify a query on previous state or as a conditional breakpoint (5). For illustration, we also show the case of multiple active dependency graphs (6) where colors indicate the performance of each node. In the enlarged detail (7) each node provides information about the signal name, type and current value.

Figure 15 shows the plugin architecture. When the plugin is



```
//    println(f10.get)

val f1 = Var(1)
val f2 = Var(1)
val f3 = Signal { f1() + f2(
val f4 = Signal { f2() + f3(
val f5 = Signal { f3() + f4(
val f6 = Signal { f4() + f5(
val f7 = Signal { f5() + f6(
val f8 = Signal { f6() + f7(
val f9 = Signal { f7() + f8(
val f10 = Signal { f8() + f9
f1.set(2)
println(f10.get)
//    f2.set(2)
```

```
Fibonacci$ (1) [Scala Application] /usr/lib/jvm/java-7-oracle/bin/java (Mar 15, 2015, 8:29:15 PM)
```

Figure 14: The Reactive Inspector Eclipse plugin.

activated, the RP library detects significant events in the application (1). Events are transmitted to the RP debugger to update the structure of the graph (2) and are recorded in an in-memory database for future use (3). The RP debugger interacts with the Scala debugger (4) for two reasons. First, it collects information about the graph, e.g., node values. Second, it controls program execution via the Scala debugger. The latter is required when the user is stepping over updates in the dependency graph or when a RP breakpoint/query is hit. The events stored in the database can be queried or can be played backwards in time to inspect the evolution of the graph (5). Internally, the implementation is based on the Esper [14] complex event processing system. Events directly generated by the program execution or replayed from the database are filtered by Esper to detect the conditions specified in the query.

Language Independence.
Reactive Inspector specifically targets REScala, but the approach is generally applicable. The plugin is compatible primarily with Scala and Java-based RP languages, but virtually also with other reactive frameworks based on languages supported in the Eclipse IDE. RP Debugging support can be achieved by a third party RP language implementing the interface that generates the events consumed by the plugin. In the case of the REScala, this functionality is implemented by a `Logging` *trait* simply mixed-in the REScala code. Obviously, integrating our plugin with another existing imperative debugger requires to interface our tool with the specific debugger implementation, e.g., to request the current value of certain nodes and to stop the execution when a breakpoint is hit.

6. EVALUATION

In this section, we present a systematic evaluation of RP Debugging aiming at answering the following research question:

RQ: Is debugging reactive applications with RP Debugging easier than with traditional debugging?

To answer RQ we organized a controlled experiment with 18 sub-

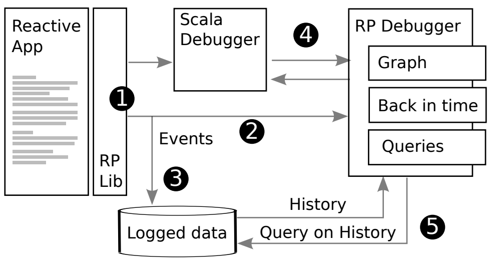


Figure 15: Architecture of Reactive Inspector.

jects. We assume *time* as a measure of simplicity. While our research speculates on RP Debugging overperforming traditional debugging, we take a neutral approach and we define 2-tailed statistical tests. We consider a single alternative to RP debugging, i.e., traditional debugging, because it is the most common solution for debugging software applications. Section 8 discusses other options.

Object of the Experiment and Methodology.

The experiment focuses on an assignment which subjects need to complete using traditional debugging or RP Debugging. The assignment is composed of 6 tasks each based on a different reactive application. Figure 16 provides a summary of the assignment. For each task, it shows the kind of application, the feature that subjects are suggested to use, a task description and the number of non-comment, non-blank lines of code. Applications belong to four categories: graphical animations, GUIs, simulations – traditional domains for reactive programming [12, 11, 33, 7] – and synthetic applications that just define functional dependencies among values and propagate changes upon updates.

In each task, a subject is shown a reactive application in the IDE and asked either to answer questions about the application behavior (tasks 1-4) or to fix a bug (tasks 5 and 6). Tasks 1-4 are motivated by the fact that understanding an application behavior is a common use case for debuggers and a preliminary step to bug fixing. To ensure that the time required is roughly the same for each task and to avoid that overhead time, e.g., to navigate the application, is significant compared to debugging activity, tasks 1-4, which are shorter, are composed of 2 or 3 subtasks.

We adopted the *think loudly* approach where subjects comment their actions [15, 22]. Our goal is not to analyze developers behavior in detail. Yet, the approach allows us to better understand what programmers are doing, e.g., to find out whether they incur in trivial mistakes. Our *between-subjects* design – traditional debugging group (TD) vs. RP Debugging group (RD) – increases variability, making harder to observe a significant effect compared to *within-subjects* designs [18, 30]. However, it is robust to learning effects, which are significant given the nature of the assignment [40]. Correctness is controlled providing tasks that are easy enough to be completed in the available time, using unit tests for checking the solution, and supervising the experiment to encourage subjects to keep working on the task if the solution is not correct.

Experiment Context.

Subjects are students from a CS program in their 4^{th} or 5^{th} year of study. They have similar academic background and they know the Eclipse IDE and its debugger. All subjects have been exposed to RP in Scala in a course. Some of them have a deeper knowledge of RP as they also attended projects or seminars on the this topic.

Both groups completed a tutorial with a few example tasks using the tool they would use in the experiment. To make sure that the tutorial is properly covered and all subjects have a minimal background, this activity is assisted by the staff and takes about 30 mins. To avoid bias towards RP Debugging, Reactive Inspector is not given to the subjects before the experiment: subjects may be curious to try Reactive Inspector more than they would refresh their knowledge of traditional debugging. However, the RD group was given the manual of Reactive Inspector a few days before the experiment.

Experiment Results.

Figure 17 shows the experiment results. We initially analyze the time that subjects required to complete the entire assignment. Descriptive statistics provides an overview of the results: The mean time for the TD group and the RD group are $\mu_{TD} = 4317.33$ respectively $\mu_{RD} = 2572.67$. Figure 18 shows the cumulative times for the TD group and the RD group. To check if the difference between the groups is significant, we formulate the hypothesis: H_0: *Times for the TD and for the RD groups are drawn from the same population.*

Since the underlying distribution is not known, we performed a non-parametric Mann-Whitney U test. The result shows that, with high significance ($p = 0.001$) H_0 can be rejected.

Group	N	Rank avg	Rank sum	p-value
TD	9	13.44	121	0.001
RD	9	5.56	50	

This result provides an answer to RQ and allows us to conclude that the RP Debugging makes debugging of RP easier.

We further analyze the data considering each task separately. Similar to the previous case, we perform a non-parametric Mann-Whitney U test. Results are in Figure 19. For tasks 1-3-4, H_0 can be rejected with high significance ($p < 0.05$), i.e., *the RD group is faster*. In tasks 2, 5 and 6, H_0 cannot be rejected ($p > 0.05$).

For tasks 2 and 6 we formulate an explanation based on the think loudly approach. In both cases, solving the task requires to detect whether a signal dependency is in place. In the code, however, after creation, the signal is passed to a function. As a result, the signal appears in (most of) the code with the name of the formal parameter. Since Reactive Inspector shows the signals in the dependency graph with their name at creation time, subjects struggled to detect the dependency. The analysis of task 2 restricted to subtask 1 – the subtask not affected by the naming issue – substantiates our explanation showing a significant difference ($p = 0.001$).

The naming issue highlights a tension between the representation via a (global) dependency graph and scope/visibility. A possible solution is to name nodes in the graph based on the name bindings in the current scope. Bringing this approach further, would allow multiple of such names based on different stack frames. We leave this research line for future work. Task 5 is harder to explain. The p value is not far from significance ($p = 0.94$) and we suspect that a larger experiment could detect an effect. A more powerful t-test, applicable because of normal distribution (Shapiro-Wilk test, $p_{TD} = 0.22$, $p_{RD} = 0.093$) also indicates a very small yet not significant p value ($p = 0.065$, no variance equality assumed).

7. DISCUSSION

Dynamic Dependency Graphs.

RP Debugging is based on a *dynamic* model. A dependency graph is a run time entity, it evolves during application progress and it does not exist before the execution starts. This is a major difference with traditional debugging where developers use the code itself as a primary model to understand the execution and drive the debugger, e.g., to set a breakpoint.

Application	**Features**	**(Sub)Tasks**	**LOC**
T1: 2D simulation	Tree Inspection: -Highlight Dependencies -Node Search	Do the following variables have a dependency ? In case describe the dependency chain 1. World.statusString ← Board.elementsChanged 2. World.statusString ← Board.elementRemoved 3. Animal.isFertile ← Animal.energy	442
T2: Fisheye animation	Tree Inspection	1. What is the range of values the signal Box.interpolationValue can have at runtime? Please give the minimum and maximum values. 2. Do the squares depend on their left or their right neighbor?	101
T3: Reactive network	Tree Inspection Reactive Breakpoints	After the execution finishes, error count is 1 due to an exception in one of the iterations. 1. On which node is the exception thrown? 2. On which iteration does the exception occur? 3. What are the values of a1, a2, a3 and a4 at the time the exception is thrown?	44
T4: Arcade Pong Game	History Queries	1. What is the value of Pong.y when the right player scores his first point? 2. What is the value of Pong.x when the ball bounces the 3rd time on the bottom border?	119
T5: RSS Feedreader	Tree Inspection	Normally the feed items of the channel are displayed in the left panel after a channel is selected but due to a bug if a news channel is selected from the channel box, nothing happens. The task is to fix the bug so that all test cases are green.	507
T6: Shapes animation	Tree Inspection Tree History	The animation contains a bug: An exception is thrown and the application stops. The task is to find the source of the exception and fix the bug. The task counts as completed, if the animation runs without exceptions.	153

Figure 16: Tasks in the controlled experiment.

Group	T1	T2	T3	T4	T1	T6	Sum
TD	1110	968	770	1115	1790	375	6128
	705	455	1065	1198	590	385	4398
	708	833	1174	649	437	355	4156
	912	535	1438	1466	1488	430	6269
	380	1330	686	303	275	465	3439
	546	525	1059	668	470	514	3782
	520	696	552	558	1406	461	4193
	455	913	579	485	1088	332	3852
	667	418	423	734	202	195	2639
Mean	667.0	741.4	860.6	797.3	860.6	390.2	4317.3
RD	269	680	393	273	150	1070	2835
	242	470	250	605	205	525	2297
	514	1235	550	310	245	350	3204
	460	904	300	835	560	290	3349
	339	464	233	200	1014	186	2436
	271	300	370	365	295	172	1773
	361	1067	632	414	615	378	3467
	261	522	280	358	260	440	2121
	223	312	248	207	414	268	1672
Mean	326.6	661.5	361.7	396.3	417.5	408.7	2572.6

Figure 17: Times in the controlled experiment.

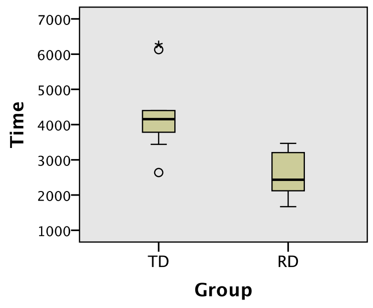


Figure 18: Cumulative time for the TD and the RD groups.

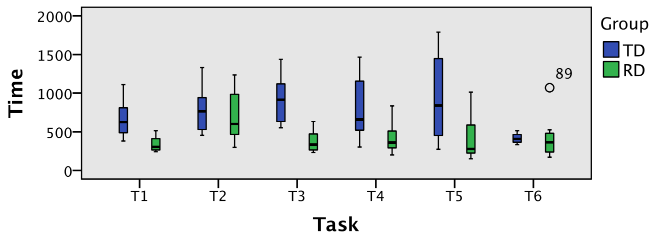


Figure 19: Time for each task for the TD and the RD groups.

Similar to object-centric debugging ([45], Section 8), accessing the debugging model (e.g., to inspect the content of a node) requires that part of the program already runs. This is not an issue in practice for two reasons. First, most RP programs define the graph structure at their very beginning and update it in the rest. Second, similar to object-centric debugging, RP Debugging does not conflict with imperative debugging, rather complements it. Users can set a breakpoint at a line and *then* refine it with a breakpoint on the graph once the execution reaches such line and the graph is built.

Scalability of the Visualization.

Our approach potentially suffers from a scalability issue common to software visualization tools – displaying a software model that is manageable in size. In most cases this is not an issue for the following reasons. First, in many applications, the dependency graph is relatively small ([10] provides some numeric evidence collected in our experience with RP), at least compared to an object graph, which would be hard to display even for minimal applications. Second, small applications, like input validation forms, are a typical domain for RP [3], hence, even an approach with limited scalability capabilities can address a major fraction of use cases.

To mitigate the effect of potentially large graphs we took some countermeasures. First, programmers can inspect the graph(s) associated to a single reactive entity, e.g., the graph(s) generated by a var. This solution, most of the times, considerably reduces the size of the displayed graph. Second, programmers can select a node and collapse the subgraph originating from there – the effect is to prune parts of the graph that are not interesting for the debugging task at hand. Finally, developers can search nodes in the graph by name and are shown a thumbnail that helps navigation in large graphs.

Limitations.

The Eclipse Scala IDE, on which Reactive Inspector is based, plugs on the Java Eclipse debugger using aspect-oriented programming technology. In the Eclipse Scala IDE, the treatment of high-level Scala abstractions that depart from Java is only partially complete. This issue is apparent, e.g., when stepping over closures. Also, step filters are not (yet) supported. Our plugin inherits those issues, at a partial detriment of the user experience. Unfortunately, this results in a time overhead that, for complex tasks, reduces the size of the observable effect and makes it harder to detect a difference between groups in the evaluation (Section 6). Finally, for now, our research does not address efficient implementation. Especially for back-in-time and omniscient debuggers, well known techniques to efficiently store and retrieve execution data exist (Section 8). We believe, that these solutions are largely orthogonal to our approach.

8. RELATED WORK

In related work we do not consider RP – outlined in Section 1. The interested reader can refer to the survey [3].

Live Programming.

ELM [8] is a functional programming language for Web GUIs and animations. It is similar to Haskell and compiles to Javascript. Recent versions of ELM support an experimental back in time debugger with hot swapping [43]. Thanks to the purity of ELM, users can play the animation backwards in time. Live programming [31, 5] is about keeping the GUI in sync with code changes. In this line of work [32], McDirmid and Edwards explored the use of *managed time* for live programming: Application time is controlled by the runtime environment – an approach inspired by automatic memory control of garbage-collected VMs. This solution can produce results analogous to back in time execution for GUI applications.

Contrarily to our work, these systems focus only on animations. In live programming, the graphical evolution of the animation over time is *itself* a model that developers can use to reason about the execution. However, this approach is only applicable for GUI software – hence our choice to use the dependency graph as the reference model for RP Debugging.

Omniscient Debugging.

Omniscient debuggers [37, 39] support back in time navigation across previous program states. In omniscient debuggers, all events from the execution are logged and can be inspected and queried. As a result, programmers can ask the debugger which events (e.g., an assignment) are responsible for a certain current state (e.g., a variable being null). In a traditional debugger, this analysis requires a tedious process, like setting a breakpoint on the assignment and stepping across the rest of the execution.

Performance is a fundamental issue in omniscient debugging because events from program execution must be processed and stored efficiently. Recent research addresses these issues with different techniques including specialized distributed databases [39], virtual machine integration [24], and efficient data structures [38]. We consider performance orthogonal to our work and we expect that those optimization techniques are also applicable to our approach.

Advanced Debuggers.

Object-centric debugging [45] supports debugging operations that are specific of the OO paradigm (e.g., set a breakpoint on an *object call*). Similar to our work, the premise of object-centric debugging is that a static representation of the program (the code) and the stack-based runtime model adopted by most debuggers are not suitable for higher level abstractions – objects in the case of object-centric debugging, RP in our case. Also, similar to our work, object-centric debugging is based on a runtime representation of the program.

Recent research investigates expressive breakpoints. Debugging with control-flow breakpoints [6] adopts aspect-oriented programming to specify breakpoint conditions based on control flow. Stateful breakpoints [4] express a stop condition based on a pattern of temporal order among other breakpoints and based on context variables captured from the program. Our query language is a less sophisticated mechanism for conditional breakpoints.

Olsson et al. [36] developed a data-flow model where users can combine low-level events from the debugging process to define complex events expressing interesting breakpoint conditions. Marceau et al. [28] adopted a similar approach to develop a scriptable debugger. In their solution, the scripting language to filter and combine events from the execution of the program *is an RP language*. In contrast to this paper, both works target debugging imperative languages.

Similar to RP, traditional debuggers are unsuitable for lazy purely functional languages. In this context, Nilsson and Fritzson propose algorithmic debugging [35], where developers are asked questions about increasingly smaller parts of the program until the bug is found. In contrast to algorithmic debugging, which targets a single paradigm, our work aims at *complementing* traditional debuggers for languages with both reactive and imperative abstractions.

Complex Event Processing and Stream Processing.

Oracle Complex Event Processing (CEP) Engine provides the CEP visualizer [44] to specify queries in a data-flow graph using a CASE interface. In contrast to RP Debugging, the support for the user is limited to query design and the tool does not provide runtime information.

The Event Flow Debugger is the debugging tool of Microsoft StreamInsight CEP system [17]. Gedik et al. [16], propose a similar tool for the Spade IBM stream processing language. Analogous to RP Debugging, in both systems, users can inspect the flow of events through the graph-like structure defined by queries. In contrast to our work, these solutions are limited to CEP and do not handle generic computations like those inside RP nodes. Also, these systems are limited to a fixed graph topology, while in RP the graph changes dynamically.

Kuo et al. designed a CASE programming environment for the StreamIt stream processing language [20] which supports inspection of events flows occurring at runtime. The focus is, however, on providing a deterministic order for facilitating the debugging of systems with parallel (distributed) event stream generators. Pauw et al. [9] developed a visualization system for stream processing. Their tool displays a graphical overview of events flow in distributed stream processing applications. In this setting with high throughput breakpoint-based debugging is not viable as it blocks a node disturbing the relative order of events. By contrast, RP Debugging targets fine grained debugging (including breakpoints) for RP applications.

9. CONCLUSION AND FUTURE WORK

In this paper, we presented RP Debugging, a debugging technique for applications written in the reactive style. In contrast to existing debugging approaches targeting the imperative programming model, our solution specifically aims at the declarative abstractions of RP. With RP Debugging developers use the dependency graph – the same mental model they adopt for reasoning about reactive applications – as the reference model to debug their applications.

We plan to further extend Reactive Inspector to support more RP languages (Section 5). On the conceptual side, we plan to extend RP Debugging to address reactive models that include complex event correlations and asynchronous signal computations.

10. ACKNOWLEDGMENTS

This work is supported by the European Research Council, grant No. 321217. We thank Oscar Nierstrasz for the useful discussion on domain-specific debugging. We thank Matthias Jahn, Simon Sprankel, Michael Hausl, Sebastian Ruhleder, Gerold Hintz and Pascal Weisenburger for contributing to the development of Reactive Inspector and to the experiments on visualizing RP programs.

11. REFERENCES

[1] ANTLR parser generator. http://www.antlr.org/.

[2] REScala reactive language website. http://www.rescala-lang.com/.

[3] 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.

[4] E. Bodden. Stateful breakpoints: A practical approach to defining parameterized runtime monitors. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 492–495, New York, NY, USA, 2011. ACM.

[5] S. Burckhardt, M. Fahndrich, P. de Halleux, S. McDirmid, M. Moskal, N. Tillmann, and J. Kato. It's alive! continuous feedback in UI programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 95–104, New York, NY, USA, 2013. ACM.

[6] R. Chern and K. De Volder. Debugging with control-flow breakpoints. In *Proceedings of the 6th International Conference on Aspect-oriented Software Development*, AOSD '07, pages 96–106, New York, NY, USA, 2007. ACM.

[7] 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.

[8] E. Czaplicki and S. Chong. Asynchronous functional reactive programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 411–422, New York, NY, USA, 2013. ACM.

[9] W. De Pauw, M. Leţia, B. Gedik, H. Andrade, A. Frenkiel, M. Pfeifer, and D. Sow. Visual debugging for stream processing applications. In *Proceedings of the First International Conference on Runtime Verification*, RV'10, pages 18–35, Berlin, Heidelberg, 2010. Springer-Verlag.

[10] J. Drechsler, G. Salvaneschi, R. Mogk, and M. Mezini. Distributed REScala: An update algorithm for distributed reactive programming. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 361–376, New York, NY, USA, 2014. ACM.

[11] C. Elliott. Functional implementations of continuous modeled animation. In *Proceedings of the 10th International Symposium on Principles of Declarative Programming*, PLILP '98/ALP '98, pages 284–299, London, UK, UK, 1998. Springer-Verlag.

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

[13] C. M. Elliott. Push-pull functional reactive programming. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Haskell '09, pages 25–36, New York, NY, USA, 2009. ACM.

[14] Esper event correlation system. http://www.espertech.com/.

[15] S. D. Fleming, E. Kraemer, R. E. K. Stirewalt, S. Xie, and L. K. Dillon. A study of student strategies for the corrective maintenance of concurrent software. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 759–768, New York, NY, USA, 2008. ACM.

[16] B. Gedik, H. Andrade, A. Frenkiel, W. De Pauw, M. Pfeifer, P. Allen, N. Cohen, and K.-L. Wu. Tools and strategies for debugging distributed stream processing applications. *Softw. Pract. Exper.*, 39(16):1347–1376, Nov. 2009.

[17] T. Grabs, S. Roman, K. Ramkumar, J. Goldstein, and R. Fernandez. Introducing Microsoft StreamInsight, 2010.

[18] S. Hanenberg. An experiment about static and dynamic type systems: Doubts about the positive impact of static type systems on development time. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 22–35, New York, NY, USA, 2010. ACM.

[19] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, robots, and functional reactive programming. In *Summer School on Advanced Functional Programming 2002, Oxford University*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer-Verlag, 2003.

[20] K. Kuo, R. M. Rabbah, and S. Amarasinghe. A productive programming environment for stream computing. In *In Proceedings of the 2nd Second Workshop on Productivity and Performance in High-End Computing*, 2005.

[21] Labview interacting debugger. www.ni.com/getting-started/labview-basics/debug.

[22] T. D. LaToza, D. Garlan, J. D. Herbsleb, and B. A. Myers. Program comprehension as fact finding. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 361–370, New York, NY, USA, 2007. ACM.

[23] T. D. LaToza and B. A. Myers. Developers ask reachability questions. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 185–194, New York, NY, USA, 2010. ACM.

[24] A. Lienhard, T. Gîrba, and O. Nierstrasz. Practical object-oriented back-in-time debugging. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ECOOP '08, pages 592–615, Berlin, Heidelberg, 2008. Springer-Verlag.

[25] Lift reactive web site. http://scalareactive.org/.

[26] I. Maier and M. Odersky. Higher-order reactive programming with incremental lists. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 707–731, Berlin, Heidelberg, 2013. Springer-Verlag.

[27] I. Maier, T. Rompf, and M. Odersky. Deprecating the Observer Pattern. Technical report, 2010.

[28] G. Marceau, G. H. Cooper, J. P. Spiro, S. Krishnamurthi, and S. P. Reiss. The design and implementation of a dataflow language for scriptable debugging. *Automated Software Engg.*, 14(1):59–86, Mar. 2007.

[29] A. Margara and G. Salvaneschi. We have a DREAM: Distributed reactive programming with consistency guarantees. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 142–153, New York, NY, USA, 2014. ACM.

[30] C. Mayer, S. Hanenberg, R. Robbes, E. Tanter, and A. Stefik. An empirical study of the influence of static type systems on the usability of undocumented software. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 683–702, New York, NY, USA, 2012. ACM.

[31] S. McDirmid. Living it up with a live programming language. In *Proceedings of the 22Nd Annual ACM SIGPLAN*

Conference on Object-oriented Programming Systems and Applications, OOPSLA '07, pages 623–638, New York, NY, USA, 2007. ACM.

[32] S. McDirmid and J. Edwards. Programming with managed time. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, pages 1–10, New York, NY, USA, 2014. ACM.

[33] 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 *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 1–20, New York, NY, USA, 2009. ACM.

[34] R. Newton, G. Morrisett, and M. Welsh. The Regiment Macroprogramming System. In *Information Processing in Sensor Networks, 2007. IPSN 2007. 6th International Symposium on*, pages 489–498, 2007.

[35] H. Nilsson and P. Fritzson. Algorithmic debugging for lazy functional languages. In M. Bruynooghe and M. Wirsing, editors, *Programming Language Implementation and Logic Programming*, volume 631 of *Lecture Notes in Computer Science*, pages 385–399. Springer Berlin Heidelberg, 1992.

[36] R. A. Olsson, R. H. Crawford, and W. W. Ho. A dataflow approach to event-based debugging. *Softw. Pract. Exper.*, 21(2):209–229, Feb. 1991.

[37] G. Pothier and E. Tanter. Back to the future: Omniscient debugging. *Software, IEEE*, 26(6):78–85, Nov 2009.

[38] G. Pothier and E. Tanter. Summarized trace indexing and querying for scalable back-in-time debugging. In *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP'11, pages 558–582, Berlin, Heidelberg, 2011. Springer-Verlag.

[39] G. Pothier, E. Tanter, and J. Piquer. Scalable omniscient debugging. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 535–552, New York, NY, USA, 2007. ACM.

[40] J. Quante. Do dynamic object process graphs support program understanding? - a controlled experiment. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pages 73–82, June 2008.

[41] Demo for the internals of the ELM debugger. www.youtube.com/watch?v=FSdXiBLpErU.

[42] Dependency graph visualization in the Unreal Engine 4, Tools Demonstration GDC 2014, minutes 6:46 and 8:45. www.youtube.com/watch?v=9hwhH7upYFE#t=384.

[43] ELM debugger. http://debug.elm-lang.org/.

[44] Oracle CEP visualizier. http://docs.oracle.com/cd/E17904_01/doc.1111/e14302.pdf.

[45] J. Ressia, A. Bergel, and O. Nierstrasz. Object-centric debugging. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 485–495, Piscataway, NJ, USA, 2012. IEEE Press.

[46] G. Salvaneschi, S. Amann, S. Proksch, and M. Mezini. An empirical study on program comprehension with reactive programming. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 564–575, New York, NY, USA, 2014. ACM.

[47] G. Salvaneschi, G. Hintz, and M. Mezini. REScala: Bridging between object-oriented and functional style in reactive applications. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 25–36, New York, NY, USA, 2014. ACM.

[48] 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.

[49] Scala Eclipse IDE. http://scala-ide.org/.