Tool Support for Reactive Programming

Tool Support für Reactive Programming
Master-Thesis von Simon Sprankel
Tag der Einreichung:

1. Gutachten: Prof. Dr.-Ing. Mira Mezini
2. Gutachten: Dr. Guido Salvaneschi

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Department of Computer Science
Software Technology Group

Tool Support for Reactive Programming
Tool Support für Reactive Programming

Vorgelegte Master-Thesis von Simon Sprankel

1. Gutachten: Prof. Dr.-Ing. Mira Mezini
2. Gutachten: Dr. Guido Salvaneschi

Tag der Einreichung:

Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 25.09.2014

(S. Sprankel)

Zusammenfassung

Reactive Programming hat in den letzten Jahren in der akademischen Welt sowie in der Industrie mehr und mehr Aufmerksamkeit erlangt. Verglichen mit traditionellen Techniken hat sich Reactive Programming als effektiver erwiesen, um leichter zu verstehende und besser wartbare Systeme zu implementieren. Leider gibt es momentan aber keine angemessene Tool Unterstützung bei der Entwicklung – vor allem im Hinblick auf den Debugging Prozess. Das Ziel dieser Abschlussarbeit ist es daher, den Debugging Prozess von reaktiven Systemen zu verbessern.

Wir präsentieren einen auf Reactive Programming spezialisierten Debugger. Dieser basiert auf dem Abhängigkeitsgraphen von reaktiven Werten und ist von fortgeschrittenen Debugging Tools wie allwissenden Debuggern inspiriert. Mit dem präsentierten System kann der Entwickler den Abhängigkeitsgraphen eines reaktiven Programms nicht nur zu einem bestimmten Zeitpunkt visualisieren, sondern auch frei in der Zeit navigieren. Eine domänenspezifische Abfragesprache bietet direkten Zugriff auf die Historie des Graphen, sodass man direkt zu interessanten Zeitpunkten springen kann. Diese Sprache hilft auch dabei, reactive-programming-spezifische Breakpoints zu implementieren. Das sind Breakpoints, die die Ausführung unterbrechen, sobald der aktuelle Programmstatus zu einer, vom Entwickler eingegebenen Abfrage passt.

Eine erste, auf verschiedenen Code-Beispielen basierende Auswertung zeigt, dass der entwickelte Debugger beim Verstehen reaktiver Systeme und dem schnellen Finden von Fehlern hilft. Vor allem übertrifft die in der Arbeit präsentierte Lösung traditionelle Debugger, da Abstraktionen von Reactive Programming direkt unterstützt werden.

Abstract

Over the last few years, reactive programming has gained more and more attention in academia and industry. Compared to traditional techniques, reactive programming has proved to be more effective to implement systems that are easier to understand and more maintainable. Unfortunately, there is no proper development tool support for the time being – especially in relation to the debugging process. The aim of this thesis is to enhance the debugging process of reactive systems.

We present a specialised debugger for reactive programming. The debugger is based on the dependency graph among reactive values and is inspired by advanced debugging tools, such as omniscient debuggers. In the system presented in this thesis, the dependency graph of a reactive software cannot only be visualised at a specific point in time, as the developer can navigate freely back and forth in time. A domain-specific query language provides direct access to the graph history, so that one can jump to interesting points in time. This language also helps in implementing reactive-programming-specific breakpoints. These are breakpoints, which interrupt the execution when the query provided by the developer matches.

A first evaluation based on various code examples shows that the developed debugger helps understanding reactive systems and finding bugs quickly. Especially, the solution presented in this thesis outperforms traditional debuggers as it directly supports reactive language abstractions.

Acknowledgements

I would like to thank my supervisor Guido Salvaneschi for his continuous support, his feedback and his patience. Thanks also go to my classmate Irina Smidt for her helpful feedback regarding language and content. I would furthermore like to thank Christine B. Maul-Pfeifer and Daniela Herrlein for their valuable feedback regarding grammar and style. Special thanks go to my girlfriend, my family and all my friends for their patience, support and appreciation. I will treat you to a drink!

Contents

1 Introduction

Reactive languages, such as FrTime, Flapjax, Scala.React or RESCALA get and deserve more and more attention, because they provide advanced mechanisms for developing reactive systems. Unfortunately, nearly all of them lack proper tool support – especially with regard to the debugging process. Hence, systems based on reactive programming are currently very hard to overview and bugs are very hard to find. This can of course lead to the situation that not many developers venture to use reactive programming. This thesis is another step towards usable debugging support for reactive programming, so that these problems may be solved in the near future.

This chapter introduces the whole topic with some background information, which will later be extended in the second chapter. The second section gives a little motivation, why better debugging support for reactive programming is necessary. The third section summarises the achieved contributions and the last section describes the structure of this thesis.

1.1 Background

Transformational systems do not need to react to changing inputs – traditional batch processes, such as compilers are an extreme example. In contrast to that, many modern systems are reactive, update their internal state according to the input and continuously interact with their environment. Changes in depending systems, changes of the environment or changes by the user can all lead to changes in the system. Users expect that modern systems react to these changes in no time and always reflect the current inputs. These systems are often referred to as reactive systems.

There are many different possibilities to implement reactive systems with different advantages and drawbacks. Roughly, in ascending attractiveness, these are the observer pattern, event-driven programming (EDP), aspect-oriented programming (AOP) as well as reactive languages. The most modern and advanced possibility is the usage of reactive languages. Hence, many reactive languages have been developed in the past years. FrTime [CK06], Flapjax [MGB+09], Scala.React [MO12] or RESCALA [SHM14] are just some examples.

The main idea of reactive languages or reactive programming is to provide language-level support for continuous changes to which the application has to react. This means that reactive languages provide the possibility to define so-called time-changing values. A traditional variable definition, such as `c = a + b` would be evaluated exactly once when this specific line is executed. If the same variable definition is somehow marked as time-changing, it would rather be interpreted as a constraint instead of a definition. There would be a dependency between `c` and the variables `a` and `b`, so that each time `a` or `b` changes, `c` would also be re-evaluated and would instantly reflect the recent input changes. These dependencies can also be visualised in a dependency graph, where each variable is a node and each dependency is represented by a directed edge. An example for a dependency graph of the aforementioned example is shown in Figure 1.1.

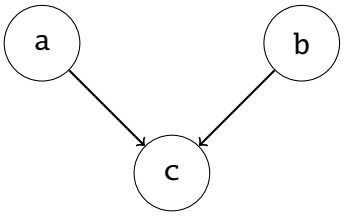


Figure 1.1.: Dependency Graph of a Reactive Application

The advantages of designs based on reactive languages over designs based on the observer pattern have often been discussed [CK06, MGB+09, MO12, BCC+13]. For instance, the usage of reactive languages leads to much less boilerplate code. Another example is that it is less error-prone, because updates do not need to be performed manually, but are automatically executed by the language.

Although the general advantages are obvious, it has not been evaluated until this year what the impact of reactive programming on the program comprehension is. But indeed both, preliminary empirical results and the first full empirical evaluation show that the program comprehensibility is enhanced by reactive programming [SM14, SAPM14]. Hence, further development in this topic seems to be worthwhile.

Besides the enormous academic interest in reactive programming, the topic also gains rising attention in the economy. An example for this is the "reactive manifesto", a document which defines criteria for reactive applications [Aut13]. Although it is written by many authors, the main authors have an economic and not academic background. The company Typesafe Inc. can be perceived as its initiator. The reactive manifesto claims that reactive applications have to be responsive, scalable, resilient and event-driven. These are the so-called reactive traits and should be understood as "design properties that apply across the whole technology stack" [Aut13]. They should not be understood as single, mutually-exclusive properties, but as connected, interrelated ones. Even though some of the reactive traits just seem to be present for marketing reasons, the reactive manifesto shows that reactive programming has arrived in the economy.

1.2 Motivation

Reactive programming is still in its infancy, so that there is actually no good tool support for it. There is especially no debugging support for it at all. And this is something which is extremely important for reactive programming. In reactive applications, changes at one place can result in changes at various different places. If a value changes, all depending values will also be updated automatically. This is something special to reactive programming, so that specialised debuggers are desirable.

Thinking about debugging, traditional breakpoint- or log-based debuggers are the first ones which come to mind. Breakpoint-based debuggers are based on the usage of line or method breakpoints. The developer creates breakpoints based on a specific line in the source code or on a specific method and the debugger will halt the execution once this point in the code is reached. Log-based debuggers are based on logging of all relevant events to a log file during the code execution and analysing this log file afterwards.

A more advanced debugging technique is omniscient debugging, also known as back-in-time, backwards, historical or reversible debugging. The idea is that one cannot only look at the

current program state and go forward step-by-step, but that one can have a look at the program state at any arbitrary time. It is also possible to go back in time during the debugging process. This is extremely comfortable and useful. A very good and easy-to-use example for such a debugger is the Elm debugger [1]. A proof-of-concept Eclipse plugin which implements an omniscient debugger for Java has also been developed by Pothier et al. [PTP07, PT09].

Although advanced debugging techniques, such as omniscient debugging exist, traditional debuggers are still commonly used and advanced debuggers are not built into modern IDEs. Additionally, the special needs when debugging systems based on reactive programming cannot be covered by traditional debuggers. A usable alternative providing debugging support for reactive programming does not exist yet. That is why there is a great need to investigate how better debugging support for reactive programming could look like and to develop appropriate tools.

1.3 Contribution

In the previous sections we introduced reactive systems, their implementation via reactive languages, the possibility to visualise them with dependency graphs, omniscient debugging as well as the great need for better debugging support for them. In this thesis, all these concepts are combined: the goal is to develop a debugger which can visualise the dependency graphs of reactive systems and provides the possibility to freely navigate through the history of these dependency graphs. In other words, an omniscient debugger for the dependency graphs of reactive systems should be developed.

In more detail, the contributions of this thesis are the following:

- It provides the possibility to visualise dependency graphs of reactive systems. Most notably, the whole history of the dependency graph is stored during the program execution and the facility to view the dependency graph at any point in time is provided.
- The history of the dependency graph cannot only be navigated manually, but can also be queried with the help of a domain-specific query language. This query language provides the opportunity to directly jump to important points in time of the dependency graph history.
- With the help of this query language, reactive-programming-specific breakpoints are implemented. These breakpoints provide the facility to halt the program execution directly when an interesting event occurred during the execution.

1.4 Structure

The thesis is structured in six chapters. Chapter 1 provides a short introduction into the topic and describes the goals of the thesis. Chapter 2 includes more background information and presents the state of the art of the important topics of this thesis: reactive systems in general, the various ways to implement them – especially reactive languages –, the problems when debugging them, advanced debuggers and REClipse – a first attempt to provide debugging support for reactive programming. The third chapter describes the system on a high-level and the design

[1] `http://debug.elm-lang.org/`, last accessed 24-09-2014

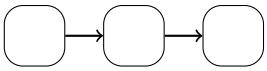
decisions that have been made. The requirements for the system are again explained in more detail. The fourth chapter explains the system on a lower level and more fine-grained. Interesting, selected implementation details are shown. Chapter 5 shows various case studies, evaluates the developed system and illustrates its usage as well as possible applications. Chapter 6 then concludes the thesis and outlines ideas for future work.

2 State of the Art

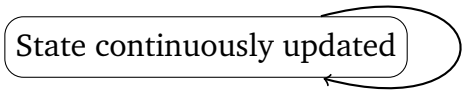
This chapter presents the state of the art of the topics relevant for this thesis. In the first section, we introduce reactive systems and the challenges in implementing them. The second section explains different implementation possibilities. Reactive programming languages in general and a specific reactive language which integrates seamlessly into an object-oriented setting – namely RESCALA– is described in the third section. Afterwards, the serious existing problems with the debugging of reactive systems are explained in the fourth section. A possible solution for these debugging problems based on advanced debuggers, particularly omniscient debuggers, is then introduced in the fifth section. The chapter finishes with an introduction to the already implemented Eclipse plugin which visualises reactive programs.

2.1 Reactive Systems

A traditional taxonomy classifies computing systems in *transformational* systems and *reactive* systems. Transformational systems receive some input, perform computations, return an output and terminate. Hence, the use of state is not essential. The different inputs and computations just lead to updates of the internal data structures:



In contrast to that, reactive systems continuously interact with their environment. They continuously update their state each time some event is fired and processed – the state is therefore essential to describe them:



2.1.1 Challenges in Implementing Reactive Systems

Implementing reactive systems is difficult because of the interaction between state and updates. Consider the following simple code, with which the basic principle of reactive systems can be explained.

```
1 val a = 1
2 var b = 1
3 val c = a + b
4 b = 5
5 println(c)
```

One would expect the output to be `2`. And this is indeed the way traditional programming approaches work. Setting the value of variable `b` to `5` will not affect the value of variable `c`, because `c` has been defined before. In reactive systems, line `2` would rather be interpreted as a constraint instead of an assignment, so that the output in the end would be `6`. The value of variable `c` will always be updated once `a` or `b` change their values. This is comparable with the way spreadsheet applications, such as Microsoft Excel work. The content of a cell depending on other cells will always be instantly updated once one of the cells it depends on is changed.

But how can such a reactive system be implemented? A manual approach would probably look similar to the following code:

```
val a = 1
var b = 1
var c = a + b
b = 5
bHasBeenUpdated()
println(c)

def bHasBeenUpdated() {
  // recalculate c, so that change of b is reflected
  c = a + b
}
```

This naturally leads to a lot of problems:

- The code which triggers updates, such as the call of `bHasBeenUpdated` above is scattered throughout the system. After each update of `b`, the triggering code has to be inserted.
- Developers may forget to insert the triggering code, so that important updates may be missed.
- Update code is executed defensively, so that it may be executed too often. The reason for it is that each change calls the update code, although changes may be combined and only one update may be necessary.
- It is not possible to compose different reactions. One cannot express new constraints based on existing ones.
- There is no separation of concerns. The update logic, the constraint definition as well as the triggering code are all tangled.
- There is a lot of boilerplate code just to define a simple constraint. In the example above, 50 % of the code is required just to express that `c = a + b` is a constraint which should always hold. This unnecessary code duplication is not desirable, makes the code less readable, less comprehensible and less maintainable.

2.2 Implementation of Reactive Systems

Apart from the unsatisfactory manual approach, there are multiple ways how reactive systems can be implemented. Initially, in this section “traditional” approaches, such as the observer design pattern, event-driven and aspect-oriented programming are surveyed. The next section focuses on reactive languages which are a central topic for the thesis.

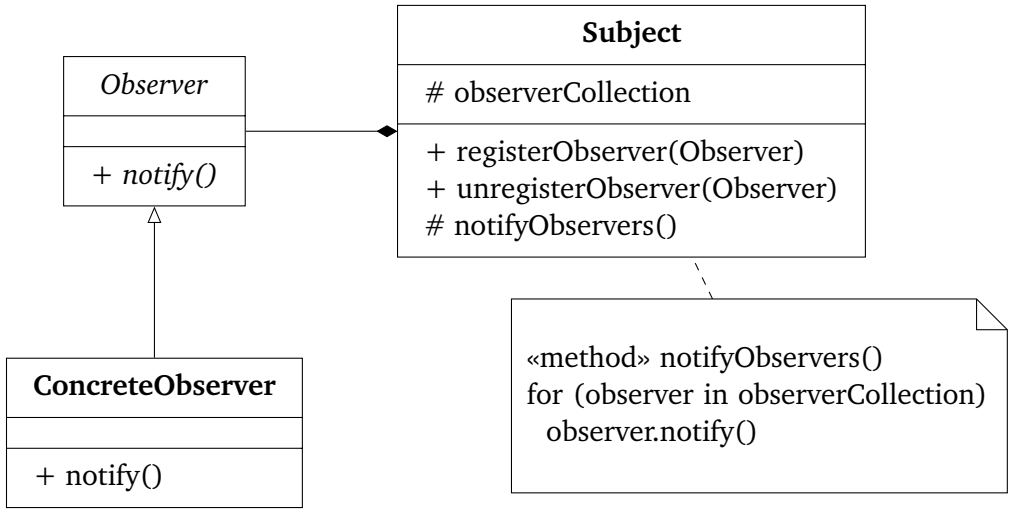


Figure 2.1.: Class Diagram of the Observer Pattern

2.2.1 The Observer Pattern

The observer pattern as a way of reacting to changes is well-known and is one of the first patterns every developer learns. It was first named in the famous book "Design Patterns: Elements of Reusable Object-Oriented Software" from the so-called "Gang of Four" [GHJV94]. As a refresher, a class diagram is depicted in Figure 2.1. The drawbacks of the observer pattern have already been broadly discussed in literature resulting in a paper "Deprecating the Observer Pattern", which explains the problems in detail and proposes built-in support for reactive programming abstractions as a solution [MRO10]. Nearly all of the aforementioned problems remain when using the observer pattern instead of the manual approach. There is only one real advantage: through the observer pattern, a better modularity is achieved. The update code is decoupled from the code that changes a value. This makes the code more readable and less error-prone.

2.2.2 Event-Driven Programming (EDP)

Programming languages are event-driven if they support events on the language-level. Examples for event-driven languages are C# [ISO06], ESCALA [GSM+11], Ptolemy [RL08] or EventJava [EJ09]. These languages support imperative, single events as well as declarative events which are composed of other events. An example for a declarative event would be `declarativeEvent = event1 || event2`, which would be fired each time `event1` or `event2` is fired. Event handlers, which are called each time an event is fired, can be registered for all kinds of events and can perform their updates. The following ESCALA example from the introductory paper shows how the code with such an event-based language could look like [GSM+11]:

```
class Drawing(val figures: List[Figure]) {
  // event which is fired each time one of the figures is invalidated
  event invalidated = figures.any(_.invalidated)
}
class View(val drawing: Drawing) {
  // attaches an event handler to the invalidated event of the given
```

```
  // Drawing instance
  drawing.invalidated += repaint
  def repaint(bounds: Rectangle) { // update code }
}
```

Event-based languages are advantageous compared to the observer pattern, since events are composable and the update code can be modularised and separated from the main code. Additionally, there is usually less boilerplate code necessary in event-based languages. But some drawbacks of the observer pattern remain: for instance, event handlers have to be registered manually, so that the dependencies are still encoded manually. Additionally, handlers still have to implement the updates explicitly – updates are not performed automatically.

2.2.3 Aspect-Oriented Programming (AOP)

Aspect-Oriented Programming has been introduced in 1997 by Kiczales et al. [KLM+97]. With AOP, it is possible to define points in the code at which some update code should be executed. These so-called pointcuts are based on events like the following:

- call of a method
- begin/end of a method execution
- begin/end of a constructor execution
- execution of an exception handler

They can usually be combined with logical operators to form more complex pointcuts. An example of a pointcut in the Java AOP library AspectJ [1] is the following:

```
call(int MyClass.myMethod(int))
```

It matches each call to a method `myMethod` of the class `MyClass`, which takes an `int` as a parameter and returns an `int` again. The update code which is executed each time a pointcut matches is called advice. The pointcut together with the respective advice is called aspect, which justifies the name of this paradigm.

Like EDP, AOP is advantageous compared to the observer pattern, because events are composable, update code can be separated from the main code and less boilerplate code is necessary. Comparing EDP and AOP, they can express different things. With EDP, it is natively not possible to run some update code each time specific methods are called. Although more specific, application-related events can be expressed with EDP more easily. AOP has the same drawbacks as EDP, e.g. that dependencies are encoded manually. That is why reactive languages have been developed, which are introduced in the next section.

2.3 Reactive Languages

Reactive languages try to solve all the problems of the aforementioned paradigms. The rationale is that the updates are not executed via events and handlers, but automatically by the respective language. Once a constraint is defined, the language takes care of updating all dependent values each time a change occurs. Recapitulate the simple example from the beginning of the chapter:

[1] `https://eclipse.org/aspectj/`, last accessed 24-09-2014

```
val a = 1
var b = 1
val c = a + b
b = 5
println(c)
```

In a reactive language, one could just define the variable `c` as a time-changing value. This would mean that it is instantly updated each time `a` or `b` change their values. The output in the example would therefore be `6`.

The first reactive language was Fran, an abbreviation for "functional reactive animation", and was implemented as a Haskell library [EH97]. It is often referred to as functional reactive programming (FRP). Examples for more recent reactive languages are FrTime [CK06], Flapjax [MGB+09] or Scala.React [MO12]. The following Scala.React example based on the introductory paper gives an impression on how reactive languages look like and work [MO12]:

```
object MyApp extends Observing {
  // quit event is fired when user quits the application via a quit
  // button or via the menu or when a fatal exception is thrown
  val quit: Events[Any] = quitButton.clicks merge
    quitMenu.click merge fatalExceptions
  def doQuit() {
    // do something before application exits
    System.exit()
  }
  // each time quit event is fired, doQuit will be executed
  observe(quit) { doQuit() }
}
```

All the aforementioned languages have a declarative approach based on functional programming, so that they do not fit in well with the object-oriented world. RESCALA is a first step towards integrating reactive behaviour as known from FRP into the object-oriented domain. RESCALA will be described in the next section in more detail, because it has been mainly used in this thesis.

2.3.1 REScala

RESCALA is a library developed by Guido Salvaneschi et al. which extends Scala to a reactive language [SHM14]. In RESCALA, there are two types of reactive values: `Vars` and `Signals`. `Vars` wrap primitive Scala values and define them as time-changing values. Examples of `Var` definitions are:

- `val a: Var[Int] = Var(10)`
- `val b: Var[String] = Var("a string")`
- `val c: Var[Boolean] = Var(true)`

The notation for changing the value of the variable `a` as defined above is `a() = 20` or `a.set(20)`.

`Signals` wrap side-effect-free Scala expressions and define them as time-changing expressions. Examples of `Signal` definitions based on the `Var` definitions above are:

- `val s1: Signal[Int] = Signal { a() + 1 }`
- `val s2 = Signal { if (c()) a() else b() }`

RESCALA also supports two kinds of events. Imperative events are defined by the developer via e.g. `val e = new ImperativeEvent[Int]()` and are also fired manually by the developer via e.g. `e(10)`. Declarative events are composed of other events with combination operators like logical operators. An example would be `val e = e1 || e2`, which defines a new event `e` which is fired each time the event `e1` or `e2` is fired. There are also more complex combination operators, such as `map`. The event `e map f` is fired each time the event `e` is fired. But the event `e map f` will call the function f on the parameter of the base event `e`, so that the parameter type of the new event is the return type of the map function.

An event handler `h` for an event `e` can be registered and unregistered at any time via `e += h` or `e -= h` respectively.

RESCALA provides conversion functions which easily convert `Signals` to events and vice versa. The basic conversion function which converts events to `Signals` is `latest`: `val s: Signal[Int] = e.latest(1)`. It returns a `Signal` which always holds the latest value of the event. The parameter defines the initial value of the `Signal`. In the other direction, the function is called `changed`: `val e: Event[Int] = s.changed`. It returns an event which is fired each time the value of the `Signal s` changes. There are also more complex conversion functions which are introduced in detail in the RESCALA manual [2].

These conversion functions are key to introduce time-changing values into object-oriented applications, which are usually event-based. So this is one of the main advantages of RESCALA: it can, as opposed to other reactive languages, seamlessly be integrated into object-oriented applications.

A fully-fledged RESCALA example, which will also be used later on, is the following Fibonacci example:

```
import rescala._
import makro.SignalMacro.{ SignalM => Signal }

object Fibonacci extends App {

  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() }
  println(f10.get)
```

[2] https://raw.githubusercontent.com/guidosalva/REScala/master/doc/manual/manual.pdf, last accessed 24-09-2014

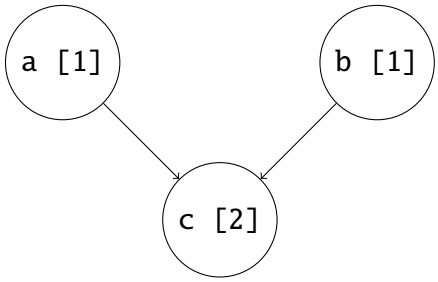


Figure 2.2.: Dependency Graph of a Very Simple Example Application

}

2.4 Debugging Reactive Programs

Debugging reactive programs can be a tedious task. While programs written in traditional, non-reactive languages can be tracked step-by-step rather easily with a breakpoint-based debugger; this is not possible with reactive programs. Assumed a reactive program would be tracked step-by-step, what should the debugger do when a value on which many other values depend is updated? Should it just skip the updates? This would not represent the inner workings at all and would skip important steps. On the other hand, switching from one update statement to the next would also be quite confusing. Since updates can trigger other updates, countless values would potentially have to be updated. Understanding and overseeing all these steps would probably not be possible. Hence, new tools have to be developed, which let developers oversee and understand reactive programs. One important aspect of this are dependency graphs. One can think of values as nodes in a graph and the dependency relation between two nodes as a simple directed edge from one node to another. The following code block has a dependency graph as depicted in Figure 2.2.

```
val a = 1
val b = 1
val c = a + b
```

There may be many dependency graphs in a reactive application, since there are normally many independent components. Inspecting these graphs will result in a good overview of a specific component of an application. This becomes even more powerful when it is combined with advanced debuggers, which are described in the following section.

2.5 Advanced Debugging

Advanced debuggers provide additional functionality to debug programs than the traditional breakpoint- or log-based debuggers. The advanced debugging technique, which is important for this thesis, is omniscient debugging. Though omniscient debugger seems to be the most prevalent term for this kind of debuggers, there are other equivalent terms, such as back-in-time, backwards, historical or reversible debugger. They not only let the developer step forward through the program, but also allow to go back in time. Hence, the developer can have a look at the state of the program at arbitrary points in time. This idea has first been formulated by Balzer

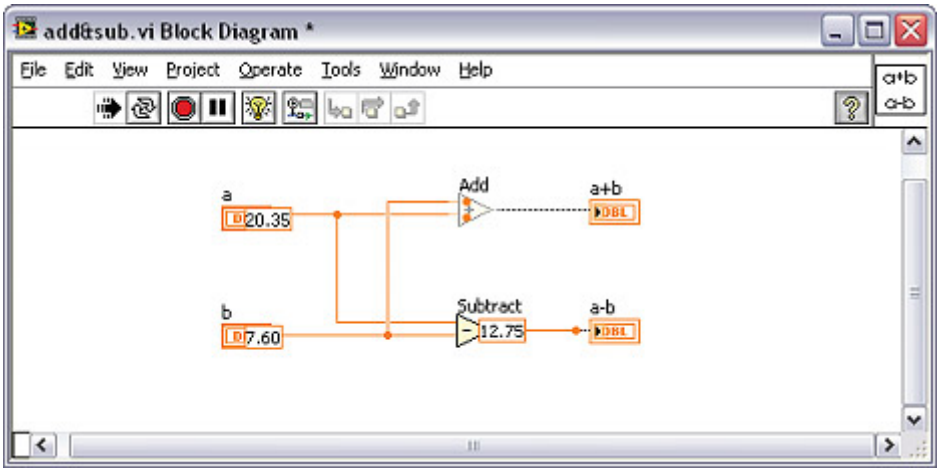


Figure 2.3.: Example of Debugging in National Instruments LabVIEW [Ins14]

in the year 1969 [Bal69]. So the idea is anything but new. Proof-of-concept systems have also been developed as early as 2003 [Lew03]. Nevertheless, there is no proper default integration of omniscient debugging into state-of-the-art IDEs. At least plugins which integrate the idea of omniscient debugging into IDEs such as Eclipse exist [PTP07, PT09].

The big benefit of omniscient debuggers is that they combine the advantages of log-based and breakpoint-based debuggers. Log-based debuggers have the advantage that the past program state is never lost – the developer knows how the program state was in the past. Breakpoint-based debuggers have the advantages that the developer can navigate interactively forward through the code execution and can have a look at the stack content at a specific point in time. Omniscient debuggers have all these advantages combined and are therefore quite worthwhile.

One of the main goals of this thesis is to apply omniscient debugging to the dependency graph of reactive programs. This gives the developer the possibility to temporally navigate back and forth the dependency graph. The developer can then track a whole lot of events:

- when nodes are created (which means when variables have been defined)
- when connections between nodes are created (which means when a constraint and therefore a dependency has been defined between two variables)
- when nodes/variables are evaluated
- how updates of variable values are passed through the program and affect other variables

A quite similar visualisation, which indeed does not allow backwards navigation, is integrated into National Instruments LabVIEW [Ins14]. LabVIEW is a graphical programming platform which allows engineers to build measurement or control systems based on physical instruments. Inputs can be easily connected with various computing units in order to build more complex system. So as to debug the system, the graphical visualisation can be observed while it is executing. The developer can see how different input signals are passed through the system and how they affect other parts of the system. An example of such a visualisation is depicted in Figure 2.3.

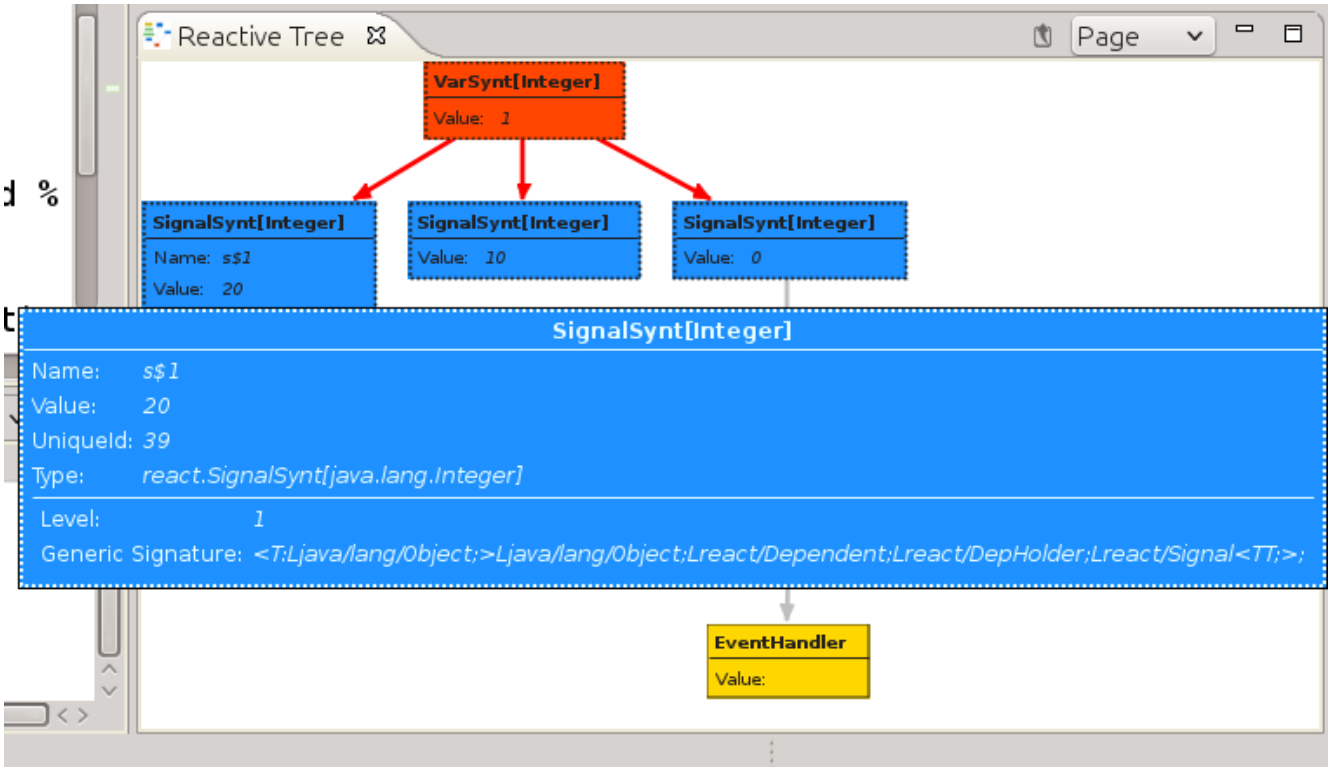


Figure 2.4.: Example Visualisation of a Dependency Graph by REClipse [Hau14]

2.6 A First Attempt: REClipse Eclipse Plugin

Michael Hausl has, as part of his diploma thesis, developed an Eclipse plugin called REClipse which can already show dependency graphs for reactive applications [Hau14]. It is written in a programming language independent way, so that various reactive programming languages can be connected to this visualisation plugin. In order to connect another reactive language, an Eclipse plugin has to be developed. In the plugin, a specific interface has to be implemented and provided via a specific extension point. Then, the base visualisation plugin can detect the new implementation and is able to visualise programs of the new language.

An example visualisation of REClipse is shown in Figure 2.4. Different layout algorithms can be chosen in order to get a different graph layout. Additionally, nodes can freely be rearranged via drag and drop. The resulting graph can then also be saved to a file. Internally, the plugin uses reflection in order to get the runtime information of the variables. This information is then shown as an overlay each time the developer hovers over the respective node.

REClipse is used as a basis for this thesis and is extended to support omniscient debugging.

3 System Design

In this chapter, we introduce the high-level requirements for the system developed in the thesis. Then, a high-level architectural overview of the system is given in the second section. Finally, we explain the design choices, such as the usage of the Eclipse plugin architecture or the usage of complex event processing in more detail.

3.1 System Requirements

The requirements for the system developed in the thesis are identified according to the general idea that the dependency graph visualisation and history improves the debugging process of reactive systems. We identified the following requirements expressed in terms of functionalities available to the end user and requirements on the software architecture.

General Availability

The new debugging functionality should seamlessly fit into an existing IDE and should be easy to install and integrate. Therefore, and in order to reach many developers, a plugin for the widely used Eclipse IDE should be developed.

Visualisation of the Dependency Graph

During the debugging process, the developer should be able to see the dependency graph based on a specific time-changing variable. Once a variable is selected, this variable and all its dependencies should be shown in a graph, so that the developer gets a good overview of the system. While stepping through the program, the graph should be automatically updated, so that new variables, new dependencies, updated values and so forth are directly visible.

Visualisation of the History of the Dependency Graph

The developer should be able to have a look at the dependency graph at any arbitrary point in time. Hence, the whole history of the dependency graph should be visualised. It should be possible to easily navigate through the time and to observe events, such as node creation, value changes, updates of dependencies and the like step-by-step.

Querying the History of the Graph

For larger programs, it is not feasible to manually navigate through the dependency graph history because it is too big. Therefore, a query language should be developed, which makes it easy to jump to specific points in the history. For instance, it should be possible to define a query which directly brings oneself to the points in time at which a specific node has been created or evaluated.

Reactive-Programming-Specific Breakpoints

Developers should be able to set breakpoints specific to reactive programming that stop the execution when a significant event occurs. For instance, it should be possible to set a breakpoint

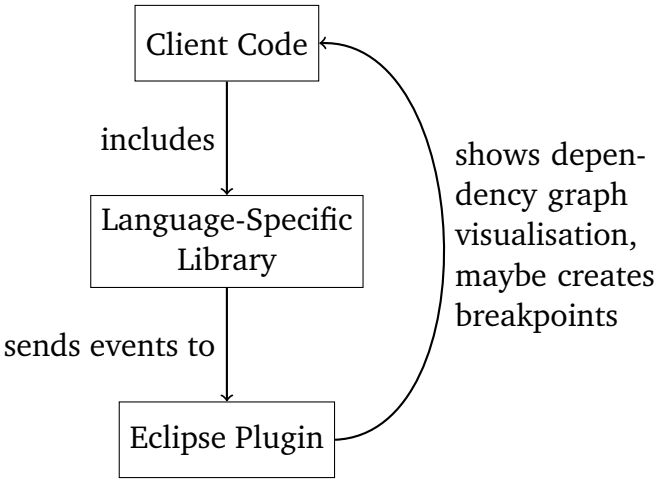


Figure 3.1.: Overview of the System Architecture

which hits when a specific node is evaluated and the evaluation yields a specific value. These breakpoints should be easy to express, so that the same query language as mentioned before should be used.

Language Independence and Extensibility

The system should not be limited to a specific reactive programming language. It should be independent of any reactive programming language, although there should be a default implementation for RESCALA. An obvious approach to implement the communication between the running system and the plugin in a language-independent manner is via a logging file. For the sake of simplicity, the communication may also be based on a language-specific concept, such as Java RMI. Since it is just a communication channel, it may be exchanged rather easily later on.

The query language should be built and defined in a way it can be easily extended. It should be possible to easily add new commands.

3.2 System Architecture Overview

The overall system design is illustrated in Figure 3.1. There are three important system components: First of all, there is the Eclipse plugin which provides the extended debugging functionality for reactive systems. Second, the language-specific library ensures that a reactive programming language, such as RESCALA is actually supported by the plugin. And as a third and last component, there is the client code which should be debugged. It has to be written in a language for which a language-specific library exists. Otherwise, such a library can be developed, so that this programming language is supported. In this thesis, an example implementation for RESCALA has been developed.

The general interaction between the different components is also shown in Figure 3.1. The client code has to include the language-specific library. This library will send appropriate events to the Eclipse plugin during the debugging process. These events are then processed by the plugin. The plugin can then show the dependency graph visualisation, provides possibilities to navigate it, may create breakpoints and so forth. The next section explains the system architecture in more detail.

3.3 System Architecture Details

Figure 3.2 shows the system architecture in more detail. The system is splitted into several projects, which correspond to the boxes in the figure. The language-specific library has to be added to the build path of the client code which should be debugged. In the case of RESCALA, this is the `REClipse_REScala` library. This library implements the logging trait provided by RESCALA. The implementing class called `REScalaLogger` has to be plugged into the RESCALA logging facility like that:

```
rescala.ReactiveEngine.log = new REScalaLogger
```

This enables the logging of all events created by RESCALA.

For the communication between the Eclipse plugin and the language-specific library, Java RMI is used. The `REScalaLogger` class gets an instance of the `LoggerInterface` class from the RMI registry and calls methods on this instance each time an event occurs. The `REClipse_LoggerInterface` project in general provides common data structures and interfaces, so that the language-specific library and the generic Eclipse plugin can communicate with each other. The language-independent Eclipse plugin is called `REClipse_ZestViewer` and implements the logger interface defined in the `LoggerInterface` project. It provides this implementation to clients by binding it to the RMI registry. In the plugin, the external library Esper is used in order to filter and especially match events on-the-fly. Another external library, ANTLR, is used in order to parse the queries of the query language.

3.4 Dependency Graph Visualisation

As explained in the previous chapter, reactive systems can be visualised with dependency graphs. Each time-changing value is a node in the graph and there is a directed connection between two nodes if one node depends on the other one. Consider the already established example extended by one more variable `d`:

```
val a = 1
val b = 1
val c = a + b
val d = 2 * c
```

If a value of a variable is changed, e.g. the value of the variable `a` is changed to `5`, the changes propagate through the dependency graph along the connections as illustrated in Figure 3.3. This gives a very good overview of the reactive system and especially of the dependencies therein. It displays vividly how changing values affect different parts of the application. This should be of great help for developers to better understand reactive systems. That is why further investigations regarding the history of dependency graphs have been made in this thesis.

The only problem which may occur is that it does not integrate well with traditional, non-time-changing variables. These are not shown in the dependency graph, so that the program may not be fully understood altogether. On the other hand, the developer can still use a traditional breakpoint-based debugger for the non-reactive part and get a good overview of the reactive part with the dependency graph history.

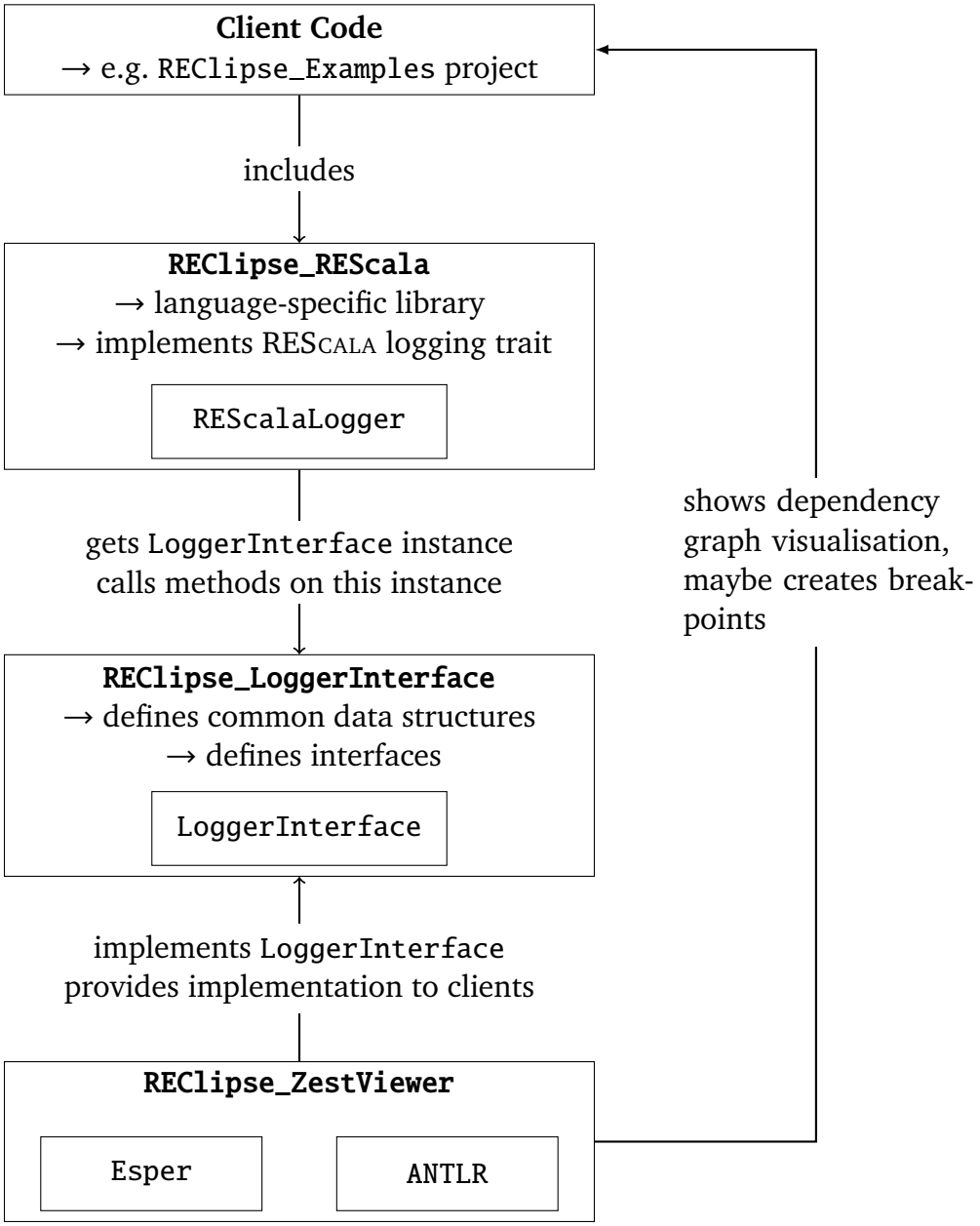


Figure 3.2.: Detailed System Architecture

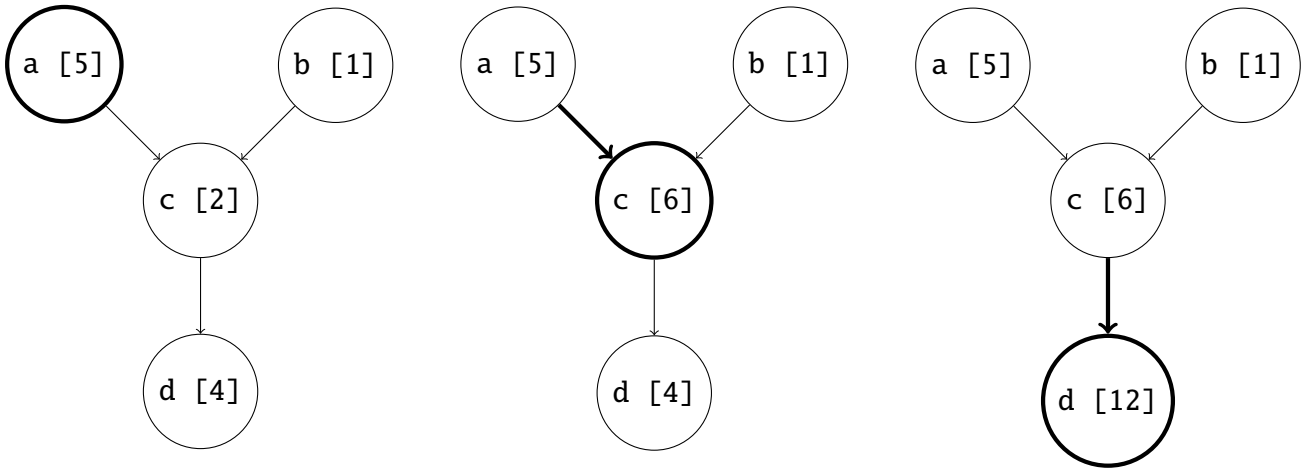


Figure 3.3.: Dependency Graph History Exemplified: Changes Propagate Through the Graph

3.5 Navigation through the Dependency Graph History

The visualisation of the dependency graph is not only instantly updated when the developer steps through the application, but it is also stored each time it changes. Hence, the developer is able to navigate through the whole history of the dependency graph. It is possible to see the dependency graph at any point in time. The developer can navigate through the history in three ways:

History Navigation

One way is to simply use the back and forth buttons which jump to the point in time directly before or directly after the current point in time.

Direct Access

Since history navigation may be impracticable for large histories, the developer can also drag the provided slider and thus quickly jump to arbitrary points in time.

History Queries

A third and last option to navigate through the history is the provided query language. By entering queries which match certain events, the developer can directly jump to the respective points in time where these events occurred.

3.6 Reactive-Programming-Specific Breakpoints

Traditional breakpoints are based on specific lines or methods. A line breakpoint hits and halts the execution each time the code execution reaches a specific line in a specific file. Method breakpoints hit each time the execution reaches a specific method of a specific class. Both breakpoint types can also be conditional, which means that they only hit if a specific condition evaluates to true. These breakpoint types do not really fit well into the reactive programming approach. Hence, reactive-programming-specific breakpoints should be implemented. They reuse the developed query language. The developer can enter an arbitrary query, start the

IDE	URL to Plugin Directory	Number of Plugins
Eclipse	`http://marketplace.eclipse.org/`	1659
IntelliJ IDEA	`http://plugins.jetbrains.com/?idea`	1187
NetBeans	`http://plugins.netbeans.org/`	859

Table 3.1.: Most Popular Java IDEs and Respective Number of Plugins, Status: 24-09-2014

debugging session and the debugger will halt the execution each time the query matches. This feature should be of great help to debug reactive systems.

As an example, consider the following, already established code again:

```
1 val a = 1
2 val b = 1
3 val c = a + b
```

A reactive-programming-specific breakpoint using the above code is the query `nodeCreated(b)`. This breakpoint hits once the variable `b` is defined. Hence, it halts the execution at line `3`. This could of course still be done with a traditional debugger by creating a line breakpoint at the respective line. But there are also examples which cannot be expressed with a traditional debugger. Especially the queries which may have many results are quite handy. For instance, one does not have to search the code oneself for all places where a specific node is evaluated and create line breakpoints there, but one could express that with the query `nodeEvaluated(c)`. Or, as a last example, the `evaluationYielded(c, "2")` query only matches if the variable `c` is evaluated to the value `"2"`.

Unfortunately, this feature partly breaks the programming language independence of the plugin. Breakpoints in Eclipse are debugger-specific. And since there are different debuggers for different languages, the creation of breakpoints had to be developed for one specific programming language. It has been developed for the Java debugger, so that the plugin depends on the Eclipse plugin `org.eclipse.jdt.debug`. This dependency is only used for the breakpoint creation feature. This dependency definitely has to be fulfilled due to the Eclipse architecture, but if another debugger is used, only the feature of reactive-programming-specific breakpoints should fail. The rest of the features should work anyway.

3.7 Plugin for the Eclipse IDE

The REClipse plugin which is extended in this thesis is already implemented as a plugin for the Eclipse IDE. But apart from that, Eclipse is a good choice in order to have a highly customisable and widely used IDE anyway. It is naturally not possible to get real usage statistics of IDEs, but there are some sources which definitely militate for Eclipse [1]. The number of plugins officially available for an IDE may be a good sign for how customisable it is and also how widespread its use is. And these numbers definitely also militate for Eclipse as Table 3.1 proves.

The implemented plugin integrates into the Eclipse architecture by using extension points. Extension points are an Eclipse mechanism, which allows to extend the functionality without editing any Eclipse core code. The plugin uses two extension points, which also serve as two

[1] see e.g. `https://www.java.net/poll/i-do-most-my-coding-using` or `http://sdtimes.com/zeichicks-take-java-java-everywhere/`, last accessed 24-09-2014

examples how Eclipse can be extended with this mechanism. First, a new view is defined, which can be opened as a tab at various places. In this view, the dependency graph visualisation is shown. Second, a popup menu is extended in order to add another action to it. This action enables the visualisation of the dependency graph and opens the respective view.

3.8 Communication via Java RMI

One goal of this thesis was to keep the independence of any specific programming language. In the base plugin REClipse, which has been extended, this has been achieved through the usage of the Eclipse extension point API: REClipse needs at least two plugins for proper functionality. One plugin is independent of any programming language and is only responsible for visualising the dependency graph. Another, programming language specific plugin has to be developed if a new programming language should be supported. By default, REClipse supports RESCALA, so that there is a RESCALA-specific plugin which uses the extension point and makes the visualisation compatible with RESCALA. When one wants to see the current dependency graph, one clicks a button. Then, the visualisation plugin will search for an appropriate plugin implementing the extension point and ask it for the information required to build the graph. Hence, the visualisation plugin gets a snapshot in-time of the dependency graph. Since the new plugin has to show the whole history of the dependency graph and not only the current state, this design decision had to be reconsidered.

The selected solution is not to ask for the required information once the developer wants to see the dependency graph, but to continuously collect the state of the dependency graph in the background. This is achieved through continuous logging of relevant events. The whole communication between the running program and the Eclipse plugin works via Java's remote method invocation (RMI) feature. The running program logs relevant events – e.g. when a time-changing value has been defined or when such a value is evaluated – by calling respective server methods. The Eclipse plugin, which acts the part of a server here, stores the given event in the database on each method call. With the stored information, the plugin can construct an appropriate dependency graph for any point in time. It can then directly show the dependency graph when the developer wants to see it by reading the information from the database. Using Java RMI is quite comfortable, because one does not have to care about any low-level concepts, but it is of course not language-independent. Nevertheless, it is just a communication channel and exchanging it is pure handwork. It does not contribute anything to the conceptual work, so that the decision for Java RMI has been made.

An alternative, truly language-independent approach for the communication would have been to base the communication on log files. That was actually the first idea and has already been fully implemented, but there is one major drawback of this approach: the communication via log files is naturally asynchronous, so that the execution of the program continues after an event has been logged. But in order to support reactive-programming-specific breakpoints, synchronous communication is needed. Because as soon as such a breakpoint hits, the plugin has to set a respective traditional line breakpoint in the editor, so that the execution stops. Hence, for the sake of this breakpoint feature, the decision for synchronous communication via Java RMI has been made, although this introduces performance and scalability issues.

3.9 Complex Event Processing (CEP)

The term complex event processing has mainly been shaped in the introductory book "The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems" by David Luckham [Luc08]. The idea is that events do not have to be stored in e.g. a classical relational database and queried afterwards, but to do it the other way round. Hence, queries are formulated beforehand and when events come in, the queries are directly executed on them. Events which do not match a specific query can then directly be discarded.

For the use case of this thesis, the events have to be stored anyhow, because it has to be possible to reconstruct the dependency graph at any point in time. But the usage of CEP is still possible and advantageous. It is of course feasible to use an event stream built from a relational database and let the CEP engine run queries on that event stream. And the usage of a CEP engine makes the system more powerful, flexible and extensible. The CEP engine Esper has been used in this thesis, because of its industry strength; because it is open source, actively maintained and easy to use: it is enough to include a few jar files into the project where Esper [2] should be used.

An example for the usage of CEP is the aforementioned reactive-programming-specific breakpoints feature. The developer enters a query such as `nodeCreated(b)` and the execution is halted once a matching event occurs. Therefore, the incoming events are sent to the Esper CEP engine which directly executes the query on them and checks if the respective event matches. If this is the case, the execution is halted.

3.10 Query Language Extensibility

The developer can put a request to the system through a domain-specific query language. For instance, it is possible to ask the system for the point in time at which a specific node has been created or evaluated. This point in time will then be shown in the dependency graph history. In general, there are two possibilities to parse a query – or, to be more specific, to match a query and read the parameters from it: First, one could simply use regular expressions. This has the advantage that they are rather easy to implement and that it is not required to use another library since regular expressions are already supported in nearly all programming languages including Java. A query which selects events when a specific node is evaluated could e.g. be defined by `nodeEvaluated(nodeName)`. This query can then be matched by the regular expression `nodeEvaluated\\(([^)]+)\\)`.

Another possibility to parse the queries is to use an external library for it. This has the advantage that it is much more powerful. It may be harder to learn in the first place, but the query language can then be extended much more easily when such a library is used. Besides, the grammar definitions of external libraries are much more readable than a set of regular expressions. Hence, for the sake of extensibility and readability, an external library is used.

There are various Java libraries out there which generally come into question. The chosen library should meet the following basic requirements:

- The library should still be actively maintained.

[2] `http://esper.codehaus.org/`, last accessed 24-09-2014

- Though subjective, the grammar definition of the library should be easy to learn, readable and rather self-explanatory.
- The code required for matching a query and reading the parameters should be terse.

Three Java libraries seem to be quite established: Xtext, JavaCC and ANTLR. Xtext [3] is a good candidate, but the grammar definition is not really self-explanatory. JavaCC [4] is another candidate, but the grammar definition also takes getting used to. ANTLR 4 [5], which is an abbreviation for "ANother Tool for Language Recognition", seems to fulfil all of the mentioned requirements. The grammar definition is quite readable and self-explanatory, as a look on it in Listing 4.1 reveals. The whole code required for parsing the query is not exactly terse since ANTLR requires some auto-generated classes, but the code one has to implement oneself is definitely terse. Additionally, it is also used by Esper and therefore seems to be the perfect choice in this case.

[3] `http://www.eclipse.org/Xtext/`, last accessed 24-09-2014
[4] `https://java.net/projects/javacc/`, last accessed 24-09-2014
[5] `http://www.antlr.org/`, last accessed 24-09-2014

4 System Implementation

This chapter is dedicated to the implementation details of the debugger. In the first section, we present the interface between the plugin and the RESCALA logger. The second section describes the implementation details of the generic Eclipse plugin. The implementation of the RESCALA-specific plugin is explained in the third section. The fourth and last section describes the graphical user interface provided by the plugin.

4.1 Interface between the Plugin and the REScala Logger

When the developer starts a debugging session, the interface between the plugin and the RESCALA logger – or potentially any logger – comes into play. This interface is based on Java RMI. The language-specific logger acts as a client and the plugin acts as a server, because the communication is basically unidirectional. The language-specific logger sends events to the generic Eclipse plugin, but the Eclipse plugin does not send anything back to the language-specific logger. It only communicates with the user via the graphical user interface and may create Java line breakpoints, which directly happens in the background via the Java development interface.

Figure 4.1 shows a class diagram of all the classes involved in the communication. For the RMI setup, the plugin will provide its implementation of the `RemoteLoggerInterface` to clients by binding it to the RMI registry. The language-specific logger – in our case called `REScalaLogger` – can then lookup this remote logger and can call methods on it. When events are sent to the plugin by invoking methods of the remote logger, the plugin stores the events into a database and possibly halts the execution if the current event matches the developer's query. The common data structures and interfaces for the communication are defined in the project `REClipse_LoggerInterface` and further introduced in the following two sections. Since the `LoggerInterface` project defines these commonly used structures, it has to be included by the plugin as well as by the logger.

4.1.1 Significant Data Structures for the Communication

Four important data structures are commonly used by the plugin and the logger:

Reactive Variable Types

In different languages, different types of reactive variables exist. The Java enumeration `ReactiveVariableType` defines the possible types:

- `VAR`: The time-changing equivalent to a primitive value.
- `SIGNAL`: A time-changing value which can contain arbitrary expressions.
- `EVENT`: An event definition.
- `EVENT_HANDLER`: A language construct which handles events.

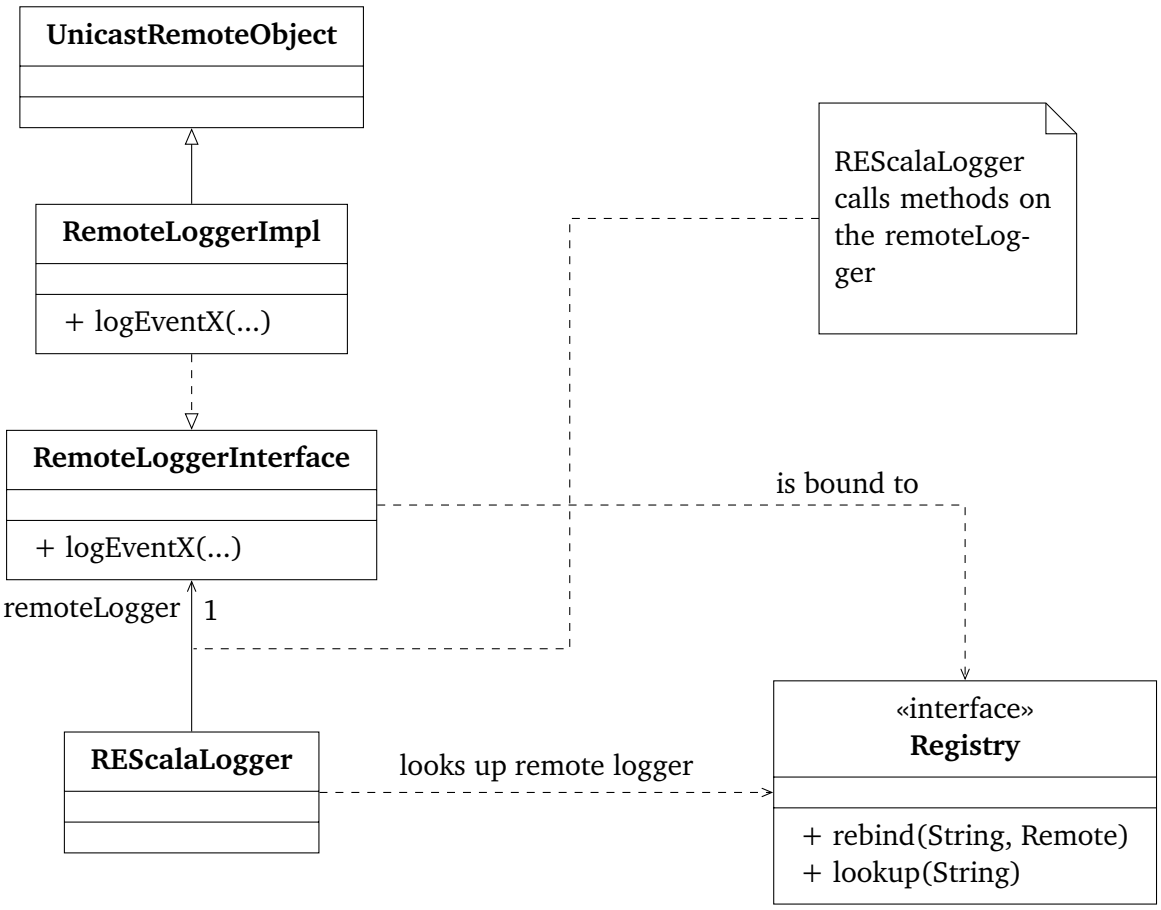


Figure 4.1.: Class Diagram of the RMI Communication Between the Language-Specific Logger and the Generic Eclipse Plugin

Event Types

A Java enumeration called `DependencyGraphHistoryType` defines all the event types which may occur:

- `NODE_ATTACHED`: When one node is attached to another, so that a dependency between the two nodes is established.
- `NODE_CREATED`: When a new time-changing value is defined, so that a new node in the dependency graph is created.
- `NODE_EVALUATION_ENDED`: When a node evaluation is finished and the node has its new value.
- `NODE_EVALUATION_ENDED_WITH_EXCEPTION`: When a node evaluation is finished and the evaluation resulted in an exception.
- `NODE_EVALUATION_STARTED`: When a node evaluation has just been started.
- `NODE_VALUE_SET`: When the value of a node has been set to a specific value.

Information about Reactive Variables

The `ReactiveVariable` class represents a reactive variable and stores all the relevant information. It contains the following fields:

- `UUID id`: A unique ID which identifies this reactive variable.
- `ReactiveVariableType reactiveVariableType`: The type of this reactive variable.
- `int pointInTime`: The point in time of the dependency graph history. Each reactive variable is stored once for each point in time, because properties may change over time.
- `DependencyGraphHistoryType dependencyGraphHistoryType`: The type of event at the aforementioned point in time.
- `String additionalInformation`: A field for arbitrary additional information. Must contain a string of the form `"ID1->ID2"` if the current event is of the type `NODE_ATTACHED`. This is used in order to detect more easily which connection is currently established.
- `boolean active`: Indicates whether this reactive variable is currently active.
- `String typeSimple`: The simple type of this reactive variable. An example value is `Signal[Integer]`. The simple type in square brackets is the respective inner type.
- `String typeFull`: The full type of this reactive variable. An example value is `rescala.SignalSynt[java.lang.Integer]`. The full type in square brackets is the respective inner type.
- `String name`: The variable name of this reactive variable in the source code. Especially important for the usage of the query language, since nodes are there identified by their name.
- `Map<String, Object> additionalKeys`: A map of additional keys which may be used freely by each reactive library. For instance, RESCALA stores the level of the reactive variable in the dependency graph here.

- `String valueString`: The current value of this reactive variable as a string.
- `Set<UUID> connectedWith`: The IDs of the nodes this node is connected with.

Breakpoints

The last important piece of information which has to be transmitted is where exactly a breakpoint may has to be created. This information is stored in the `BreakpointInformation` class. It only contains three fields which are necessary in order to create a respective breakpoint: The path to the source code of the class, the name of the class and the line number.

4.1.2 The Remote Logger Interface

The `RemoteLoggerInterface` defined in the `REClipse_LoggerInterface` project is a simple interface which defines all methods which can be called by the language-dependent logger on the Eclipse plugin implementing this interface. The methods correspond one-to-one to the event types defined by the `DependencyGraphHistoryType` enumeration. For instance, for the `NODE_EVALUATION_ENDED` event, the respective method is called `logNodeEvaluationEnded`. Each time a specific event occurs, the logger will call the respective method which sends this event to the server. All methods have the `ReactiveVariable` instance of the currently active reactive variable as well as a `BreakpointInformation` instance as parameters. Two methods have one more parameter each: first, the `logNodeAttached` method also takes the ID of the newly added, dependent node as a parameter. Second, the `logNodeEvaluationEndedWithException` method also takes the respective `Exception` instance as a parameter.

4.2 Plugin Logic

In this section, we describe selected implementation details of the plugin logic. We introduce the implementation of the remote logger interface, the deployment thereof as well as the implementation of the query language.

4.2.1 Implementation of the Remote Logger Interface

The implementation of the `RemoteLoggerInterface` is the component in the Eclipse plugin which receives and processes all the events sent by the language-specific library. It uses Esper in order to directly match the incoming events against the query the developer may have entered. It is implemented in the generic `REClipse_ZestViewer` project. The implementing class is called `RemoteLoggerImpl`. The sequence diagram in Figure 4.2 gives a rough overview of the interaction between the `RemoteLoggerImpl`, another internal plugin class called `EsperAdapter` and the whole Esper engine from the Esper library. The methods in the `RemoteLoggerImpl` are called via Java RMI. These methods first store the new event into the database. The database which is used therefore is called `reclipse` and the respective table is called `revars`. The columns in this table directly correspond to the fields of the `ReactiveVariable` class. The database structure is set up once Eclipse is started.

Afterwards, the Esper engine comes into play in order to implement the reactive-programming-specific breakpoints. The `ReactiveVariable` instance, which is a parameter

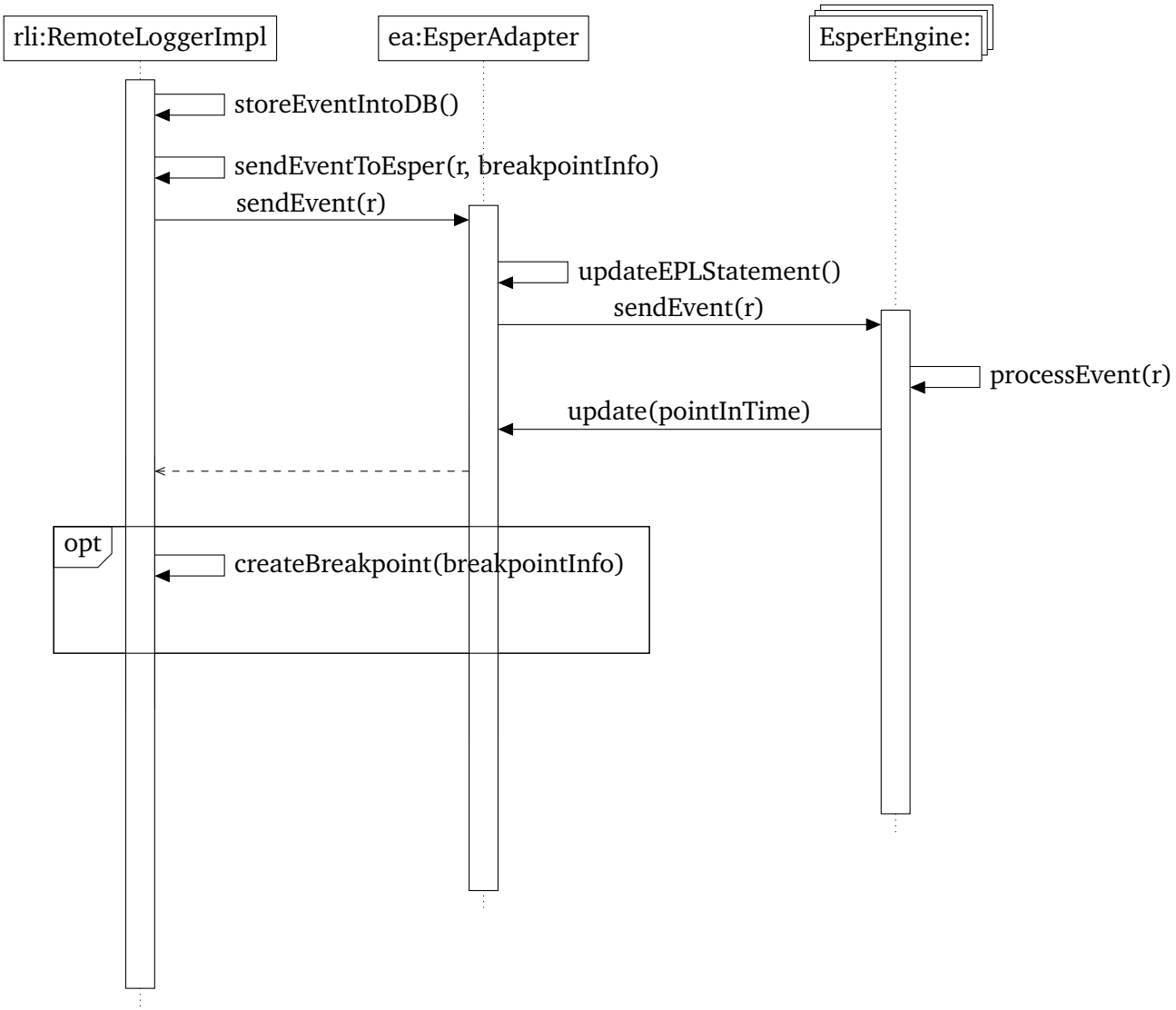


Figure 4.2.: Sequence Diagram Visualising the Remote Logger Interface Implementation

of the RMI methods, is sent to the `EsperAdapter`. If the developer entered a query, the `EsperAdapter` now creates and starts an Esper statement which represents this query. Additionally, the `EsperAdapter` registers itself as a subscriber to this statement, which means that it is called by the Esper engine if an event matches this statement. The `ReactiveVariable`, which also acts as an event object for the Esper engine, is then sent to the Esper engine. The Esper engine processes the event and calls the `update(pointInTime)` method of the `EsperAdapter` again if the query matched the current event. The `pointInTime` given to the `update` method is the point in time at which this event occurred.

If the event really matched the query entered by the developer, a traditional line breakpoint has to be created, so that the debugger halts the execution at the next possible place. This is done by the `RemoteLoggerImpl` via the Java development interface provided by `org.eclipse.jdt.debug`. The breakpoint is created with the method `JDIDebugModel.createLineBreakpoint()`. The information where exactly this breakpoint has to be created comes from the `BreakpointInformation` parameter of the RMI method.

4.2.2 Deployment of the Remote Logger Interface Implementation

So that the language-specific library can call methods on the remote logger interface implementation `RemoteLoggerImpl` and therefore send events to it, the implementation somehow has to be made available to the clients. This is done via an RMI registry. The implementation has to be provided as early as possible, because it is directly needed when the developer starts a debugging session. Since there is no reliable way to detect the start of a debugging session, the Eclipse extension point `org.eclipse.ui.startup` is used. With this extension point, it is possible to define a class implementing the interface `org.eclipse.ui.IStartup` whose `earlyStartup()` method is called after the workbench has been initialised. In this method, the `RMIServer` class which implements the `Runnable` interface is started. The `run()` method then basically does the following:

1. If no security manager has been established yet, a new `RMISecurityManager` is installed. Therefore, two system properties are set: first, the `java.security.policy` property is set, which defines the path to the server policy file which also comes with the plugin. Second, the `java.rmi.server.hostname` is set to the local IP address `127.0.0.1`. This should prevent any RMI security issues.

2. A new instance of the `RemoteLoggerImpl` class is created.

3. It is tried to create and export a new registry on the default port. If this is not possible because such a registry has already been exported, an `ExportException` is thrown. In the case that this exception occurs, it is caught and the respective registry is read from the `LocateRegistry`.

4. Now that there is an instance of the `RemoteLoggerImpl` and a registry, the instance is bound to the registry, so that clients can access it via the registry.

4.2.3 The Query Language

With the help of the query language, the developer can query the dependency graph history for interesting events. As described in the previous chapter, this can be done in two ways. First, the history can be queried after the program execution in order to find relevant events more easily. Second, the query can also be entered beforehand, so that the plugin will halt the execution once a matching event occurs. This is known as the reactive-programming-specific breakpoint feature. The query language supports six commands for the time being, but can easily be extended. The single commands are listed and explained in Table 4.1. The query language is implemented using the ANTLR 4 library, which is comprehensively described in "The Definitive ANTLR 4 Reference" by Terrence Parr [Par13].

ANTLR 4 grammar definitions are logically structured. At first, a name is assigned to the grammar, which has to be equal to the file name. Afterwards, all starting with lowercase letters, the parser rules are defined. They define the high-level structure of the defined language. They make use of lexer rules, which start with an uppercase letter and which define various datatypes. Often, lexer rule names are solely written in uppercase letters. The border between parser and lexer rules is not exactly clear-cut. There are many cases in which the grammar designer has to decide if a rule should be expressed as a parser or a lexer rule.

Command	Parameters	Description
`nodeCreated(name)`	`name`: a name of a node	specific node is created
`nodeEvaluated(name)`	`name`: a name of a node	specific node is evaluated
`nodeValueSet(name)`	`name`: a name of a node	value of a specific node is set
`dependencyCreated(name1, name2)`	`name1/name2`: names of nodes	dependency between 2 specific nodes is created
`evaluationYielded(name, value)`	`name`: a name of a node `value`: a String (!) value, such as `"value"`	evaluation of specific node yields specific value
`evaluationException(name)`	`name`: a name of a node	evaluation of specific node throws an exception

Table 4.1.: Commands of the Query Language

In the `Reclipse` grammar, the decision between parser and lexer rules was quite straightforward, because it is a rather simple language. The grammar is shown in a shortened form in Listing 4.1. It should be – especially together with the commands in Table 4.1 – understandable as is. In lines 1–16, the grammar name and the parser rules are defined. This already covers the important parts of the language. Lines 17–19 contain parts of the lexer rules. The language has been designed so that the `NODE_NAME` rule matches the usual identifiers in modern programming languages. The `VALUE` matches any String value enclosed in quotes, such as `"String value"`. Hence, the definitions have been taken from the official ANTLR 4 grammar for Java [1]. These definitions take about 70 lines, so that they are omitted here. The last line only claims that all whitespace in between the different parts of the query are ignored or “skipped”.

```
1  grammar Reclipse;
2
3  query: nodeCreatedQuery
4       | nodeEvaluatedQuery
5       | nodeValueSet
6       | dependencyCreated
7       | evaluationYielded
8       | evaluationException;
9
10 nodeCreatedQuery: 'nodeCreated(' NODE_NAME ')';
11 nodeEvaluatedQuery: 'nodeEvaluated(' NODE_NAME ')';
12 nodeValueSet: 'nodeValueSet(' NODE_NAME ')';
13 dependencyCreated: 'dependencyCreated(' NODE_NAME ',' NODE_NAME ')';
14 evaluationYielded: 'evaluationYielded(' NODE_NAME ',' VALUE ')';
15 evaluationException: 'evaluationException(' NODE_NAME ')';
16
17 NODE_NAME: Identifier;
18 VALUE: StringLiteral;
19 // definitions of the Identifier and StringLiteral rules omitted
20 WS: [ \t\r\n\u000C]+ -> skip;
```

Listing 4.1: ANTLR 4 Grammar for the Query Language

Defining the grammar in a standardised and readable format is one contribution of ANTLR. But what makes the query language usable are the parser, lexer, visitor and listener which are auto-generated by ANTLR. The lexer and the parser just tokenise and parse the code in a proper

[1] `https://github.com/antlr/grammars-v4/blob/master/java/Java.g4`, last accessed 24-09-2014

way, so that it can be processed further. Visitors and listeners can then do something with the parsed code. As written in the reference book, the "biggest difference between the listener and visitor mechanisms is that listener methods are called by the ANTLR-provided walker object, whereas visitor methods must walk their children with explicit visit calls. Forgetting to invoke visit() on a node's children means those subtrees don't get visited." [Par13]. Since we want to have full control over the visited parts of the code and since they are rather simple to implement, a visitor is used for processing the query language code. A typical control flow between the lexer, the parser and a visitor looks like the following:

```
ReclipseLexer lexer = new ReclipseLexer(
        new ANTLRInputStream("nodeCreated(f1)")
);
CommonTokenStream tokens = new CommonTokenStream(lexer);
ReclipseParser parser = new ReclipseParser(tokens);
ParseTree tree = parser.query();
ReclipseVisitorMySQLImpl visitor = new ReclipseVisitorMySQLImpl();
String conditions = visitor.visit(tree);
```

In this case, the visitor is implemented so that it returns a `String`. But visitors as well as listeners are generic, so that they can return any type of data. Here, the visitor translates the query into MySQL conditions. There is also another implementation called `ReclipseVisitorEsperImpl`, which translates the query into Esper conditions. These conditions can then be used in order to write MySQL queries or Esper statements with the respective conditions.

4.3 REScala Logger

In this section, we describe selected implementation details of the RESCALA logger. We explain the already existing logging functionality of RESCALA, the implementation of the logging class as well as the retrieval of the logged data.

4.3.1 The Logging Classes

So that the Eclipse plugin can visualise the dependency graph and provide the discussed functionality, it depends on the events which are sent to it. Therefore, a language-specific library, such as `REClipse_REScala` has to be able to get these events from the reactive language somehow. Fortunately, RESCALA already provides such a logging functionality, which was quite handy for this thesis. All the events used in this thesis are covered by this logging functionality. Some of them have been revised or added to the RESCALA code base during the development of this thesis though. The logging functionality in RESCALA is implemented in the file `rescala.log.Logging.scala`. In this file, a trait called `Logging` is defined which contains all methods required for the logging events. Additionally, an implementation of this trait which does not log anything at all is provided. This implementation called `NoLogging` is used as a default by RESCALA. If one wants to do something with the log events, one has to implement its own class implementing the `Logging` trait. So that the custom logging implementation e.g. called `MyCustomLogger` is used, the code in which the events should be logged has to include the following line:

```
rescala.ReactiveEngine.log = new MyCustomLogger
```

The implementation of the logger used by this thesis is called `REScalaLogger`. It basically just creates instances of the `ReactiveVariable` and `BreakpointInformation` classes which contain all the important information and then calls the respective RMI methods. The Eclipse plugin then receives these RMI calls and can process the given events. So that the respective methods can be called via Java RMI, the implementation of the `RemoteLoggerInterface` has to be read from the RMI registry. The steps in order to get this implementation are quite similar to the process of providing the implementation to the clients on the server-side:

1. If no security manager has been established yet, a new `RMISecurityManager` is installed. Therefore, two system properties are set again: first, the `java.security.policy` property is set, which defines the path to the client policy file. This file is also included in the `REClipse_REScala` project. Second, the `java.rmi.server.hostname` is set to the local IP address `127.0.0.1` again.

2. The implementation of the `RemoteLoggerInterface` is looked up from the RMI registry.

4.3.2 Logged Data

When the developer debugs a program, events of the reactive language have to be logged. As written in the previous section, the `REScalaLogger` therefore creates instances of the `ReactiveVariable` and `BreakpointInformation` classes in order to store all the relevant data and sends it to the Eclipse plugin. This section explains how this data is retrieved.

The log methods called by RESCALA get the respective `Reactive` instances as a parameter. From this instance, the according `ReactiveVariable` instance can be created. The unique ID can directly be read from the `Reactive` instance by reading out the public `id` field. The inner type and value strings can only be read from `Signal` instances, so that the `Reactive` is cast accordingly and the values are read from the `Signal` instance if possible. The short and long type names of the `Reactive` are constructed via Java's `Object.getClass()` method. The variable name is read by an external class called `SrcReader`, which is introduced in the next section. As a last important field, the `connectedWith` field of the `ReactiveVariable` has to be filled correctly. Therefore, the `Reactive` is cast to a `DepHolder` instance if possible. If this is the case, the connections can be read out through the public `dependents` field and added to the `ReactiveVariable` accordingly.

The `BreakpointInfomation` class contains three fields: the path to the source file, the name of the class and the line number. The class name as well as the line number are constructed by analysing the stack trace. From the stack trace of the current thread, all calls on classes from the current project or from the Java, Scala or RESCALA library are ignored. The first stack trace element which calls a method from a class outside of these libraries is the relevant element for us. It represents the point in the source code where the respective reactive variable is defined or used in some way. The place where the breakpoint may have to be set is exactly one line after it. The path to the respective source file is already constructed by the `SrcReader` class and can be read out from it. In the next section, this `SrcReader` is explained a bit further.

4.3.3 Retrieving the Variable Names

Variable names are indispensable for the plugin since the developer has to be able to easily see which node in the graph represents which variable in the program. Additionally, the variable names are indispensable for writing the code to query the dependency graph. Unfortunately, getting the variable names of a running program is not easy at all. Variable names are usually removed by the compiler and the Java reflection API does not provide any possibility to get them. The only proper solution seems to be to parse the source code and read the variable names directly from the code. This is what has been done in this thesis. The code is originally from Gerold Hintz who wrote it for a logging application.

The idea is that a specific method is called as soon as a time-changing value is defined. Then, the stack trace will be analysed in order to get the information in which file and in which line this value is defined. The source file is then read and in the respective line, the variable name is extracted via a regular expression. For performance reasons, this is only done once per source file and the results are cached accordingly. In order to make that work, the singleton pattern is used, which is natively supported by Scala via the `object` keyword. Through Scala's comfortable and compact language constructs, the class which implements this idea is only around 80 lines long. That is enough to read the respective variable names.

4.4 Graphical User Interface

The graphical user interface of the plugin is an Eclipse view which can be opened in a new tab during the debugging session. The GUI is basically divided into two main parts as shown in Figure 4.3. It can be activated by right-clicking a variable in the debug view and selecting the action "Open Reactive Tree". Then, the respective view will be opened in a new tab. The first main part of the view just shows the dependency graph and provides some basic actions to do something with it and manipulate it. This is mainly the functionality which already worked before: the developer sees the dependency graph, can move nodes around and additional information to the nodes can be shown if the developer hovers with the mouse over a specific node. One can choose different layout algorithms which organise the nodes in different manners. The current visualisation can also be stored in a png-file. Zooming functionality is also provided.

These features have been extended: Nodes can now be collapsed and extended. If the developer double-clicks on a node, the status of the node is toggled – if it has previously been collapsed, it is extended and vice-versa. If a node is collapsed, all child nodes of this node are recursively hidden. This can make the dependency graph way more clear. If the developer is not interested in a specific part of the graph, this part can easily be hidden with this feature. Besides, the graph can now be moved directly with the mouse. Before, one had to use the scrollbars in order to move the graph around. Now, one can just click on some free whitespace and the graph will move proportionally to the mouse movement.

The second part of the view is the main contribution of this thesis. It provides the possibility to navigate through the dependency graph history and to query it. The slider provides a simple possibility to navigate through the history by dragging it. The dependency graph will then be instantly updated and shows the state of the graph at the respective point in time. Below the slider, the developer can enter the queries explained in Section 4.2.3. If a valid query is entered and the enter key is pressed or the "Submit Query" button is pressed, the visualisation will jump

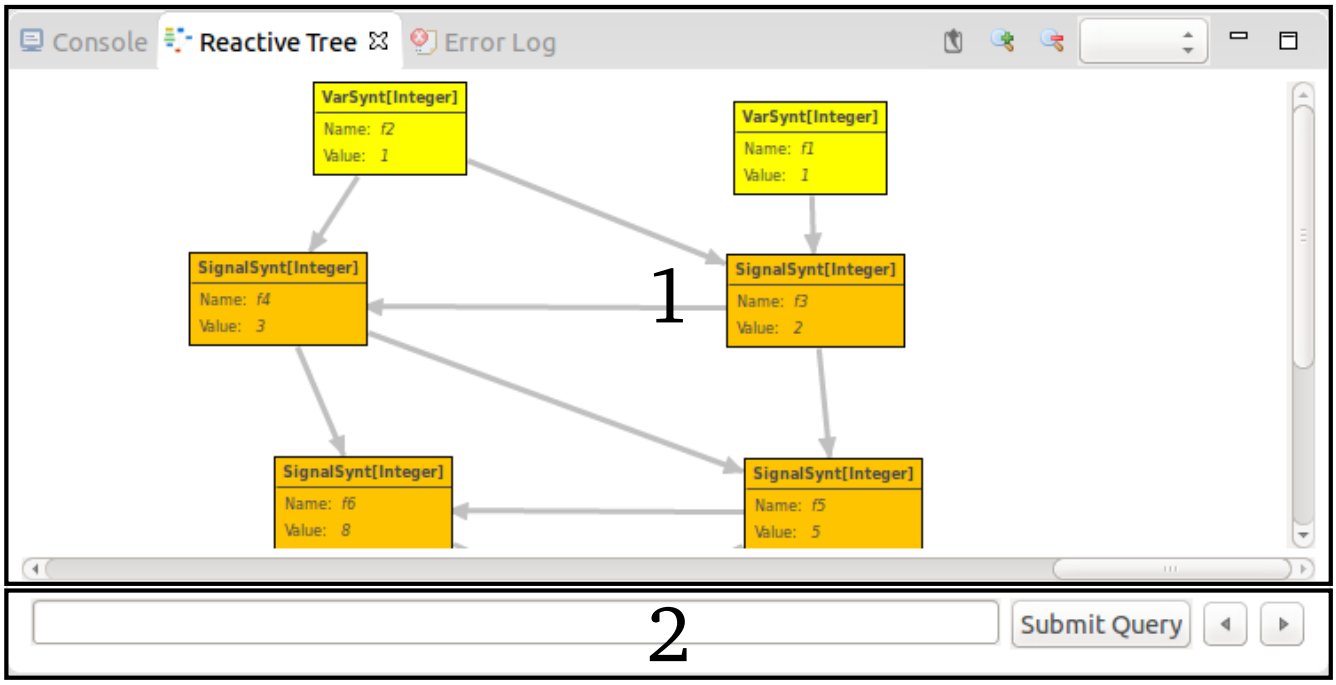


Figure 4.3.: Screenshot of the GUI

to the first point in time at which the entered query matches. The arrow keys let the developer then navigate through the different matched points in time if applicable. If no matching results are found, a respective message is shown in a pop-up.

The visualisation of the dependency graph is based on Zest [2], a visualisation toolkit from Eclipse. It is a sub-project of the graphical editing framework named GEF, which is a more general toolkit to create rich graphical editors and views. Zest makes it quite straightforward to visualise graphs in Eclipse. Since its modelling has been orientated towards the Eclipse-standard, it integrates seamlessly into the Eclipse GUI. A discussion of the reasons for choosing Zest over other graph visualisation tools can be found in the thesis of Michael Hausl [Hau14]. The containers, buttons and different actions are naturally based on Eclipse SWT [3], the open source standard widget toolkit used throughout Eclipse.

[2] `http://www.eclipse.org/gef/zest/`, last accessed 24-09-2014
[3] `http://www.eclipse.org/swt/`, last accessed 24-09-2014

5 Evaluation

In this chapter, various case studies are introduced which demonstrate the use of the plugin. They have been chosen in order to illustrate the most important features of the plugin and therefore the contributions of this thesis. They are all written in RESCALA.

For the sake of brevity and clarity, we skip the package and import declarations in the code examples in the next sections. The required imports are shown in Appendix A. The first section basically shows how the plugin already worked before – it could visualise reactive systems by showing the respective dependency graph. The second section exemplifies the first contribution of the thesis: it shows how the dependency graph history helps in understanding reactive systems. The developer can observe the evolution of the shape of the graph as well as the change propagation. The third section illustrates how bugs can easily be found by using the developed query language. The fourth section then explains how the reactive-programming-specific breakpoint feature can help to find bugs even faster under certain circumstances. The sections five and six show again how bugs can quickly be found. Once primarily with the help of the visualisation and once primarily with the help of the query language. The last section summarises and concludes the evaluation.

5.1 Visualisation of the Dependency Graph: The Fibonacci Example

The Fibonacci numbers with the calculation procedure $F_n = F_{n-1} + F_{n-2}$ and the start values $F_1 = 1, F_2 = 1$ are commonly known. Due to the high number of regular dependencies, it is a nice example to show the visualisation feature of the Eclipse plugin as depicted in Figure 5.1. It shows one node for each variable. `Var` instances are shown in yellow, `Signal` instances in orange. Currently active nodes are green. There is a directed edge between a node `x` and another node `y` if the value of `x` has a direct influence on the value of `y`. For instance, the value of `f1` as well as the value of `f2` have a direct influence on the value of `f3`, so that there is a respective edge in each case. The respective source code which defines the first ten Fibonacci numbers in RESCALA is rather terse:

```
object FibonacciExample extends App {

  rescala.ReactiveEngine.log = new REScalaLogger

  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() }
```

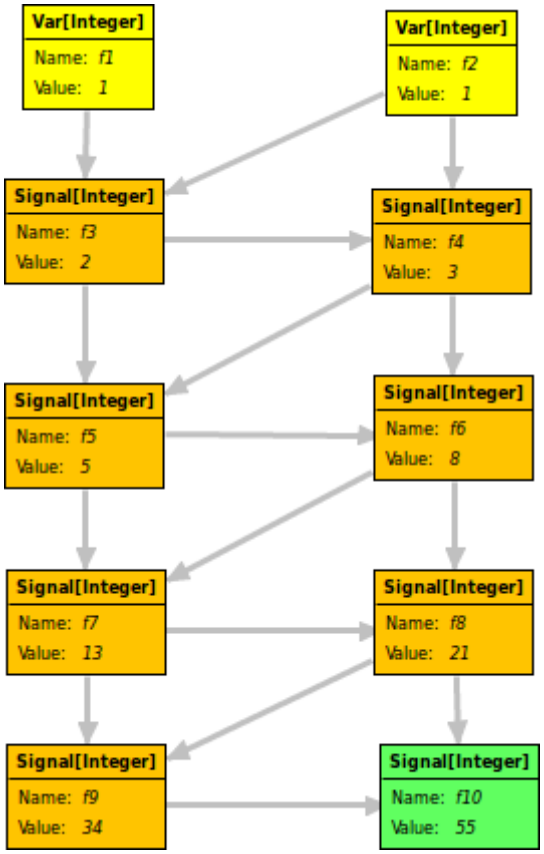


Figure 5.1.: Dependency Graph Visualisation of the Fibonacci Example

```
  val f9 = Signal { f7() + f8() }
  val f10 = Signal { f8() + f9() }
  println(f10.get)
  f1.set(2)
  println(f10.get)
  f2.set(5)
  println(f10.get)

}
```

This example illustrates how the plugin helps to get a rough overview of a reactive system. If dependent variables stretch over multiple files and classes and one does not know the system, this is a great possibility to understand the reactive system without the need to study the whole source code.

5.2 Navigating through the Dependency Graph History

One of the contributions of this thesis is the storage of the whole dependency graph history and the possibility to navigate through it. The basic navigation is done via a simple slider as shown in Section 4.4. With this navigation feature, one can observe two important aspects of the reactive system: first, one can observe the evolution of the shape of the dependency graph. One can see how nodes are created and how dependencies are established. Second, one can observe how changes propagate through the dependency graph. Both features help

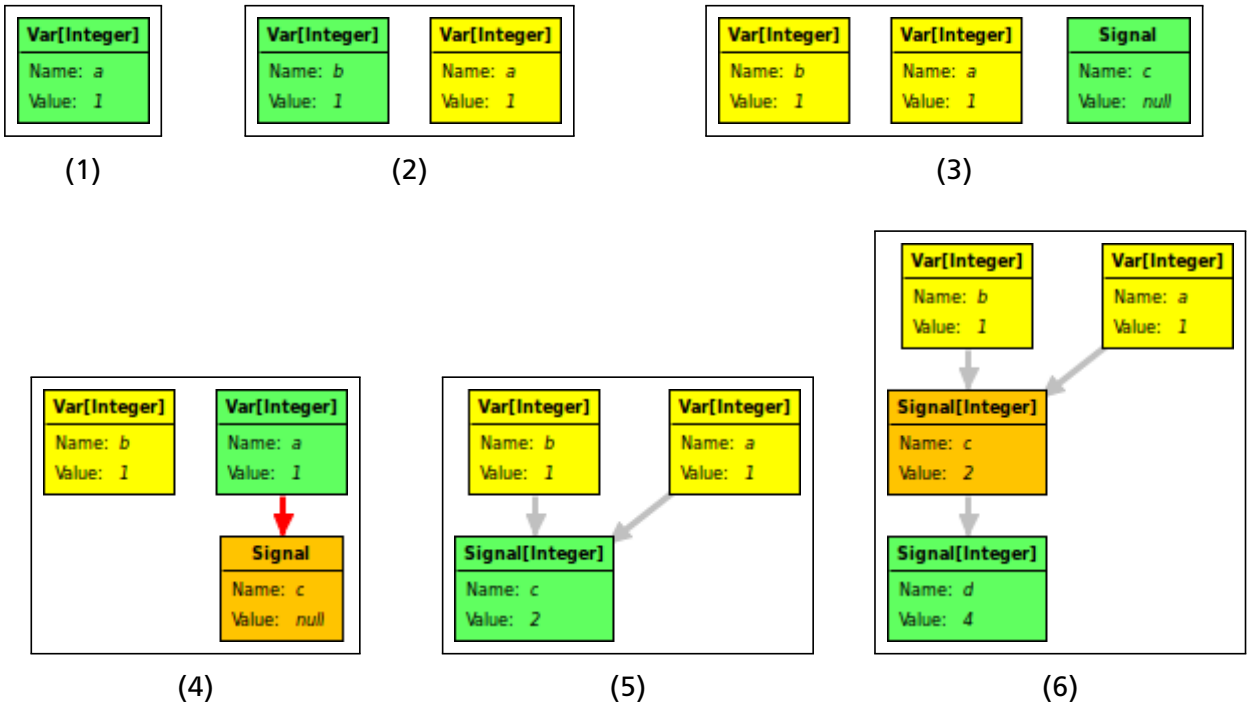


Figure 5.2.: Evolution of the Shape of the Dependency Graph

to comprehend the execution of the reactive system step-by-step. Doing this with a traditional debugger would potentially mean that it jumps around in the source code, so that one cannot comprehend the steps at all. With the help of the dependency graph visualisation, the steps are easily comprehensible and shown in a more natural format. The two features are exemplified with the example we already introduced in the previous chapters:

```
object SimpleExample extends App {

  rescala.ReactiveEngine.log = new REScalaLogger

  val a = Var(1)
  val b = Var(1)
  val c = Signal { a() + b() }
  val d = Signal { 2 * c() }
  println(d.get)
  a.set(5)
  println(d.get)
}
```

Evolution of the Shape of the Dependency Graph

The evolution of the shape of the dependency graph is visualised in Figure 5.2. In the first three steps, the three nodes a, b and c are created. Then, the dependency between a and c as well as the dependency between b and c are established in the fourth and fifth step resulting in the evaluation of c. In the last step, node d is created, connected to c and evaluated. Some steps like the start and end of node evaluations have been omitted for the sake of better clarity.

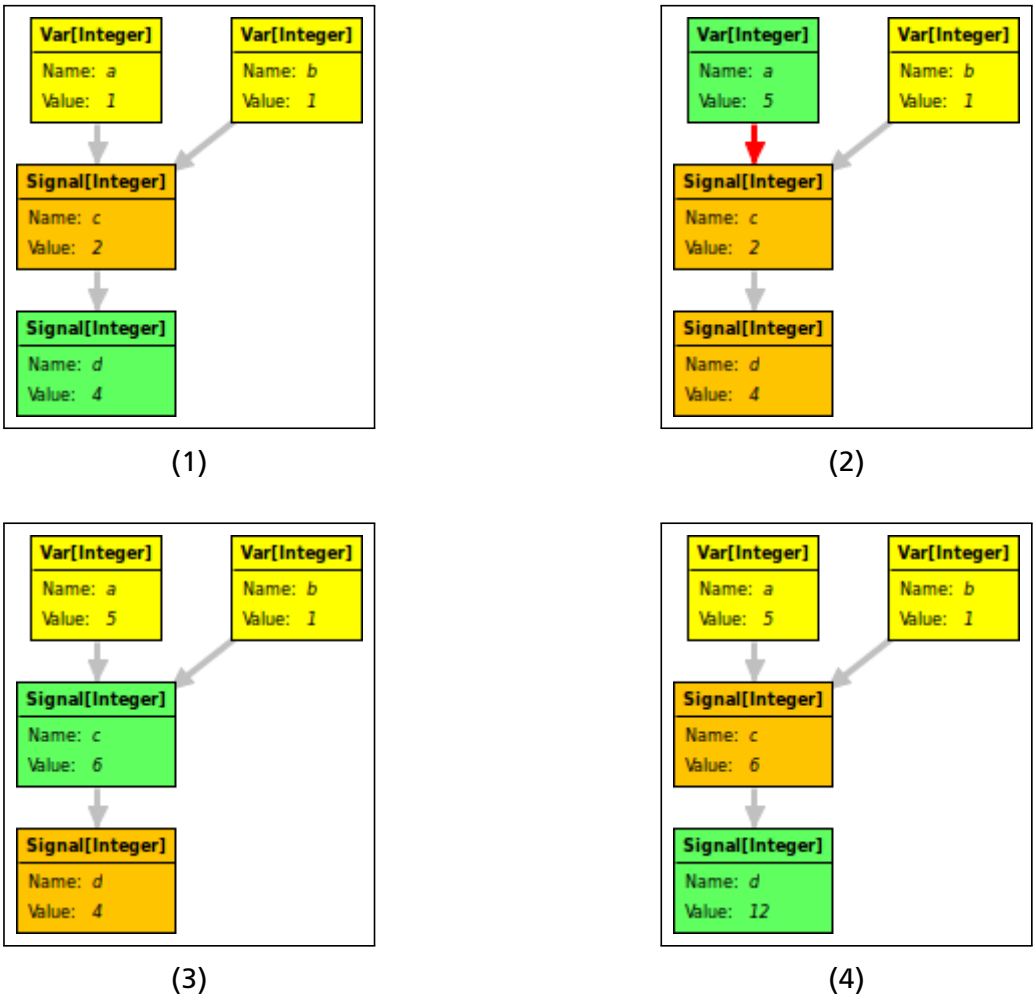


Figure 5.3.: Change Propagation in the Dependency Graph

Change Propagation in the Dependency Graph

The propagation of changes in the dependency graph is visualised in Figure 5.3. The first step is the same as the last one from the previous example. In the next three steps, the propagation of the value change can easily be observed. At first, the node a changes its value to 5. Then, this change is further propagated and also c reflects the recent change, so that its value changes to 6. In the last step, the value change arrived at node d and its value is updated to 12. Some steps have again been omitted for the sake of better clarity.

5.3 Querying the Dependency Graph History: The File Example

In the following example, we show how the developed plugin supports the developer to directly find a bug in a reactive system. First, we execute the program and then we query the dependency graph history. In the next section, we basically do the same, but define the query in advance in order to use the reactive-programming-specific breakpoint feature. The source code for this example is the following:

```
1 class FileExample {
2
3   rescala.ReactiveEngine.log = new REScalaLogger
```

```
4
5     val dir: Var[String] = Var(null)
6     val filename: Var[String] = Var(null)
7     val path = Signal {
8       if (dir() != null && filename() != null) {
9         dir.get + filename.get
10      } else {
11        null
12      }
13    }
14    def setDir(theDir: String) {
15      dir.set(theDir)
16    }
17    def setFilename(theFilename: String) {
18      if (theFilename.endsWith("txt")) {
19        filename.set(theFilename)
20      } else {
21        filename.set(null)
22      }
23    }
24    def getFile(): File = {
25      new File(path.get)
26    }
27  }
28  object FileExample extends App {
29
30    val fe = new FileExample
31    fe.setDir("/tmp/")
32    fe.setFilename("something.txt")
33    val f = fe.getFile()
34
35    fe.setFilename("something.log")
36    val f2 = fe.getFile()
37  }
```

The idea is that there is a class called `FileExample` with which files can be created. The directory and path can be defined separately and will be constructed to a full path with a `Signal` expression. The problem is that the setter for the file name only accepts file names which have a `txt` ending. If it does not have a `txt` ending, it sets the file name to null, which may finally result in a `NullPointerException` once the `File` instance is created.

When the program is executed, a `NullPointerException` is thrown. From the stack trace, the developer will see that the problem lies in line 25, because the `path` variable is `null`. Furthermore, he will see that this is the case when line 36 is executed. Hence, he sets a breakpoint before line 36 and looks at the dependency graph history. Since the problem is that the `path` variable is null, he can enter the query `evaluationYielded(path, "null")`. There are multiple results for this query, but of course the last result is the relevant result in this case. Navigating

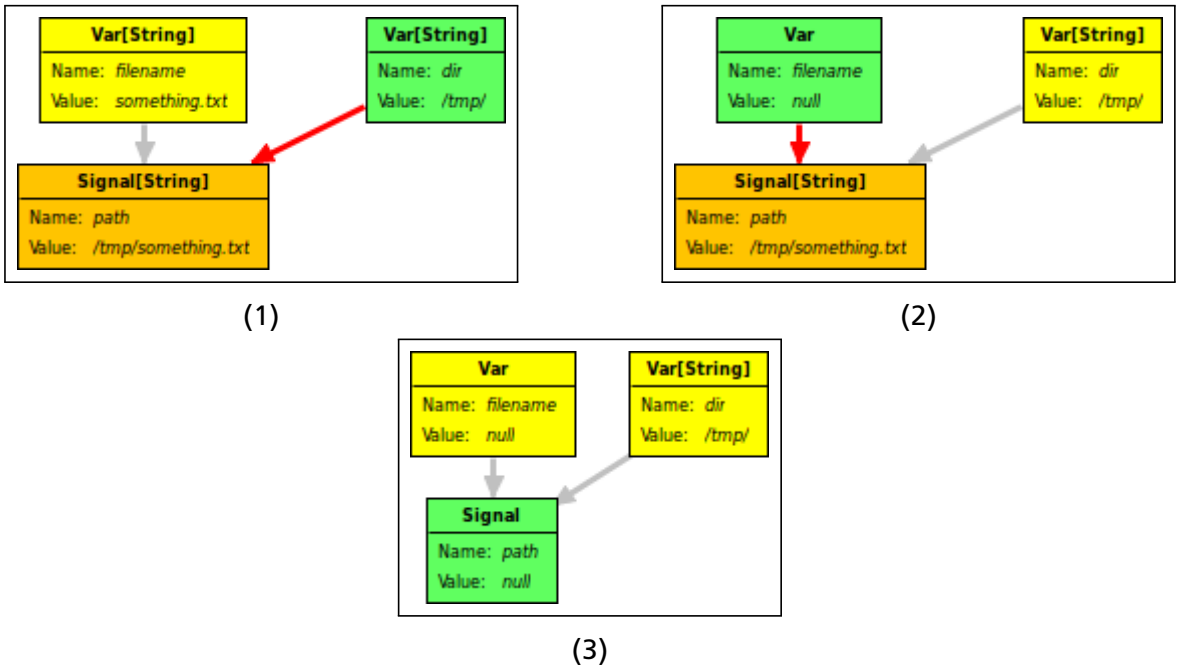


Figure 5.4.: Finding the Bug in the File Example

from this result back and forth the history, he sees the evolution depicted in Figure 5.4 and soon realises that the `filename` variable is the real problem. It is for some reason set to `null`, so that the `path` is also `null`. Having a look at the `filename` setter then directly reveals the problem: it only accepts `txt` files, but it is used with a `log` file in this case. Hence, the developer can either use the class only with `txt` files or he can fix the class to also accept other file types.

5.4 Reactive-Programming-Specific Breakpoints: The File Example Revisited

For fast-running programs, it is not really a problem to execute them first and then define and execute queries on the results. But certainly in situations where programs take minutes or hours to execute, it is extremely important to be able to have the possibility to define queries beforehand and the debugger halts the execution once a matching result occurs. This feature has been discussed before as the reactive-programming-specific breakpoints feature. We illustrate that with the file example introduced above again. Before the program is executed, the query `evaluationYielded(path, "null")` is entered. Then, the debugger is started as before. It will then halt the execution each time the evaluation of the variable `path` yields the value `null`. The developer then directly sees that this is the case when the `filename` or `dir` variable is `null`. Especially, he sees that at the problematic step, the `path` evaluates to `null`, because the `filename` is `null`. He sees exactly the same steps as shown in the previous section in Figure 5.4. Hence, he looks into the `filename` setter and sees that only `txt` files are accepted. He not only finds the bug, but got a much better overview of the whole system.

5.5 Finding Bugs with the Visualisation

The two examples introduced in this section show how the visualisation part of the system helps in finding bugs in reactive applications. The second example additionally shows that the

system works flawlessly together with REScala's conversion functions, which create `Signals` from `Events` and vice versa.

5.5.1 The Logging Example

The logging example shows how the visualisation feature of the plugin can help in finding a typical bug in a reactive system. Additionally, it shows that events as well as event handlers are also visualised in the dependency graph. The source code for this example is the following:

```
1  object LoggingEventExample extends App {
2
3    rescala.ReactiveEngine.log = new REScalaLogger
4
5    object LoggingModule1 {
6      val err = new ImperativeEvent[String]
7      def sth() = {
8        // simulate that an error occurs sometimes
9        if (Math.random() < 0.2) {
10         err("An␣error␣occurred␣in␣LoggingModule1")
11       }
12       42
13     }
14   }
15   object LoggingModule2 {
16     val err = new ImperativeEvent[String]
17     def sthElse() = {
18       // simulate that an error occurs sometimes
19       if (Math.random() < 0.2) {
20         err("An␣error␣occurred␣in␣LoggingModule2")
21       }
22       "Answer␣to␣the␣Ultimate␣Question␣of␣Life,␣the␣Universe"
23         + "and␣Everything"
24     }
25   }
26   val err = LoggingModule1.err || LoggingModule2.err
27
28   // developer erroneously attached the log handler only to the err
29   // event of the LoggingModule1, not to the combined event
30   LoggingModule1.err += { s => println("ERR:␣" + s) }
31
32   // just some sample calls
33   var i = 0
34   for (i <- 1 to 30) {
35     LoggingModule1.sth()
36     LoggingModule2.sthElse()
37   }
38 }
```

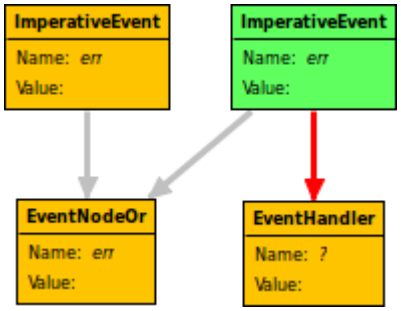


Figure 5.5.: Dependency Graph Visualisation of the Logging Example

In this example, two modules provide an event which is fired each time an error occurs. The developer wants to do something with these events, e.g. logging them by printing them to the console. Therefore, he attaches a respective event handler to the events. But he erroneously attaches this log handler only to the first module's event. This could just be a careless mistake, but there are also other possible explanations. Maybe there was only one module at first and the developer added the second one afterwards. He implemented the combined event in line 26, but did not change the event handler attachment in line 30 accordingly.

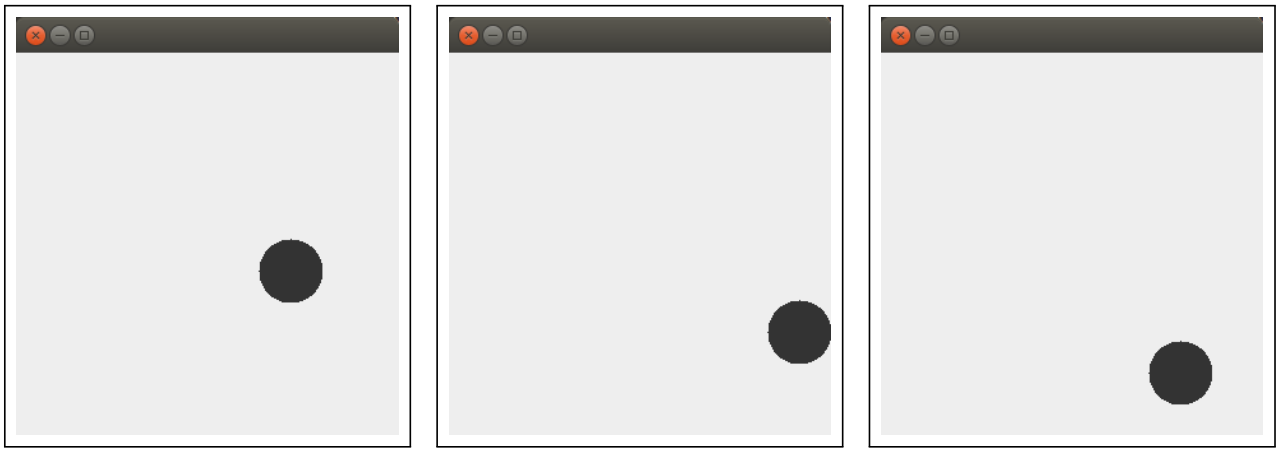
So the developer wonders why only errors of the first module are logged and there are no entries from the second module. In order to find the bug, he has a look at the dependency graph depicted in Figure 5.5. This visualisation directly reveals that the event handler is only attached to one of the error events and not to the combined event. The bug is then easily fixed by simply changing line 30 to `err += { s => println("ERR: " + s) }`.

5.5.2 The Bouncing Example

The example discussed in this section is basically taken from the "official" RESCALA examples [1]. It is a graphical application which, if implemented correctly, shows a ball bouncing off the edges of a specific area as shown in Figure 5.6. This is an example where reactive programming may be helpful in the development of reactive animations or graphical user interfaces. The source code for this example, which includes a tiny, but unpleasant bug, is the following:

```
1  object BouncingExample extends SimpleSwingApplication {
2
3    rescala.ReactiveEngine.log = new REScalaLogger
4
5    lazy val application = new BouncingExample
6    def top = application.frame
7
8    override def main(args: Array[String]) {
9      super.main(args)
10     while (true) {
11       Swing onEDTWait { application.tick() += 1 }
12       Thread sleep 20
13     }
```

[1] https://github.com/guidosalva/REScala-examples, last accessed 24-09-2014



(1) Ball Moving Downright (2) Ball Bouncing off the Edge (3) Ball Moving Bottom Left

Figure 5.6.: Screenshots of the Bouncing Example

```
14    }
15  }
16
17  class BouncingExample {
18    val Size = 50
19    val Max_X = 300
20    val Max_Y = 300
21    val initPosition = new Point(20, 10)
22    val speed = new Point(10, 8)
23    val tick = Var(0)
24
25    // Signals for x and y position
26    // entirely functionally dependent on time (ticks)
27    val x = Signal {
28      val width = Max_X - Size
29      // erroneously, tick.get instead of tick() is used here
30      val d = speed.x * tick.get + initPosition.x
31      if ((d / width) % 2 == 0) d % width else width - d % width
32    }
33    val y = Signal {
34      val width = Max_Y - Size
35      val d = speed.y * tick() + initPosition.y
36      if ((d / width) % 2 == 0) d % width else width - d % width
37    }
38
39    tick.changed += ((_: Int) => frame.repaint)
40
41    // drawing code
42    val frame = new MainFrame {
43      contents = new Panel() {
```

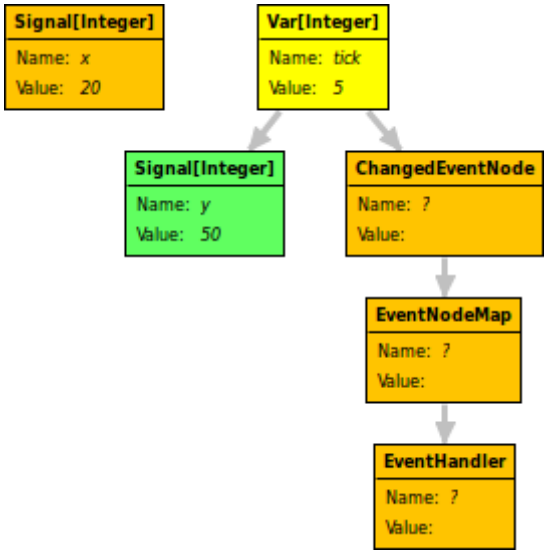


Figure 5.7.: Dependency Graph of the Faulty Bouncing Example

```
44          preferredSize = new Dimension(Max_X, Max_Y)
45          override def paintComponent(g: Graphics2D) {
46            g.fillOval(x.get, y.get, Size, Size)
47          }
48        }
49      }
50    }
```

In line 30, the developer erroneously used the method `tick.get` instead of using the correct form `tick()`. The tiny, but important difference is that `tick.get` does not create a dependency between tick and the variable `d`, so that `d` and therefore the `x` value of the position is not automatically updated each time `tick` changes. A dependency is only created when `tick()` is used.

We selected this example for two reasons. First, it shows again how the dependency graph visualisation can help to find bugs in an application. A look at the dependency graph as shown in Figure 5.7 reveals that `x` definitely lacks a dependency on the `tick` value. It immediately looks faulty that `y` has a dependency on this value and `x` does not have any dependency at all. A look into the definition of `x` and the comparison to the definition of `y` then reveals that `tick()` instead of `tick.get` should have been used in line 30. Additionally, if the developer navigates through the dependency graph, he directly sees that only the `y` value is updated each time the contents of the frame are painted again.

Second, the example shows that the system works flawlessly with RESCALA's conversion functions. In this case, the conversion function `changed` is used in line 39. It simply converts a `Signal` into an `Event` and fires each time the `Signal` changes its value. The thereby created `Event` is also shown in the dependency graph. All `Vars`, `Signals` or `Events`, no matter if created directly or created via one of the conversion functions, are shown in the dependency graph.

5.6 Finding Bugs with the Query Language: The External Library Example

This section introduces an example in which a traditional debugger fails short. Instead, developed debugger makes it easy to find the included bug. Therefore, we simulate the situation that we want to use an external reactive library which is badly documented and we do not want or do not have the time to have a look at the internals. Hence, we use the external library in a way we think it is supposed to be used as shown in the following source code:

```
1  class ExternalLibrary {
2
3    private[this] val v = Var(1)
4    private[this] val s = Signal { v() + 2 }
5
6    def getSVal(): Int = s.get
7
8    def setV(newVarVal: Int) {
9      v.set(newVarVal)
10   }
11   // other stuff ...
12 }
13 object ExternalLibraryExample extends App {
14
15   rescala.ReactiveEngine.log = new REScalaLogger
16
17   // assuming s and v are unrelated
18   val t = new ExternalLibrary
19   println(t.getSVal()) // prints "3"
20   t.setV(20)
21   println(t.getSVal()) // prints "22"
22 }
```

The library provides, besides other things, a getter for a variable `s`, which is a `Signal` and a setter for a variable `v`, which is a `Var`. The developer thinks that these variables are completely unrelated and hence uses them accordingly. But then he encounters the problem that the returned value of `s` changes unexpectedly. With a traditional debugger, one cannot really do something – since reactive programming abstractions are not supported, one cannot comprehend why the value of `s` changes. But with the developed debugger, the developer can set a breakpoint in line 21 and have a look at the dependency graph. He would directly see that there is a dependency between `v` and `s`, so that it is clear that `v` has an influence on the value of `s`. But more specifically, he could jump to the relevant point in time where `s` evaluates to `22` by executing the query `evaluationYielded(s, "22")` on the dependency graph history. Navigating back and forth the history, the developer would see the evolution as shown in Figure 5.8 and would then quickly realise that `s` changed because of the value change of `v`. He then knows why the problem exactly occurs and can further analyse the issue by e.g. checking how exactly the two values depend on each other.

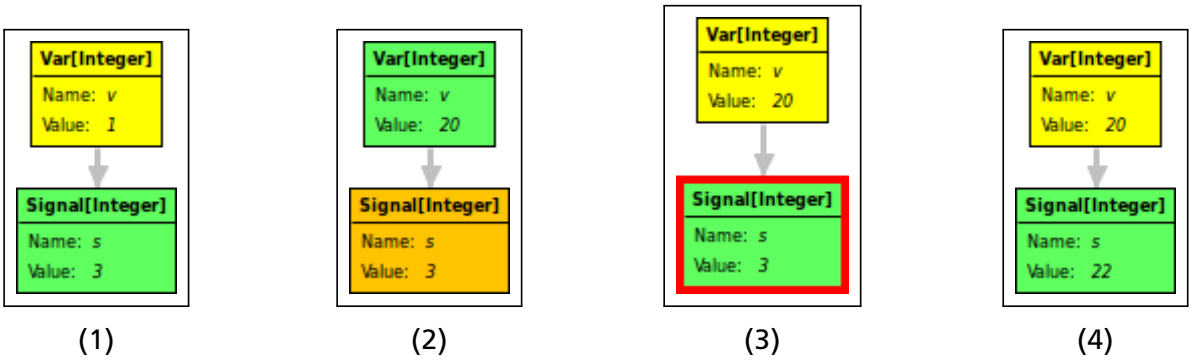


Figure 5.8.: Finding the Bug in the External Library Example

5.7 Summary of the Evaluation

The preceding evaluation illustrated with the help of various examples what the contributions of this thesis are. We showed how the developed debugger can be used in order to find bugs in reactive applications more easily and more quickly. Additionally, we showed how the visualisation can help in getting a good overview of a reactive system and understand its internals at a glance.

Of course we tried to keep the examples manageable, so that they can be fully included, read and understood. That is why they suffer from the problem that they are somewhat artificial and the introduced bugs and problems may seem obvious. But these problems may not be obvious at all when they occur in bigger systems that are not so easy to overview. The reader should imagine that these bugs are usually "hidden" in big systems with countless lines of code, which one usually did not write oneself. Hence, an absolutely worthwhile direction for future work is to apply the debugger in the everyday developer life and thoroughly test it on bigger applications. This and also other directions for future work are outlined in the next chapter.

6 Conclusion and Future Work

In this chapter, the thesis is concluded by shortly summarising the main contributions. Additionally, various ideas for future work are introduced, so that the development can proceed without any break.

6.1 Final Remarks

The developed system is based on an already existing Eclipse plugin called REClipse, which already provides the functionality to visualise dependency graphs. The contribution of this thesis is to extend this functionality with the history of the dependency graph and thus develop an advanced debugger for reactive programming.

With the system developed in the thesis it is possible to visualise the whole history of the dependency graph and simply navigate through it with the help of a slider. Additionally, advanced methods to navigate the history have been developed. A domain-specific query language provides the possibility to formulate queries on the history and directly jump to relevant points in time. Moreover, reactive-programming-specific breakpoints have been developed. They allow to halt the program execution as soon as a relevant event occurs. These events are again formulated by the developed query language.

As shown in the evaluation, advanced debugging mechanisms for reactive programming have the potential to ease the development of reactive systems. Not only can reactive systems be overseen more easily, but bugs can be found easier as well. This still needs to be verified in case studies, which is one possibility for future work. Many ideas for further improvements are introduced in the next and also last section.

6.2 Outlook

Since the development of debugging tools for reactive programming is still in its infancy and this thesis is just one of the first approaches, there are various options for future work:

Plugin Implementation

Regarding the implementation of the plugin, a good idea would be to tackle the scalability issues. Since the plugin heavily depends on synchronous communication via Java RMI, it does not scale very well for big applications with many and highly involved time-changing values. One could replace Java RMI by other concepts, such as CORBA [1] or sockets for performance reasons and in order to make the plugin more language-independent. Furthermore, the query language could be extended to support more commands. One obvious idea would be the possibility to combine already existing commands with logical operators in order to create more advanced commands. For instance, this would give the developer the possibility to define a query which fires each time a specific variable is activated – no matter if it is created, evaluated, or whatsoever.

[1] `http://www.corba.org/`, last accessed 24-09-2014

Additional Features

The biggest area for possible improvements are new features. Here is a non-exhaustive list of ideas:

- A scoping feature could reduce the complexity of the dependency graph history, so that it is easier for the developer to figure it out and to get a good overview. It would provide the possibility to exclude certain classes or packages to generate any events or the possibility to explicitly define which classes or packages should generate events exclusively. Another scoping idea would be to exclude certain event types beforehand so that they are not logged at all. For instance, it may be advantageous to be able to ignore all events fired when nodes are created or attached. This would also improve the performance of the debugging execution.
- It may be helpful to have some form of bookmarking feature. If the developer is interested in one specific point in time of the dependency graph history, it may be helpful if it can be bookmarked for easier future access.
- A great help would be to improve the debugger in order to make it a so-called "edit-and-continue debugger". This would give the developer the possibility to change values in the dependency graph on-the-fly and continue the execution of the program afterwards. Adding or deleting certain nodes on-the-fly could also be interesting.
- The plugin could also provide great assistance with performance analysis. It would be very helpful to visualise the number of times a node is evaluated or the amount of time these evaluations take. This could be visualised with colours in different brightnesses. The mean time needed for evaluating a node could be shown in the node or via an overlay. This would easily and comfortably provide relevant performance information.
- It may be helpful to have the possibility to navigate directly from the dependency graph to the code. A double click on a node could e.g. directly open the respective place in the code where the respective variable is defined.

Case Studies and Evaluation

Another area for possible improvements are case studies and evaluation. It would be good to inspect how the plugin helps developers to understand reactive systems and to find bugs in them. How is the plugin exactly used? Does it really help the developers to better understand reactive systems? Does it really help to find bugs in them? With the help of user surveys, one could also discover further ideas for future work. Developers who work with the plugin on a daily basis will surely have various ideas for new features. Additionally, user surveys could help to prioritise the features mentioned in the last paragraph, so that the features most important to the developers are developed first.

A Imports for the Code Examples

For the sake of brevity and clarity, import and package declarations have been omitted in the source code examples. All examples from the evaluation chapter reside in the `de.tu_darmstadt.stg.reclipse.examples` package. The following import declarations are valid for all examples:

```
1 import de.tu_darmstadt.stg.reclipse.rescala.REScalaLogger
2 import makro.SignalMacro.{ SignalM => Signal }
3 import rescala.Var
```

The first line includes the language-specific logger from the imported `REClipse_REScala` project, so that it can be plugged into the logging facility of RESCALA. The second line enables the usage of `Signals` and the last line enables the usage of `Vars`.

Some examples require some more imports, such as the `BouncingExample` which requires some Swing and AWT imports or the `LoggingEventExample` which also imports the required event `rescala.events.ImperativeEvent`. Since these imports are rather obvious, they have been completely omitted.

Bibliography

[Aut13] Various Authors. The reactive manifesto, September 2013. `http://www.reactivemanifesto.org`, last accessed 24-09-2014.

[Bal69] R. M. Balzer. Exdams: Extendable debugging and monitoring system. In *Proceedings of the May 14-16, 1969, Spring Joint Computer Conference*, AFIPS '69 (Spring), pages 567–580, New York, NY, USA, 1969. ACM.

[BCC+13] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. A survey on reactive programming. *ACM Comput. Surv.*, 45(4):52:1–52:34, August 2013.

[CK06] Gregory H. Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In Peter Sestoft, editor, *Programming Languages and Systems*, volume 3924 of *Lecture Notes in Computer Science*, pages 294–308. Springer Berlin Heidelberg, 2006.

[EH97] Conal Elliott and Paul 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.

[EJ09] Patrick Eugster and K.R. Jayaram. Eventjava: An extension of java for event correlation. In Sophia Drossopoulou, editor, *ECOOP 2009 – Object-Oriented Programming*, volume 5653 of *Lecture Notes in Computer Science*, pages 570–594. Springer Berlin Heidelberg, 2009.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.

[GSM+11] Vaidas Gasiunas, Lucas Satabin, Mira Mezini, Angel Núñez, and Jacques Noyé. EScala: Modular event-driven object interactions in scala. In *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*, pages 227–240, New York, NY, USA, 2011. ACM.

[Hau14] Michael Hausl. Eclipse plugin for reactive programming. Diploma thesis, TU Darmstadt, March 2014.

[Ins14] National Instruments. Debugging tools in NI LabVIEW, August 2014. `http://www.ni.com/getting-started/labview-basics/debug`.

[ISO06] ISO/IEC. *Information technology – Programming languages – C#*. ISO/IEC, September 2006.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97 — Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin Heidelberg, 1997.

[Lew03] Bil Lewis. Debugging backwards in time. In *Proceedings of the Fifth International Workshop on Automated Debugging*, AADEBUG 2003, 2003.

[Luc08] David Luckham. The power of events: An introduction to complex event processing in distributed enterprise systems. In Nick Bassiliades, Guido Governatori, and Adrian Paschke, editors, *Rule Representation, Interchange and Reasoning on the Web*, volume 5321 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2008.

[MGB+09] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: A programming language for ajax applications. In *Proceedings 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.

[MO12] Ingo Maier and Martin Odersky. Deprecating the Observer Pattern with Scala.react. Technical report, EPFL, 2012.

[MRO10] Ingo Maier, Tiark Rompf, and Martin Odersky. Deprecating the Observer Pattern. Technical report, EPFL, 2010.

[Par13] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

[PT09] Guillaume Pothier and Éric Tanter. Back to the future: Omniscient debugging. *Software, IEEE*, 26(6):78–85, November 2009.

[PTP07] Guillaume Pothier, Éric Tanter, and José Piquer. Scalable omniscient debugging. *SIGPLAN Not.*, 42(10):535–552, October 2007.

[RL08] Hridesh Rajan and Gary T. Leavens. Ptolemy: A language with quantified, typed events. In Jan Vitek, editor, *ECOOP 2008 – Object-Oriented Programming*, volume 5142 of *Lecture Notes in Computer Science*, pages 155–179. Springer Berlin Heidelberg, 2008.

[SAPM14] Guido Salvaneschi, Sven Amann, Sebastian Proksch, and Mira Mezini. An empirical study on program comprehension with reactive programming. FSE'14, 2014. To appear.

[SHM14] Guido Salvaneschi, Gerold Hintz, and Mira 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, April 2014. ACM.

[SM14] Guido Salvaneschi and Mira Mezini. Towards reactive programming for object-oriented applications. In Shigeru Chiba, Éric Tanter, Eric Bodden, Shahar Maoz,

and Jörg Kienzle, editors, *Transactions on Aspect-Oriented Software Development XI*, volume 8400 of *Lecture Notes in Computer Science*, pages 227–261. Springer Berlin Heidelberg, 2014.

List of Figures

List of Tables