

A Debugging Tool for Javascript Reactive Libraries

Master-Thesis von Waqas Abbas

Tag der Einreichung: 05. Juni 2017

1. Gutachten: Prof. Dr. Guido Salvaneschi
2. Gutachten: M.Sc. Pascal Weisenburger



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Reactive Programming Technology

A Debugging Tool for Javascript Reactive Libraries

Vorgelegte Master-Thesis von Waqas Abbas

1. Gutachten: Prof. Dr. Guido Salvaneschi
2. Gutachten: M.Sc. Pascal Weisenburger

Tag der Einreichung: 05. Juni 2017

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Masterarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in dieser oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Die schriftliche Fassung stimmt mit der elektronischen Fassung überein.

Darmstadt, den 05. Juni 2017

Waqas Abbas



Abstract

Over the last few years, reactive programming has proved to be useful in several domains such as GUIs, animations, web applications, robotics and sensor networks. Reactive programming has been preferred over traditional techniques to implement reactive systems because it makes applications more comprehensible, composable, easy to understand and less error-prone. Unfortunately, it is hard to debug reactive code because there is no proper tool-support available. Traditional debugging tools are engineered for procedural programming and are not suited for debugging reactive applications. The goal of this thesis is to improve the debugging process of reactive systems in the web domain.

In this thesis, we present a special debugger as an extension to Google Chrome DevTools for debugging reactive applications based on JavaScript reactive libraries (**RxJS** and **Bacon.js**), called **Chrome Reactive Inspector (CRI)**. RxJs and Bacon.js, Javascript libraries for asynchronous data streams, bring together the best from functional and reactive programming concepts to the web domain. The debugger is an implementation of **RP Debugging** in the web domain where the dependency graph is used to model reactive applications. The system presented in this thesis makes it possible to debug a specific point back-in-time, as the developer can navigate history back and forth freely. A domain-specific query language provides direct access to the graph history so that one can jump to points of interest. This query language also enables programmers to set reactive-programming-specific breakpoints. These are breakpoints which interrupt execution when a query provided by the developer matches a given criteria.

We present a preliminary evaluation based on various sample applications taken from the internet. The evaluation shows that the debugger helps in the understanding of reactive systems. It also enables bugs to be found quickly by outperforming traditional debuggers as it directly supports abstractions of reactive libraries.



Contents

1. Introduction	3
1.1. Background	3
1.2. Motivation	4
1.3. Our Contribution	4
1.4. Outline	5
2. State of the Art	7
2.1. Implementation of Reactive Systems	7
2.1.1. Observer Design Pattern	7
2.1.2. Aspect-Oriented Programming	8
2.1.3. Callbacks	8
2.1.4. Promises	8
2.1.5. Iterator Vs Observer Pattern	9
2.2. Reactive Programming	9
2.2.1. Reactive Programming with JavaScript	10
2.2.2. ReactiveX and RxJS	11
2.2.3. Important Concepts of RxJS	11
2.2.4. Bacon.js	14
2.3. Debugging and Tools Support	16
2.3.1. Debugging JavaScript	16
2.3.2. Debugging in Reactive Programming	17
2.3.3. Debugging Reactive Extensions of JS	17
2.4. Jalangi	18
2.5. Chrome Developer Tools	19
2.5.1. Extending Chrome DevTools	20
2.6. Related Work	21
2.6.1. Beyond Traditional Debugging	21
2.6.2. Related Tools	22
3. System Design	23
3.1. System Requirements	23
3.2. System Architecture	24
3.2.1. Event Sniffing - Options	26

3.3. System Features and Design Choices	27
3.3.1. Dependency Graph Visualization	28
3.3.2. History of Dependency Graph	29
3.3.3. Reactive Breakpoints	29
3.3.4. Extension to Chrome DevTools	29
3.3.5. Scoping and Snapshot	29
3.3.6. Reactive Libraries Support	30
3.3.7. Data Storage and Management	30
3.3.8. Mapping Stream to JS variable	30
3.3.9. Chrome Extension Scripts	31
3.3.10. Scripts Injection from Extension	31
4. Implementation	33
4.1. CRI - Implementation Alternatives	33
4.1.1. Alternative - 1	33
4.1.2. Alternative - 2	35
4.2. Significant Data Structures	36
4.3. Communication between Extension Components	38
4.4. The Query Language	40
4.5. Graphical User Interface	40
5. Evaluation	43
5.1. Feasibility of the Technical Approach	43
5.1.1. Evaluated Web Applications	43
5.1.2. Evaluation Criteria	44
5.1.3. Evaluation Results	45
5.2. Case Studies	46
5.2.1. Understanding Operators	46
5.2.2. Understanding Reactive Systems with Dependency Graph History	49
5.2.3. Querying the Dependency Graph History	53
5.2.4. Reactive-Programming-Specific Breakpoints	55
5.3. More Advance Applications	59
5.3.1. RxJS - Canvas Painting Board	59
5.3.2. RxJS - Who to Follow	62
5.3.3. Baon.js - Drawing Bar Chart	65
5.3.4. Bacon.js - Live Wikipedia Updates Over Websockets	65
5.4. Summary of the Evaluation	67

6. Summary	71
6.1. Conclusion	71
6.2. Future Work and New Features	72
List of Figures	75
Listings	77
Bibliography	79
Appendices	85
A. APPENDIX	87
A.1. App#1 - Operators and Events	87
A.2. App#2 - Father-Son Wallet War	87
A.3. App#3 - Up-Down Counter	87
A.4. App#4 - Form Validation	88
A.5. App#5 - Movie Search	88
A.6. App#6 - Bar Chart	88
A.7. App#7 - WebSocket - Wikipedia	88
A.8. App#8 - Smart Counter	89
A.9. App#9 - State Sotrage	89
A.10.App#10 - Letter Counter	89
A.11.App#11 - Father-Son Wallet War	90
A.12.App#12 - Movie Search	90
A.13.App#13 - Follow The Mouse	90
A.14.App#14 - Drag and Drop	90
A.15.App#15 - Canvas Painting	91
A.16.App#16 - Twitter Follow Box	91
A.17.App#17 - REST API Call	91
A.18.App#18 - Spotify Artist Search	92
A.19.App#19 - Image Sampler	92
A.20.App#20 - Mullti Select Cards	92
A.21.App#21 - True - False Logger	92
A.22.App#22 - Drawing App	93



1 Introduction

In the last two decades, there has been increasing interest in web applications as compared to traditional desktop applications. JavaScript has been predominately used to make web applications dynamic [61]. Additionally, today's software applications have become highly interactive with their environment. However, reactive systems can be developed and maintained more efficiently with **reactive programming (RP)** as compared to imperative programming. Indeed reactive languages like FrTime [7] and Flapjax [37] have gotten high recognition. Recently, JavaScript has emerged as a general purpose programming language and has become popular for server-side programming with platforms such as Node.js. To benefit from RP techniques, the JavaScript community produced libraries like RxJs[46] and Bacon.js [4], these have made it possible to do RP with JavaScript. Unfortunately, most reactive languages and libraries need proper tool support for better understanding and debugging process. Systems based on RP contain very abstract code that make it difficult to understand and debug.

This thesis is another step towards usable debugging support for RP in general and for JavaScript-based reactive libraries more specifically. This chapter introduces the topic with some background knowledge, which will be later explained in more detail in the second chapter. The second section addresses the motivation behind the thesis and why better debugging support for JavaScript-based reactive libraries is necessary. The third section presents the outcomes of this thesis and the last section outlines the structure of this thesis.

1.1 Background

Transformational systems have no interaction with their environment, whereas reactive systems react to environmental changes and to the current input by updating their internal state. A reactive system maintains an ongoing interaction with its environment, activated by receiving input events from the environment and producing output events in response [41]. Web applications reacting to user input originating from a mouse or keyboard by enabling or disabling the input component is a classical example of the reactive system. Most of the today's software applications are reactive systems because they continuously interact with the systems in its surroundings and react to any change in real time. According to the document “reactive manifesto”[2], responsiveness, scalability, resilience and being event-driven are design properties that all reactive systems should have. Before the birth of RP paradigm, reactive systems were implemented with observer pattern, event-driven programming (EDP) or aspect-oriented programming (AOP). The RP paradigm aims to simplify the development of reactive systems. Concepts of RP are implemented either as an extension to the already existing language or in the

form of a library. Examples of reactive languages are FrTime [7], Flapjax[37], Scala.React [34] and RESCALA [51]. JavaScript libraries like RxJS[46] and Bacon.js [4] introduced RP to the web domain.

In essence, the RP paradigm provides language-level abstractions for modelling and supports time-changing values. Designs based on RP are declarative where developers define dependencies among different components. Propagation of changes among dependent components is implicit. RP provides operators to compose event streams. With RP, computation is driven by events flowing into the system. Designs based on RP requires less boiler code and is less error-prone because propagation of updates and dependencies are managed by the language itself. Further advantages of RP over other paradigms have often been discussed [7, 37, 34, 5]. Empirical evaluation proves that the program comprehensibility is improved by RP [50].

1.2 Motivation

Program comprehension is the most important and time-consuming process in software development. Developers spend 50% to 80% of their time trying to understand code [36, 9]. Previous research has confirmed that developer tools can considerably improve program comprehension [60] and thus reduce overall development time. Debugging is an important aspect of programming and debugging tools can help developers find errors in code as well as assist in comprehension. Reactive programming is a relatively new programming paradigm and is different from traditional imperative programming. Unfortunately, there is no debugging tool support currently available. Traditional debuggers are not useful for debugging reactive applications because the code based on RP is very abstract and changes are propagated to dependent components by the language itself. This work is inspired by **RP Debugging** [52], a methodology for effectively debugging reactive programs proposed by Guido Salvaneschi and Mira Mezini at the Technical University of Darmstadt, Germany. The proposed debugging model covers some advanced debugging techniques like omniscient debugging, which allows one to look at the program state at any arbitrary moment. Their model uses dependency graphs as a primary runtime representation of the program during the debugging process, where each variable is a node and each dependency is represented by a directed edge. We found **RP Debugging** very useful and therefore applied their proposed debugging model to the web domain resulting in a debugging tool for web applications based on JavaScript reactive libraries.

1.3 Our Contribution

In the last two sections, we introduced reactive systems, their implementation in web domain via reactive libraries and the possibility to model them with dependency graphs and omniscient debugging. We also found that tool support for better understanding and debugging purpose

is missing. The goal of this thesis is to develop an interactive debugger which can visualize the dependency graphs of reactive systems based on JavaScript reactive libraries and provide advanced debugging features like back in time debugging and reactive breakpoints. This thesis focuses on RxJS and bacon.js. Both are reactive libraries helping RP with JavaScript.

As a summary, this thesis makes the following contributions:

- Model reactive applications into dependency graphs.
- Save history of dependency graph during the execution of a program.
- Ability to download dependency graph at any point in time.
- Search for nodes in dependency graph by node name.
- Navigate through the history of dependency graph with the help of a slider.
- Navigate history with the help of a query language.
- Reactive breakpoints to provide the facility to halt a program execution directly when a specific event occurs during execution.
- Evaluation of implemented system with sample applications from the internet.

1.4 Outline

After providing a short introduction to the topic with the motivation and goals behind this work, the thesis is further structured as follows: Chapter 2 includes more information about the implementation of reactive systems without RP. The basic concept of RP in general and RP with JavaScript is discussed in more detail. The reason why traditional debuggers are not suitable for debugging reactive code is also explained in the second chapter. Next, we present a JavaScript analysis framework called Jalangi. The process of extending Chrome DevTools along with related work is presented at the end of the second chapter. The third chapter illustrates the system on a high-level and system requirements are presented in more detail. This chapter also includes the design options with the decisions that have been made. Chapter 4 explains the system implementation in detail. Two alternative implementations are shown. The GUI of the system and its usage is also presented. Chapter 5 evaluates the developed system with the help of already existing reactive applications from the internet and illustrates all features of the implemented system with sample applications. Chapter 6 finally concludes this thesis and presents possible ideas for future work.



2 State of the Art

This chapter presents the state of the art of the topics related to this thesis. In the first section, we introduce reactive systems and different ways to implement them. The second section explains RP and the two libraries from the web domain allowing RP with JavaScript. Traditional debugging tools and techniques along with their limitations with RP are described in the third section. Afterwards, an analysis framework for JavaScript applications called Jalangi is presented in the fourth section. The fifth section presents basic features of Chrome developer tools with a brief guide on writing an extension to add functionality to Chrome DevTools. The chapter finishes with related work from two different aspects.

2.1 Implementation of Reactive Systems

The state is an essential and important part of reactive systems because the state of the system changes as a result of processing events. According to the **reactive manifesto** [2], reactive systems respond in a timely manner even during a failure or variable workload and ensure loose coupling between its components through asynchronous message passing. Reactive systems are those systems that can react to events (event-driven) and to its users (responsive). We can think of web applications as reactive systems, where events happen to DOM [11] elements, and result in a change in the state, which can be visible or hidden to the end user [67]. Continuously updating the state of systems and the propagation of every single change into systems make the implementation of reactive systems difficult. The manual approach would be to trigger update code after every single change to the state of the system, this is very complex and error-prone. With the manual approach, there will be no separation of concerns and a lot of boiler code would be required to define simple constraints.

2.1.1 Observer Design Pattern

The observer design pattern has been used for a long time to implement interactive applications. This pattern does not eliminate the problems of the manual approach to the propagation of changes, but it does modularize the process of propagation of any change to its dependents. The observer pattern defines dependency between objects so that if one object changes its state, all the other dependent objects are notified automatically [20]. With the observer design pattern, the amount of code for handling events is quite large and scattered, which leads to a complex and error-prone code [37]. A report “**Deprecating the Observer Pattern**” [34], proposed a way to minimize the amount of code in an application to handle events by abstracting the logic of event handling as part of a programming language.

2.1.2 Aspect-Oriented Programming

Aspect Oriented Programming (AOP) is a technique for expanding the behaviour of objects, methods, and functions without modifying them. AOP allows us to define points in the code at which some update code should be executed. These points are termed as **pointcuts** and the update code that executes on matching pointcuts is called **advice**. Pointcuts are based on events like **call of a method** or **begin/end of a method execution** etc.. So when implementing reactive systems, the code responsible for updating the state of the systems can be implemented with AOP. Using AOP, separation of concerns can be achieved with less boiler code as compared to the manual approach. Like many open source JavaScript libraries, dojo[19], jQuery AOP plugin[23] and AspectJS[33] helps one to do AOP with JavaScript. However AOP is not the best choice to implement reactive systems because dependencies still have to be managed manually.

2.1.3 Callbacks

Before the RP paradigm, event handling in interactive applications was done with the help of asynchronous callbacks known as event handlers. Instead of blocking program execution while waiting for the result of an asynchronous computation, callbacks are used to receive the results of asynchronous tasks by registering a method that executes as soon as the result of computation is ready. Nesting callbacks to handle multiple asynchronous tasks in a program mostly leads to a well known problem called “**callback hell**” [31] also known as asynchronous spaghetti [38]. Callback hell basically is a state of the program that tries to benefit from callbacks but gets too complex to understand, maintain and give reason about that code. Another drawback of using callbacks is that it needs a shared mutable state and event listeners are hard to compose.

2.1.4 Promises

In JavaScript, the term promise is used for a proxy object that represents a future value that is not available initially and yet to be computed in the future [31]. In other programming languages, a promise may be called future, delay or deferred. Promises are already implemented by some open source libraries with minor differences in syntax include Q, jquery, deferred.js, vow. In JavaScript, promises became standard with the release of ECMAScript version ES6 in 2015 [30]. Replacing nested callbacks with promises gives more structure to the code, which in turn becomes easier to handle and understand. Because promises are first class objects, they are easily composable [31].

2.1.5 Iterator Vs Observer Pattern

The iterator pattern enables the consumer to pull data progressively from any data structure, one item at a time. However, in the case of the observer pattern, the producer sends/pushes data to the consumer when data is ready. The consumer gives a callback to the data producer and the producer pushes the data to the consumer using that callback. In the iterator pattern, the consumer decides when to pull data from the producer, while in the observer pattern, its producer decides when the consumer is going to receive data. Both patterns are explained in detail in the book “Design Patterns: Elements of Reusable Object-Oriented Software by Gang of four” [20]. The observer pattern is well-suited for UI events while the iterator pattern is better for traversing different types of collections in a consistent way.

2.2 Reactive Programming

In recent years, advancements in technology and high expectations from application users has changed software applications dramatically and resulted in more interactive applications. Handling large amounts of data, quick response times, not having any downtime and being able to run the application on various platforms are some of the current applications requirements [2]. Applications need to be more responsive to the events happening outside and within an application environment like never before. Users tend to regularly use responsive systems while non-responsive systems rapidly lose their users. To fulfil the above mentioned requirements, sequential and imperative programming techniques are not enough because of unpredictable behaviour of events and their effects. Before the RP model, any changes in state, its effects and order of performed actions were managed by programmers manually. This was very complex and likely to fail or cause errors in results [12]. Reactive programming came into existence as a programming model that assists in developing event-driven and interactive applications. RP provides abstractions for representing values that change over time, event handling and state management. Time-changing values, data flow management, propagation of change and tracking of dependencies are key concerns of this paradigm [35]. The programmer specifies constraints among the variables and then all the required operations to fulfill those constraints are performed by RP runtime [49]. The RP model is similar to the spreadsheet model in the sense of handling computation dependencies automatically, like changing the value of one cell can effect other cells automatically in a spreadsheet [5]. In RP if some variable is defined as a time changing value, then it means every time there will be a change in the value of that variable, reactive language or library will automatically propagate that change to all parts of the program that depends on this particular variable. For example, if we have expression ***var a = b + c;*** traditional programming languages will treat this expression as just an assignment of the sum of values of ***b*** and ***c***, while in RP this is treated as a constraint. So, whenever the

value of ***b*** or ***c*** get changed, the value of ***a*** will be updated automatically. By using RP, the composition and comprehension of the software improves as compared to the traditional observer pattern [37, 5, 34]. Design benefits of RP could be verified with preliminary empirical results of REScala and more recently, the claimed advantage of RP about better comprehension also has been empirically evaluated in detail [50]. This programming model has more emphasis on scalability and responsiveness, so we can say that RP is a programming model that help us to build a scalable architecture that is resilient and quick to react to any change. An event-driven programming model has more focus on handling single events while RP focuses on data flows and propagating changes all over. Thus far, most reactive languages have been implemented on top of functional programming paradigm, hence, this is also known as functional reactive programming.

Event-based languages and abstraction in order to represent reactive values on top of existing languages are two different approaches used to implement reactive applications [51]. Ptolemy [42], EventJava [14], EScala [21] and DominoJ [68] are all examples of event-based languages that support the event on the language level. Events are composable and update code can be easily separated from the main code to achieve a basic design principle of separation of concerns. Manual registration of event handlers and dependency management makes this approach an unfavourable choice for reactive applications. Reactive languages have direct representation of reactive values and allow composition by dedicated abstraction. Constraints are defined and runtime of language take care of propagating any change to all dependents. Fran [13] implements the concept of reactive values as Haskell library. It is a collection of data types and functions for composing richly interactive and multimedia animations. Other reactive languages like Scala.React [34], FrTime [7] also implement the same concepts in a new fashion. The next section focuses on this second approach and more specifically, the reactive libraries in the web domain which are a central topic for this thesis

2.2.1 Reactive Programming with JavaScript

JavaScript is a well-known programming language for developing client-side web applications and was ranked as the seventh overall most popular programming language in February 2017 [63]. JavaScript is considered to be a dynamic programming language because it provides features at runtime that non-dynamic programming languages provide during compile time. Treating functions as objects, allowing insertion and evaluation of code at runtime using **eval**, providing an interface to make requests to web servers and manipulating DOM at runtime make JavaScript a dynamic programming language [66]. Ajax revolution in 2005 further increased the popularity of JavaScript as a significant programming language. Competition between browser vendors (Mozilla, Microsoft, Apple, Opera, and Google) as resulted in vast improvements in JavaScript performance [6]. Ryan Dahl, the author of Node.js [10], was looking to

build non-blocking I/O server platform. He tried C and Lua as the programming language for his project but eventually turned to JavaScript because it lacked a robust input/output model (meaning that he could write his own new one) and had the fast and fully programmable V8 runtime readily available [65]. Closures and first class functions in JavaScript also make it a powerful match for his project [62]. In 2009, Ryan Dahl presented an early version of Node.js library at JSConf in Berlin [8]. It is built on Google Chrome's V8 JavaScript runtime engine and allows programmers to do server side programming using JavaScript. JavaScript applications are portable as most of the modern day devices have a browser that runs JavaScript applications [45]. Web applications nowadays are not static HTML pages anymore. They have become more interactive and complex, containing asynchronous behaviours [61]. Before RP, the above mentioned requirements were managed with the help of callbacks, but those are difficult to handle and error prone. To manage the complexity of modern day interactive web applications, some JavaScript libraries have been developed in recent years to implement RP paradigms. We will discuss the two most commonly used JavaScript reactive libraries in detail as these will be the main focus of this thesis.

2.2.2 ReactiveX and RxJS

ReactiveX is basically a pool of libraries based on different well-known programming languages like Java, PHP, JS etc.. It provides libraries on top of these different programming platforms to add abstraction to those languages and enable programming in a more declarative and reactive manner. Sometimes the term functional RP has been misused for ReactiveX library, but functional reactive programming is a bit different. The major difference between these two is that functional reactive programming operates on values that change continuously over time while ReactiveX operates on discrete values that are emitted over time [46]. The Reactive extension for JavaScript(RxJS) is a set of libraries to develop asynchronous and event-based programs. Basically, RxJS is composed of three components, namely Observables, Operators, Schedulers. Observables are used to represent asynchronous data streams. A stream can be defined as a sequence of ongoing events ordered in time. Mouse movement, clicks, HTTP requests and UI events are examples of a stream. Operators are used to transform those event streams from one form to another and use schedulers to handle concurrency in between event streams [43].

2.2.3 Important Concepts of RxJS

RxJS Reactive Pattern

In ReactiveX anything can be represented as a stream of data or events which are called Observable. Later those streams can be functionally transformed to another stream or can be combined with other streams to make a new stream. Finally, each stream can be consumed by any num-

ber of subscribers. An observer subscribes to an observable and reacts to items emitted by an observable. This concept ensures concurrency because the observer does not need to block itself while waiting for data from an observable. The marble diagram is a great way to visualize this pattern. In figure 2.1, a marble diagram is used to explain observables and transformation of observables to another form.

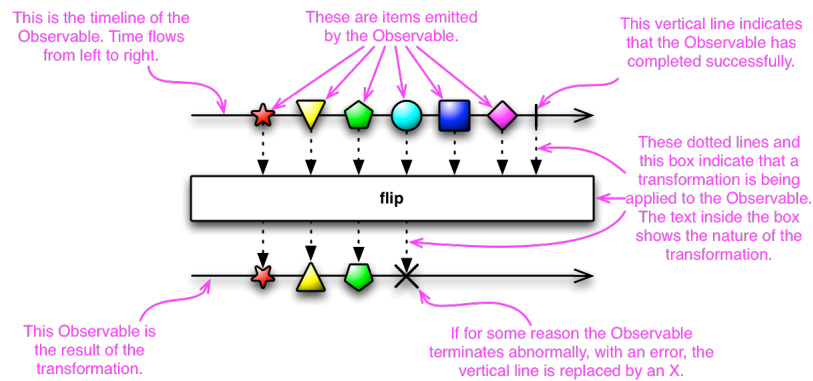


Figure 2.1.: Reactive pattern [47]

Observable and Observer

As we have already mentioned, the Observable pattern is the 'push' equivalent of the Iterator pattern. However, the Gang of Four's Observer pattern [20] still misses two key semantics of the Iterator pattern. The first one is the producer not signalling to its consumer that there is no more data available. The second is that the producer is not informing the consumer if an error takes place. With observable type, an observable calls its observer's `onCompleted` method to implement first semantic and calls its observer's `onError` method for the second missing semantic. With these additions, the iterable and observable types becomes more equivalent to each other. The only difference between them is the direction in which data flows. This equivalence makes it possible to perform any operation on observable that we can perform on iterable [46]. The observable is the basic building block of RxJS. It is the name of another abstract data type just like an array and other collections in programming languages. It represents any set of values over any amount of time. An Observable contains a sequence of values that a data producer pushes to the consumer. An Observable can also signal to its listener that it has been completed and will not send any more data. Using an array, all the data is stored in memory and using the Observable, there is no data stored in memory and items arrive asynchronously over time. We can also name the observable object as a provider because it represents the object that sends notifications to the observer object. An Observable gives us an object that represents an event stream and we can use that object to perform different methods on that object, as can be done with an array. For example, we can traverse an observable like we can traverse an array. In RxJs, the Observable can hot or cold. Hot observable starts emitting values as soon as it has been created and cold observable starts emitting data when an observer subscribe to it.

Operators

Besides extending the observer pattern to support sequences of data and/or events, ReactiveX provides a large number of operators that allow you to compose sequences together declaratively while abstracting away concerns about things like threading, synchronization and concurrency. RxJS provides a bundle of operators that we can apply on observables. An operator takes one observable as input and generates another observable as output. When an operator is called, it does not change the existing Observable instance. Instead, it returns a new Observable, which has different properties than the source observable. A list of operators with their details is available in the reactiveX online documentation [48]. Most of these operators operate on observable and produce another observable which make it possible to chain operators one after another.

RxJS Code Structure

In general, we can divide RxJS code into three parts. The first part defines and creates a source observable. The second part involves applying different operators to transform the source observable into the desired observable. Lastly, the desired observable has to subscribe in order to receive emitted values from the desired observable. Let's examine the code structure of RxJS with a simple example in Listing 2.1. As we have already seen that the observable is a collection of values over time, we need to define and create an observable before using it. RxJS library gives the possibility to convert single or multiple values, arrays, events, and callbacks into observables. We can also create our own observable by wrapping any functionality that produces values over time.

```
1 // 1. Srouce Observable Creation
2 var sourceObservable = Rx.Observable.interval(1000);
3 // 2. Transformation by applying different operators
4 var transformedObservable = sourceObservable.map(function(x) {
5     return x * 10;
6 })
7 .filter(function(x) {
8     return x !== 20
9 })
10 .take(5);
11 // 3. Subscribe to desired Observable
12 var subscription = transformedObservable.subscribe(
13     function(x) {
14         console.log('Next: ' + x);
15     },
16     function(err) {
17         console.log('Error: ' + err);
18     },
19     function() {
20         console.log('Completed');
```

```
21    });  
22    // OUTPUT  
23    Next: 0  
24    Next: 10  
25    Next: 30  
26    Next: 40  
27    Next: 50  
28    Completed
```

Listing 2.1: RxJS Simple Example

In the example code at line 2 **sourceObservable** is defined as observable using **interval** operator. Interval operator creates an observable sequence that produces a value after each period. In this case, it emits a sequence of integers spaced by 1000 milliseconds. In line 4 to 10, three operators are chained together to get the desired observable as **transformedObservable**. Map operator is applied to sourceObservable, that multiplies every emitted value from source observable with 10. So our source observable is supposed to emit values like 0,1,2,3,4.... and after applying map operator our observable will become 0,10,20,30,40..... . There is a filter operator next to map operator in the chain that filters value of 20 from its source observable. After applying the filter operator emitted values are supposed to be 0,10,30,40... The last operator in the list is the take operator that restricts the emitted values to a defined number, in this case to 5. After transforming the source observable to our desired observable we need to subscribe to our transformed observable. Line 12 to 21 in the example code contains the subscription. Subscribe method take three callbacks, the first one executes whenever a new value emitted by the observable, the second callback is for error handling and executes in the case of any error occurring and the third callback is executed to signal that the observable has been completed and will not emit values anymore. Line 22 to 28 displays the output of the example code.

2.2.4 Bacon.js

Bacon.js is a functional reactive programming library for JavaScript that assists in dealing with the asynchronous nature of JavaScript code. It is comparable to Underscore.js [1], which is a JavaScript library which provides a collection of useful functional programming utilities for common tasks like map, filter, invoke etc.. Underscore.js is for data which is available already like arrays. Bacon.js turns the way in which individual events are handled with imperative programming model into a functional programming model and works on events streams. So using this library will reduce the complexity of handling events individually into code that will look more logical, easy to understand and functional.

EventStream and Property

In Bacon, these are two flavors of Observables. EventStreams and Property are two basic concepts of Bacon.js that are basically known as events and behaviours in the literature of FRP

EventStreams are sources of events. For example mouse clicks, keyboard events can be modeled into an EventStream object. EventStreams are composable. So we can combine several EventStream objects to create another EventStream. Property which is also known as behaviours and signals, is an abstraction for the values that vary over time [22]. Properties are very similar to EventStreams and share most of the functionality with EventStream but Property also has the concept of current value and we can create a Property from an event stream by using toProperty or scan method [3].

Bacon Example

```
1 var arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
2 var baconSourceStream = Bacon.sequentially(1000, arr);
3 var baconFilteredEvenStream = baconSourceStream.filter(function(x) {
4   return x % 2 == 0;
5 });
6
7 baconFilteredEvenStream.onValue(function(val) {
8   console.log('Next: ' + val);
9 });
10
11 baconFilteredEvenStream.onEnd(function() {
12   console.log('Completed');
13 });
14 baconFilteredEvenStream.onError(function(err) {
15   console.log('Error: ' + err);
16 });
17
18 // OUTPUT
19 Next: 2
20 Next: 4
21 Next: 6
22 Next: 8
23 Next: 10
24 Completed
```

Listing 2.2: Bacon.js Example

The example code is a simple code written using the bacon.js library and is very similar to the RxJS example presented earlier. Line 2 uses the method **Bacon.sequentially** to create a source event stream containing values from a given array delivered one by one with a given interval in milliseconds. So after every 1000ms one item from the given array will be emitted. Line 3 to 5 apply a filter operator to source event stream to emit only even values. From line 7 to 9, subscription of a given handler function to the observable is done, using **onValue** method.

Lastly, **onEnd** and **onError** methods are used to trace the completion and error handling of streams.

2.3 Debugging and Tools Support

In software engineering literature, the process of finding errors in a computer program is called debugging. The error can be syntax or logical error. Violation of any syntax rule of a programming language such as misspelled keywords, a missing bracket, or a missing closing parenthesis is categorized as a syntax error. Syntax error can be found easily because a program that includes syntax errors, won't be executed. Nowadays, IDEs and debugging tools detect and highlight these errors as you type. If you try to execute a program that includes syntax errors, you will get error messages on your screen and the program will not be executed. Logical errors allow the program to execute but might result in an unexpected outcome. It is difficult to detect a logical error because there is no warning on execution. Debugging tools can help to revisit the whole program to detect the logical errors. Debugging tools are always the helping hands of programmers whether they are trying to find an error in their own program or they are trying to understand the workings of programs written by other developers. Most debugging tools give its users some visual representation of the code and gives users the privilege to control how the program executes.

2.3.1 Debugging JavaScript

Dynamic and reflective nature of JavaScript makes it hard to debug and analyse JavaScript code [45, 53]. Before the big advancements in the developer tools of browsers, application developers mostly relied on **alert** statements to inspect values of variables and the output of functions but for this, they are required to change the code. Besides using `console.log` and debugger statements, now modern browsers give more built-in debugging functionalities, that developers can use to debug JavaScript programs. Programmers can set breakpoints to halt the execution of a program and can examine the complete context in which any statement or function executes. Developers can also observe performance and memory consumption of code. There is a long list of similar features but these are beyond the scope of this work. If we compare debugging support from language itself, then we come to know that the JavaScript language provides very limited support for debugging. It does not offer any API for debugging purpose. Unlike JavaScript other programming languages like C/Java provides APIs or packages to developers for debugging purpose. For example the Sun Java Development Kit (the JDK) includes the Java package `sun.tools.debug`, which provides a simple interface into the Java Virtual Machine. This API allows another program, probably a debugger, to connect and communicate with the Java Virtual Machine to get low-level information about a currently executing Java application [64].

Another reason why it is hard to debug JavaScript code is that, unlike other languages JS code does not need to compile before execution, so bugs can only be found at runtime.

2.3.2 Debugging in Reactive Programming

Nowadays advanced debugging tools are an essential part of good IDEs. Those tools are unsuitable for declarative and data flow oriented models of RP [52]. So for RP, debugging tools intend to manage conceptual change of RP as compared to traditional imperative programming. As the RP paradigm is still a new and emerging model, there are not many examples of tools that are built to debug reactive programs. RP debugging technique models reactive application into dependency graph where nodes represent the values/signal that may change over time and edges between them depict dependency among various signals. RP Debugging technique has been implemented as an Eclipse plugin to debug software in reactive style [52]. This thesis is the result of the inspiration of the above-mentioned techniques.

2.3.3 Debugging Reactive Extensions of JS

Bacon and RxJS are most commonly used reactive libraries for JavaScript. Debugging asynchronous code is not an easy task. As mentioned before that RxJs and Bacon work with streams of events asynchronously. That's why it's not feasible to use traditional browser developer tools to debug RxJS and Bacon based applications. Like other RP languages or extensions, there is no debugging tool available for these libraries. These libraries are gaining popularity day by day but they are still missing tools support that can help developers to debug and understand programs written in these libraries. In this article [56], the author explains why traditional dev tools are not enough to debug RxJS code. Because applications developed using these libraries are abstract code and not procedural code anymore, that is why browser developer tools and breakpoints do not help while debugging or understanding code. In this article, the author writes that to debug the RxJS application you have to rely on drawing a dependency graph and marble diagrams by hand. The dependency graph has been introduced already. A marble diagram is the visual representation of input and output of operators available in these libraries [57]. So there is a strong requirement to extend existing debugging tools to support debugging for JS libraries built to implement the concepts of RP. This thesis is a step towards developing debugging tool for these libraries. We implemented an extension to chrome dev tools where developers can visualize and debug code written in RxJS or Bacon library. Details of this tool will be presented in coming chapters.

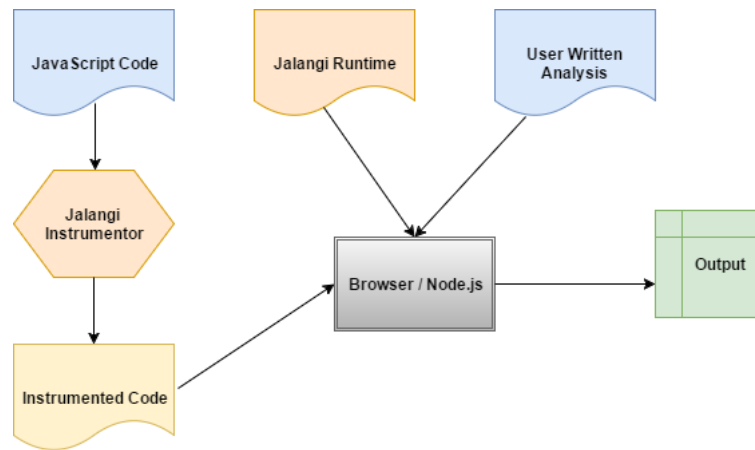


Figure 2.2.: How Jalangi Works

2.4 Jalangi

Jalangi is dynamic analysis framework for JavaScript applications. Using this framework, it becomes possible to monitor every operation of the JavaScript application and its API makes it possible to write one's own program analysis modules. Jalangi framework works independently of the platform where the code eventually runs. Basically this framework instruments all JavaScript code given to it and creates hooks in the resultant code. Hooks inserted by this framework are used to monitor each operation at runtime, like read from, write to variable, function calls etc.. [54]

The basic components of Jalangi framework and their working is illustrated in figure 2.2. First of all, Jalangi instrumentation module takes JavaScript code and instruments that code to be executed in the browser. Beside this instrumented code, Jalangi runtime framework code also executes in the browser, which implements those hooks called in the instrumented code. Those hooks keep the semantics of the target code and invoke callback functions defined in user-written analysis code. The user-written analysis is the code written by third-party program analysis developers, overriding those predefined APIs and thus allows one to intercept those execution events and do program analysis.

```

1 // Before Instrumentation
2 x = y + 1
3 // After Instrumentation
4 x = Write( 'x' , Binary ( '+' , Read( 'y' , y ) , Literal(1) , x )
  
```

Listing 2.3: Jalangi Instrumentation

The first code snippet is before instrumentations and the second one is the code after instrumentation is done by Jalangi. Jalangi framework runtime code implements those hook callbacks (Read, Write, Binary, Literal etc..) . This framework can be used to check different kinds of correctness bugs and performance bugs, doing various program analysis (e.g., debugging, Per-

formance analysis, Monitoring dynamic behaviours, Record and replay, runtime call graph etc..) We use this framework in our extension to find the reference of a node of dependency graph to JavaScript variable name. Details of this usage will be presented in further sections.

2.5 Chrome Developer Tools

Over the last 5 years, the Google's Chrome browser has been the most popular internet browser used worldwide. In the month of February-2017, Chrome has been used by 52.72 % users, according to StatCounter, the independent website analytics company. Useful features for developers in the form of developer tools and a big pool of extensions also makes the Chrome browser the best choice for web developers. Chrome developer tools (DevTools) allows web developers extensive access into the internals of the browser and runtime state of their web application [24]. Efficiently tracking down layout issues, setting JavaScript breakpoints, and getting insights for code optimization are common use cases of DevTools. The DevTools can be open for any web page in the Google Chrome browser by right clicking the mouse and selecting **Inspect** option. Keyboard shortcut **Ctrl+Shift+I (Windows)** or **Cmd+Opt+I (Mac)** can also be used to open DevTools in chrome browser. One can also alter the position of DevTools to the bottom or side of the browser window or it can even be opened as a separate window. DevTools group different tools and put it in separate panels as we can see in the figure 2.3.

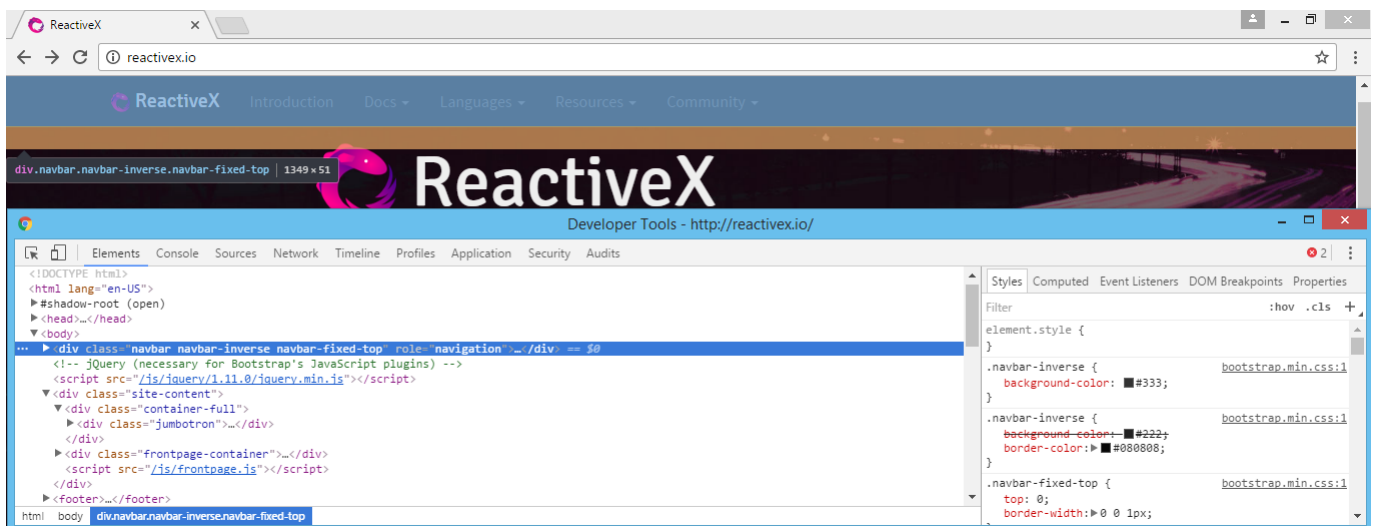


Figure 2.3.: Chrome Developer Tools

Using the Elements panel, we can see the raw HTML, raw CSS styles, the Document Object Model. One can manipulate HTML or CSS in real time and can see how it is effecting the web page. Web developers use Elements panel quite extensively when they need to inspect HTML and corresponding CSS of part of a web page. This tool also presents minified or ugly HTML in formatted and readable structure. The console panel is used to log and display debugging information along with all errors or warnings generated by the current web page. From the

Console panel, we can also enter arbitrary JavaScript and programmatically interact with our page [29]. Web developers especially JavaScript developers use Sources panel to inspect all the JavaScript code that is being used by web page under inspection. One can change running JavaScript code on the fly, one can set breakpoints and can inspect values of variables at runtime [28]. The Network panel can be used for optimization. It provides detailed real-time insights into resources requested and downloaded over the network. More on DevTools can be found from online documentation [24].

2.5.1 Extending Chrome DevTools

One does not need to hack Chrome browser code to add functionality to Chrome Devtools. Chrome allows everyone to add functionality to Chrome DevTools by writing extensions to it. To write an extension to DevTools, we need to have knowledge of core web technologies such as HTML, CSS, and JavaScript. We can also distribute our extension by publishing to the Chrome Web Store. By extending DevTools one can add new UI panels and sidebars, interact with the inspected page, get information about network requests, and much more. Dev-Tools extensions have access to a big pool of JavaScript APIs [25] and some DevTools-specific extension APIs. List of sample DevTools extensions can be found here [27]. The manifest file called **manifest.json**, contains metadata about that particular extension in JSON format. It contains properties like your extension's name, description, version number and so on. At a high level, we will use this file to declare to Chrome what the extension is going to do, and what permissions it requires in order to function properly. Background page, content script and DevTools page are three main components of DevTools extension as shown in figure 2.4. These three components communicate with each other by message passing [26].

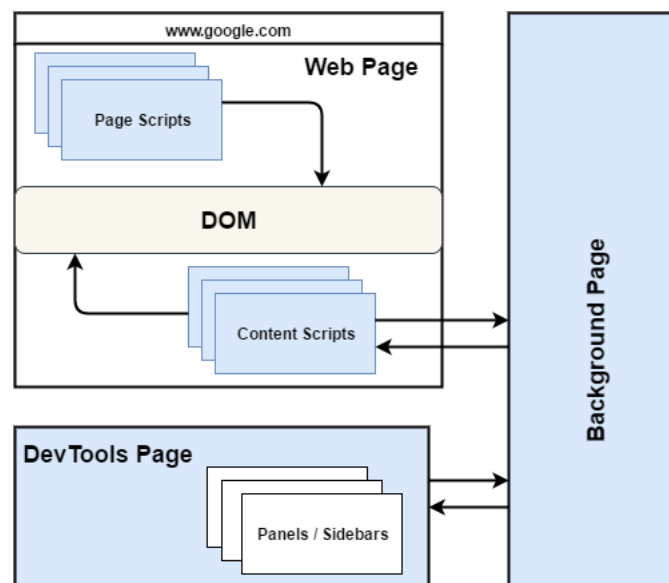


Figure 2.4.: The Structure of Chrome DevTools Extension

Content Script helps to interact with web pages the browser visits and gives access to shared DOM. The content script is JavaScript code injected by an extension to run in the context of the page that has been loaded into the browser. Content scripts can be injected statically by defining the name of JavaScript file and parameter that tells the extension whether to execute at the start or at the end of the document. The DevTool page can also inject content script dynamically using chrome API. The content script can access and manipulate the DOM of the web page that is currently under inspection. Content script and script belonging to the web page do not conflict with each other because they both execute in an isolated manner. Background Page is an invisible page that holds the main logic of the extension. Persistent background pages and event pages are two types of background pages to control the behaviour of the extension. Event pages run as needed and persistent background pages run all the time. Besides containing the logic of the extension, background page serve as a communication channel between the inspected page and DevTools page through content scripts. The **devtools_page** property in the manifest file, must point to an HTML page and that reference to JavaScript file. So DevTools page is not something that you visually need to see, rather it is just something that you need to have in order to run some JavaScript code in DevTools window while it is opened. DevTools page uses devtools APIs, to setup panels/sidebars, to interact with the inspected window and to get information about network requests. DevTools page communicates with the background page using Message Passing.

2.6 Related Work

We split this section into two parts. Firstly, we discuss different perspectives of advancements in debugging techniques, related to our work. In the second part we present some example tools in web domain and aids developers to understand abstraction in the running code.

2.6.1 Beyond Traditional Debugging

Log-based debugging and breakpoint-based debugging are two traditional debugging techniques. The first one requires manual modification of source code to trace program execution whereas the second one allows setting breakpoints within the source code to step into code execution. Omniscient debuggers overcome the issues of traditional debuggers and combine the positives of both traditional approaches. During the execution of the program, all events get recorded and presented to the programmer, which can later be used by the developer to see the transformation of states caused by different events [40]. Omniscient debuggers help developers to visualize data and control flow in an application with the ability to navigate forward and backward through the program execution. Besides the powerful advantages of omniscient debuggers, it has some serious concerns regarding performance that is why this technique is

not adopted by the software industry as well as in education sector [55]. It may require special techniques to process and store events in large, real-world applications. There is a lot of recent research addressing the optimization of omniscient debugging technique [40, 39, 32]. Hence, we tried to adopt properties of the omniscient approach in our implementation and thus the pros and cons of this technique also applicable to our work.

Traditional debuggers are mostly based on runtime stack information and do not consider any other abstraction within the running code. Object-centric debugging uses objects as abstraction and allow the developer to perform debugging operations based on objects instead of the execution stack [44]. Similar to our work, Object-centric debugging is also a representation of the running system. In our case, we focus on declarative abstractions implemented by Reactive Programming paradigm.

2.6.2 Related Tools

During this work we have found an application called RxVision [17], that was made for debugging and visualizing RxJs reactive streams. This tool provides an online web page [18], where we can write and run RxJS code and can also see a visual representation of that code. This application does not support the latest version of RxJS and was not developed further. The visual representation of RxJs code is very hard to understand and it is very hard to map visual representation to the code because this application does not map event streams to the JavaScript variable. We also looked into the code which is available on GitHub [17], and we found that they override almost everything of RxJS library to log all internal activities, that is the reason why it does not support the latest version of RxJS.

Cycle.js DevTool for Chrome is another browser extension which aims to help while debugging and understanding the abstract code [59]. This extension is to debug or visualize data flow in Cycle.js [58] apps. This application has more resemblance to our work because it also displays event stream in the form of dependency graph but this also has the same problem of not getting the reference to the JavaScript variable name. It is not possible to track the history data of streams.

React is an open source JavaScript library developed by facebook [15], for building user interfaces of web applications. Fortunately, this library already has a tool support for debugging, in the form of chrome browser extension [16]. This tool is very helpful for developers to debug the code that is based on React library. Using React dev tool, we can inspect React tree, components and properties passed to each component and the state of each component with some live editing features. This extension is a live and working example that is implemented as a browser extension to explain the abstraction in the code, to the developers.

3 System Design

The first section of this chapter presents the high-level requirements for the system developed in this thesis. The system architecture is explained in detail in the second section. Finally, we explain the design choices and system features with examples.

3.1 System Requirements

Considering the fact that traditional debuggers are not well suited for debugging reactive applications, a new system is developed in this thesis to debug web applications based on JavaScript reactive libraries. We identified the requirements for the system developed in this thesis with regards to the requirements of the software architecture and features available to the end user.

General Availability

The new system should be easy to install and integrate. In order to reach many developers, the system should be developed as an extension to the widely used Chrome DevTools.

Debugging without Changing Your Code

The new system should work without making any change to the application code. Developers should be able to analyze and debug reactive system without modifying the application code.

Visualization of Application at Macro-Level

To get the big picture of the application under inspection, the developer should be able to see the dependency graph based on specific time-changing variables (streams/observables). For better understanding of the application, dependencies among different variables should be clearly visible. During the execution of the application, the dependency graph should be automatically updated, so that new variables, new dependencies, updated values and so forth are immediately visible. Transformation of data streams by different operators should also be visible. Especially, when the chain of operators is being applied to an observable, the transformation should be visible by showing emitted values by each node after applying every single operator in that chain.

Back-in-Time Debugging

The developer should be able to have a look at the dependency graph at any arbitrary point in time. Hence, one should be able to visualise the whole history of the dependency graph. Step by step navigation through the time to observe events, such as node creation, value changes,

updates of dependencies and the like should be possible.

Querying the History of the Graph

Manual step by step navigation is not practical for large programs. Therefore, it should also be possible for the user to jump to a specific point in history with the help of query language. For example, by defining a query, one should be able to jump to the point in time at which a particular node has been created or evaluated.

Reactive Breakpoints

Developers should be able to set breakpoints specific to RP that stop the execution when a significant event occurs. For example, it should be possible to set a breakpoint 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 should be used as mentioned before.

Extension Helpers

The developer should have an option to download the dependency graph at any point in time as an image so that it can be shared with other developers. For larger programs, it would be helpful to have a search functionality that finds a node within the dependency graph. For application with long execution cycle, it would be advantageous to have a feature to pause and resume updates to the dependency graph at any point during the execution of an application.

Library Independence and Extensibility

The system should not depend on a specific version of reactive libraries implementations, so that if a new version of the reactive library comes then the system should work with little or no modification. It should also be possible to add support for more reactive libraries. The system should be easy to extend, for example, one can easily extend the query language.

3.2 System Architecture

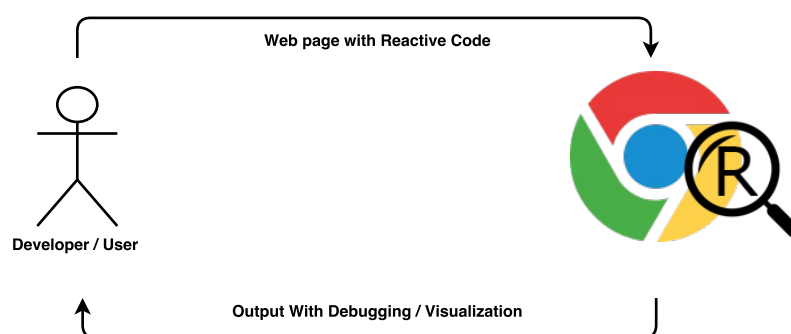


Figure 3.1.: Simple Use Case Diagram

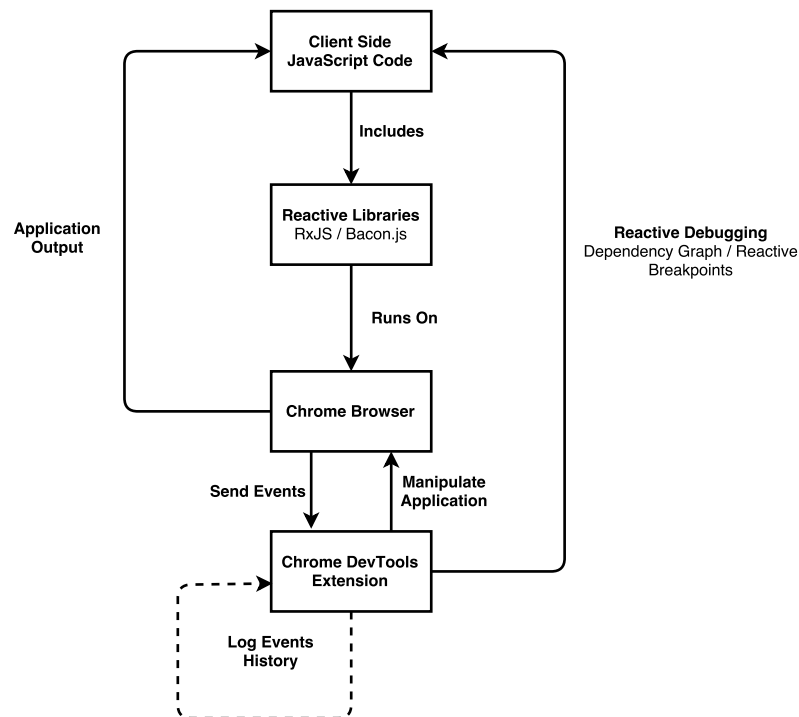


Figure 3.2.: System Components Overview

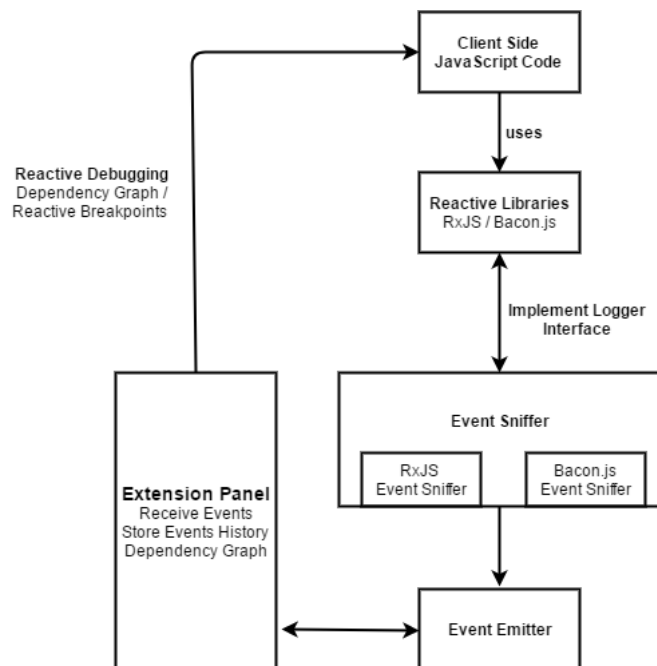


Figure 3.3.: Detailed System Architecture

Figure 3.1 shows a simple use case diagram where the developed DevTools extension (CRI) is installed on Google Chrome browser. The user is loading a web application that contains reactive code, in the browser. Chrome browser is returning the output of web application with debugging and visualisation features.

Figure 3.2 presents the overview of system components and their interaction. CRI manipulates the application code before executing it in the browser. The manipulation makes it possible to sniff all internal events from the reactive library. CRI stores the event history to the browser storage which makes back in time debugging possible.

Figure 3.3 depicts the system architecture in detail. The application code is using the reactive library (RxJS / bacon.js). Event Sniffer is a core component of CRI; it contains library specific implementations to sniff all the internal events from the respective library. In this thesis, we have implemented event sniffers for RxJS and bacon.js. One can easily extend event sniffer to add support for other libraries. Event Sniffer passes the information to the event emitter, which is further sent to the CRI extension panel. The CRI extension panel receives all event information, it puts this information into defined data structure and saves it to browser storage. The CRI extension panel also creates a dependency graph out of prepared data structure.

3.2.1 Event Sniffing - Options

To implement the systems architecture mentioned above, the fundamental task at hand was to sniff all internal events from the reactive libraries. Let's have a look at possible options to implement event sniffer for a particular library.

3.2.1.1 Option 1

The general concept of this option can be visualised in figure 3.4, where bypass of selected calls to the reactive library is shown. We intercept the connection between application code and reactive library and bypass selected calls through injected script. Injected script is JS code that is injected into the web page by CRI. This bypass of calls is implemented using AOP technique, which has already been explained in section 2.1.2.

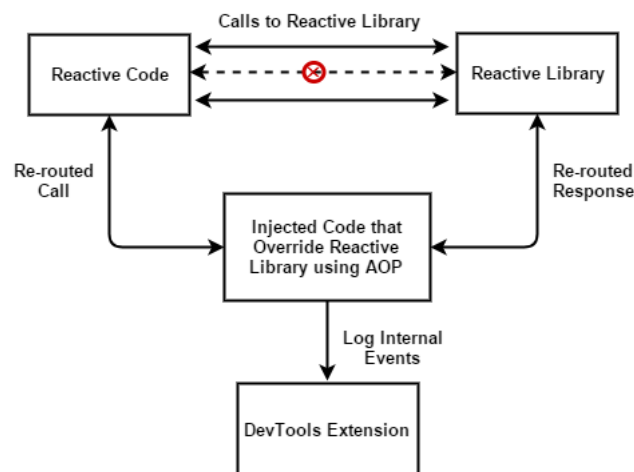


Figure 3.4.: Event Sniffing - Override Calls To Reactive Library with AOP

We use this option in event sniffer for RxJS library because RxJS library did not provide any API interface to log all internal activities of the library. Although RxJS library provides significantly improved internal architecture from version 5, where they have implemented observables operators in term of **lift**¹. Lift is a function that takes source observable and an observable factory function and returns a new observable. That means every observable operator of the library calls lift function. We take advantage of this change in the RxJS library, and we just bypass call to lift function of RxJS library using AOP technique. By overriding the only single method of RxJS library, we make it possible to get all observables defined in the application code and then we subscribe to those observables to get their values.

3.2.1.2 Option 2

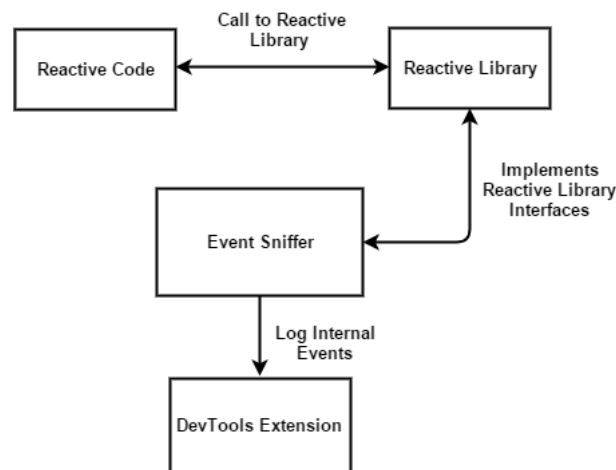


Figure 3.5.: Event Sniffing - Implementing Interfaces Provided By Reactive Library

Figure 3.5 depicts the second option to implement event sniffer for the reactive library. For this option, it is necessary that the target library provides some interface to get all internal activities. Bacon.js provides such interface in term of **Bacon.spy(f)**² method. It adds the given function as a spy that will get notified on all new observables. The CRI subscribes to all observables received by the spy interface to sniff all internal events of the library.

3.3 System Features and Design Choices

¹ <https://github.com/ReactiveX/RxJS/issues/60> , last accessed 23-05-2017

² <https://baconjs.github.io/api.html> , last accessed 23-05-2017

3.3.1 Dependency Graph Visualization

As described in chapter 2, reactive systems can be understood and visualised with dependency graphs. A dependency graph consists of nodes and edges where observables are represented by nodes and edges which describe the dependencies among the observables.

```
1 // following observable emit 1 when element with id 'up' is clicked
2 var upClicks = $('#up').asEventStream('click').map(1);
3 // following observable emit -1 when element with id 'down' is clicked
4 var downClicks = $('#down').asEventStream('click').map(-1);
5 // above two event streams are merged and scanned to get the resultant value
6 var counter = upClicks.merge(downClicks).scan(0, function(x, y) {
7   return x + y;
8 });
```

Listing 3.1: Bacon.js Example of Up - Down Counter

Consider the following example in listing 3.1 where **upClicks** and **downClicks** are two observables. **upClicks** is a stream of events which emit 1 on every click event of DOM element with id is **up**. Similarly **downClicks** observable emits -1 on every click event of DOM element with id is **down**. Line 6 of the example merges both event streams and scan results to the sum of both. The figure 3.6 models our example as a dependency graph and shows the propagation of changes through nodes and edges of the dependency graph. It displays how relevant events affect different parts of the application. Modelling of reactive systems with dependency graphs should be a great help for developers to understand and debug reactive systems.

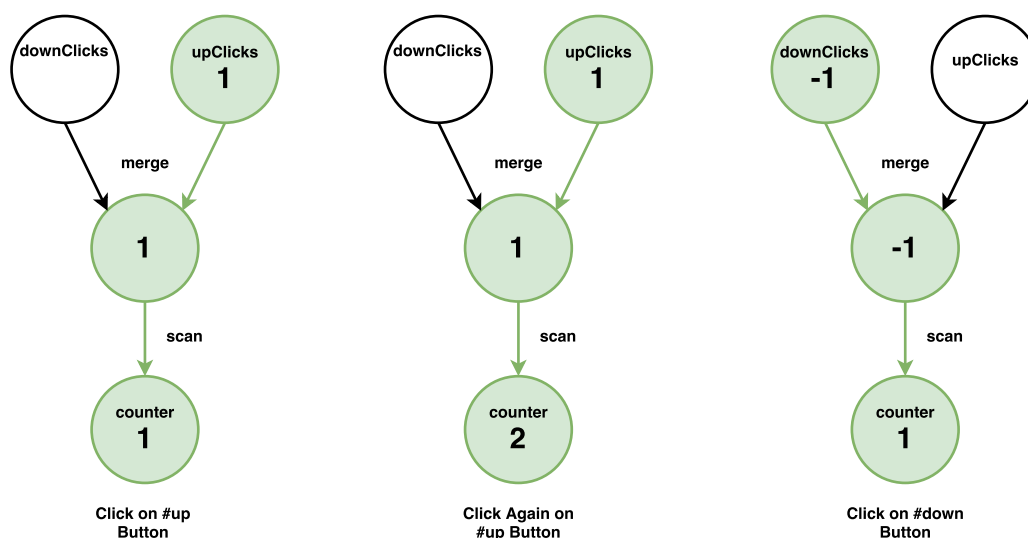


Figure 3.6.: Propagation of Changes through Dependency Graph

3.3.2 History of Dependency Graph

During the execution of the reactive application, every change in the dependency graph is recorded as a history of the dependency graph. Hence, the developer has access to the whole history of the dependency graph. It is possible to see the dependency graph at any point in time. There are three possible ways to navigate through the history. One can use back and forth buttons to navigate step by step through the history of the dependency graph. However step-by-step navigation is not practical with large histories. Developers can use a slider to navigate to arbitrary points in time. A third option to browse through the history is by using the provided query language. Using queries which match certain events, the developer can directly jump to the respective points in time where these events occurred.

3.3.3 Reactive Breakpoints

Native Chrome DevTools support different type of breakpoints to halt the application execution. Most commonly used breakpoints types are, **line-of-code** and **conditional line-of-code** breakpoints ³. These breakpoints are not well suited for RP. Hence, reactive-programming-specific breakpoints should be implemented. For example, we can set a reactive breakpoint for listing 3.1, `evaluationYielded[nodeIdOfCounter, "2"]` query only matches if the **counter** is evaluated to the value **2**. This could of course still be done with native DevTools by creating a conditional line-of-code breakpoint at the respective line. But there are also cases which cannot be handled with native DevTools. Especially if there is a chain of operators being applied to an observable, then we cannot set breakpoints to intermediate streams with the native debugger.

3.3.4 Extension to Chrome DevTools

The System implemented in this thesis is an extension to Chrome DevTools, which are a set of web authoring and debugging tools built into Google Chrome. Chrome allows the addition of more debugging features to its native debugging tools by creating an extension to DevTools. The developed extension would add a new Panel into the existing Developer Tools of Google Chrome which provides the UI of features discussed so far. The developed extension is so general that the same implementation can also be adopted for others browsers like firefox.

3.3.5 Scoping and Snapshot

The scoping feature aims to decrease performance overhead of the developed extension. As CRI instruments the JavaScript code of the target application with a library called Jalangi. With

³ <https://developers.google.com/web/tools/chrome-devtools/javascript/breakpoints> , last accessed 23-05-2017

Scoping feature one can define the target JavaScript file names that should be considered during the instrumentation process. The snapshot feature allows users to download the dependency graph as an image, which they can later share with others.

3.3.6 Reactive Libraries Support

As mentioned earlier, the system implemented in this thesis supports two reactive libraries, namely RxJS and Bacon.js. In future, one can easily add support for other libraries by implementing event sniffer for that particular library.

3.3.7 Data Storage and Management

As explained in section 2.5.1, different components of Chrome DevTools extension can communicate with each other by message passing. Besides direct communication among extension components, we also need to store data somewhere temporarily. For example, we need temporary storage for the scoping feature, so that we can store the string of file names that a user wants to consider during the instrumentation process. We also need to save reactive breakpoints, so that they can be processed during the execution of target application next time. We use **chrome.storage**⁴ API to store, retrieve and track changes to the user data specific to the features mentioned above. Using this API, we can save objects to local storage of the browser and can access it later from any component of the extension (content script, background, panel).

3.3.8 Mapping Stream to JS variable

During the development of the CRI extension, it was quite challenging to map reactive streams (observables, event stream, property) to the JavaScript variable name defined by the user. For example, if we look into the second line of Listing 3.1. We can see that there we have an observable that emits 1 when a particular DOM element is clicked. In JavaScript, there is no way to find variable names at runtime. Because the JavaScript interpreter in a browser is implemented as a single thread, only one thing can happen at a time and other actions are queued in execution stack. At runtime, any function can access variables/functions from outside of its own context, but an outside context can not access variables/functions declared inside. So we can sniff events from our event sniffer without referring those to JavaScript variables names defined by the developer.

To tackle this issue, we used a third party JavaScript library called Jalangi, which is already introduced in section 2.4. It instruments JavaScript code so that the functionality remains unchanged and the hooks added during the instrumentation process make it possible to monitor

⁴ <https://developer.chrome.com/extensions/storage> , last accessed 23-05-2017

every operation (e.g., variable read/write, unary/binary operation, function/method call, etc.) performed by the execution.

3.3.9 Chrome Extension Scripts

Lets the architecture of Chrome DevTools extension from section 2.5.1 and group the type of JavaScript codes runs in Chrome extension.

1) Web Page Scripts: Web Page scripts are the JavaScript code that is part of the webpage, it has full access to the DOM.

2) Content Scripts: This runs in a scope between the extension and the web page. It has partial access to some of the Chrome APIs and full access to the page's DOM. It executes in a special environment called an isolated world. They do not have access to any JavaScript variables or functions created by the page. It looks to each content script as if there is no other JavaScript executing on the page it is running on. Similarly, Web page scripts cannot call any functions or access any variables defined by content scripts. An extension can add content script via the manifest file or by using **chrome.tabs.executeScript** from background script.

3) Background Scripts: This behaves as a middle tier in between content and panel script. It can also use to inserting any script as a content script into a page programmatically. Background script has full access to all permitted chrome.* APIs.

4) Panel Script: This is responsible for communicating with the background page of extension and the logic of panels and sidebars.

3.3.10 Scripts Injection from Extension

A Chrome extension can inject JavaScript code into the target application which can be run either in the context of the web page script or in the context of the content script. If the content script's code should always be injected, then one should register it in the extension manifest using the `content_scripts` field⁵. The content script can also be injected by using **executeScript** method of tabs API⁶ of Chrome.

```
1 var scriptElement = document.createElement('script');  
2 scriptElement.src = chrome.extension.getURL('myscript.js');
```

⁵ https://developer.chrome.com/extensions/content_scripts , last accessed 23-05-2017

⁶ <https://developer.chrome.com/extensions/tabs> , last accessed 23-05-2017

```
3 (document.head || document.documentElement).appendChild(scriptElement);
4 scriptElement.onload = function() {
5   scriptElement.parentNode.removeChild(scriptElement);
6 };
```

Listing 3.2: Injecting Script into Web Page Context

If we want to inject a script that should be executed in the same context where target web scripts are being run, then we have to inject it by adding new script tag. Listing 3.2 can be used to inject **myscript.js** file from the content script into the target web page. We use all of these options to inject script into the web page in our implementations.

4 Implementation

This chapter presents the implementation details of the CRI extension. In the first section, we present two alternative implementations of the CRI extension. The second section describes the significant data structures used by the extension. The communication in between different components of the extension is explained in the third section. The fourth section presents the developed query language and its usage. The last section describes the graphical user interface provided by the extension.

4.1 CRI - Implementation Alternatives

In this section, we present two possible implementations of CRI extension. To determine the best possible alternative is yet to discover by looking into the pros and cons of both alternatives and is mentioned as one of the possible future work. We tried our hands at both alternatives, but for the final version, we focused on the first alternative. So except this section, everything presented in this thesis is based on the first alternative.

4.1.1 Alternative - 1

As described in section 2.5.1, in Chrome DevTools extension, page script and content script runs in two different execution contexts, and both have access to same DOM. Figure 4.1 depicts component level details of the first alternative, where the user has an application with reactive code to debug or analyse. Before execution takes place, the application code is intercepted by DevTools extension and the manipulated code is run in the browser which provides the same output as the original code. All components residing within the big dotted round cornered box are run as a content script. **Interceptor** module is responsible for preventing JavaScript code in the target application from executing in the browser. For this purpose, we created a script that runs as a content script at document start. It uses **window.stop** to halt the normal loading of the web page to the browser. Then, we make a XHR(XMLHttpRequest) request to get the content of the target document. Replace the **type** attribute of all script tags with some custom string and write it to the document, this will stop the browser to execute that script. After that sequentially loads the JavaScript files and instruments them if required and injects them into the web page as a content script. After instrumentation, when manipulated code runs in the browser then it invokes the Jalangi API on different events like whenever any value is written to JavaScript variable. Instrumentor receives JavaScript code from the interceptor module and performs instrumentation on a given JavaScript code with Jalangi library. Instrumentor returns instrumented code to the interceptor module.

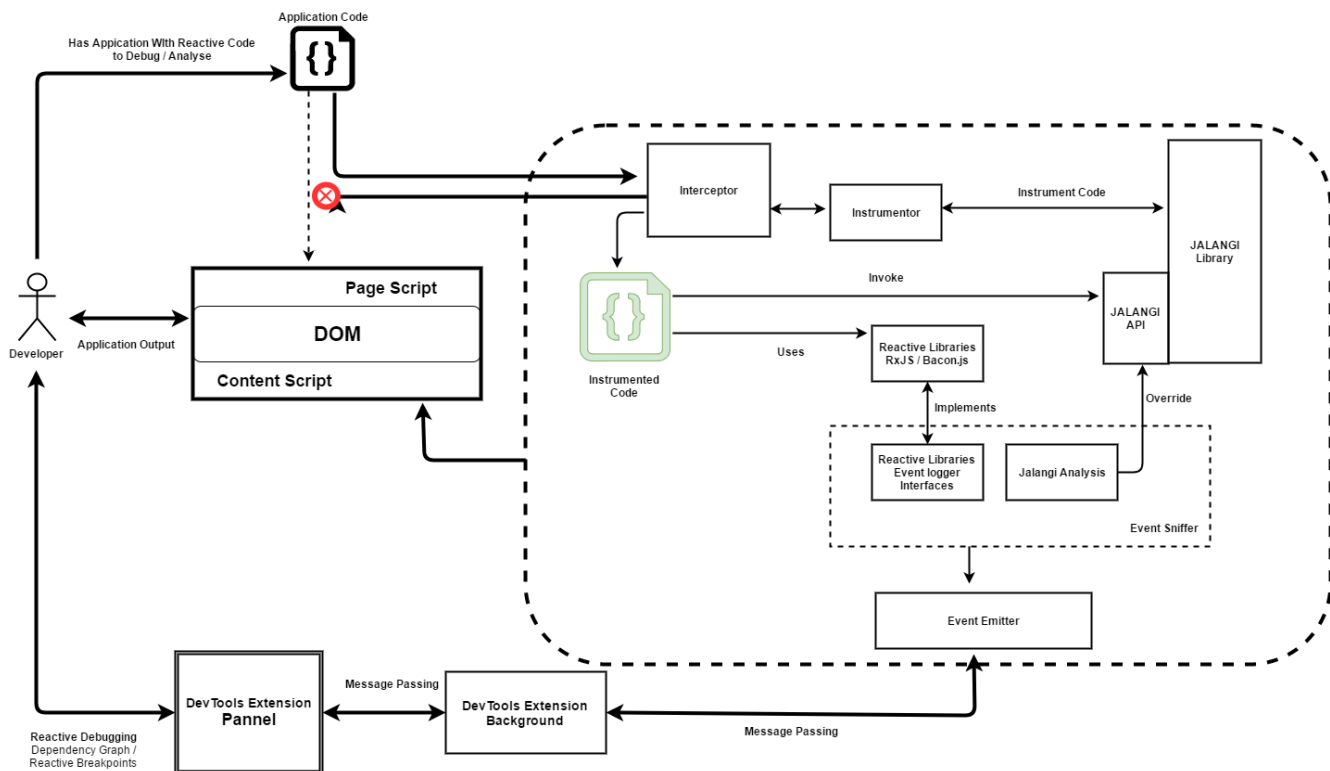


Figure 4.1.: System Components Detail - Alternative 1

Event sniffer comprises of two sub-components, the first one contains the reactive library specific implementation to log all internal activities such as the creation of observable, and values emitted to subscribers. The second sub-component of Event sniffer is Jalangi Analysis that overrides some methods of Jalangi API, which make it possible to map the reactive streams to the JavaScript variable name. Event Emitter is also part of the content script, it receives events information from event sniffer, it changes them to defined message structure and passes this as a message to extension's background page. The communication in between content script, background page and extension panel is done with message passing¹. Extension's background page receives messages of a defined pattern from the event emitter and passes them to extension panel. Similarly, background page also receives messages from extension panel and forwards them to content script. Extension's panel receives events information as a message from background page, parse those messages and responds according to the type of the message. The message can be of type **saveNode** or **saveEdge**. If the message type is **saveNode** then we check whether it is an update to an existing node or it is a new node. On receiving a message from the background page, extension panel update the dependency graph and save the graph state in the browser storage. Whenever the new state of dependency graph is saved to the browser storage, the step slider is also updated. Finally, the developer sees the updated dependency graph, to which he can interact with by setting reactive break points or navigating through the history of the dependency graph.

¹ <https://developer.chrome.com/extensions/messaging>, last accessed 24-05-2017

4.1.2 Alternative - 2

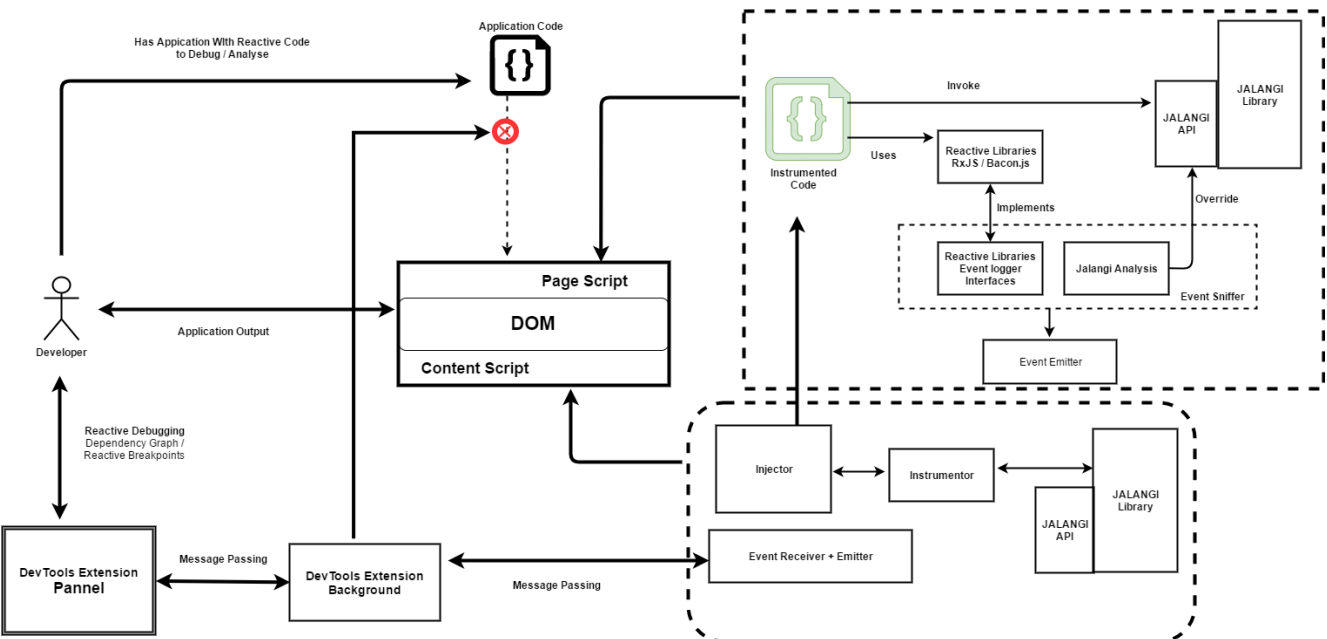


Figure 4.2.: System Components Detail - Alternative 2

Figure 4.2 presents the implementation details of the second approach. Most of the modules are common to the first option. With this approach, instead of preventing the whole page from loading to the browser, we use **chrome.webRequest**² API in background page of the extension to only intercept the HTTP request to specific files that are defined in the field of the `scopes` feature. Injector script is run as a content script by extension at the end of the document loading; it is responsible for getting target JavaScript files with XHR request and instrumenting it with Jalangi. All modules within the dotted square box are injected by injector script into the target web page as page scripts. Event sniffer works similarly as in the first approach. Event emitter from the page script passes messages to **Event receiver + emitter** using **Window.postMessage()**³, which is then received in the content script by **Event receiver + emitter**. From here onwards everything works as described in the first approach. The noticeable differences in this method are as follows: 1) instrumented code, and event sniffer modules run in page script context. 2) Intercepting of target JavaScript code is being done from background script of extension.

² <https://developer.chrome.com/extensions/webRequest>, last accessed 25-05-2017

³ <https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage>, last accessed 25-05-2017

4.2 Significant Data Structures

There are two important data structures defined in CRI. The first one is used while sending a message from content script to the panel and the other is used while saving the history of dependency graph into Chrome storage. Listing 4.1 shows those significant data structures.

In CRI, the communication between content script (event emitter) and the panel has a defined data structure. The message object has two attributes named as **content** and **action**. The **action** attributes define which type of event has occurred, and the **content** attribute contains detailed information about that message. Two important values of action attribute are **saveNode** and **saveEdge**.

In the case of **saveNode**, the **content** attribute contains **node** object and **node** object contains the following properties:

- **nodeId**: defines the unique id of the node.
- **nodeType**: specifies the type of reactive variable, it can be observable, eventStream or property. Different reactive libraries have their own implementations for time-changing variables.
- **nodeMethod**: determines the method name which results in this stream.
- **nodeRef**: contains the JavaScript variable name, identified by jalangi. It can be empty in the case of intermediate streams; those are not directly assigned to a variable.
- **nodeValue**: contains the current value of this reactive variable as a string.
- **sourceCodeLine**: holds the line number from the source code, where this reactive variable is defined.

In the case when **action** attribute of the message is set to **saveEdge**, the **content** attribute contains edge object and **edge** object is defined by the following properties:

- **edgeStart**: The node id of the parent reactive variable, on which another reactive variable depends.
- **edgeEnd**: The node id of the child reactive variable, which results in the application of some operator on parent reactive variable.
- **edgeLabel**: The name of the operator which causes this dependency.

```

1 // Message object in case of dependency creation
2 {
3   content: {
4     "edgeStart": '',
5     "edgeEnd": '',
6     "edgeLabel": ''
7   },
8   action: "saveEdge",
9   destination: "panel"
10 }
11
12 // Message object in case of new reactive stream defined / existing stream gets a
    new value
13 {
14   content: {
15     'nodeId': '',
16     'nodeType': '',
17     'nodeMethod': '',
18     'nodeRef': '',
19     'nodeValue': '',
20     'sourceCodeLine': ''
21   },
22   action: "saveNode",
23   destination: "panel"
24 }
25
26 // Data Structure for history of dependency graph
27 {
28   "stageId": '',
29   "stageEvent": '',
30   "stageCurrentNode": '',
31   "stageCurrentEdgeStart": '',
32   "stageCurrentEdgeEnd": '',
33   "stageData": {
34     "nodes": [],
35     "edges": []
36   }
37 }

```

Listing 4.1: Significant Data Structures used by CRI

History of dependency graph is saved to chrome storage as an array of **stage** objects, and **stage** object comprises of the following properties. Each stage represents a single step of the history of the dependency graph.

- **stageId**: represents the id of the stage/step for the history of the dependency graph.
- **stageEvent**: represents the event that causes this stage. (saveEdge, newNode, updateNode)
- **stageCurrentNode**: represents the id of the current node. It is used to highlight a node in the dependency graph.
- **stageCurrentEdgeStart**: If current stageEvent is saveEdge then this property contains the parent node id.
- **stageCurrentEdgeEnd**: If current stageEvent is saveEdge then this property contains the child node id.
- **stageData**: is an object that contains an array of all nodes and edges objects.

4.3 Communication between Extension Components

Since the content scripts run in the context of a web page and not the extension, they often need some way of communicating with the rest of the extension. In our case of CRI implementation, event sniffer script run as a content script. It has to forward the information about logged events from reactive libraries to the panel, which further reacts to those messages by updating dependency graph in the panel GUI and saving this information to browser storage for later access. Communication between extension components works by using message passing. Two possible ways to communicate between extension components are simple one-time requests and long-lived connections⁴. We use long-lived connections because event sniffer would send messages continuously. Extension's panel creates a channel to background script with the name **reactive-debugger** and adds listeners to receive messages from the background script. Listing 4.2 shows the relevant code that panel script uses to create a connection with background script.

```
1 //Create a port with background page for continuous communication
2 var port = chrome.extension.connect({
3   name: "reactive-debugger" //Given a Name
4 });
5 // Listen to messages from the background page
6 port.onMessage.addListener(function (message) {
7   // Here we get message from background,
8   // that gives us information about events like,
9   // New reative variable is defined / Dependency established / New value of
   reactive stream
10
11 });
```

⁴ <https://developer.chrome.com/extensions/messaging> , last accessed 26-05-2017

Listing 4.2: Panel Script to Communicate with Background

Listing 4.3 shows the relevant code for background script to communicate with content scripts and panel script. To handle incoming connections in background script, we set up an **extension.onConnect** event listener. When **extension.connect** is called from the panel script, **extension.onConnect** event is fired along with **runtime.port** object that we can use to send and receive messages through that connection. The **Event Emitter** module of CRI which also runs as content script send messages to background script by using **chrome.extension.sendMessage**, which is received by background script and forwarded to panel script.

As in the second alternative 4.1.2, the event emitter is injected into the web page context instead of running as content script so in this case the event emitter passes messages that it receives from event sniffer to the content script by using **window.postMessage**, which is received in content script **Event Reciever + Emitter** module. Beside direct communication between extension components by message passing, there is an indirect communication which is being done by **chrome.storage** API already explained in section 3.3.7.

```
1 chrome.extension.onConnect.addListener(function(port) {
2
3   // Listens to all incoming messages
4   // Message can be from content script or from panel
5   chrome.extension.onMessage.addListener(extensionListener);
6
7   // function to identify message source and forward its destination
8   var extensionListener = function(message, sender, sendResponse) {
9     // identify destination of message and forward it
10    if (message.destination == "panel") {
11      // message from content script so send it to panel
12      port.postMessage(message);
13    } else {
14      if (message.tabId && message.content) {
15        // message from panel and send to content script
16        chrome.tabs.sendMessage(message.tabId, message, sendResponse);
17      } else {
18        // message from content script and send to panel script
19        port.postMessage(message);
20      }
21    }
22    sendResponse(message);
23  }
24
25  // Remove message listner when we close DevTools window
26  port.onDisconnect.addListener(function(port) {
27    chrome.extension.onMessage.removeListener(extensionListener);
```

```
28    });  
29    });
```

Listing 4.3: Background Script to Communicate with Panel and Content Scripts

4.4 The Query Language

Two important features of the CRI extension are based on query language.

- A) Querying the dependency graph history after the execution of the application.
- B) Setting reactive-programming specific breakpoints.

Let's take a deeper look into the implementation of these two features.

The single commands are listed and explained in Table 4.1. The commands for feature **A** is based on nodeName's and for feature **B** commands are based on nodeId's. In the first feature, extension gets the query when the user submits the query and it parses the submitted query with regular expressions. By applying regular expression we identify the query and its parameters. After identifying the query and its parameters, we apply the search operation to the data of dependency graph. For the second feature, panel script stores the submitted queries in chrome storage and accesses it from the content script. The content script parses then those queries to apply breakpoint on the relevant place of the code.

Command	Parameters	Description
nodeCreated[nodeName]	nodeName: name of the node	specific node is created
nodeUpdated[nodeName]	nodeName: name of the node	specific node is updated, e:g node gets a new value
evaluationYielded[nodeName][value]	nodeName: name of the node value: a String	evaluation of specific node yields specific value
dependencyCreated[nodeName1][nodeName2]	nodeName1: name of the parent node nodeName2: name of the child node	dependency between two specific nodes is created

Table 4.1.: Commands of the Query Language

4.5 Graphical User Interface

The graphical user interface of the debugger is Chrome DevTools panel view which can be opened just like any other panel in DevTools. The GUI is basically divided into six parts as shown

in figure 4.3. It can be activated by opening Chrome DevTools and navigating to **Reactive-Inspector** panel.

The first part of the view just shows a text field to input the file names for scoping feature. In this field, one can enter JavaScript files name which should be considered for debugging. Multiple files name can be entered by separating them with a comma. If this particular field is left empty then the debugger will consider all JS files for debugging. The second part of the view contains some helper elements for the developer. One can search for a node by name within the dependency graph, the resultant node will be highlighted and remaining dependency graph will be faded out. One can also reset the current dependency graph and can start or pause debugging anytime. The current visualisation can also be downloaded as an image by pressing the **Download** button.

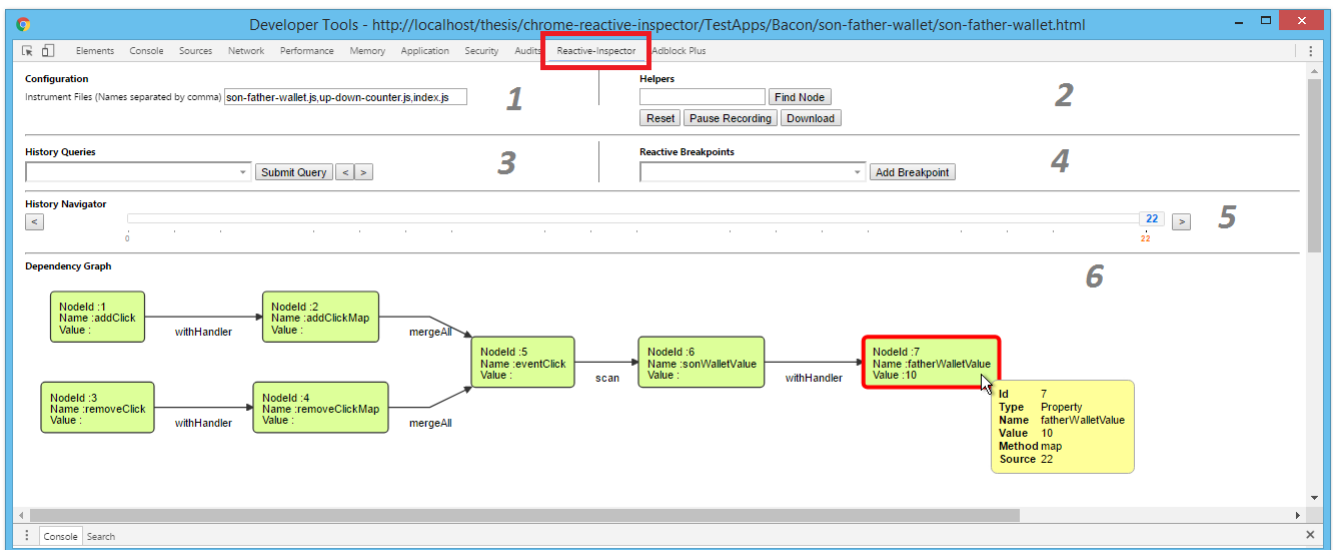


Figure 4.3.: CRI - Graphical User Interface

The third part of the view is to query the dependency graph, where the developer can enter the queries explained in section 4.4. If a valid query is entered and the **Submit Query** button is pressed, the visualisation will jump to the first point in time at which the entered query matches. In the case of more results to a query, one can use navigation buttons available in this part to traverse through query results. The fourth part of the view is to set reactive breakpoints using the same query language that is used to search through the history of the dependency graph. The fifth part of the view is to navigate through the history of dependency graph with the help of a slider. One can navigate step-by-step with the help of buttons available on the left and right side of the slider. One can also use the mouse by dragging and dropping the pointer of the slider to any step. Every change to slider will result in changes to the dependency graph to a specific state.

Last but not the least, the sixth part of the view contains the dependency graph. One can use the mouse to zoom in or out the current visualisation. Additional information to the nodes will be

presented as a tooltip, if the developer hovers with the mouse over a specific node. The current node is highlighted with a red border around node box in the dependency graph.

The visualisation of the dependency graph is based on `dagre-d3`⁵, which is a JavaScript library that acts as a front-end to `dagre`⁶, providing actual rendering using `D3`⁷.

⁵ <https://github.com/cpetitt/dagre-d3> , last accessed 25-05-2017

⁶ <https://github.com/cpetitt/dagre> , last accessed 25-05-2017

⁷ <https://d3js.org/>, last accessed 25-05-2017

5 Evaluation

This chapter presents the evaluation of the CRI extension. The goal of this is to determine how well the CRI works while debugging and understanding reactive web applications. All the applications used to evaluate the extension are taken from the internet, and they are built on RxJS or Bacon.js library. The first section defines a methodology for the evaluation and applies it to the pool of applications. The second section presents various case studies in detail, which demonstrate the use of the extension. Every subsection of the second section focuses on one feature and evaluates that feature with real applications. One application from both RxJS and Bacon.js is evaluated in every subsection. In the third section, some advanced web applications are debugged with CRI to check the robustness of CRI. The last section summarises and concludes the evaluation.

5.1 Feasibility of the Technical Approach

This section presents the evaluation and its result that was performed by assessing the feasibility of the technical approach. The goal is to determine how well CRI works together with real applications. We take 22 reactive web applications, based on RxJS and Bacon.js library, from the internet. We run these applications and tried to debug and analyse them with CRI.

5.1.1 Evaluated Web Applications

Because RP paradigm is still new to the web domain, it was very hard to find a large number of reactive web applications for the purpose of evaluation. However, we found 22 applications for evaluation (10 applications are using Bacon.js, and the rest 12 are using RxJS library). Evaluated applications are listed in Table 5.1. Along with the title of the application, we also present some details like how many lines of codes that application comprises of, how many Eventstreams or Observables are defined, and the number of used operators. More details of the evaluated applications are available in Appendix A.

As we have already seen in chapter 4, we are using Jalangi for instrumenting JavaScript code before executing it in the browser. Because jalangi does not support “**arrow functions**” (which comes as a new feature in ECMAScript 6 and is the new syntax of writing functions in JavaScript), we replaced the arrow function syntax with normal syntax from applications before the evaluation. Another important fact is that CRI only supports from version 5 of RxJS which is still very new. So some applications were based on earlier versions of RxJs, we modified those to make them work with version 5 before performing the evaluation.

No.	Application Title	Library	Code Lines	Streams	Unique Operators	Total Op-erators
1	Operators and Events	Bacon.js	74	8	7	15
2	Father-Son Wallet War	Bacon.js	19	2	3	5
3	Up-Down Counter	Bacon.js	8	2	3	4
4	Form Validation	Bacon.js	42	3	5	12
5	Movie Search	Bacon.js	34	3	4	5
6	Bar Chart	Bacon.js	81	1	3	3
7	WebSocket Wikipedia	Bacon.js	180	2	6	9
8	Smart Counter	RxJs	29	2	7	8
9	State Sotrage	RxJs	37	3	3	5
10	Letter Counter	RxJS	12	1	2	2
11	Father-Son Wallet War	RxJS	23	2	4	5
12	Movie Search	RxJS	33	2	2	5
13	Follow The Mouse	RxJS	45	1	8	2
14	Drag and Drop	RxJS	19	3	3	3
15	Canvas Painting	RxJS	62	6	8	10
16	Twitter Follow Box	RxJS	57	5	5	8
17	REST API Call	RxJS	30	1	3	3
18	Spotify Artist Search	RxJS	35	2	8	9
19	Image Sampler	RxJS	38	2	2	3
20	Mullti Select Cards	Bacon.js	79	3	5	15
21	True - False Logger	Bacon.js	44	7	5	16
22	Drawing App	Bacon.js	38	4	4	4

Table 5.1.: Evaluated Web Applications

5.1.2 Evaluation Criteria

We prepared a recipe for the evaluation that comprises of eight questions. The primary goal of these questions was to see whether the use of the CRI benefits the developer to understand and to find bugs in the target application. The summary of evaluation criteria is listed in Table 5.2. First of all, we evaluated if the behaviour of the target application remains unchanged while debugging it with the CRI. Secondly, we saw whether the CRI was able to model the target application into a dependency graph. Next, we observed that if the propagation of changes through nodes of the dependency graph are visible to the developer. We navigated through the history of the dependency graph for the target application to see if that helped us to understand the target application better. The fifth point in the evaluation recipe is to test RP specific break-

points. The next three steps are to gather some statistics about the outcomes of the CRI like getting the total number of nodes in the dependency graph, the total number of green nodes that represents the identified streams assigned to JS variables and the number of steps recorded while debugging.

Q1	While the CRI was active, did the target application run as it was running before?
Q2	Did the CRI deliver the dependency graph of the target application?
Q3	Was the propagation of changes to the dependency graph be visible?
Q4	Did traversing through the history of the dependency graph helps to understand the working of target application?
Q5	Did the execution of application pause, when we set a reactive breakpoint?
Q6	What is the total number of nodes in the dependency graph?
Q7	What is the total number of green nodes in the dependency graph?
Q8	What is the total number of recorded steps?

Table 5.2.: Evaluation Criteria

5.1.3 Evaluation Results

We performed the evaluation recipe, one by one to all of the applications listed in Table 5.1. Table 5.3 summarises the results of the evaluation. For application 7 and 8, the results are unstable, and this is because these applications contain observables that emit values after a very short time interval. The CRI tries to log all events and sends a message to the panel to update values to corresponding nodes of the dependency graph. In the case of application 7, there is a web socket connection, and the application receives all incoming updates and afterwards samples them with a sampling interval of 2 seconds. So while debugging this application, the stream that receives updates from the web socket connection also gets subscribed which then cause high resource consumptions and leads to an uncertain behaviour. In the case of application 8, there is an observable that emits values after every 20 milliseconds and causes high resource consumption because the extension communicate with panel after every 20 milliseconds.

To reproduce the evaluation, one should follow the following steps:

- Get and set up the target application. See Appendix A for more information for each application.
- Run the application without activating CRI extension to understand and observe its working.
- Check Appendix A for the required refactoring of the target application.
- Activate the CRI extension, open **reactive inspector** panel from Chrome DevTools and execute the target application.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
App-1	Yes	Yes	Yes	Yes	Yes	35	4	140
App-2	Yes	Yes	Yes	Yes	Yes	7	7	21+
App-3	Yes	Yes	Yes	Yes	Yes	6	5	16+
App-4	Yes	Yes	Yes	Yes	Yes	19	9	50+
App-5	Yes	Yes	Yes	Yes	Yes	42	5	125+
App-6	Yes	Yes	Yes	Yes	Yes	4	5	16+
App-7	Yes	Yes	Unstable	Unstable	Yes	14	15	20+
App-8	Yes	Yes	Unstable	Unstable	Unstable	7	0	16+
App-9	Yes	Yes	Yes	Yes	Yes	8	4	23+
App-10	Yes	Yes	Yes	Yes	Yes	3	4	27+
App-11	Yes	Yes	Yes	Yes	Yes	8	5	25+
App-12	Yes	Yes	Yes	Yes	Yes	6	6	21+
App-13	Yes	Yes	Yes	Yes	Yes	9	3	27+
App-14	Yes	Yes	Yes	Yes	Yes	8	4	7+
App-15	Yes	Yes	Yes	Yes	Yes	19	7	55+
App-16	Yes	Yes	Yes	Yes	Yes	32	9	134+
App-17	Yes	Yes	Yes	Yes	Yes	4	1	30+
App-18	Yes	Yes	Yes	Yes	Yes	9	5	29+
App-19	Yes	Yes	Yes	Yes	Yes	5	2	11+
App-20	Yes	Yes	Yes	Yes	Yes	54	6	118+
App-21	Yes	Yes	Yes	Yes	Yes	17	8	37+
App-22	Yes	Yes	Yes	Yes	Yes	13	4	50+

Table 5.3.: Evaluation Results

- Apply the evaluation criteria mentioned in Table 5.2.

5.2 Case Studies

In this section, various case studies are presented in detail, which demonstrate the use of the CRI extension. They have been chosen in order to illustrate the most important features of the extension. The goal of these case studies is to show how a developer new to RP could use the CRI to understand the target application and solve potentially complex problems. The case studies do not fix the bugs in the target application but often lead the developer into a direction to understand and find the bugs quickly.

5.2.1 Understanding Operators

Operators provided by reactive libraries has already been presented in section 2.2.3. While developing reactive applications, one needs to use different operators for transforming, combining, filtering, mathematical operations. To chose the correct operator, one needs to understand

its workings in detail. The CRI helps the developer to understand the abstract workings of operators. We used some operators provided by RxJs and Bacon libraries in the sample application and then tried to use the CRI to understand the working details of those operators. This section illustrates how the CRI helps to understand the working details of operators provided by reactive libraries with sample applications.

5.2.1.1 Bacon.js - Operators

Listing 5.1 shows a sample application that uses the filter, fold and scan operators of Bacon.js library. From line 1 to 13 in the source code, the usage of filter operator is shown. Line 5 uses **sequentially** to create a stream containing given values from an array delivered with a given interval in milliseconds. Line 6 to 8 apply filter operator on the created stream to filter odd values from the array, that results in a stream of values. Line 9 to 11 is the subscription of the filtered stream. Line 15 to 23, **fold** operator is used to calculate the sum of all elements of the array. Line 25 to 33, **scan** operator is used to calculate the sum of all elements of the array. Figure 5.1 shows the dependency graph of the target application of listing 5.1. When we click on the button **Run (filter)**, the filter operation can be observed in the dependency graph created by the CRI. Two nodes on the top represent filter operation. The first node represents the source stream that emits values from the array one by one after every 1000 milliseconds. The second node represents the filtered stream that emits only even values. On hovering over the node, a popup opens up with detailed information about that particular node. According to Bacon.js API documentation¹, fold operator is like **scan** operator but only emits the final value. This behaviour can be easily visualised in the dependency graph.

```
1 // Array data source
2 var arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
3 // Filter
4 $("#runBaconFilter").click(function () {
5     var baconSourceStream = Bacon.sequentially(1000, arr);
6     var baconFilteredEvenStream = baconSourceStream.filter(function (x) {
7         return x % 2 == 0;
8     });
9     baconFilteredEvenStream.onValue(function (val) {
10         $("#baconFilter").text(val);
11     });
12
13 });
14 // Sum of array elements with Fold
15 $("#runBaconFold").click(function () {
16     var baconStream = Bacon.sequentially(500, arr);
```

¹ <https://baconjs.github.io/api.html> , last accessed 22-05-2017

```

17  var baconFoldStream = baconStream.fold(0, function (x, acc) {
18      return x + acc;
19  });
20  baconFoldStream.onValue(function (val) {
21      $("#baconFold").text(val);
22  });
23  });
24  // Sum of array elements with Scan
25  $("#runBaconScan").click(function () {
26      var baconStream = Bacon.sequentially(500, arr);
27      var baconScanStream = baconStream.scan(0, function (x, acc) {
28          return x + acc;
29      });
30      baconScanStream.onValue(function (val) {
31          $("#baconScan").text(val);
32      });
33  });

```

Listing 5.1: Understanding Bacon.js Operators

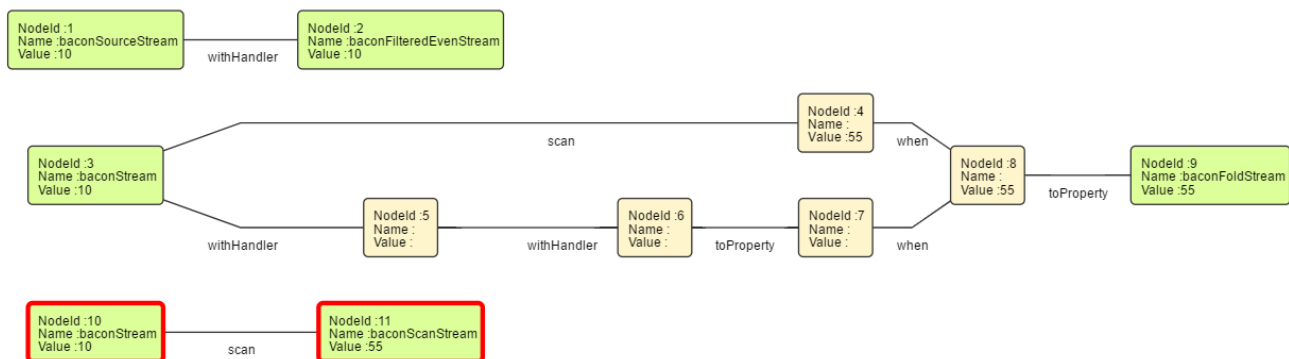


Figure 5.1.: Dependency Graph of Bacon.js Operators

5.2.1.2 RxJS - Operators

Listing 5.2 shows a sample application that makes use, **mapTo**, **delay** and **merge** operators from RxJS library. When the program is executed, the dependency graph created by the CRI is shown in figure 5.2. The dependency graph shows four observables emitting different strings with a specified delay value. Merging these four observables then results in a desired observable which is assigned to a variable called **message**. A subscriber of the resultant observable receives a string of values one by one with a specified delay.

```

1  //emit one item

```



```

2 var example = Rx.Observable.of(null);
3 //delay output of each by an extra second
4 var message = Rx.Observable.merge(
5   example.mapTo('Hello'),
6   example.mapTo('World!').delay(1000),
7   example.mapTo('Goodbye').delay(2000),
8   example.mapTo('World!').delay(3000)
9 );
10 //output: 'Hello'...'World!'...'Goodbye'...'World!'
11 var subscribe = message.subscribe(function (val) {
12   return console.log(val);
13 });

```

Listing 5.2: Understanding RxJS Operators

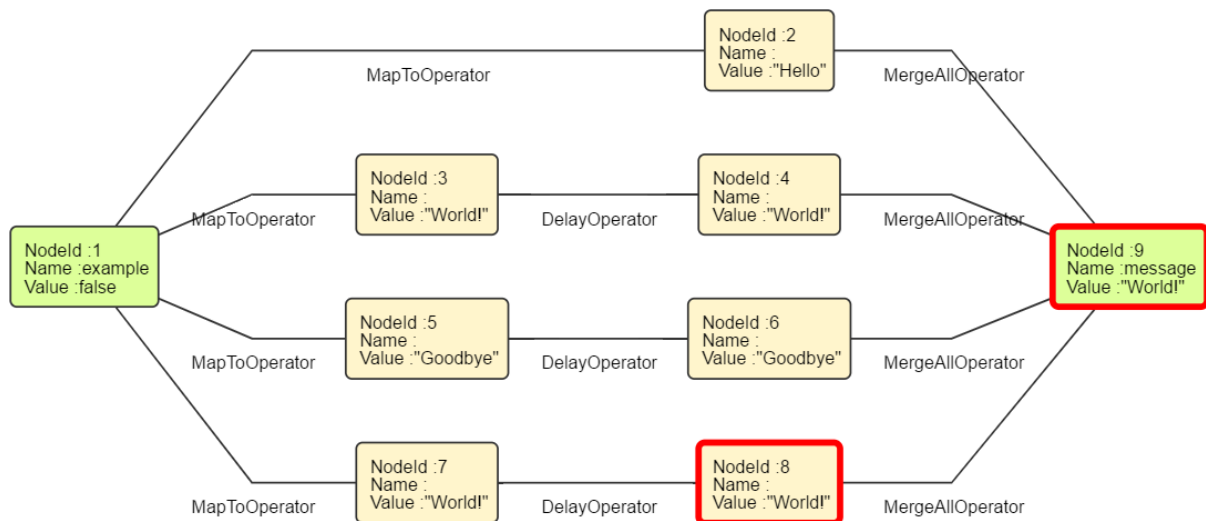


Figure 5.2.: Dependency Graph of Bacon.js Operators

5.2.2 Understanding Reactive Systems with Dependency Graph History

Besides modelling the reactive applications into a dependency graph, the storage of the whole dependency graph history and the possibility navigating through it are the important features of the CRI. One can use a simple slider to navigate through the history of the dependency graph as described in Section 4.5. With this navigation feature, one can see the evolution of the shape of the dependency graph. One can see how nodes are created and how dependencies are established. One can also observe how changes propagate through the dependency graph. The above mentioned features help to comprehend the execution of the reactive application step-by-step and dependencies among reactive variables are easily visible. Doing this with a native

DevTools 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. This section will further evaluate these features with some sample applications.

5.2.2.1 Bacon.js - Up-Down Counter

Listing 5.3 presents the relevant source code of an up-down counter application based on Bacon.js. GUI of this application contains two buttons (up and down) and one DOM element to display the final result. Figure 5.3 shows the evolution of the dependency graph when the target application is run by activating the CRI extension. Figure 5.4 represent the state of the dependency graph when the user clicks twice on the DOM element with id **up**. The complete application can be easily overviewed with the dependency graph. One can easily see that two streams are being merged and scan to deliver desired results.

```
1 var up = $('#up').asEventStream('click');
2 var down = $('#down').asEventStream('click');
3 var upClick = up.map(1);
4 var downClick = down.map(-1);
5 var counter = upClick.merge(downClick).scan(0, function (x, y) {
6   return x + y;
7 });
8 counter.assign($('#counter'), 'text');
```

Listing 5.3: Bacon.js - Up-Down Counter

5.2.2.2 RxJS - State Management

Let us look at another case from RxJS applications. This application manages the state of three DOM elements. The relevant code is in Listing 5.4. The **state** object comprises of **count** and **inputValue**. The **count** attribute of the state object is linked to two buttons, which increase and decrease the count value. **inputValue** is the current value of a text field. The dependency graph of this application is depicted in figure 5.5. One can clearly see in the dependency graph that three streams are being merged. The resultant stream is scanned to get the required stream which always emits the current state of the system on any change in the values of relevant DOM elements.

```
1 var increaseButton = document.querySelector('#increase');
2 var increase = Rx.Observable.fromEvent(increaseButton, 'click')
3 // Again we map to a function the will increase the count
4 .map(function () {
5   return function (state) {
```

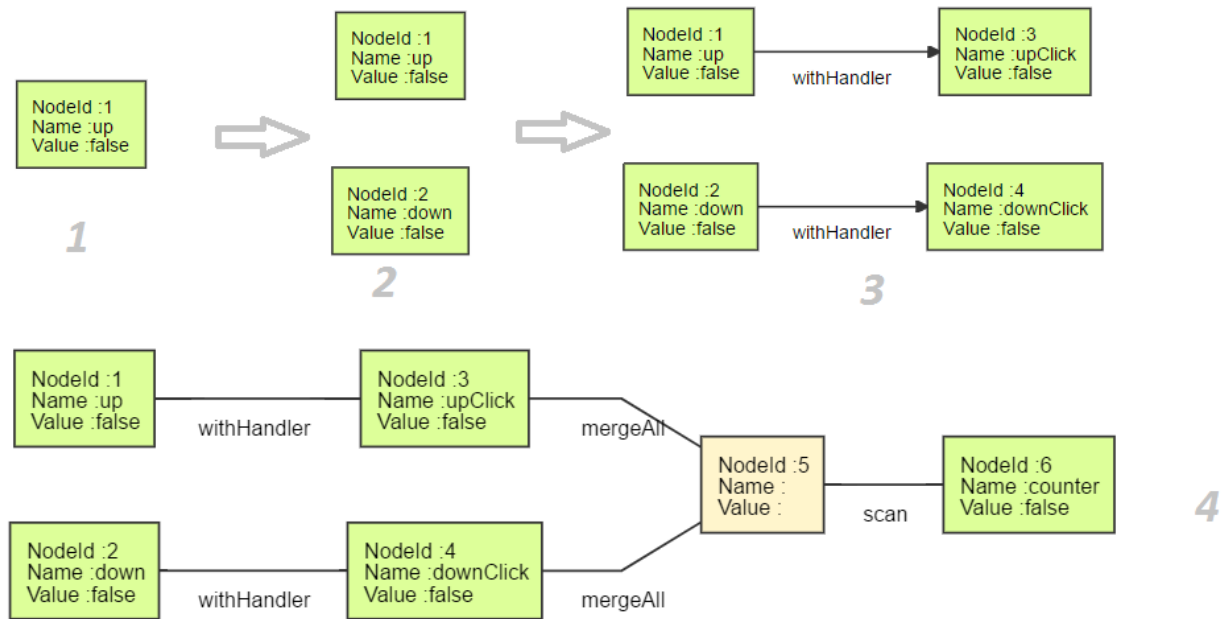


Figure 5.3.: Up-Down Counter - Evolution of the Dependency Graph

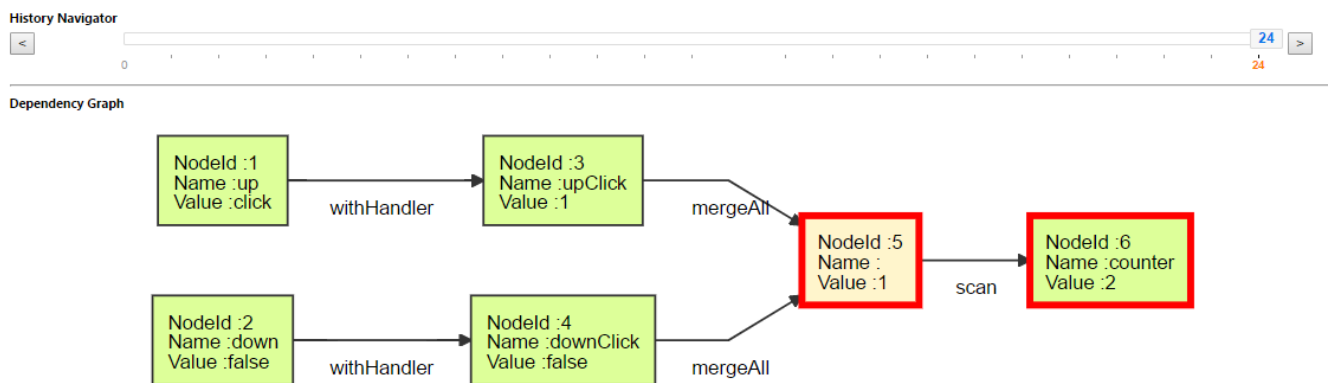


Figure 5.4.: Bacon.js - Up-Down Counter

```

6     return Object.assign({}, state, { count: state.count + 1 });
7   };
8 });
9
10 var decreaseButton = document.querySelector('#decrease');
11 var decrease = Rx.Observable.fromEvent(decreaseButton, 'click')
12 // We also map to a function that will decrease the count
13 .map(function () {
14   return function (state) {
15     return Object.assign({}, state, { count: state.count - 1 });
16   };
17 });
18
19 var inputElement = document.querySelector('#input');

```

```

20 var input = Rx.Observable.fromEvent(inputElement, 'keyup')
21 // Let us also map the keypress events to produce an inputValue state
22 .map(function (event) {
23     return function (state) {
24         return Object.assign({}, state, { inputValue: event.target.value });
25     };
26 });
27
28 // We merge the three state change producing observables
29 var state = Rx.Observable.merge(increase, decrease, input).scan(function (state,
    changeFn) {
30     return changeFn(state);
31 }, {
32     count: 0,
33     inputValue: '',
34 });
35
36 // We subscribe to state changes and update the DOM
37 // has actually changed
38 var prevState = {};
39 state.subscribe(function (state) {
40     if (state.count !== prevState.count) {
41         document.querySelector('#count').innerHTML = state.count;
42     }
43     if (state.inputValue !== prevState.inputValue) {
44         document.querySelector('#hello').innerHTML = 'Hello ' + state.inputValue;
45     }
46     prevState = state;
47 });

```

Listing 5.4: RxJS - State Management

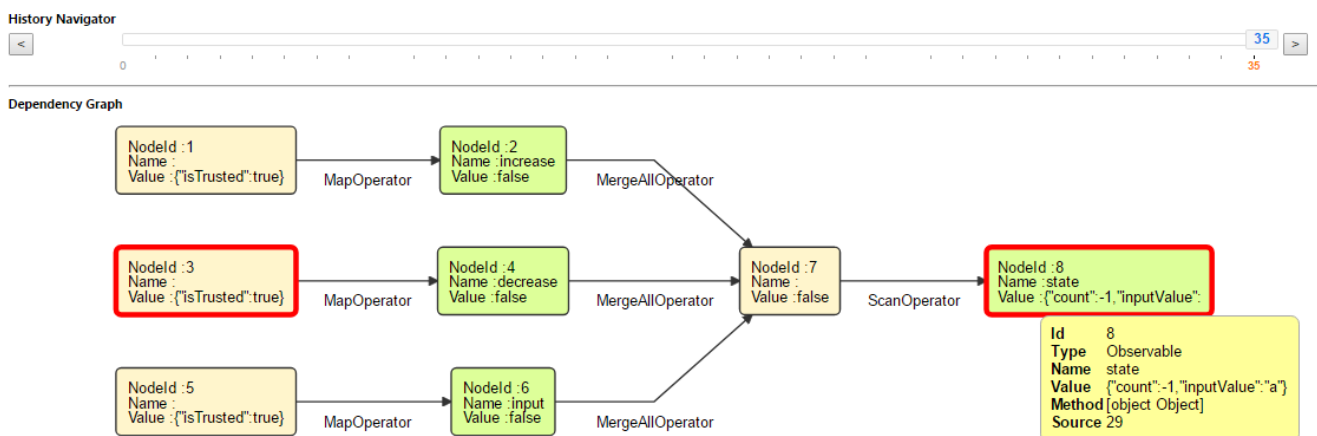


Figure 5.5.: RxJS - Dependency Graph of State Management

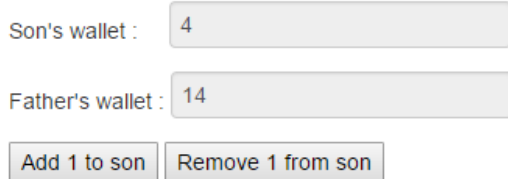
5.2.3 Querying the Dependency Graph History

In this section, we show how the developer can query the history of the dependency graph using query language presented in chapter 4. The developer can use this feature either to find bugs in the application or to observe the data flow through different nodes of the dependency graph. First, we execute the program and then we query the dependency graph history.

5.2.3.1 Bacon.js - Father and Son Wallet War

Let's now look at an interesting application called **Father and Son Wallet War**. This application is based on a rule that the father will always have ten dollars more than his son. The initial value of father wallet is ten. Figure 5.6 is a screenshot of the GUI of the application. Listing 5.5 show the relevant source code of this application. When we execute this application, the CRI extension models it into the dependency graph in 21 steps. Then we have done the following series of actions on the application +, +, +, -, +, +, +, -, -, + (where + represent one click on **Add 1 to son** and - represent one click on **Remove 1 from son** button) Figure 5.7 show the state of the dependency graph after applying this series of actions. Then we ran the following query to **History Queries** section of the CRI. `evaluationYielded[sonWalletValue][3]` We found three results that matched our query where the son wallet value was three at step 36, 46 and 66. Now we want to know that during execution of the program, how many node `addClickMap` results to 1, we execute the following query for that `evaluationYielded[addClickMap][1]`, which gives us 7 results, so it means add click button was clicked 7 times, which is true.

The father will always have 10 dollars more than his son.



Son's wallet : 4

Father's wallet : 14

Add 1 to son Remove 1 from son

Figure 5.6.: GUI - Father And Son Wallet War

```
1 $sonWallet = $('#wallet-son');
2 $fatherWallet = $('#wallet-father');
3 $addSon = $('#add-son');
4 $removeSon = $('#remove-son');
5
6 // change the value in the son's wallet
7 addClick = $addSon.asEventStream('click');
```

```

8 addClickMap = addClick.map(1);
9 removeClick = $removeSon.asEventStream('click');
10 removeClickMap = removeClick.map(-1);
11
12 eventClick = addClickMap.merge(removeClickMap);
13 function plus(a, b) {
14     return a + b
15 }
16 sonWalletValue = eventClick.scan(0, plus)
17 fatherWalletValue = sonWalletValue
18 .map(function (value) {
19     return value + 10
20 })
21 sonWalletValue.assign($sonWallet, "val");
22 fatherWalletValue.assign($fatherWallet, "val");

```

Listing 5.5: Bacon.js - Father And Son Wallet War

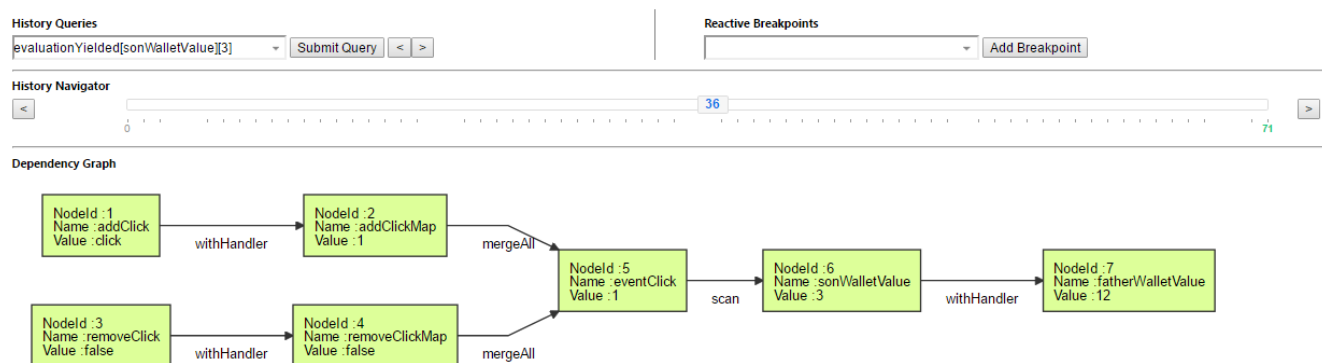


Figure 5.7.: History Query - Father And Son Wallet War

5.2.3.2 RxJS - Movie Search

Listing 5.6 shows a Movie Search application that makes HTTP GET request to the third party server and shows the results on the target page. Figure 5.8 shows the GUI of the target application. Keyup event on the text field is mapped to get the value of the text field, which is further filtered for checking the minimum length. **debounceTime** operator is used to limit the number of requests to the server. It emits values only after 750ms have passed since it emitted last time. Figure 5.9 shows the dependency graph of the target application. While debugging this application, we search through the history of the dependency graph using the query language to see what values at node **afterDebounce** are received and then we can easily adjust the debounce time value accordingly.

```

1 // Search Movies for a given term

```

```

2 function searchMovie(query) {
3   return jQuery.get(
4     '//api.themoviedb.org/3/search/movie?api_key=9eae05e667b4d5d9fbb75d27622347fe&
      query=' + query
5     ).then(function (r) {
6       return r.results;
7     });
8 }
9 var $input = $('#textInput');
10 // Get all distinct key up events from the input
11 // and only fire if long enough and distinct
12 var keyup = Rx.Observable.fromEvent($input, 'keyup');
13 var mappedValue = keyup.map(function (e) {
14   return e.target.value; // Project the text from the input
15 });
16 var filteredValue = mappedValue.filter(function (text) {
17   return text.length > 2; // Only if the text is longer than 2 characters
18 });
19 var afterDebounce = filteredValue.debounceTime(750 /* Pause for 750ms */);
20 var distinctUntilChnaged = afterDebounce.distinctUntilChanged(); // Only if the
      value has changed
21 // Final Stream
22 var searcher = distinctUntilChnaged.switchMap(searchMovie);

```

Listing 5.6: RxJS - Movie Search

Movie Search

Example to show combining input events such as keyup with Ajax requests

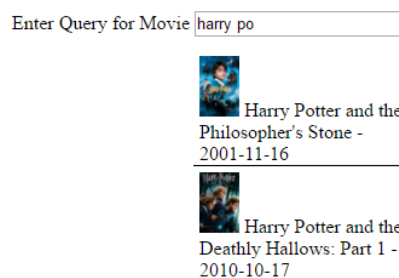


Figure 5.8.: RxJS - Movie Search UI

5.2.4 Reactive-Programming-Specific Breakpoints

For fast-running applications with a short life cycle, it is not a problem to find an interesting point in the history of the dependency graph after the program has completed its execution. There are situations where programs either take a long time to execute or the occurrence of the

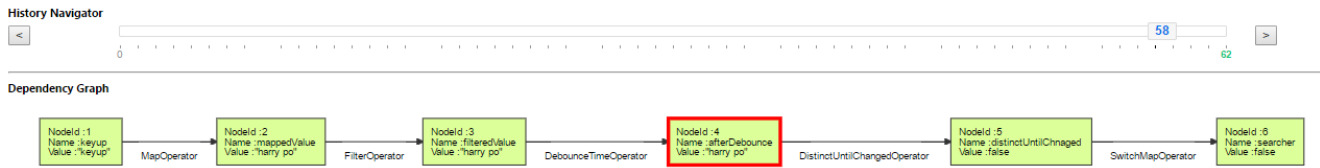


Figure 5.9.: RxJS - Dependency Graph of Movie Search

interesting event is uncertain. In such cases, it is very 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. In this section, we illustrate this with a few real applications. Before the program is executed, the query is defined. Then, the debugger is started as before. It will then pause the execution each time it matches to the defined queries. Let's have a look at a few applications to evaluate this feature.

5.2.4.1 Bacon.js - Registration Form

This application implements form validation logic in a reactive way. This application validates the availability of a username from the server by sending Ajax request. The full name field is also mandatory. Figure 5.10 shows the GUI of this application, the left-hand side shows the case where the given username is not available. The user gets a message about the unavailability of the username and the submit button remains disabled. The right side shows the case where the entered username is available and the full name is also given, so the submit button is enabled.



Figure 5.10.: Bacon.js - GUI of Registration Form

Listing 5.7 shows the source code of the target application. Figure 5.11 shows the dependency graph in the state when the form is validated and the submit button is enabled. We have observed a case where the entered username is available and full name is also given but still, the submit button is not enabled. We tried to debug this issue with the CRI extension. For this,

we set a reactive break point by defining a query `evaluationYeilded[3][false]`. While executing, we noticed that as soon as user typed any character from ä, ö or ü , the node with id 3 emit false. Figure 5.12 shows the screenshot of the debugging session where the program execution paused by the reactive breakpoint. We looked into the relevant code and found that there is a restriction of umlaut characters.

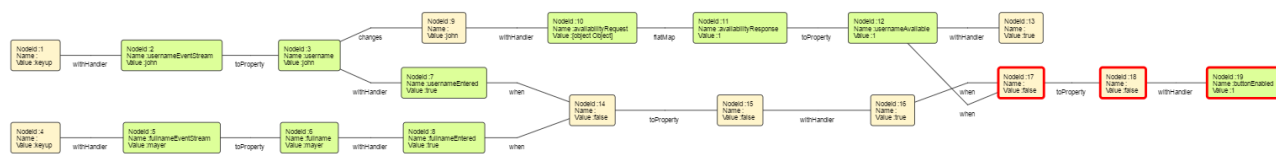


Figure 5.11.: Bacon.js - Dependency Graph of Registration Form Application

```

1 function show(x) { console.log(x) }
2 function nonEmpty(x) { return x.length > 0 }
3 function setVisibility(element, visible) {
4   element.toggle(visible)
5 }
6 var usernameEventStream = $("#username input").asEventStream("keyup").map(function
    (event) {
7   var typedValue = $(event.target).val();
8   // Umlaut Characters are not allowed
9   if( (typedValue.indexOf('ä') !== -1) || (typedValue.indexOf('ö') !== -1) || (
        typedValue.indexOf('ü') !== -1)){
10    return 'false';
11   }else{
12    return typedValue;
13   }
14 });
15 var username = usernameEventStream.toProperty("");
16 var fullnameEventStream = $("#fullname input").asEventStream("keyup").map(function
    (event) {
17   return $(event.target).val()
18 });
19 var fullname = fullnameEventStream.toProperty("");
20 var usernameEntered = username.map(nonEmpty)
21 fullnameEntered = fullname.map(nonEmpty)
22 function toResultStream(request) {
23   return Bacon.fromPromise($.ajax(request))
24 }
25 availabilityRequest = username.changes().map(function(user) { return { url: "/test
    .php?uname=" + user }}});
26 availabilityResponse = availabilityRequest.flatMap(toResultStream)
27 usernameAvailable = availabilityResponse.toProperty(true)
28 usernameAvailable.not().onValue(function (show) {
29   setVisibility($("#username-unavailable"), show);

```

```

30 })
31 var buttonEnabled = usernameEntered.and(fullnameEntered).and(usernameAvailable)
32 //var buttonEnabled = fullnameEntered.and(usernameAvailable)
33 buttonEnabled.onValue(function(enabled) {
34     $("#register button").attr("disabled", !enabled)
35 })

```

Listing 5.7: Bacon.js - Registration Form



Figure 5.12.: Bacon.js - Reactive Breakpoint in Registration Form Application

5.2.4.2 RxJS - Follow the Mouse Pointer

This application binds three DOM elements to the movement of the mouse pointer. Wherever the user moves the mouse pointer, those three DOM elements reposition themselves on the screen. Each DOM element changes its position with a certain delay which is defined in source code. Listing 5.8 shows the source code of target application. Figure 5.13 depicts the dependency graph of this application, where the user can see the defined streams and operators applied to those streams. The features of reactive breakpoints can be useful in this kind of application where the application does not have a limited life cycle.

```

1 (function () {
2   function extractClientX(e) { return e.clientX; }
3   function extractClientY(e) { return e.clientY; }
4   function setLeft(x) { this.style.left = x + 'px'; }
5   function setTop(y) { this.style.top = y + 'px'; }
6   function add(x, y) { return x + y; }
7   var partialAdd = function (x) { return add.bind(null, x); }

```

```

8  var delay = 300;
9  var mousemove = Rx.Observable.fromEvent(document, 'mousemove');
10 var left = mousemove.map(extractClientX);
11 var top = mousemove.map(extractClientY);
12 // Update the mouse
13 var themouse = document.querySelector('#themouse');
14 left.subscribe(setLeft.bind(themouse));
15 top.subscribe(setTop.bind(themouse));
16 // Update the tail
17 var mouseoffset = themouse.offsetWidth;
18 var thetail = document.querySelector('#thetail');
19 left.map(partialAdd(mouseoffset))
20 .delay(delay)
21 .subscribe(setLeft.bind(thetail));
22 top.delay(delay)
23 .subscribe(setTop.bind(thetail));
24 // Update wagging
25 var wagDelay = delay * 1.5;
26 var wagging = document.querySelector('#wagging');
27 var mouseandtailoffset = mouseoffset + thetail.offsetWidth;
28 left.map(partialAdd(mouseandtailoffset))
29 .delay(wagDelay)
30 .subscribe(setLeft.bind(wagging));
31 top.delay(wagDelay)
32 .subscribe(setTop.bind(wagging));
33 }());

```

Listing 5.8: RxJS - Follow the Mouse Pointer

5.3 More Advance Applications

This section will evaluate the robustness of the CRI extension with some advanced applications.

5.3.1 RxJS - Canvas Painting Board

This application is built with RxJS. Figure 5.14 shows the GUI of Canvas Painting Board application. The user can select a colour and can draw on the canvas with the mouse. Listing 5.9 shows the relevant source code. The target application can easily understand by viewing its dependency graph. Figure 5.15 shows a snapshot of the dependency graph. The user can easily see defined streams and applied operators. Every event while drawing something on canvas can be observed in the dependency graph.

```

1  Rx.DOM = {};

```

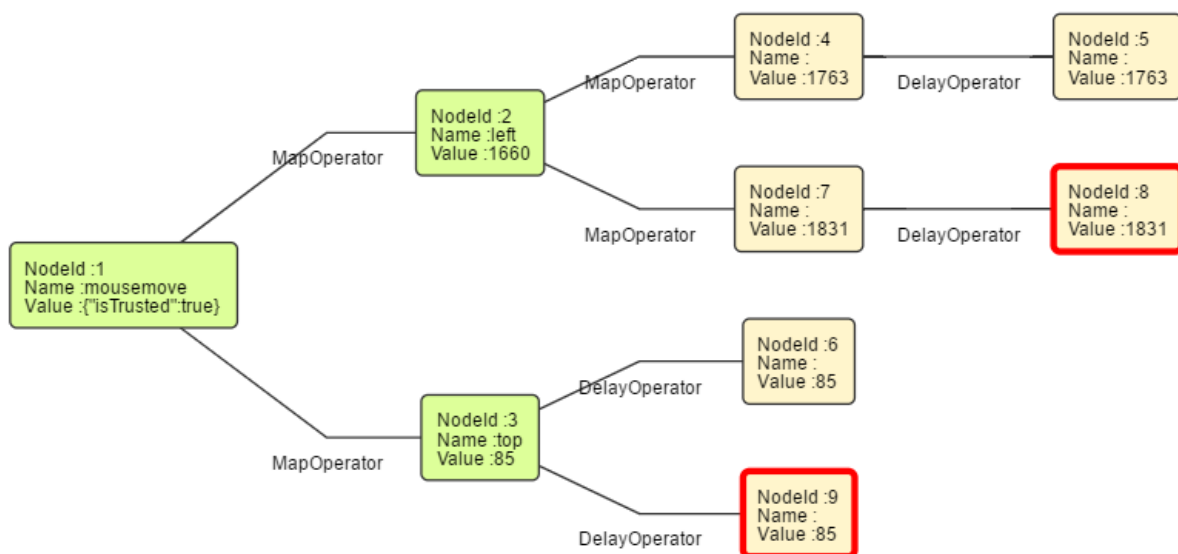


Figure 5.13.: RxJS - Follow the Mouse Pointer

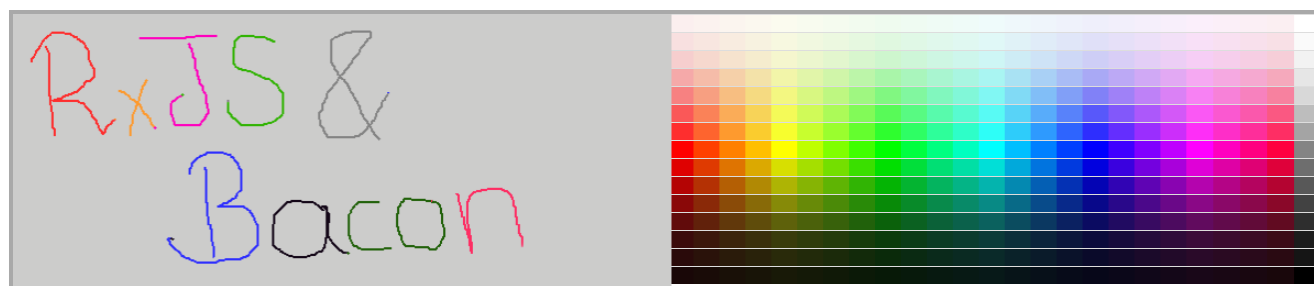


Figure 5.14.: RxJS - Canvas Painting Board

```

2 ['mousemove', 'mouseup', 'mousedown', 'click'].forEach(function (name) {
3   Rx.DOM[name] = function (element, selector) {
4     return Rx.Observable.fromEvent(element, name, selector);
5   };
6 });
7 Rx.DOM.ready = function () {
8   return Rx.Observable.create(function (o) {
9     function handler() {
10       o.next();
11       o.complete();
12     }
13     var added = false;
14     if (document.readyState === 'complete') {
15       handler();
16     } else {

```

```

17     console.log("else");
18     document.addEventListener('DOMContentLoaded', handler, false);
19 }
20 }).take(1);
21 };
22
23 // Calcualte offset either layerX/Y or offsetX/Y
24 function getOffset(event) {
25     return {
26         offsetX: event.offsetX === undefined ? event.layerX : event.offsetX,
27         offsetY: event.offsetY === undefined ? event.layerY : event.offsetY
28     };
29 }
30
31 function intialize() {
32     var canvas = document.getElementById('tutorial');
33     var colorchart = document.querySelectorAll('#colorchart tr td');
34     var ctx = canvas.getContext('2d');
35     ctx.beginPath();
36
37     // Get mouse events
38     var mouseMoves = Rx.DOM.mousemove(canvas),
39     mouseDowns = Rx.DOM.mousedown(canvas),
40     mouseUps = Rx.DOM.mouseup(canvas);
41
42     // Get the table events
43     var colorValues = Rx.DOM.click(colorchart)
44     .do(function () {
45         ctx.beginPath();
46     })
47     .map(function (e) {
48         return e.target.bgColor;
49     })
50     .startWith('#000000');
51
52     // Calculate difference between two mouse moves
53     var mouseDiffs = mouseMoves.zip(mouseMoves.skip(1), function (x, y) {
54         return {first: getOffset(x), second: getOffset(y)};
55     });
56
57     // Get merge together both mouse up and mouse down
58     var mouseButton = mouseDowns.mapTo(true)
59     .merge(mouseUps.mapTo(false));
60
61     // Paint if the mouse is down

```

```

62 var paint = mouseButton.switchMap(function (down) {
63     return down ? mouseDiffs : Rx.Observable.never();
64 })
65 .combineLatest(colorValues, function (pos, color) {
66     return {pos: pos, color: color};
67 });
68
69 // Update the canvas
70 paint.subscribe(function (x) {
71     ctx.strokeStyle = x.color;
72     ctx.moveTo(x.pos.first.offsetX, x.pos.first.offsetY);
73     ctx.lineTo(x.pos.second.offsetX, x.pos.second.offsetY);
74     ctx.stroke();
75 });
76 }
77 Rx.DOM.ready().subscribe(initialize);

```

Listing 5.9: RxJS - Canvas Painting Board

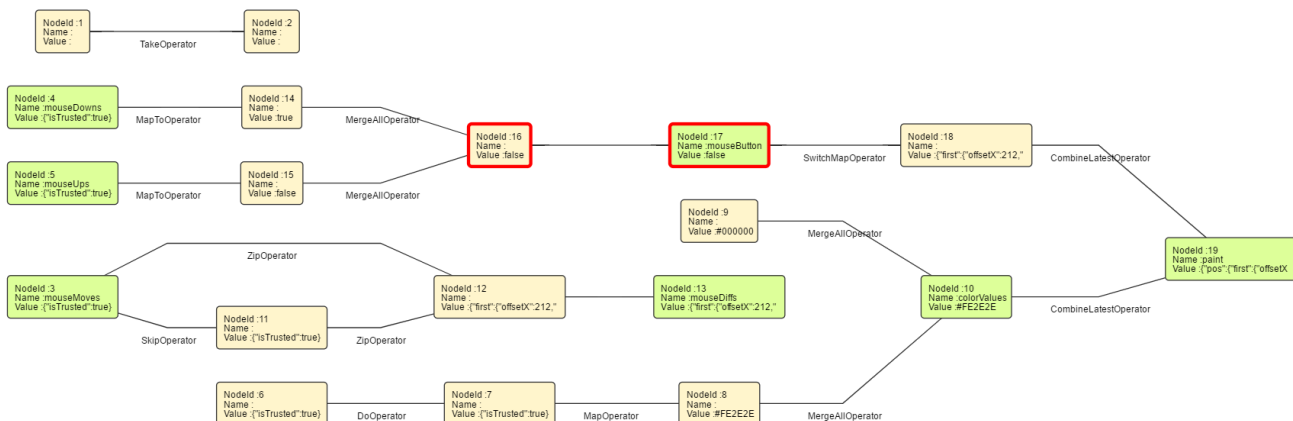


Figure 5.15.: Bacon.js - Dependency Graph of Canvas Painting Board

5.3.2 RxJS - Who to Follow

The Listing 5.10 shows the relevant part of a **Who to Follow** application that is based on RxJS. This application imitates the core features of Twitter UI element that suggests other accounts you could follow. Figure 5.17 depicts the GUI of this application. This application loads accounts data from the API on start up and displays three suggestions. When the user clicks on **Refresh**, it loads three other account suggestions into the three rows. Clicking on the **x** button on an account row will remove only that current account and display another suggestion.

There are five different event streams defined, and five different operators are being used. If we take a closer look into its source code, it seems very short and abstract. It is very hard to

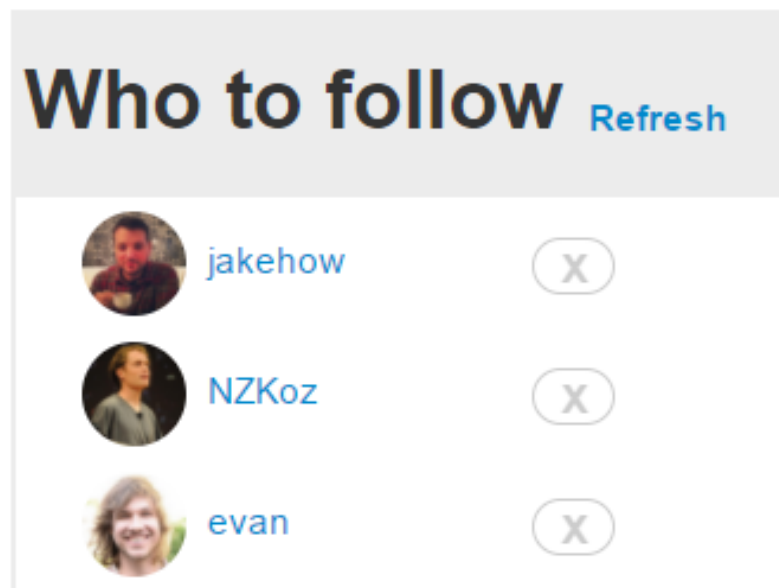


Figure 5.16.: RxJS - Who to Follow

understand this kind of abstract code by just looking into the code. We used the CRI extension to get a better understanding of this abstract source code. Figure 5.17 shows the dependency graph generated by the CRI. The dependency graph and the propagation of changes while interacting with the application gives a broad understanding of the source code of this application.

```
1 // UI Elements Selectors
2 var refreshButton = document.querySelector('.refresh');
3 var closeButton1 = document.querySelector('.close1');
4 var closeButton2 = document.querySelector('.close2');
5 var closeButton3 = document.querySelector('.close3');
6 // Click Streams
7 var refreshClickStream = Rx.Observable.fromEvent(refreshButton, 'click');
8 var close1ClickStream = Rx.Observable.fromEvent(closeButton1, 'click');
9 var close2ClickStream = Rx.Observable.fromEvent(closeButton2, 'click');
10 var close3ClickStream = Rx.Observable.fromEvent(closeButton3, 'click');
11 // Request Stream
12 var requestStream = refreshClickStream.startWith('startup click')
13 .map(function() {
14     var randomOffset = Math.floor(Math.random()*500);
15     // Just so we don't max out the anonymous github api req
16     // limit, I've cached a page. We'll pretend for now.
17     // To really hit the API, use
18     // 'https://api.github.com/users?since=' + randomOffset;
19     return 'users.json'
20 });
```

```

21 // Response Stream
22 var responseStream = requestStream
23   .flatMap(function (requestUrl) {
24     return Rx.Observable.fromPromise($.getJSON(requestUrl));
25   });
26 // Create Suggestion Stream
27 function createSuggestionStream(closeClickStream) {
28   return closeClickStream.startWith('startup click')
29     .combineLatest(responseStream,
30       function(click, listUsers) {
31         return listUsers[Math.floor(Math.random()*listUsers.length)];
32       }
33     )
34     .merge(
35       refreshClickStream.map(function(){
36         return null;
37       })
38     )
39     .startWith(null);
40 }
41 // Creating Streams
42 var suggestion1Stream = createSuggestionStream(close1ClickStream);
43 var suggestion2Stream = createSuggestionStream(close2ClickStream);
44 var suggestion3Stream = createSuggestionStream(close3ClickStream);
45
46 // Rendering
47 function renderSuggestion(suggestedUser, selector) {
48   var suggestionEl = document.querySelector(selector);
49   if (suggestedUser === null) {
50     suggestionEl.style.visibility = 'hidden';
51   } else {
52     suggestionEl.style.visibility = 'visible';
53     var usernameEl = suggestionEl.querySelector('.username');
54     usernameEl.href = suggestedUser.html_url;
55     usernameEl.textContent = suggestedUser.login;
56     var imgEl = suggestionEl.querySelector('img');
57     imgEl.src = "";
58     imgEl.src = suggestedUser.avatar_url;
59   }
60 }
61 // Subscription to Streams
62 suggestion1Stream.subscribe(function (suggestedUser) {
63   renderSuggestion(suggestedUser, '.suggestion1');
64 });
65 suggestion2Stream.subscribe(function (suggestedUser) {

```



```

66   renderSuggestion(suggestedUser, '.suggestion2');
67 });
68 suggestion3Stream.subscribe(function (suggestedUser) {
69   renderSuggestion(suggestedUser, '.suggestion3');
70 });

```

Listing 5.10: RxJS - Who to Follow

5.3.3 Bacon.js - Drawing Bar Chart

```

1  var initial = {
2    time: 1297110663,
3    value: 70
4  }
5
6  function walk(prev, rand) {
7    return {
8      time: prev.time + 1,
9      value: ~~Math.max(10, Math.min(90, prev.value + 10 * (rand - .5)))
10   };
11 }
12
13 var intervalStream = Bacon.interval(15000);
14 var mappedToRandom = intervalStream.map(Math.random);
15 var ScanTo = mappedToRandom.scan(initial, walk);
16 var withSlidingWindow = ScanTo.slidingWindow(10)
17 withSlidingWindow.onValue(redraw);

```

Listing 5.11: Bacon.js -Drawing Bar Chart

Drawing bar chart is an application based on Bacon.js, which draws bar chart to the target web page. After every 15000 milliseconds, it adds a bar to the chart with random height. The Listing 5.10 shows the relevant code of the target application. Figure 5.18 is a screenshot of a debugging session with this application, where the left side depicts the GUI of the application, and the right side shows the dependency graph. With the passage of time, we can observe the values emitted by observables by hovering over the nodes of the dependency graph.

5.3.4 Bacon.js - Live Wikipedia Updates Over Websockets

```

1
2  // Create our websocket to get wiki updates
3  var ws = new WebSocket("ws://wiki-update-sockets.herokuapp.com/");
4  var updateStream = Bacon.fromEventTarget(ws, "message").map(function(event) {

```

```

5   var dataString = event.data;
6   return JSON.parse(dataString);
7   });
8
9   // Filter the update stream for newuser events
10  var newUserStream = updateStream.filter(function(update) {
11      return update.type === "newuser";
12  });
13  newUserStream.onValue(function(results) {
14      var format = d3.time.format("%X");
15      updateNewUser(["New user at: " + format(new Date())]);
16  });
17
18  // Filter the update stream for unspecified events, which we're taking to mean
19  // edits in this case
20  var editStream = updateStream.filter(function(update) {
21      return update.type === "unspecified";
22  });
23  editStream.onValue(function(results) {
24      updateEditText(["Last edit: " + results.content]);
25  });
26
27  // Calculate the rate of updates over time
28  var updateCount = updateStream.scan(0, function(value) {
29      return ++value;
30  });
31
32  var sampledUpdates = updateCount.sample(samplingTime);
33  var rateUpdates = sampledUpdates
34      .map(function (value) {
35          // timestamp all samples
36          return {date: new Date(), value: value};
37      })
38      .slidingWindow(2)
39      .skip(2) // we'll ignore the first two results, they don't contain
40      enough samples to determine a rate
41      .map(function(updates) {
42          // Determine rate of the last two samples
43          var previous = updates[0];
44          var current = updates[1];
45          return {
46              x: current.date,
47              y: (current.value - previous.value) / (current.date - previous.date)
48              * 1000 // Delta is in ms, but we need s.
49          }

```

```
48         //
49     });
50
51     var graphData = rateUpdates
52     .slidingWindow(maxNumberOfDataPoints)
53     .onValue(function (updates) {
54         update(updates)
55     });
```

Listing 5.12: Bacon.js - Live Wikipedia Updates Over Websockets

Listing 5.12 shows an excerpt from a Bacon.js application that creates a websocket connection to third party server. This websocket connection receives messages that contain different types of update notifications from Wikipedia. The stream of recieved messages is converted into EventStream named as **updateStream**. Target application filter **updateStream** further into three streams, which are **newUserStream**, **editStream** and **updateCount**. These filtered streams are used to create a graph that updates on the fly and updates other information to the target web page. Figure 5.19 and 5.20 shows the GUI and dependency graph of the target application.

5.4 Summary of the Evaluation

While these case studies do not prove the CRI extension's performance, they demonstrate its usefulness when debugging real web applications containing reactive code. In these case studies, the user was able to get a better grasp of the internal workings of the target application while quickly viewing the dependency graph and being able to navigate through the history of the dependency graph. These case studies indirectly present some of the deficiencies of the CRI extension as well. Firstly, if the target application has some event streams that emits values after a very short time interval, this can be very resource and memory consuming. Secondly, if the target application code contains arrow functions syntax, then it might not work because Jalangi does not support it. The CRI extension offers powerful tools, demonstrated in these case studies, that could make the CRI more efficient than native DevTools. Our extension works fine with most of the applications we took from the internet, and we are confident to say that the technical approach we have used would work more accurately with a little bit of fine tuning.

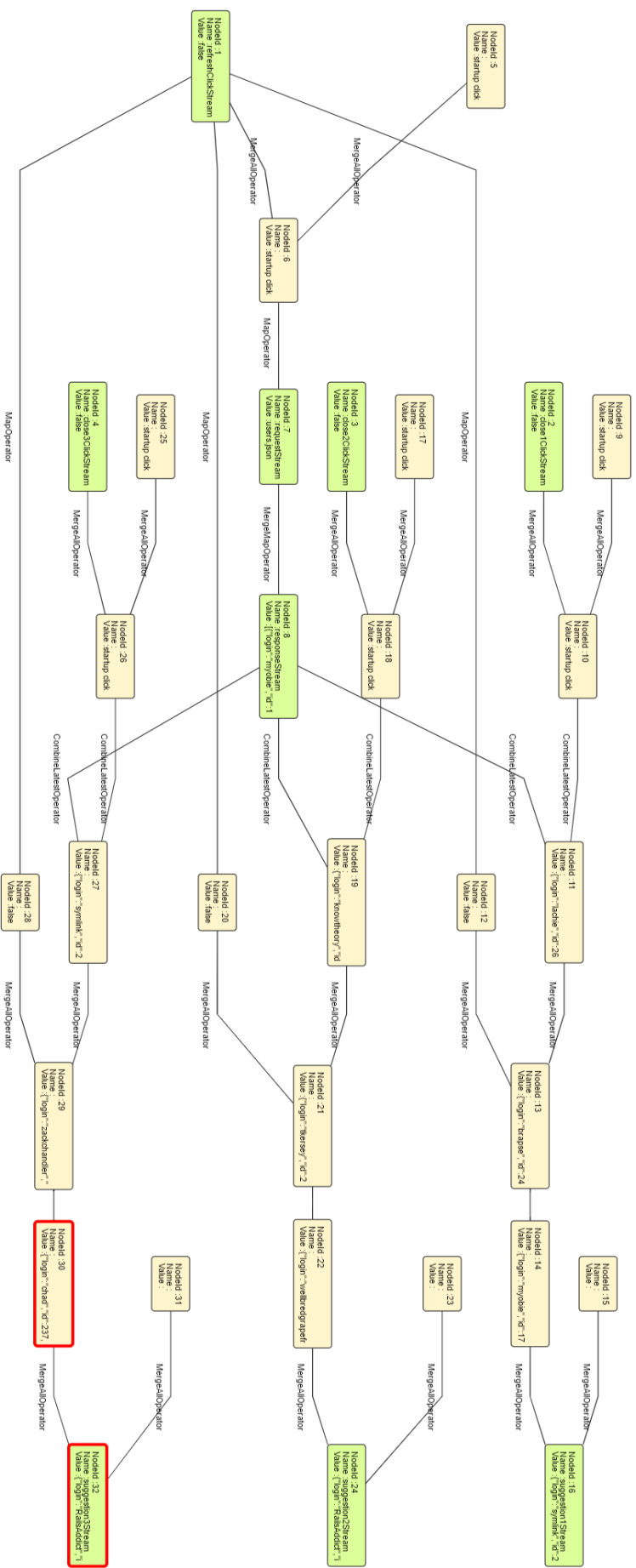


Figure 5.17.: RxJS - Dependency Graph of Who to Follow

d3 barchart with bacon

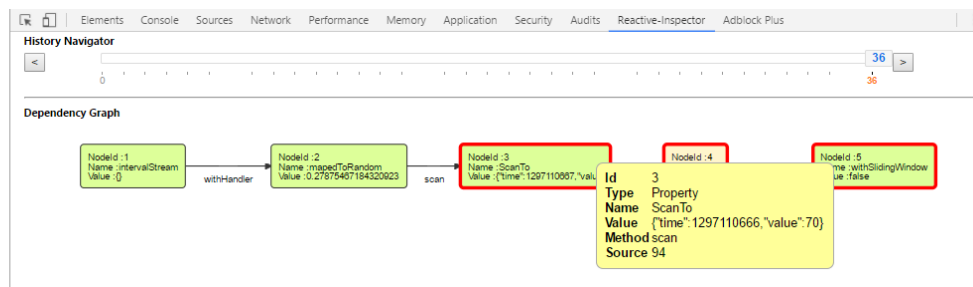


Figure 5.18.: Bacon.js - UI and Dependency Graph of Drawing Bar Chart

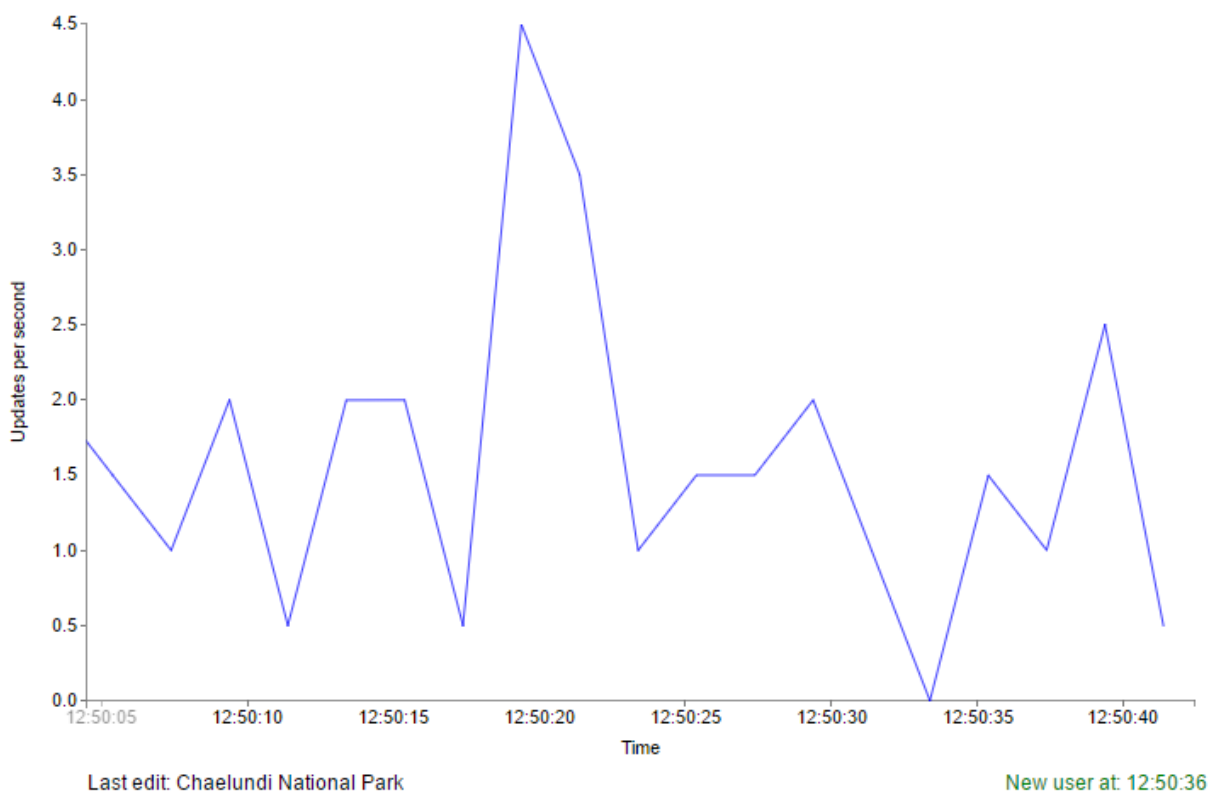


Figure 5.19.: Bacon.js - Live Wikipedia Updates Over Websockets

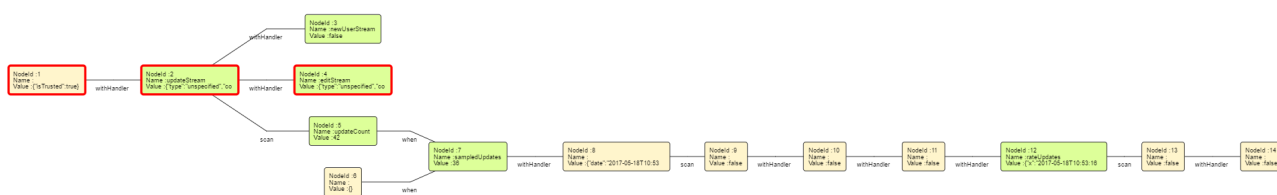


Figure 5.20.: Bacon.js - Dependency Graph of Live Wikipedia Updates Over Websockets



6 Summary

This chapter presents a short summary of the main contributions along with the implications and results of this thesis. Additionally, various ideas for future research and new features for the developed debugger are proposed.

6.1 Conclusion

The main goal of this thesis is to improve the debugging process for reactive applications based on JavaScript reactive libraries by designing and implementing a new developer tool for debugging. Developer tools have been found to be useful in supporting programmers in software comprehension and debugging, thus improving their productivity.

The debugging tool developed in this thesis is a result of an inspiration from Reactive Inspector, which is a debugger for reactive programs integrated with the Eclipse Scala IDE that allows debugging software in the reactive style. The contribution of this thesis is to implement the same technique in the web domain and thus develop an extension to Chrome DevTools to assist in the debugging process for RP. CRI (Chrome Reactive Inspector), a debugging tool for JavaScript reactive libraries like RxJS and Bacon.js, is designed and implemented in this thesis. The tool models reactive application code into dependency graphs, which has proved to be useful for understanding applications overall. Additionally, CRI also provides the possibility to see the history of dependency graphs which gives a clear view of data flow in the application. Moreover, reactive-programming-specific breakpoints have been developed. They allow breaking the program execution as soon as a relevant event occurs.

CRI is implemented specifically for JavaScript libraries RxJS and Bacon.js as an extension to Chrome DevTools. However, the technical approach used by the developed system can easily be used to add support for other JavaScript reactive libraries as well as to build similar tools for other browsers like an extension to Firebug of Firefox browser. We believe that the system developed in this thesis removes one of the biggest hurdles in the increasing popularity of RP in the web domain by providing a debugging tool. As demonstrated in chapter 5, the CRI improves and simplifies developer workflow while developing reactive web applications. In the future, the productivity improvements of Chrome Reactive Inspector should be assessed by conducting a user study.

6.2 Future Work and New Features

Regarding future research, it would be good to perform a user study on how the CRI helps developers to understand reactive systems and to find bugs in them. User surveys can help to get feedback on required new features and improvement to the existing features.

In chapter 4, we presented two alternative implementations of CRI extension, it would be very useful to have a deep look into both alternatives, and recommend one of them for future use. As we used Jalangi framework to map reactive values to JavaScript variables. Because our extension intercepts application code and instruments it with Jalangi framework, it creates performance overhead and we do not have the original code in DevTools to refer to. Although Jalangi instrumentation is optional in the debugging process and we can also limit instrumentation with scoping feature, it would be nice to find some other way to find reference of reactive values to JavaScript variables more elegantly.

CRI is one of the very first debugging tools for RP in the web domain and has much potential for future research and new features. In fact, the current implementation can be used as the foundation, on top of which new features can be built. New features that could be added to CRI include:

- It would also be worthwhile to implement the same extension for other browsers like Firefox and Safari.
- We already have Scoping feature, that only excludes/include JS files for instrumentation. This feature can be extended to exclude/include files or some event types from logging, this will simplify the dependency graph and help to boost performance.
- The extension could also be made more interactive by allowing developers to set or edit values of nodes at any specific time.
- The extension could also present the performance analysis of the application under inspection. It would be very helpful to see how many times a node is evaluated or the amount of time these evaluations take.
- Although current implementation gives the node reference to the line number in the code but it would be helpful if the user can jump to the code in DevTools by clicking on the node in the dependency graph.
- Currently, CRI only keeps the history of a dependency graph until the DevTools window is opened. It would be nice to have the ability to save debugging sessions to share with other developers later.
- As presented in chapter 5, in the background of CRI extension, every intermediate stream is subscribed that updates the dependency graph. This can sometimes cause performance

overhead. In the future, a useful feature could be allowing a developer to activate and deactivate logging events of nodes of dependency graph on a from an individual node level.



List of Figures

2.1. Reactive pattern [47]	12
2.2. How Jalangi Works	18
2.3. Chrome Developer Tools	19
2.4. The Structure of Chrome DeveTools Extension	20
3.1. Simple Use Case Diagram	24
3.2. System Components Overview	25
3.3. Detailed System Architecture	25
3.4. Event Sniffing - Override Calls To Reactive Library with AOP	26
3.5. Event Sniffing - Implementing Interfaces Provided By Reactive Library	27
3.6. Propagation of Changes through Dependency Graph	28
4.1. System Components Detail - Alternative 1	34
4.2. System Components Detail - Alternative 2	35
4.3. CRI - Graphical User Interface	41
5.1. Dependency Graph of Bacon.js Operators	48
5.2. Dependency Graph of Bacon.js Operators	49
5.3. Up-Down Counter - Evolution of the Dependency Graph	51
5.4. Bacon.js - Up-Down Counter	51
5.5. RxJS - Dependency Graph of State Management	52
5.6. GUI - Father And Son Wallet War	53
5.7. History Query - Father And Son Wallet War	54
5.8. RxJS - Movie Search UI	55
5.9. RxJS - Dependency Graph of Movie Search	56
5.10. Bacon.js - GUI of Registration Form	56
5.11. Bacon.js - Dependency Graph of Registration Form Application	57
5.12. Bacon.js - Reactive Breakpoint in Registration Form Application	58
5.13. RxJS - Follow the Mouse Pointer	60
5.14. RxJS - Canvas Painting Board	60
5.15. Bacon.js - Dependency Graph of Canvas Painting Board	62
5.16. RxJS - Who to Follow	63
5.17. RxJS - Dependency Graph of Who to Follow	68
5.18. Bacon.js - UI and Dependency Graph of Drawing Bar Chart	69
5.19. Bacon.js - Live Wikipedia Updates Over Websockets	69

5.20.Bacon.js - Dependency Graph of Live Wikipedia Updates Over Websockets	69
--	----

Listings

2.1. RxJS Simple Example	13
2.2. Bacon.js Example	15
2.3. Jalangi Instrumentation	18
3.1. Bacon.js Example of Up - Down Counter	28
3.2. Injecting Script into Web Page Context	31
4.1. Significant Data Structures used by CRI	37
4.2. Panel Script to Communicate with Background	38
4.3. Background Script to Communicate with Panel and Content Scripts	39
5.1. Understanding Bacon.js Operators	47
5.2. Understanding RxJS Operators	48
5.3. Bacon.js - Up-Down Counter	50
5.4. RxJS - State Management	50
5.5. Bacon.js - Father And Son Wallet War	53
5.6. RxJS - Movie Search	54
5.7. Bacon.js - Registration Form	57
5.8. RxJS - Follow the Mouse Pointer	58
5.9. RxJS - Canvas Painting Board	59
5.10. RxJS - Who to Follow	63
5.11. Bacon.js -Drawing Bar Chart	65
5.12. Bacon.js - Live Wikipedia Updates Over Websockets	65



Bibliography

- [1] J. Ashkenas: Underscore.js. <http://underscorejs.org>. last accessed: 10-02-2017.
- [2] V. Authors: The reactive manifesto. <http://www.reactivemanifesto.org/>. last accessed: 08-02-2017.
- [3] Bacon: Bacon Property. <https://baconjs.github.io/api.html#property>. last accessed: 10-02-2017.
- [4] Bacon: Bacon.js. <https://baconjs.github.io/>. last accessed: 03-06-2017.
- [5] E. Bainomugisha et al.: “A Survey on Reactive Programming”. In: *ACM Comput. Surv.* 45.4 (Aug. 2013), 52:1–52:34. ISSN: 0360-0300. DOI: 10.1145/2501654.2501666. URL: <http://doi.acm.org/10.1145/2501654.2501666>.
- [6] M. Cantelon: Node.js in action. Shelter Island: Manning, 2014. ISBN: 9781617290572.
- [7] G. H. Cooper; S. Krishnamurthi: “Embedding Dynamic Dataflow in a Call-by-Value Language”. In: *Programming Languages and Systems: 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006. Proceedings*. Ed. by P. Sestoft. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 294–308. ISBN: 978-3-540-33096-7. DOI: 10.1007/11693024_20. URL: http://dx.doi.org/10.1007/11693024_20.
- [8] R. Dahl: Node.js by Ryan Dahl - JSConf in Berlin. http://www.jsconf.eu/2009/video_nodejs_by_ryan_dahl.htm. last accessed: 28-03-2017.
- [9] J. W. Davison; D. M. Mancl; W. F. Opdyke: “Understanding and addressing the essential costs of evolving systems”. In: *Bell Labs Technical Journal* 5.2 (2000), pp. 44–54. ISSN: 1089-7089. DOI: 10.1002/bltj.2221.
- [10] N. Developers: Node.js. <https://nodejs.org>. last accessed: 12-02-2017.
- [11] W. DOM: Document Object Model. <https://www.w3.org/DOM/>. last accessed: 14-02-2017.
- [12] J. Edwards: “Coherent Reaction”. In: *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*. OOPSLA ’09. Orlando, Florida, USA: ACM, 2009, pp. 925–932. ISBN: 978-1-60558-768-4. DOI: 10.1145/1639950.1640058. URL: <http://doi.acm.org/10.1145/1639950.1640058>.
- [13] C. Elliott; P. Hudak: “Functional Reactive Animation”. In: *SIGPLAN Not.* 32.8 (Aug. 1997), pp. 263–273. ISSN: 0362-1340. DOI: 10.1145/258949.258973. URL: <http://doi.acm.org/10.1145/258949.258973>.

-
- [14] P. Eugster; K. R. Jayaram: “EventJava: An Extension of Java for Event Correlation”. In: *ECOOP 2009 – Object-Oriented Programming: 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*. Ed. by S. Drossopoulou. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 570–594. ISBN: 978-3-642-03013-0. DOI: 10.1007/978-3-642-03013-0_26. URL: http://dx.doi.org/10.1007/978-3-642-03013-0_26.
- [15] Facebook: React. <https://facebook.github.io/react/>. last accessed: 10-02-2017.
- [16] Facebook: React Chrome Developer Tools. <https://facebook.github.io/react/blog/2014/01/02/react-chrome-developer-tools.html>. last accessed: 10-02-2017.
- [17] J. Forsyth: RxVision. <https://github.com/jaredly/rxvision>. last accessed: 10-02-2017.
- [18] J. Forsyth: RxVision Playground. <http://jaredforsyth.com/rxvision/examples/playground/>. last accessed: 10-02-2017.
- [19] J. Foundation: Dojo Toolkit. <https://dojotoolkit.org>. last accessed: 19-03-2017.
- [20] E. Gamma et al.: Design Patterns: Elements of Reusable Object-oriented Software. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.
- [21] V. Gasiunas et al.: “EScala: Modular Event-driven Object Interactions in Scala”. In: *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*. AOSD ’11. Porto de Galinhas, Brazil: ACM, 2011, pp. 227–240. ISBN: 978-1-4503-0605-8. DOI: 10.1145/1960275.1960303. URL: <http://doi.acm.org/10.1145/1960275.1960303>.
- [22] Github: FRP. https://en.wikipedia.org/wiki/Functional_reactive_programming. last accessed: 10-02-2017.
- [23] gonzalocasas: jQuery AOP. <https://github.com/gonzalocasas/jquery-aop>. last accessed: 19-03-2017.
- [24] Google: Chrome Developer Tools. <https://developer.chrome.com/devtools>. last accessed: 15-03-2017.
- [25] Google: Chrome DevTools - JavaScript APIs. https://developer.chrome.com/extensions/api_index. last accessed: 18-03-2017.
- [26] Google: Chrome DevTools - Message Passing. <https://developer.chrome.com/extensions/messaging>. last accessed: 18-03-2017.
- [27] Google: Chrome DevTools - Sample Extensions. <https://developer.chrome.com/extensions/samples#devtools>. last accessed: 18-03-2017.
- [28] Google: Get Started with Debugging JavaScript in Chrome DevTools. <https://developers.google.com/web/tools/chrome-devtools/javascript/>. last accessed: 15-03-2017.
- [29] Google: Using the Console - Chrome Developer Tools. <https://developers.google.com/web/tools/chrome-devtools/console/>. last accessed: 15-03-2017.

-
- [30] E. International: ECMAScript Promise. https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Promise. last accessed: 14-02-2017.
- [31] K. Kambona; E. G. Boix; W. De Meuter: “An Evaluation of Reactive Programming and Promises for Structuring Collaborative Web Applications”. In: *Proceedings of the 7th Workshop on Dynamic Languages and Applications*. DYLA '13. Montpellier, France: ACM, 2013, 3:1–3:9. ISBN: 978-1-4503-2041-2. DOI: 10.1145/2489798.2489802. URL: <http://doi.acm.org/10.1145/2489798.2489802>.
- [32] A. Lienhard; T. Gîrba; O. Nierstrasz: “Practical Object-Oriented Back-in-Time Debugging”. In: *ECOOP 2008 – Object-Oriented Programming: 22nd European Conference Paphos, Cyprus, July 7-11, 2008 Proceedings*. Ed. by J. Vitek. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 592–615. ISBN: 978-3-540-70592-5. DOI: 10.1007/978-3-540-70592-5_25. URL: http://dx.doi.org/10.1007/978-3-540-70592-5_25.
- [33] D. T. Ltd: AspectJs - Proxies in JavaScript. <http://www.aspectjs.com/index.htm>. last accessed: 19-03-2017.
- [34] I. Maier; M. Odersky: Deprecating the Observer Pattern with Scala.react. Technical report. 2012.
- [35] A. Margara; G. Salvaneschi: “We Have a DREAM: Distributed Reactive Programming with Consistency Guarantees”. In: *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. DEBS '14. Mumbai, India: ACM, 2014, pp. 142–153. ISBN: 978-1-4503-2737-4. DOI: 10.1145/2611286.2611290. URL: <http://doi.acm.org/10.1145/2611286.2611290>.
- [36] C. McMillan et al.: “Portfolio: finding relevant functions and their usage”. In: *2011 33rd International Conference on Software Engineering (ICSE)*. 2011, pp. 111–120. DOI: 10.1145/1985793.1985809.
- [37] L. A. Meyerovich et al.: “Flapjax: A Programming Language for Ajax Applications”. In: *SIGPLAN Not.* 44.10 (Oct. 2009), pp. 1–20. ISSN: 0362-1340. DOI: 10.1145/1639949.1640091. URL: <http://doi.acm.org/10.1145/1639949.1640091>.
- [38] T. Mikkonen; A. Taivalsaari: Web Applications - Spaghetti Code for the 21st Century. Technical report. 2007.
- [39] G. Pothier; É. Tanter: “Summarized Trace Indexing and Querying for Scalable Back-in-Time Debugging”. In: *ECOOP 2011 – Object-Oriented Programming: 25th European Conference, Lancaster, Uk, July 25-29, 2011 Proceedings*. Ed. by M. Mezini. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 558–582. ISBN: 978-3-642-22655-7. DOI: 10.1007/978-3-642-22655-7_26. URL: http://dx.doi.org/10.1007/978-3-642-22655-7_26.

-
- [40] G. Pothier; E. Tanter; J. Piquet: “Scalable Omniscient Debugging”. In: *SIGPLAN Not.* 42.10 (Oct. 2007), pp. 535–552. ISSN: 0362-1340. DOI: 10.1145/1297105.1297067. URL: <http://doi.acm.org/10.1145/1297105.1297067>.
- [41] R. R. Pucella: “Reactive programming in Standard ML”. In: *Proceedings of the 1998 International Conference on Computer Languages (Cat. No.98CB36225)*. 1998, pp. 48–57. DOI: 10.1109/ICCL.1998.674156.
- [42] H. Rajan; G. T. Leavens: “Ptolemy: A Language with Quantified, Typed Events”. In: *ECOOP 2008 – Object-Oriented Programming: 22nd European Conference Paphos, Cyprus, July 7-11, 2008 Proceedings*. Ed. by J. Vitek. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 155–179. ISBN: 978-3-540-70592-5. DOI: 10.1007/978-3-540-70592-5_8. URL: http://dx.doi.org/10.1007/978-3-540-70592-5_8.
- [43] Reactive-Extensions: RxJS. <https://github.com/Reactive-Extensions/RxJS>. last accessed: 08-02-2017.
- [44] J. Ressa; A. Bergel; O. Nierstrasz: “Object-centric Debugging”. In: *Proceedings of the 34th International Conference on Software Engineering*. ICSE ’12. Zurich, Switzerland: IEEE Press, 2012, pp. 485–495. ISBN: 978-1-4673-1067-3. URL: <http://dl.acm.org/citation.cfm?id=2337223.2337280>.
- [45] G. Richards et al.: “An Analysis of the Dynamic Behavior of JavaScript Programs”. In: *SIGPLAN Not.* 45.6 (June 2010), pp. 1–12. ISSN: 0362-1340. DOI: 10.1145/1809028.1806598. URL: <http://doi.acm.org/10.1145/1809028.1806598>.
- [46] Rx: ReactiveX. <http://reactivex.io/intro.html>. last accessed: 08-02-2017.
- [47] Rx: ReactiveX Observable. <http://reactivex.io/documentation/observable.html>. last accessed: 09-03-2017.
- [48] Rx: ReactiveX Operators. <http://reactivex.io/documentation/operators.html>. last accessed: 10-02-2017.
- [49] G. Salvaneschi; P. Eugster; M. Mezini: “Programming with Implicit Flows”. In: *IEEE Software* 31.5 (2014), pp. 52–59. ISSN: 0740-7459. DOI: 10.1109/MS.2014.101.
- [50] G. Salvaneschi et al.: “On the Positive Effect of Reactive Programming on Software Comprehension: An Empirical Study”. In: *IEEE Transactions on Software Engineering* PP.99 (2017), pp. 1–1. ISSN: 0098-5589. DOI: 10.1109/TSE.2017.2655524.
- [51] G. Salvaneschi; G. Hintz; M. Mezini: “REScala: Bridging Between Object-oriented and Functional Style in Reactive Applications”. In: *Proceedings of the 13th International Conference on Modularity*. MODULARITY ’14. Lugano, Switzerland: ACM, 2014, pp. 25–36. ISBN: 978-1-4503-2772-5. DOI: 10.1145/2577080.2577083. URL: <http://doi.acm.org/10.1145/2577080.2577083>.

-
- [52] G. Salvaneschi; M. Mezini: “Debugging for Reactive Programming”. In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE ’16. Austin, Texas: ACM, 2016, pp. 796–807. ISBN: 978-1-4503-3900-1. DOI: 10.1145/2884781.2884815. URL: <http://doi.acm.org/10.1145/2884781.2884815>.
- [53] M. Schäfer: “Refactoring Tools for Dynamic Languages”. In: *Proceedings of the Fifth Workshop on Refactoring Tools*. WRT ’12. Rapperswil, Switzerland: ACM, 2012, pp. 59–62. ISBN: 978-1-4503-1500-5. DOI: 10.1145/2328876.2328885. URL: <http://doi.acm.org/10.1145/2328876.2328885>.
- [54] K. Sen et al.: “Jalangi: A Selective Record-replay and Dynamic Analysis Framework for JavaScript”. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2013. Saint Petersburg, Russia: ACM, 2013, pp. 488–498. ISBN: 978-1-4503-2237-9. DOI: 10.1145/2491411.2491447. URL: <http://doi.acm.org/10.1145/2491411.2491447>.
- [55] B. Siegmund et al.: “Studying the Advancement in Debugging Practice of Professional Software Developers”. In: *2014 IEEE International Symposium on Software Reliability Engineering Workshops*. 2014, pp. 269–274. DOI: 10.1109/ISSREW.2014.36.
- [56] Staltz: How to debug RxJS. <http://staltz.com/how-to-debug-rxjs-code.html>. last accessed: 10-02-2017.
- [57] Staltz: Marble Diagrams. <http://rxmarbles.com>. last accessed: 10-02-2017.
- [58] Staltzs: Cycle.Js. <https://cycle.js.org>. last accessed: 10-02-2017.
- [59] Staltzs: Cycle.Js Devtool. <https://github.com/cyclejs/cyclejs/tree/master/devtool>. last accessed: 10-02-2017.
- [60] M. A. Storey: “Theories, methods and tools in program comprehension: past, present and future”. In: *13th International Workshop on Program Comprehension (IWPC’05)*. 2005, pp. 181–191. DOI: 10.1109/WPC.2005.38.
- [61] A. Taivalsaari; T. Mikkonen: “The Web as an Application Platform: The Saga Continues”. In: *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*. 2011, pp. 170–174. DOI: 10.1109/SEAA.2011.35.
- [62] P. Teixeira: Professional Node.js : building Javascript based scalable software. Hoboken, N.J. Chichester: Wiley John Wiley distributor, 2012. ISBN: 978-1-118-18546-9.
- [63] TIOBE: Tiobe index for february 2017. <http://www.tiobe.com/tiobe-index/>. last accessed: 13-02-2017.
- [64] G. Vanderburg: Tricks of the Java Programming Gurus. Tricks of the Java Programming Gurus. Sams.net, 1996. ISBN: 9781575211022. URL: https://books.google.de/books?id=Gxt_QgAACAAJ.

-
- [65] M. Wandschneider: *Learning Node.js: A Hands-On Guide to Building Web Applications in JavaScript*. Addison-Wesley Professional, 2013. URL: <https://www.amazon.com/Learning-Node-js-Hands-Applications-JavaScript-ebook/dp/B00DI9SVS8?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=B00DI9SVS8>.
- [66] A. White: *JavaScript Programmer's Reference*. New York: John Wiley & Sons, 2010. ISBN: 978-0-470-57784-4.
- [67] D. Zanarini; M. Jaskelioff: "Monitoring Reactive Systems with Dynamic Channels". In: *Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security*. PLAS'14. Uppsala, Sweden: ACM, 2014, 66:66–66:78. ISBN: 978-1-4503-2862-3. DOI: 10.1145/2637113.2637120. URL: <http://doi.acm.org/10.1145/2637113.2637120>.
- [68] Y. Zhuang; S. Chiba: "Method Slots: Supporting Methods, Events, and Advices by a Single Language Construct". In: *Proceedings of the 12th Annual International Conference on Aspect-oriented Software Development*. AOSD '13. Fukuoka, Japan: ACM, 2013, pp. 197–208. ISBN: 978-1-4503-1766-5. DOI: 10.1145/2451436.2451460. URL: <http://doi.acm.org/10.1145/2451436.2451460>.

Appendices



A APPENDIX

Details on the applications used for evaluation in the chapter 5 are presented here. For each application, we present the source from where we took the application. We also mention the required refactoring for each application.

A.1 App#1 - Operators and Events

This application contains lot of sub-applications to explain the operators and event handling with Bacon.js library.

Application Source:

This application is one of the examples from the GitHub repository of Bacon.js library.

<https://github.com/baconjs/bacon.js/blob/master/examples/examples.html>

Application Setup/Refactoring:

On page JavaScript code has been moved to separate JavaScript file, which is then included in HTML file.

A.2 App#2 - Father-Son Wallet War

This application is based on a rule that father will always have ten dollars more than his son.

Application Source:

Detailed logic of this application can be found here.

<https://alfredodinapoli.wordpress.com/2011/12/24/functional-reactive-programming-kick-starter-guide/>

We found Bacon.js implementation in jsfiddle <http://jsfiddle.net/EeJgZ/> mentioned in one issue of GitHub repository of Bacon.js

<https://github.com/baconjs/bacon.js/issues/51>

Application Setup/Refactoring:

We copied the HTML from the jsfiddle to the HTML file and JavaScript code into a JS file, include it in HTML file with other dependencies like JQuery and Bacon.js libraries.

A.3 App#3 - Up-Down Counter

It counts up and down as we click the buttons.

Application Source:

This example is taken from the home page of Bacon.js library.

<https://baconjs.github.io>

Application Setup/Refactoring:

Only relevant JavaScript code is given with demo, we build it by copying relevant HTML from the source code of demo.

A.4 App#4 - Form Validation

This application validates form in reactive style.

Application Source:

This example is also taken from the tutorial page of Bacon.js library.

<https://baconjs.github.io/tutorials.html>

Application Setup/Refactoring:

For ajax request to check the availability of the username we created a PHP script that validates all usernames of even length.

A.5 App#5 - Movie Search

This application search for movies from third party API as user types in the text field.

Application Source:

This example is taken from the home page of Bacon.js library.

<https://baconjs.github.io>

Application Setup/Refactoring:

Only relevant JavaScript code is given with demo, we build it by copying relevant HTML from the source code of demo.

A.6 App#6 - Bar Chart

This application adds a bar of random height to the bar chart.

Application Source:

This application is one of the examples from the GitHub repository of Bacon.js library.

https://github.com/baconjs/bacon.js/blob/master/examples/sliding_window.html

Application Setup/Refactoring:

On page JavaScript code has been moved to separate JavaScript file, which is then included in HTML file.

A.7 App#7 - WebSocket - Wikipedia

This application gets updates from Wikipedia by establishing web socket connection to third party API.

Application Source:

This application is presented on the following webpage.

<http://blog.scottlogic.com/2014/07/23/frp-with-bacon-and-d3.html>

Source code can be found on GitHub, within the following repository.

<https://github.com/DanGorst/frp-with-bacon> - Commit f6fc841

Application Setup/Refactoring:

We copied relevant files from the GitHub repository and runs it from the locally configured server.

A.8 App#8 - Smart Counter

This application counts and display numbers starting from zero the defined number.

Application Source:

This application is taken from GitHub repository.

<https://github.com/btroncone/learn-rxjs/blob/master/recipes/smartcounter.md> - Commit eb4f535

Application Setup/Refactoring:

This application is refactored for arrow functions.

A.9 App#9 - State Sotrage

This application manages the state linked to DOM elements.

Application Source:

This application is taken from sample applications available on GitHub repository of RxJS library - version 4.

<https://github.com/ReactiveX/rxjs/blob/master/doc/tutorial/applications.md> - Commit 3d88a99

Application Setup/Refactoring:

This application is refactored for arrow functions and to make it work with RxJS - version 5.

A.10 App#10 - Letter Counter

This application counts the number of letters of the text in the text field, it updates the count value of text field change.

Application Source:

This application is taken from sample applications available on GitHub repository of RxJS library.

<https://github.com/Reactive-Extensions/RxJS/tree/master/examples/lettercount> - Commit f5fe0af

Application Setup/Refactoring:

A.11 App#11 - Father-Son Wallet War

This application is based on a rule that father will always have ten dollars more than his son.

Application Source:

The concept of this application is taken from following web page.

<https://alfredodinapoli.wordpress.com/2011/12/24/functional-reactive-programming-kick-starter-guide/>

Application Setup/Refactoring:

We developed this application by using RxJS library.

A.12 App#12 - Movie Search

This application makes HTTP GET request to the third party server and shows the results on the target page

Application Source:

This application is taken from sample applications available on GitHub repository of RxJS library.

<https://github.com/Reactive-Extensions/RxJS/tree/master/examples/autocomplete> - Commit f8bf877

Application Setup/Refactoring:

We just replace ajax call, instead of calling wikipedia to search, we call third party API to search for movies.

A.13 App#13 - Follow The Mouse

This application moves three DOM elements with some delay as mouse pointer move.

Application Source:

This application is taken from sample applications available on GitHub repository of RxJS library.

<https://github.com/Reactive-Extensions/RxJS/tree/master/examples/followthemouse> - Commit 5fe0af

Application Setup/Refactoring:

We copied relevant files from the GitHub repository and runs it from the locally configured server.

A.14 App#14 - Drag and Drop

This application provides a feature to drag and drop a DOM element.

Application Source:

This application is taken from sample applications available on GitHub repository of RxJS library.
<https://github.com/Reactive-Extensions/RxJS/tree/master/examples/dragndrop> - Commit f5fe0af
Application Setup/Refactoring:
We copied relevant files from the GitHub repository and runs it from the locally configured server.

A.15 App#15 - Canvas Painting

This application provides painting features on canvas.
Application Source:
This application is taken from sample applications available on GitHub repository of RxJS library.
<https://github.com/Reactive-Extensions/RxJS/tree/master/examples/canvaspaint> - Commit f5fe0af
Application Setup/Refactoring:
We copied relevant files from the GitHub repository and runs it from the locally configured server.

A.16 App#16 - Twitter Follow Box

This application imitates the core features of Twitter UI element that suggests other accounts you could follow.
Application Source:
This application is presented on the following webpage.
<https://gist.github.com/staltz/868e7e9bc2a7b8c1f754>
Source code can be found on following jsfiddle. <http://jsfiddle.net/staltz/8jFJH/48/>
Application Setup/Refactoring:
We copied the HTML from the jsfiddle to the HTML file and JavaScript code into a JS file, include it in HTML file with other dependencies like JQuery and RxJS libraries. We also refactored to make it work with RxJS - version 5.

A.17 App#17 - REST API Call

This application query wikipedia as we type in given text field.
Application Source:
This application is presented on the following webpage.
<https://www.sitepoint.com/rxjs-functions-with-examples/>
Source code can be found on following jsfiddle.
<http://jsfiddle.net/mattpodwysocki/AL8Mj/>
Application Setup/Refactoring:

We copied the HTML from the jsfiddle to the HTML file and JavaScript code into a JS file, include it in HTML file with other dependencies like JQuery and RxJS libraries. We also refactored to make it work with RxJS - version 5.

A.18 App#18 - Spotify Artist Search

This application searches the artist name as we type in given text field, by querying Spotify API.
Application Source:

This application is grabbed from the following Demo page.

<https://rxjs.firebaseio.com/>

Application Setup/Refactoring:

We copied the resources from demo page and runs it from the locally configured server.

A.19 App#19 - Image Sampler

This application provides the colour when we hover over the image. On click, it saves the colour as a sample.

Application Source:

This application is grabbed from following Demo page.

<https://rxjs.firebaseio.com/>

Application Setup/Refactoring:

We copied the resources from demo page and runs it from the locally configured server.

A.20 App#20 - Mullti Select Cards

This application provides five DOM elements as cards with multiple selection functionality of those cards.

Application Source:

<https://codepen.io/wolfflow/pen/ceDtw>

Application Setup/Refactoring:

We copied the HTML from the codepen to the HTML file and compiled JavaScript code into a JS file, include it in HTML file with other dependencies like JQuery and Bacon.js libraries.

A.21 App#21 - True - False Logger

This application provides buttons for logging true and false. It also displays the time span of when these two buttons were clicked. Summary of clicks events is also displayed.

Application Source:

<https://jsfiddle.net/egman24/rjhphx7z/>

Application Setup/Refactoring:

We copied the HTML from the jsfiddle to the HTML file and JavaScript code into a JS file, include it in HTML file with other dependencies like JQuery and RxJS libraries.

A.22 App#22 - Drawing App

This application provides drawing functionality on canvas.

Application Source:

<https://codepen.io/wolfflow/pen/cudiJ>

Application Setup/Refactoring:

We copied the HTML from the codepen to the HTML file and compiled JavaScript code into a JS file, include it in HTML file with other dependencies like JQuery and Bacon.js libraries.