A Debugger for Reactive Javascript Libraries

Master-Thesis von Pradeep Baradur
Tag der Einreichung: 03. Oktober 2017

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

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Reactive Programming Technology

A Debugger for Reactive Javascript Libraries

Vorgelegte Master-Thesis von Pradeep Baradur

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

Tag der Einreichung: 03. Oktober 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 03. Oktober 2017 Pradeep Baradur

Abstract

Reactive programming is a programming paradigm designed to provide developers with the right abstractions for creating systems that uses streams of data. Reactive language abstractions are used to specify data dependencies between inputs and outputs. These dependencies are propagated by the language runtime, and thus reducing the burden on the user from managing dependencies. The execution of reactive programs is mostly driven by data flow, debugging such programs by traditional debuggers is hard. Due to lack of support for the provided abstractions, developers have to use the most readily available debug tool: console.log-debugging.

The lack of support from traditional debuggers created the need for a debugging tool that aids comprehension and debugging of reactive systems. **Chrome Reactive inspector**(CRI), a Chrome extension, helps the debugging process of applications built using reactive javascript libraries (**RxJS** and **Bacon.js**). In this thesis, existing CRI is extended to provide extensive support for all the operators and Subjects from RxJS library. The extended version mentioned above also helps the user to query the history using query language and set up breakpoints. A user can also search for a particular node, to gain an understanding of its dependents and dependencies. There is an option for the user to specify node IDs to exclude from logging the values to dependency graph which reduces performance overhead. We present the evaluation based on several real-time applications that show how the extension provides an easier and intuitive way of understanding data streams, and how they mutate through the flow of the program.

Contents

1 Introduction

1.1 Reactive Applications

Software applications have been evolving in the past decade and requirements of today's applications are not same anymore as they were decades ago[5]. Due to rise in the use of portable devices such as mobiles, tablets etc., there has been more demand for web applications instead of conventional desktop or native mobile applications as the operating system of these devices vary[10]. Due to the need of cross-platform applications, there emerged a concept of hybrid applications. Javascript is one of the popular programming languages for client-side web programming. Evolution of browser technologies and Javascript frameworks/engines in the recent years have increased the use of Javascript in rich internet applications[54][51].

Software applications often need to be very responsive to the user interactions such as external events and data flows. To achieve such responsive systems, needed an architectural changes which provides **responsive**, **resilient**, **elastic** and **message driven** systems, which are referred as **Reactive Systems**[5]. Traditionally, to develop Reactive systems, developers were dependent on Callbacks which mostly lead to a known problem “Callback hell”. Reactive programming (RP) is a subset of **Asynchronous programming** and archetype where the accessibility of new data drives the rationale forward as opposed to having control flow driven by thread-of-execution. Reactive programming helps to develop reactive systems more efficiently. There are several Reactive programming languages like FrTime[12] and FlapJax[34], RxJava[43]. RP has been further got popularized by the likes of Microsoft’s ReactiveX frameworks[36]. As a result, Javascript community introduced libraries like RxJS[44], BaconJs[6] and many more.

1.2 Motivation

In recent years, even though Reactive programming is widely adopted by the developers, it is apparently very difficult to debug reactive applications. In reactive programs, changes in one value trigger updates on all the other dependent values. The study[47] describes the challenges occurs in debugging reactive applications and infeasibility of traditional debuggers to debug the reactive applications. Traditional debuggers support imperative style of programming, where users can specify breakpoints on specific lines of code and the program execution will pause when it encounters the breakpoint. This kind of debugging is obsolete for reactive applications which uses declarative programming style and are data driven. This work is inspired by Reactive Inspector[45], an Eclipse plugin which is developed by Guido Salvaneschi at the Technical

University Darmstadt. The plugin uses dependency graph to represent the flow of the program during debugging process, where each variable is represented by a node and each dependency is represented by a directed edge. We have found Reactive inspector very useful. Therefore, used the same methodology for debugging Javascript libraries in web domain.

1.3 Thesis contribution

This thesis focuses on RxJS and BaconJS libraries, the currently trending reactive javascript libraries. We discuss Reactive programming in general and RxJS, BaconJS specifically. We also explain the complexities involved in debugging reactive applications and the applications built using RxJS and BaconJS. We are extending already existing chrome extension developed at Technical University Darmstadt by Waqas Abbas[1], which aids the debugging process of reactive applications. The extension provides a visual representation of how the applications are evolved over the time. The representation shows the dependency graph, where it shows which part of the programs depends on other parts of the program. The visualization helps the user understand the flow of the program and which in turn, helps in pinpointing the operators of interest that may need further scrutiny. The nodes in the dependency graph in each flow will represent each evolution step. The extension provides interactivity, where a user can travel back in time through all states of the dependency graph, dependent nodes for a particular node and set a breakpoint at a specific point. Setting a breakpoint enables debugging at a specific point and helps the user to examine node values at that point of execution. Breakpoints can also be disabled at any point in time to continue the normal execution of the program.

1.4 Outline

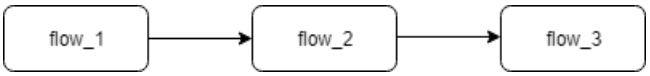
The structure of thesis is as follows. Chapter 2 introduces the fundamental concepts of Reactive programming in general and specifically about RxJS and BaconJS libraries. Further, we look at the details of Chrome reactive inspector a google chrome extension and also understand how the Google chrome devtools can be extended. In Chapter 3, we will have more insight into the design of Chrome Reactive inspector. The implementation details covering the inherent challenges are covered in Chapter 4. After that, we will look at the sample cases where our tool is helpful. As a part of an evaluation, we will explore the use of our extension for real-time applications built on using RxJS and BaconJS, and impact that the usage of our tool has on the reactive applications in chapter 5. Chapter 6, Conclusion and Future Work summarizes the work undertaken in the context of the thesis and present future research directions.

2 State of the Art

This chapter represents the state of the art of the topics relevant to this thesis. In the first section, we introduce reactive systems and challenges in implementing them. The second section explains different ways to implement reactive systems. The third section explains RP and the two javascript libraries which allows RP in the web domain. The fourth section describes the traditional debugging tools and their limitations to RP. Chrome developer tools' features with a brief guide on extending the functionality of Chrome DevTools is presented in the fifth section. The chapter finishes with related work from different aspects.

2.1 Reactive Systems

According to **reactive manifesto** [5], a reactive system is a set of architectural design principles for developing systems that are capable of meeting the increasing demands of responsive applications today. A traditional taxonomy classifies computational systems into transformational and reactive systems. Transformational systems receive some input, perform required computations, return an output and terminate. Hence, the use of state is not essential. Various inputs and computations lead to updates of an internal data structures.



In contrast to that, reactive systems continuously interact with the environment. They constantly keep updating their state whenever some event is fired. Hence, state is essential in reactive systems.

2.1.1 Challenges in Implementing Reactive Systems

Implementing reactive systems is hard, because of continue interaction between states and updates. With the help of following example, reactive systems can be explained in detail.

```
1 var a = 1;
2 var b = 2;
3 var c = a + b;
4 b = 4;
5 console.log(c)
```

Listing 2.1: Sample example 1

The output would be 3 because of the traditional programming approach. Change in the value of variable *b* has not effect on variable *c* because *c* has been defined before. But in reactive systems, the line 2 is rather interpreted as a constraint instead of an assignment, so that the output would be 5. The value of variable c is always updated whenever value of *a* or *b* changes. To implement such reactive system, a manual approach would look similar to the following code:

```
1  var a = 1;
2  var b = 2;
3  var c = a + b;
4  b = 5;
5  valueChanged();
6  console.log(c);
7
8
9  // This function will recalculates the value of c
10 function valueChanged(){
11   c = a + b;
12 }
```

Listing 2.2: Sample example 2

Above method naturally leads to following problems:

- The function *valueChanged* is scattered throughout the system. After each update of a or b, the triggering code should be inserted.
- There is high chance of developer may forget to insert the code and an important update can be missed.
- It is not possible to compose different reactions. One cannot express new constraints based on existing ones.
- There is no separation of concern.
- There is a lot of boilerplate code just to define a simple constraint. In the above example, just to express c = a + b, there is unnecessary code duplication which is less readable and less maintainable.

2.2 Implementation of Reactive Systems

Apart from the manual approach, there are multiple ways how the reactive systems can be implemented. We will discuss the traditional approaches such as aspect-oriented programming, observer design pattern, promises and callbacks.

2.2.1 The Observer Pattern

The Observer Pattern(also known as Publish-Subscribe Pattern)[55] is a way of reacting to changes. This pattern defines one-to-many relationships between objects such a way that when a state of one object changes, all dependent objects are updated automatically. It is suitable for any scenario, which requires push-based notification[55]. As a refresher, a UML diagram is depicted in Figure 2.1. The drawbacks of the observer pattern are broadly discussed in literature "Deprecating the Observer Pattern", which explains the problems in detail and concludes by proposing a built-in support for reactive programming abstractions as a solution[32]. The pattern still suffers from nearly all aforementioned problems when using a manual approach. The only real advantage: through the observer pattern, a better modularity is achieved. The updated code is decoupled from the code that changes a value. This makes the code more readable and less error-prone.

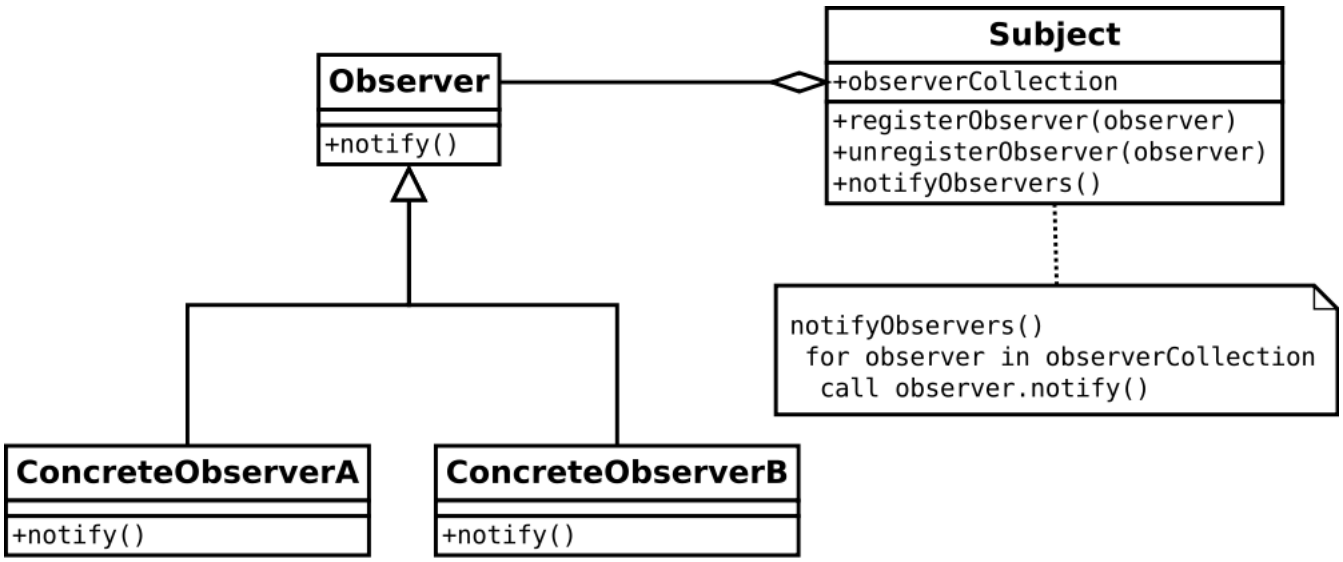


Figure 2.1: UML Class Diagram of the Observer Pattern

2.2.2 Aspect-Oriented Programming (AOP)

Aspect-Oriented Programming is a subset of Object-Oriented Programming. AOP aims at modularizing the CCC(cross-cutting concerns) throughout the application. These concerns are related

to any code, which cannot be cleanly decomposed from rest of the application and leads to code duplication. With AOP, it is possible to define points in the code at which some update code should be executed. These so-called pointcuts are based on events such as call of a method, execution of exception handler. These events can be combined with logical operators to generate more complex pointcuts. To implement reactive systems, the code which is responsible for updating the states in the system could be implemented using AOP. There are several Javascript libraries such as Dojo, JQuery AOP plugin, AspectJS that promise to address the CCC. However, AOP still has some drawbacks where dependencies have to be encoded manually.

2.2.3 Callbacks

For quite some time now, programmers have been simulating the management of events/notifications and corresponding responses with traditional programming techniques and asynchronous event handles like **callbacks**. Callbacks are nothing but Javascript functions. These are called by any other function which accepts the first function as a parameter. Most of the time, a "callback" is a function that is called whenever an event happens. The difficult thing when trying to understand is the order in which the callbacks execute as the program runs. Nesting callbacks to achieve asynchronous tasks often leads to a problem called "**callback hell**"[2]. The only way to delay computation so that it runs after the asynchronous call return is to put the delayed code inside a callback function. Callback hell problem can be handled to some extent by keeping the code shallow, modularizing the code and handling every single error.

2.2.4 Promises

The problem imposed by callbacks can be dealt with **promises**. D. Wise and Daniel Friedman[57] proposed the term promise as a proxy object that represents an unknown result that is yet to be computed. Libraries such as Q, JQuery, vow have already implemented promises with some minor differences in syntax. Furthermore, libraries like AngularJs[20], MVC-based Javascript library for web applications and Dart, a class-based web programming language have provided the abstractions for implementation of promises. Promises provide the modularization of the code and easy to handle and understand. One problem of using **promises** is that once a **promise** is triggered, there is no way to stop it. Once fired, they will end and call callback method, either in a successful way out in an error.

2.3 Reactive Programming

Developers have been using **procedural** and **imperative** programming from a long time. Object Oriented (OO) languages made programming more organized and structured bringing in the

concepts of inheritance and modularity. These programming paradigms consisted of a sequence of statements to be executed sequentially as encountered by the compiler to reach particular system state. Functional programming made the programming more declarative. In **functional programming**, the output of a program depends only on the input and refers to no external state, also has no side effects[19]. Hence the program that takes a particular input always returns the same output irrespective of any other conditions. Functional programs are designed to be reusable and composable. They also enforce immutability and functions can be composed of a chain to obtain the desired output. In functional programming, functions could be written more expressively also with a more intuitive handling of the current object with the *this* keyword. The verbosity in a traditional way of defining functions is greatly reduced using lambda styled expressions.

Most of the modern systems are reactive and they respond to a specific event of interest by changing state. Designing, developing and debugging such systems is hard because most computations are triggered with asynchronous inputs which makes it difficult to trace the control flow back[33]. The birth of Reactive programming was to address the very same problem. A RP language provides abstractions with which a program is expressed as a series of reactions to observable events[7]. RP is based on the idea of "Define, Watch & React"[28] where the entities of interest are defined, those would be observed and the reactions are triggered based on the changes observed on the interested entities. In a non-RP setup, a value assignment to a data element is generally a "snapshot assignment" indicating that the value assigned would be based on a calculation at the time of assignment. Dependencies between data elements are never established. The values are read and assigned at particular assignment points in a program. RP aims to establish data dependencies between elements. Assuming a variable a depends on variable b, a is assigned value based on the value of b at a certain point of time in the program. If value of b changes, this is propagated to reflect and update the value of a accordingly. In simple terms, RP is modeled on the spreadsheet like model where cell dependencies are defined and an update on one cell triggers a series of dependent cells[7].

RP handles an asynchronous sequence of data from the defined observable sequences. With the introduction of observer design pattern, observables could send multiple values over a period of time in response to a registration request from an observer. These values from the observable are asynchronous in nature and the observer has to handle them on a continuous basis over time. There are certain issues with the observer pattern that RP elegantly overcomes like the observer missing out on values that were sent from the observable before registration, the inversion of control with the observer pattern, the verbose non-declarative pattern of linking observables and observers etc. An author Stephen Blackheath, lists a few scenarios where designs based on observer pattern fail that RP overcomes [9]. The article[32] talks extensively about the shortcomings of the observer pattern. In traditional observer pattern, the programmer is expected to handle the logic around the registration of the observers and manage notifications;

however, with RP, the programmer declaratively specifies the dependencies between the various observables and the reactions. The language runtime is expected to handle the propagation of values also reducing the errors and ensuring correctness. The study[48] makes a detailed empirical evaluation on the comprehensibility of RP in comparison to the traditional observer pattern in object-oriented programming. The study also highlights the reduction in side effects of callbacks and also the boilerplate code around the traditional observer pattern in object-oriented style when RP is used in place.

Scala.React[32], is library that implements composable reactive abstractions on Scala. Flapjax[34] is Javascript framework for event-driven and reactive evaluation. These are Reactive languages that are based on the Functional programming style but do not integrate well with the mutable state of objects in the object-oriented style. These demand that, objects have to be recomputed from scratch in response to a change in a dependency. These are defined in a declarative way and updates on dependencies are handled by the runtime.

REScala[46] is a reactive language that bridges the gap between event-driven languages, functional reactive programming, and the object-oriented languages. The study[46] discusses the drawbacks of each of the above mentioned approaches applied in isolation and provides REScala as the solution that brings in the best of these into a single language. ReactiveX[36] is a library that also brings in the declarative and functional programming into the OO world also providing event compositions with LINQ[35] styled operators, which we will look at in more details in the further sections.

Distributed REScala[14] brings reactive programming to distributed systems. Distributed software accounts to a huge chunk of software systems today and Distributed REScala brings in the reactive programming language support to distributed software across multiple hosts. The study[14] shows that existing algorithms for the update propagation in a single system are not suitable for a Distributed scenario and propose an algorithm (Source Identifier Update Propagation) for glitch-free propagation of dependency updates in distributed systems. The algorithm also makes no assumptions about the knowledge of a centralized topology of dependencies between values.

Distributed Reactive Middleware (DREAM) [33] is a middleware completely implemented in Java that focuses on consistency guarantees in a reactive distributed system. DREAM highlights the lack of research on the consistency guarantees of signal propagation between components in a distributed reactive scenario and proposes a middleware support that the components in a distributed reactive setup can utilize to define suitable properties and enforce the required consistency guarantees for the propagation of changes between the components.

We will look at ReactiveX, RxJS and BaconJS in detail in the next section as these are most commonly used reactive javascript libraries.

2.3.1 ReactiveX

ReactiveX is an API based on the idea of asynchronous programming based on events using observable streams. An author André Staltz, defines Reactive programming as programming with asynchronous data streams[53]. ReactiveX is a programming API based on RP principles. It enables to programmatically express [13] all the properties that are desired of Reactive systems[5]. In ReactiveX, various data types can be viewed as an observable that emits only a single data item. User input events can also be considered as observables that emit streams of data. The consumer deals with the incoming data when it is notified of the data from the stream.

In addition to this, there are range of operators that can be applied on the streams. A filter applied to a stream observes the stream and emits a new stream of values that satisfy the filter criteria. In ReactiveX, every stream is immutable and the operator observing a stream, emits a new stream of values, leaving the original unchanged. In essence, a key aspect of ReactiveX(RP in general) is the flexibility to compose [34] asynchronous data into observable sequences.

ReactiveX facilitates notification of error and complete also in case of an error in an operation or when a stream is complete respectively by specifying functions in the observer to be notified of the corresponding events. The error and complete notifications, unlike the data notifications, are terminal and the association between the observable and the observer is ended. However, the error and complete functions are optional and an observer is free to leave them unimplemented. The observer can unsubscribe too from the observable to stop receiving notifications. Further, we will discuss RxJS, a Reactive Extension for javascript.

2.3.2 RxJS

RP has attracted more attention due to its ease of programming user interfaces[50][7]. The idea of ReactiveX has been employed for various platforms and programming languages and RxJS is Javascript library that allows users the means to employ Rx concepts into user interfaces.

Asynchronicity and user-system interactivity [31] has been increasing consistently on the user interface of web applications especially with the introduction of Ajax technologies[18] on the web pages and RxJS makes an attempt to take the user experience to the next level of responsiveness through event streams and event compositions. RxJS employs the ReactiveX practices along with the Javascript's inherent query operators to build desired observables and have the observers notified of the events asynchronously.

Asynchronous delivery of a value is readily available in vanilla javascript as of ES6[3] and callbacks have been used before promises. However, observables[42] facilitate delivering multiple values to subscribers asynchronously. observables provide abstractions over a stream of events. Promises are synchronous executables with a single return value, whereas observables are asynchronous executables that return multiple values over time.

Observables

Observables in ReactiveX can be conceptually viewed as the push equivalent of Iterables[36]. The subscribers pull data from iterables, whereas the observers are notified of the availability of new information from observables through push mechanism, the *next* method defined on the observers. With this idea, the subscribers are not locked in a synchronous request and there can be an infinite sequence of *next* emissions from the observables to the observer.

Observables in Rx also add another important idea of *completed* and *error* functions being defined at the observer, which are invoked by the observable on respective events. This lets the observer know that an observable has exhausted sending values, or an error has occurred when performing an operation on the observable. The calls to *next* are generally referred to as emissions and the *completed* and *error* are called notifications. It is evident how elegantly errors are handled and pushed as normal data to the consumer. An error is not something that is dealt by the observable but is passed on to the subscriber along the pipeline and the subscriber is expected to handle the errors. There is no such explicit definition in the case of iterables.

Observable is the basic building block of RxJS. The data produced by the producer is stored in observable which the consumer consumes. If the consumer is subscribed to the producer, it receives a signal from the observable whenever the data is pushed from the producer. Data available in the observable is not stored in the memory as opposed to an array which stores data in the memory. An observable represents event stream on which we can perform different methods as we can do on an array. For example, we can map the each value in an observable. There are two type of observables: Hot and Cold. According to an author Ben Lesh[1], if the underlying producer is created and activated during subscription, it is *cold observable* and an observable is *hot* if the underlying producer is either created or activated outside of its subscription.

Operators

RxJS is a combination of the best ideas from iterator pattern, observer pattern and functional programming[36]. Before data is handed over to Subscriber by an observable, there is a possibility that range of operators can be applied on the data streams. Each operator outputs an another observable without modifying the original data streams. This can be best understood

[1] https://medium.com/@benlesh/hot-vs-cold-observables-f8094ed53339 , last accessed 23-08-2017

with a representation using Marble diagrams. In the figure 2.2, a marble diagram is used to explain observables and transformation to another observable when an operator is applied.

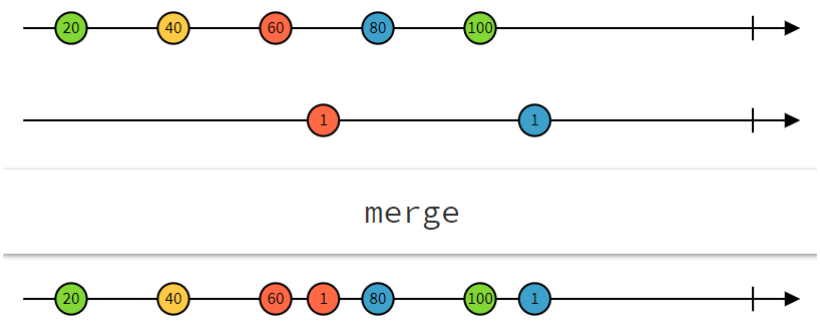


Figure 2.2: Marble diagram

Operators can be chained one of the other on observables to derive the desired results. Since the operators are chained, each operator is applied on the output of the previous operator and not on the original observable data streams. All the available operators in RxJS can be found in the online documentation[2].

Subjects

Subjects can be both **observable** and an **observer**[37]. It is a special type of observable which allows values to be multicasted to many observers. Observables are unicast in nature. Subjects maintain the list of observers. Whenever different observers subscribe to the source observable is intercepted by the Subjects and Subjects multicast the data from source observable to the subscribed observers. When source publishes the value, the Subject receives the value and it further broadcasts that value to the observer that are subscribed to the source observable.

Subjects are implicitly used when an observable is shared between observers, which is the case when using *hot observables*. When a *share* function is called on an observable, a Hot observable is implicitly created and Subject acts as a proxy between the observable and the observers. The Subjects handles the registration and disposal of subscriptions to the observable. The Subject also acts as a single observer on the source observable. There are four types of Subjects: **AsyncSubject**, **BehaviorSubject**, **PublishSubject** and **ReplaySubject**.

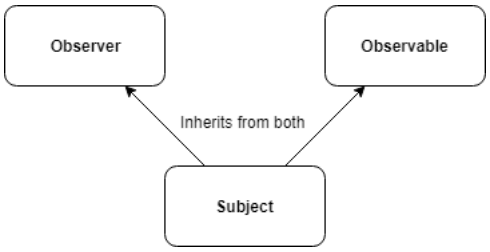


Figure 2.3: Subject

[2] http://reactivex.io/documentation/operators.html

RxJS - Design

Understanding the internal workings of RxJS involves knowledge of design and functioning of the RxJS framework. RxJS code can be divided into four parts. The first part will define the creation of source observable. The second part, to derive the desired observable, we will apply some operators on the source observable. In the third part, we will subscribe to the source observable to receive the emitted values after the operators are applied. Lastly, we will unsubscribe from the source observable. Unsubscribing is an optional feature in RxJS as RxJS handles this by default but it makes sense to unsubscribe manually by a user if the source observable is emitting the values continuously and the user does not need values after a specific point of time. RxJS example code can be seen in a example listing 2.3. As we discussed earlier, an observable is a collection of the data stream, we first need to create an observable. We can use functionality provided by RxJS to create an observable or we can create our own observable from scratch by wrapping any functionality that produces values over time. With the support of RxJS, we can convert multiple values, arrays, events into observables.

```
/*
Increment value every 1s, emit numbers 0, 1, 2, ....
*/
const observable = Rx.Observable.create(function(observer) {
let value = 0;
const interval = setInterval(() => {
observer.next(value);
value++;
}, 1000);

return () => clearInterval(interval);
});

const evenNumbers = observable.map(function(x) {
// return the square of each value
return x * x;
}).filter(function(x) {
// filter the values which are even
return x % 2 === 0
}).take(3);

// variable subscribe is subscribing to evenNumbers observable
const subscribe = evenNumbers.subscribe(val => console.log(val));

//unsubscribe after 10 seconds
setTimeout(() => {
subscribe.unsubscribe();
```

```
28 }, 10000);
29
30 //Output to the console
31 0
32 4
33 16
```

Listing 2.3: RxJS example

In the example code, from line 4 to 12, variable *observable* is defined as observable, which emits a sequence of numbers after 1000 milliseconds. In line 14 to 20, three operators are chained together and applied on the *observable* to derived the desired observable *evenNumbers*. Map operator is applied on *observable* and returns the square of each emitted values. For example, *observable* emits 0,1,2,3,4... and after **map** operator is applied, the new observable will emit the values 0,1,4,9,16.... The **filter** operator will then filter out the even values (0,4,16,36,64..) and pass on to the next operator. Lastly, the **take** operator will take only first 3 values(0,4,16) from the data stream. Thus, *evenNumbers* observables holds the values 0,4,16. In the line 23, a subscriber *subscribe* will subscribe to *evenNumbers* observable and print the values emitted to the console. After 10000 milliseconds, we will unsubscribe to the *evenNumbers* observable, which can be seen in the lines 26 to 28. After unsubscribing, the values emitted by *evenNumbers* observable are discarded.

2.3.3 BaconJS

BaconJS is a functional reactive programming module for events in javascript which can transform event listener/handler to a functional event stream. BaconJS turns event streams into clean and declarative data streams by switching from imperative way to functional way. For example, replacing nested for-loops with functional programming concepts like map and filter. BaconJS emphasizes on working with event streams instead of individual events. **eventstream** and **property** are the two kinds of observables in BaconJS. EventStream represents discrete values over the time. One can think of eventstreams as lists of events occurring over the time. For example, a promise that resolves after getting data from the API consists of events which can be modeled into event streams. The power of eventstreams is that they are composable. The tools to handle arrays for event streams are also be used on eventstreams such as filter events, map one event value to another. In the case of **properties**, they introduce the notion of continuous values which change over time. Any event streams can be easily converted to a property. Properties are very much similar to **eventstreams** in behavior but properties may or may not have an initial value, properties are continuous whereas eventstreams are discrete. At any given point of time, an eventstream can be converted to property using inbuilt methods such as *toProperty()*, *scan()* and *fold()* [30].

2.4 Debugging Software Programs

Developing Software programs includes testing, updating and maintenance. Usually, every software contains bugs or errors, which are removed over the time. The errors can be coding errors, design errors, complex interactions and system failures. The process of removing the bugs by finding out the root cause is called Debugging. In debugging process, software programs are compiled and executed to identify and eliminate the bugs. The rise of IDEs(Integrated Development Environment) has reduced the syntax related errors to some extent but to its hard to detect logical errors in the program. To help the debugging process, developers use the Debugger program. With the help of debuggers, users can step-through statements in a program by setting up breakpoints wherever necessary. When the program hits the breakpoint, the execution of the program is paused and a user can see the state of the program at that point in time. Debuggers also help user to see current execution stack which displays the stack history through which the control reached the specified breakpoint. Generally, debuggers are integrated with the development IDEs. Also, there are other independent debuggers such as [40] and [26] which run from command line and take a target program for analysis and inspection. In general, debuggers are developed for a particular programming language but there are debuggers such as [26] which can handle programs of multiple programming languages.

2.4.1 Debugging Javascript

Debugging is hard. But fortunately, modern browsers ship with a built-in javascript debugger. Javascript does not provide many debugging tools and, therefore it is challenging to debug any javascript program. In programming languages such as C++, Java, all objects are defined before the program is written and the user can use these definitions during the debugging process[27]. Whereas, Javascript is an interpretive language in which objects can be described dynamically. Javascript gives a user the power to assign new properties to objects as the program is executed. Such nature of javascript makes it hard to debug. There are different ways to debug javascript code such as using *alert* statements, printing out values to console, using *debugger* statement and using browser inbuilt debuggers. To understand the flow of the execution, a developer used to write alert or console.log statements which was extra code to the program. Using browser inbuilt debuggers, the developer can set breakpoints to a specific line in the program and examine the complete context of execution. After examining the values, the developer can resume the execution of the program. With the help of *debugger;* statements user can achieve the similar functionality as setting up a breakpoint.

2.4.2 Debugging Reactive Programs

Debugging reactive programs is a tedious task. In imperative programming paradigm, program execution can be tracked step-by-step with a breakpoint-based debugger; this is not possible with reactive programs. Assuming a reactive program can be tracked step-by-step, what should the debugger do when a value on which many other values depend is updated? Should it skip them? This would skip important steps. On another hand, moving from one update statement to another would also be quite confusing. Since some updates depend on other updates, countless values would have to be updated. Tracking all these updates with traditional debuggers is quite not possible.

In the article[52], the author explains why debugging RxJS programs is hard with the help of traditional debuggers. The applications developed using reactive javascript libraries are more abstract than procedural code. In the article, author mentioned the ways RxJS application can be debugged. One has to rely on drawing a dependency graph or marble diagrams manually. An example of marble diagram is shown in the figure 2.2. It is not feasible to draw a dependency graph or marble diagram manually for large applications. So there is a need of new debugging tools to support debugging for reactive javascript libraries. In this thesis, we take a step forward to implement a google chrome extension to visualize and debug the application. We will discuss the tool in coming chapters.

2.4.3 Jalangi Framework

Jalangi framework[51] is a powerful browser-independent(dynamic) analysis framework for JavaScript. The framework helps the user understand few useful abstractions and an API that simplifies implementation of dynamic analyses for Javascript. The framework is independent of browser and an Operating system. The framework instruments the javascript code and allows a user to further implement various dynamic analysis techniques. When the framework instruments the code, it creates hooks in the output and those hooks are monitored whenever there is an operation such as read or write to variable, function calls etc.

Working of **Jalangi** framework is illustrated in the figure 2.4. First, the framework reads the javascript code and performs instrumentation of the code and outputs instrumented code. Then, the instrumented code can be run in the browser or Node.js environment. We will not discuss how we can use an instrumented code in the Node.js environment as it is out of the scope for this thesis. With the help of API provided by the framework, the user can invoke callback functions. By invoking the callback functions, the user can intercept the execution of events and perform further analysis if required.

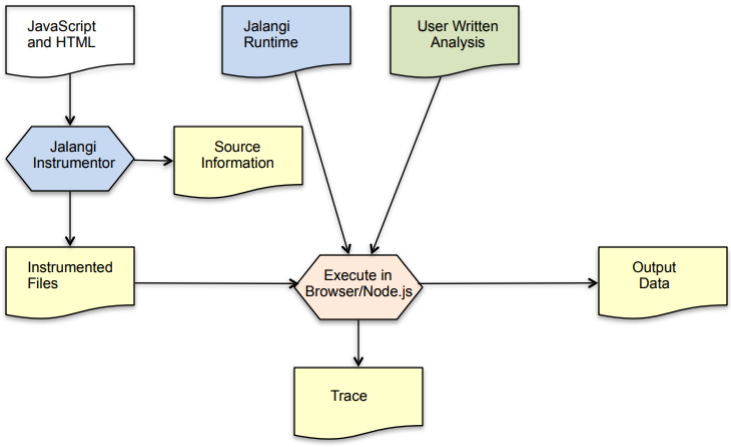


Figure 2.4: Jalangi framework components

2.5 Google Chrome Developer Tools

The Chrome Developer Tools(DevTools) are a group of debugging and web authoring tools build into Google Chrome[21]. The DevTools provide an interface where web developers get deep access into the internals of the browser and their web application. To access the DevTools, open a web page in Google chrome and use keyboard shortcuts *Ctrl+Shift+I(Windows)* and *Cmd+Opt+I(Mac)*. By right-clicking on a web page and selecting an option *Inspect* also opens up DevTools. By default, DevTools ships with panels such as Elements, Console, Sources, Network etc.

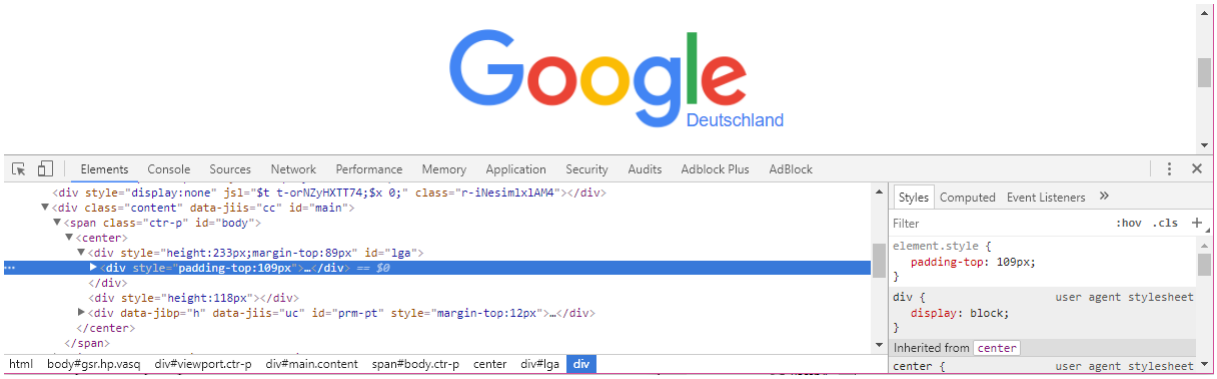


Figure 2.5: Chrome Developer Tools

In figure 2.5, we can see how DevTools looks when opened. The Elements tab window is split into two panels with HTML on the left and CSS, javascript debugging on the right. HTML panel gives the overview of Document Object Model(DOM) of the page. Developer can modify the HTML or CSS code and the effect can be seen on the page. The console tab shows all the information logged into the console by the running javascript code. The logged values can be either warning messages or error messages or the information logged by the user in the code. This tab is extensively used by web developers to debug the javascript code. Sources tab provides

an overview of all the files related the current web page. The user can set breakpoints in the javascript files for the debugging purpose. The Network tabs provide a detailed information regarding various HTTP calls made by the browser.

2.5.1 Extending Chrome Developer Tools

DevTools can be extended to add additional features. A DevTools extension is any functionality that is added in the form of an extension to enhance the debugging capabilities. An extension can be embedded as a new tab or a sidebar in the Elements tab. The extension is structured very similar to any other Chrome extensions[24]. With the help of the extension, a developer can interact with the current web page, get access to debugger etc. DevTools provides user a large pool of javascript APIs[21]. The elements of the manifest file are listed in [25]. The manifest file(manifest.json) tells Chrome, information about the extension such as name, permissions, version etc. The file declared as *devtools_page* in the manifest file has access to DevTools API, **chrome.devtools**. The Background page is created when an extension is loaded and it handles the user events and corresponding views for the events. Architecture of chrome DevTools extension is depicted in the figure 2.6. Components communicate with each other by message passing.

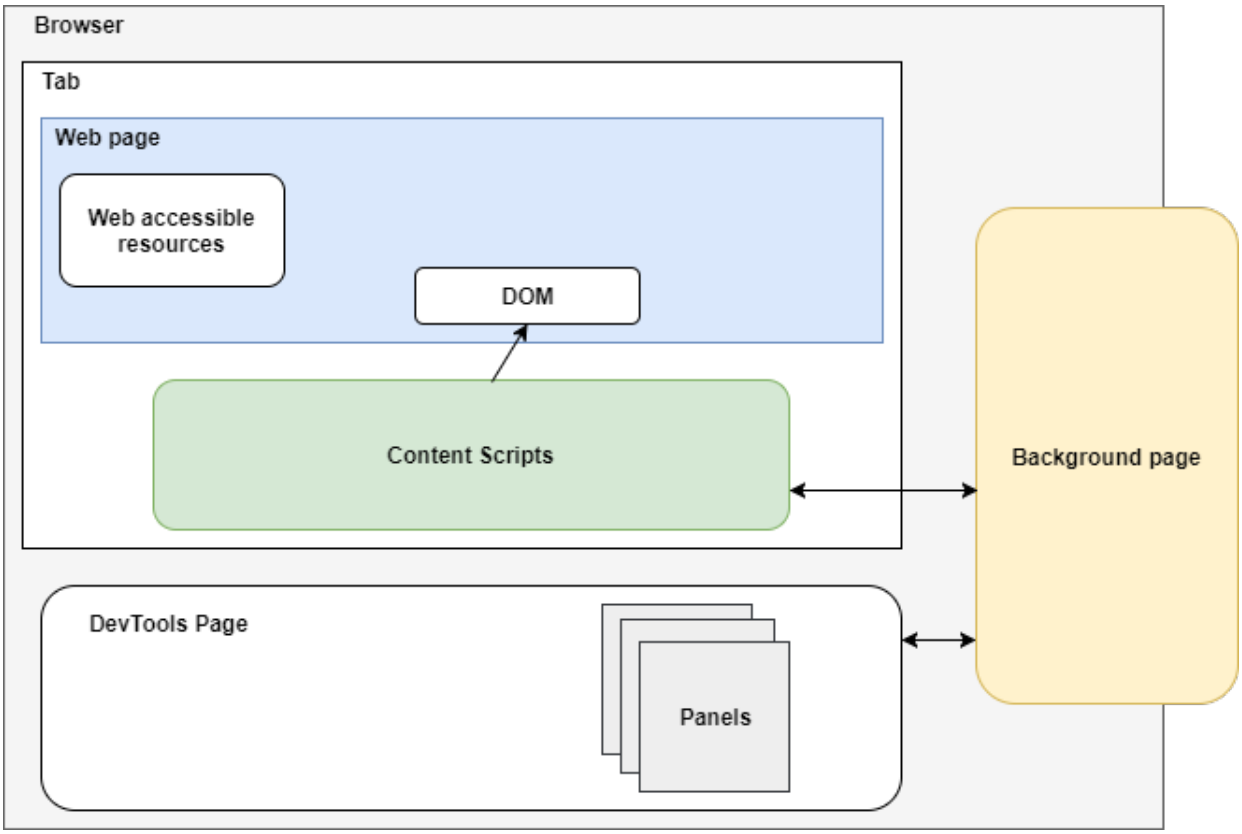


Figure 2.6: Chrome Extension Architecture

Context scripts in chrome extension are the list of javascript files which run in the context of web pages when the page is loaded[23]. They have the ability to modify the current web page since they have access to the DOM. But they do not have the ability to modify the loaded javascript code for the current page. There are two ways to inject content scripts: inject statically for all the pages or programmatically. In programmatically injection method, the *devtools_page* requests the background page to inject the content script. A message is sent to background page with the content script file names to load. The background page then handles the injection using **chrome.tabs.executeScript** API. In the statically injection method, the list of all the content scripts files to be injected are mentioned in the manifest file. In the manifest file, the developer can also specify other information like the time of injection whether before injecting all the scripts or after. Here the background page acts as a bridge between content scripts of the page and devtools_page of the extension.

Content scripts are executed in a special environment which is often referred to as an Isolated world[23]. Javascript defined in the inspected page and Content scripts do not have any knowledge of each other. They both run in isolation and handle their respective events on their own[23]. According to google chrome developers documents, chrome prevents direct access between Content script and inspected pages'javascript file due to security reasons. Although content script has no knowledge about the web page's javascript, it can still access the DOM of the page. The communication between inspected page and the content script can only happen through message passing API window.postMessage[23]. With the help of API call, specific event listeners defined in the content script and the inspected page can communicate with each other.

2.6 Related Work

As we discussed earlier, it is difficult to debug reactive programs. Due to the limitations of traditional javascript debuggers, there emerged a need of new debuggers. Some of them we will discuss in this section. The blog[56] suggests users to use the *do* operator for debugging the code. The *do* operator does not modify data in the observable but helps the user to log the subsequent values to the console. The operator can be used as shown in the below example.

```
1 var shortLowerCaseName$ = name$
2 .map(function (name) {
3 return name.toLowerCase(); })
4 .do(console.log);
```

Listing 2.4: Do operator usage

In the above example, *do* operator in the line no. 4 will log all the values which are passed from *map* operator after converting values to lowercase. The issue with the *do* operator is, it is an overhead code which should be removed in the production environment.

RxVision

While learning RxJS, an author Jared Forsyth developed the tool RxVision[16]. The tool helps the user to understand the flow of the data stream by visualizing the streams in real time. The author also provides a user an online playground where users can write RxJS code and the tool will visualize it instantly[17]. Unfortunately, this tool is not under active development and it does not support latest RxJs version. The tool not only visualizes the code but also provide an information such as source code line number. It is hard to understand the visualization since it does not give much information regarding variable names and map them to the visualization.

RxFiddle

This is another tool developed to debug Reactive Extensions by Herman Banken as a part of Master thesis[8]. This tool is similar to RxVision but supports both RxJS version 4 and 5. The tool visualizes the data flow through observable in more detailed manner. Currently, it supports RxJS code snippets and lacks the support for input streams[8]. This tool can be used in both browser and Node environment.

3 System Design

In this chapter, we discuss high-level requirements for the system to be developed in this thesis and the system architecture is explained in detail. At the end of the chapter, we will discuss the design choices and system features.

3.1 System Requirements

The requirements for the system developed in this thesis are analyzed according to the idea that the dependency graph visualization and history improve the debugging process of an application based on reactive javascript libraries. We identified following requirements expressed in terms of functionalities available to the end user.

Availability
The new system should seamlessly work and should be easily installable. Therefore, in order to reach many web developers, an extension for Google Chrome is being developed.

Visualization of the Dependency Graph
While debugging, the developer should be able to see the dependency graph generated based on the developers' code. For the selected variable, all the dependents and dependencies of the variable should be shown in the graph, so that the developer understands the overview of the system. When the new values are generated, the graph should be automatically updated.

Visualization of the History of the Dependency Graph
Once the graph is generated, the developer should be able to have a look at the graph at any arbitrary point in time. Thus, the whole history of the evolution of the graph can be visualized. It should also be possible for a developer to easily navigate through the time and observe the events such as node creation, node updates, dependency updates etc.

Querying the History of the Graph
Depends on the size of the application, the size of graph grows. For large applications, it is not feasible to manually navigate through each step of the graph evolution. Therefore, a query language should be developed such that it makes easier for the developer to jump to specific points/events in the history. For example, the system should be able to jump to a specific point at which a node is created or updated.

Breakpoints

Sometimes developer wants to halt the execution of the program by setting breakpoints at specific events. Our system should also provide developer an option to set breakpoints. For instance, it should be possible to set a breakpoint which is hit when a specific node is created or evaluated.

Helpers

System provides following additional features to the developer.

- Search node by name in dependency graph.
- Pause or resume logging all values to the graph.
- Exclude selected nodes from logging values to the graph.
- An option to chose whether developer wants to print all the logged values to the console.
- Show dependents and dependencies of the selected node.

3.2 System Architecture Overview

The overall system design is illustrated in Figure 3.1. There are two system components: One is Chrome extension which provides extended debugging functionality for reactive applications and another one is client code which needs to be debugged. The general interaction between client code and extension is shown in the figure 3.1.

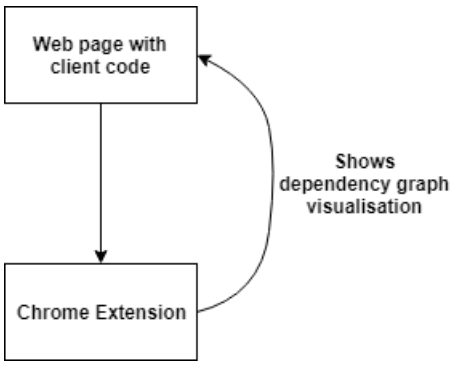


Figure 3.1: Overview of system design

The detailed system architecture is depicted in figure 3.2. The application is written using reactive javascript libraries(RxJS/BaconJS). Analyzer is a core component of the system. It helps the extension to catch all the events happening at both the libraries. In the current implementation, we support two reactive javascript libraries, RxJS and BaconJS. Analyzer analyzes all the events and passes the relevant information further to Chrome Reactive Inspector(CRI) panel.

The information received by the panel is then stored in browser storage and also displayed as a dependency graph. CRI stores in-between steps or data to browser storage which helps developer for back in time debugging.

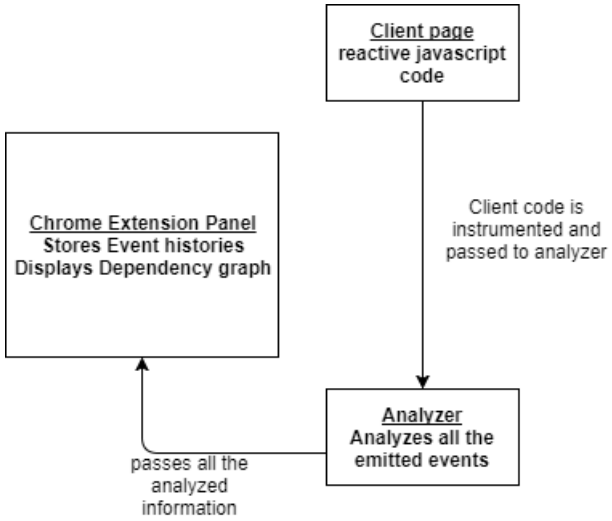


Figure 3.2: System Architecture

3.2.1 Analyzer

As we said earlier, Analyzer is the main building block of our system. It receives all the events and analyzes them. We will now look into how Analyzer is designed in detail.

RxJS library does not provide any interfaces to the developer for debugging purpose yet. But BaconJS provides a user an interface in the form of **Bacon.spy** method. Using this method, the developer can catch all the events emitted by clients' BaconJS code. Keeping these in mind, we designed Analyzer as shown in the figure 3.3.

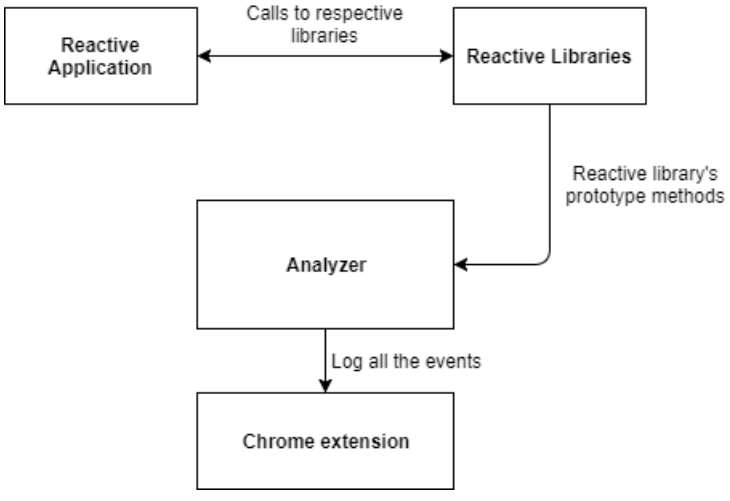


Figure 3.3: Analyzer Design

Here, we intercept and rewrite the required functions of both the libraries using prototype[38]. Every object in javascript has an internal property called Prototype. Using this property, we are overriding required function calls of both libraries. We capture the required information and handover the call to original library call for further computations. Overriding all the function calls is not feasible and scalable. We observed that calls to various operators returned respective new observables. For instance, map operator returns MapObservable in RxJS. All types of observables call their respective subscribe functions which internally calls parent subscribe method. Thus, we only override parent subscribe method instead of individual subscribe methods.

3.3 System Architecture Details

3.3.1 Dependency Graph Visualization

As explained earlier, dependency graphs help user visualize reactive applications. Each time-changing value is a node in the graph and a directed connection between two nodes if one node depends on another node. Consider the following RxJS code snippet for example.

```
1 var shortName = name.map(name => name.toLowerCase())
2 .filter(name => name.length < 5);
3
4 var bmi = weight.combineLatest(height, (weight, height) =>
5 Math.round(weight / (height * height * 0.0001))
6 );
7
8 var fullInfo = shortName.combineLatest(bmi);
```

Listing 3.1: RxJS code example

In the above example, in Line 1, variable *shortName* depends on variable *name*. Similarly at Line 4, *bmi* depends on both *height* and *weight*, at Line 8 *fullInfo* depends on both *shortName* and *bmi*. The figure 3.4, how dependency graph looks after modeling it. This gives a very good overview of the reactive system and especially of the dependency therein. This should be of great help for developers to understand and analyze the reactive applications.

3.3.2 Navigating through the Dependency Graph History

The visualization of a graph is not only updated whenever an event occurs, but it also stores each stage when it changes. This gives the developer a possibility to navigate through the whole history. The developer can navigate through the history in following ways:

History Navigation

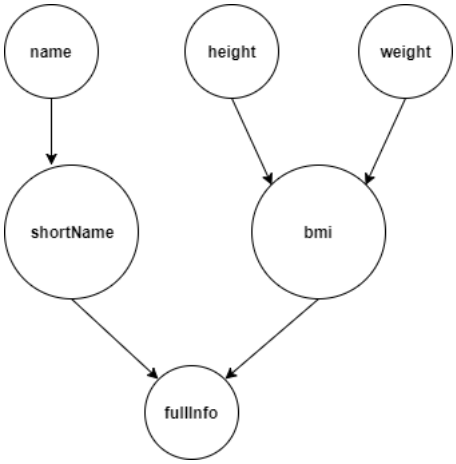


Figure 3.4: RxJS Dependency Graph Example

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

Direct Access

History navigation may not be practical for application with large histories. The developer should be able to drag the slider and thus quickly jump to the desired point in time.

History Queries

A third and last option is to navigate through the history using provided query language. By entering valid queries, the developer can directly jump to the respective points in time when these events happened.

3.3.3 Breakpoints

In traditional debuggers, breakpoints are based on specific line or conditions. A breakpoint hits and halts the execution of program each time the code reaches specific line or encounters specific condition evaluates to true. These breakpoint type does not really fit well into the RP approach. Hence, RP specific breakpoints should be developed. They reuse the developed query language. The developer can enter a query, start to debug the application and the debugger will halt the execution each time the query matches. For instance, if developer enters a query *NodeCreated[NodeId, 1]*, the debugger will halt the execution when a node with Id 1 is created.

3.3.4 Chrome extension

In this thesis, we are implementing a debugger in the form of an extension to Google Chrome DevTools. With the help of chrome APIs, we can add more debugging capability to our extension. The extension adds a new panel to DevTools which provides all the features mentioned so far

and also adds an options page where a user can use optional features. The same extension can be further extended and adapted to support other browsers like Mozilla Firefox[39].

3.3.5 Scoping and other features

The Jalangi framework, which we discussed earlier, instruments the given javascript code. It is performance hindering if we are instrumenting all other code which is not RxJS. So we provide scoping feature where a user can mention the file name to be instrumented. Another feature is, a user can search any existing node by name. This feature is useful when the dependency graph is large. The user can also figure out dependents and dependencies of any given node. As optional features, a user can exclude nodes to log events to the graph, can choose whether to log all the values to console.

3.3.6 Chrome Storage

The Google chrome provides data storage facility to manage data for specific requirements[22]. There are options of local storage, session storage and an HTML5 storage available in chrome. The data is stored in the form of key-value pair. Local storage is persistent and has no expiration until the user deletes it explicitly. On the other hand, session storage is persistent only for current browser session and is erased when a browser is closed. We use local storage in our extension to store the data such as file name used for scoping feature, list of breakpoints, list of nodes to be excluded from logging events to the graph.

4 System Implementation

This chapter is dedicated to the implementation details of the debugger. The first section describes the implementation method used in our chrome extension. In next section, we explain the data structures that capture all the data. In the third section, we explain how communication happens between different components of the extension. Finally, we discuss different components of the extension with the help of graphical user interface.

4.1 Implementation method

This section provides in-depth knowledge of how we implemented chrome reactive inspector(CRI).

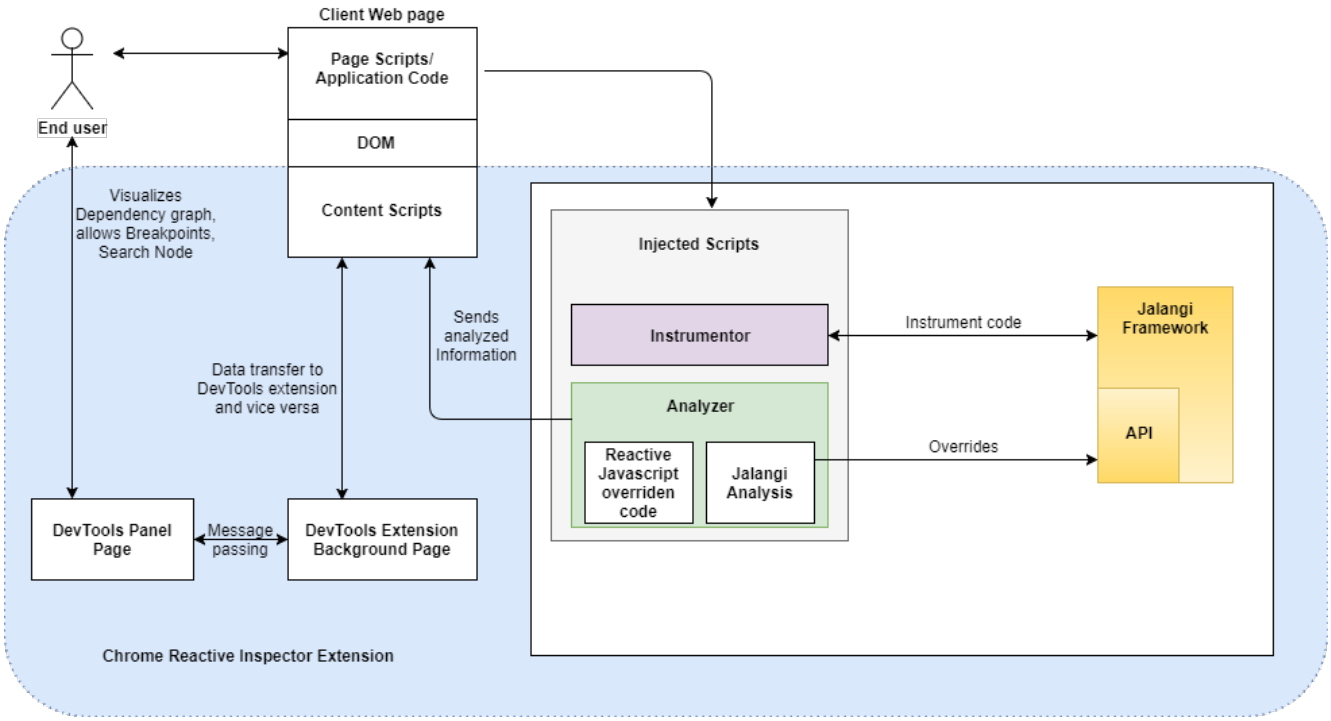


Figure 4.1: Detailed System Components

As we explained earlier in the section 2.5.1, current web page script and content script from extension have access to DOM. Figure 4.1 depicts detailed system design. First, the user has to load a web page which contains reactive javascript code. Then open DevTools and select a panel *Reactive Inspector*. The moment user selects the panel, we are then injecting two javascript files on-the-fly to our extension as shown in below code.

```
1 chrome.tabs.executeScript(tabId, {file: "analyzer.js"}, function(){
2   chrome.tabs.executeScript(tabId, {file: "instrumentor.js"}, function(){
```

```
3       //all injected
4     });
5   });
```

Listing 4.1: Injecting Javascript Files

In the above code, file "instrumentor.js" works as a *Interceptor*. Interceptor is responsible for loading all the client side javascript files sequentially and instrument them if required. Jalangi framework will help instrument all the javascript code. Then we evaluate instrumented code using *eval()* function. The instrumented code invokes Jalangi API functions when it encounters operations such as reading a variable, reading a function etc.

We override Jalangi API to catch all the operations performed by Jalangi framework. Analyzer component is responsible for further actions. It comprises of both overridden Jalangi API methods and catching all the events emitted by reactive libraries. Analyzer, with the help of Jalangi API, maps the reactive data streams to the respective variable names mentioned in the client code. Analyzer helps to catch the reactive library specific events such as creating an observable, subscriber subscribing to an observable, application of operators on observable, values emitted by observables etc. After catching the events, it analyzes them and then forwards the processed data to content-script. Content script then forwards it to background page and then finally it is forwarded to a panel in which an end user can see the generated dependency graph. All communication between content scripts, background page and panel are done via Message passing. Messages received by panel can be of type *saveNode*, *saveEdge* or *sendAllNodesAndEdges*. In *saveNode* message, we pass the node details such as nodeId, value, sourceCodeLine etc. and in *saveEdge*, we specify between which nodes there should be a directed graph drawn in the graph. For each message received by the panel, the current state of the graph is saved for further computations such as traveling back and forth in history graph, history queries etc. Finally, a user can see the generated dependency graph, options to search nodes, find dependencies and add breakpoints.

4.2 Significant Data Structures for the Communication

The three important data structures commonly used by the CRI are listed below:

Information about Reactive variables

Each node in the dependency graph contains following information.

- nodeId: Each node in the graph is identified by unique Id
- nodeType: Every node in the graph are of type, which can be different kind of observables or subscribers in RxJS and eventStream or property in BaconJS.
- nodeRef: It contains variable name provided by user and identified by Jalangi. This can be empty in case of intermediate streams.

- nodeValue: It holds current values of respective variable.
- sourceCodeLine: It holds the line number of a variable which is defined in the client code and is identified by Jalangi.

Data structure used in communication

Content script sends messages to panel vie background page to save node details and the edge details. To communicate, content script uses data structure defined in 4.2. The *action* attribute differentiates type of message, whether it is to save node details or save edge details. And *destination* attribute is always set to *panel* in our case.

```
1  // This data structure used in case of defining new reactive stream or updating
       the existing ones with new nodeValue
2  content: {
3    'nodeId': '',
4    'nodeType': '',
5    'nodeRef': '',
6    'nodeValue': '',
7    'sourceCodeLine': ''
8  }, action: "saveNode", destination: "panel"
9
10 // This data structure used in case of defining a new dependency between two
       reactive streams
11 content: {
12   "edgeStart": '',
13   "edgeStartName": '',
14   "edgeEnd": '',
15   "edgeEndName": '',
16   "edgeLabel": ''
17 },
18 action: "saveEdge",
19 destination: "panel"
```

Listing 4.2: Data structure for communication

To send edge details, we define another data structure as presented in the listing 4.2. Each attribute are defined as follows:

- edgeStart: It denotes the parent node on which another node depends on.
- edgeStartName: It holds name of parent node.
- edgeEnd: It denotes the child node which is subscribed to parent node.
- edgeEndName: It holds name of child node.

- edgeLabel: It denotes how parent and child node are related.

History entry

For history queries feature, we need to save all the events happening in CRI. The data structure used for this purpose is defined in listing 4.3.

- stageId: It holds value which defines current stage number.
- type: It holds type of history entry such as *dependencyCreated* etc.
- nodeName: holds node name in string form.
- nodeId: It holds id of the node.
- nodeValue: It holds value of node at that point of time.

```
historyEntry = {
  'stageId': '',
  'type': '',
  'nodeName': '',
  'nodeId':'',
  'nodeValue': ''
}
```

Listing 4.3: History entry Data structure

4.3 Communication between components

The building block of components working together is communication. We know that content scripts and other scripts of an extension run in the different context of the plugin. In CRI, content scripts include interceptor and analyzer. These need a way to communicate with the panel where dependency graph is displayed and a user can interact with it. Also for every new message, dependency graph should be updated. Chrome provides two ways of communication APIs. One way is using simple API for one-time requests[1] and another is using complex API that allows long-lived connections[2]. Since content scripts and background page needs to communicate continuously, we use long-lived connections. When a user opens DevTools, background script opens up a unique port for the panel named *Reactive-Inspector* and adds message listeners to receive messages from background script. The code mentioned in listing 4.4 is executed when an user opens DevTools. Here, it is creating a panel and creating a channel to post a message to the background page.

[1] https://developer.chrome.com/apps/messaging#simple , last accessed 13-09-2017
[2] https://developer.chrome.com/apps/messaging#connect , last accessed 13-09-2017

```
1  //Creates a panel with name "Reactive-Inspector" in DevTools
2  chrome.devtools.panels.create("Reactive-Inspector", "reactive-debugger.png", "
       panel.html", function (extensionPanel) {
3    // Opens a port with unique port number for the created panel
4    var port = chrome.runtime.connect({name: 'Reactive-Inspector'});
5    // this listens to any messages sent by background page
6    port.onMessage.addListener(function(msg) {
7    // Perform the actions on received messages such as
8    // creating new or updating existing variable, dependency created etc.
9    });
10 });
```

Listing 4.4: Creating a channel for communication between Panel and Background pages

Similarly, there should be another channel for background script to communicate with panel script. Listing 4.5 contains the relevant code snippet. Here, we set up a channel to receive messages sent to background script as shown in line 46. When DevTools with CRI is opened in two different browser tabs, it is hard to differentiate panels by panel name because both panels have the same name. Hence to differentiate between panels of different tabs, we need to store port numbers of both panels to avoid cross panel communication.

```
1  // holds objects of key-value pairs of tabid and port number
2  var tabPorts = {};
3
4  // creates a listener to listen to messages sent by Panel script
5  chrome.runtime.onConnect.addListener(function (port) {
6
7    var tabId;
8    port.onMessage.addListener(function(message){
9      if (!tabId) {
10       // add tabId to each message
11       tabId = message.tabId;
12       // maps port number to tabId
13       tabPorts[tabId] = port;
14     }
15     // explained in section 4.1
16     chrome.tabs.executeScript(tabId, {file: "analyzer.js"}, function(){
17       chrome.tabs.executeScript(tabId, {file: "instrumentor.js"}, function(){
18         //all injected
19       });
20     });
21   });
22
23   // function called for every message received by background page
24   var extensionListener = function (message, sender, sendResponse) {
```

```
25      // holds current port number
26      const port = sender.tab && tabPorts[sender.tab.id];
27      // if port is open and message destination is 'panel', forward message to
        Panel
28      if (port && message.destination === "panel") {
29        port.postMessage(message);
30      } else {
31        // if no destination provided, forward message to content script
32        if (message.tabId && message.content) {
33          chrome.tabs.sendMessage(message.tabId, message, sendResponse);
34        } else {
35        if(port)
36          // if no tabId, then forward message to panel from content script
37          port.postMessage(message);
38          else
39          return false;
40        }
41      }
42      sendResponse(message);
43    };
44
45    // Listens to messages sent from the panel
46    chrome.runtime.onMessage.addListener(extensionListener);
47
48    // listener to DevTools disconnection
49    port.onDisconnect.addListener(function (port) {
50      // remove port number when DevTools is disonnected and remove listener for
       message receivers
51      delete tabPorts[tabId];
52      chrome.extension.onMessage.removeListener(extensionListener);
53    });
54  });
55
56  // Deletes current port when current browser tab is closed
57  chrome.tabs.onRemoved.addListener(function (tabId) {
58    delete tabPorts[tabId];
59  });
```

Listing 4.5: Channel for background script to communicate with Panel script

4.4 CRI User Interface

The user interface(UI) of the CRI is a panel which can be seen when user opens chrome DevTools. After opening DevTools, CRI can be activated by clicking on “Reactive-Inspector” panel. The UI can be divided into 6 main parts as shown in Figure 4.2.

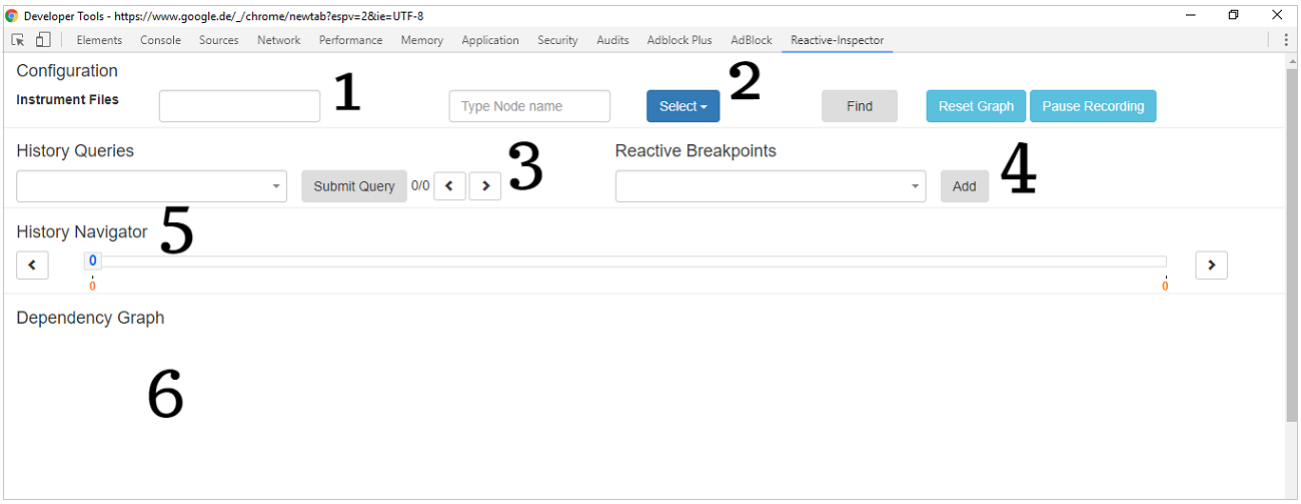


Figure 4.2: Chrome Reactive Inspector

Part 1 - Scoping Feature

This view is responsible for *Scoping* feature. The user can mention which files to be included for instrumenting by Jalangi framework. For example, if a user enters file named ’index.js’, only this file will be instrumented and other files are excluded. It is an optional feature. If the field is empty, then Jalangi instruments all the client-side javascript files.

Part 2 - Node details

This view is responsible for 3 features: search node by name, find dependents of the given node and find dependencies of a node. The user can select which feature he/she wants to use and click on Find button. The results will be shown in the dependency graph.

Part 3 - History Queries

The user can use this part to query all the history entries. We use query language for querying the history. When expanded, the views looks as shown in the Figure 4.3. The user can select any one of the options and edit it as per the requirement. For instance, to query creation of node named ’x’, user selects an option *NodeCreated* and edit it to *NodeCreated[x]*.

Part 4 - Reactive Breakpoints

This feature is responsible for adding or removing reactive breakpoints. This looks similar to

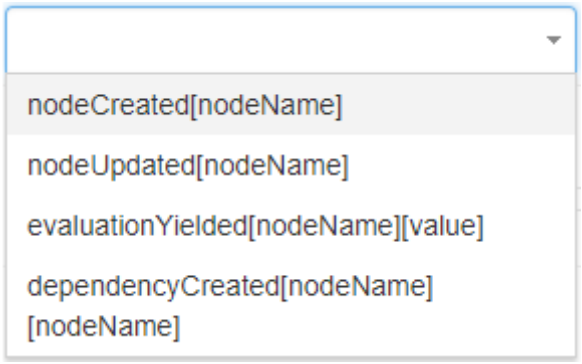


Figure 4.3: History Query

figure 4.3 but we use nodeId instead of nodeName for queries. The user selects an option and clicks on Add button to add a new breakpoint. List of all the current breakpoints is displayed right side of the button. The user can remove breakpoints if they are not required anymore.

Part 5 - History Navigator

To navigate to and forth between the history of the dependency graph evolution, we have provided a slider. A user can either slide the slider or use the buttons < or > to navigate.

Part 6 - Dependency Graph

This view is responsible to display the current state of a dependency graph. Dependency graph updates as soon as there is a new event. Each node in the graph provides node details such as nodeId, name, current value and source code line number. Additionally, a user can mouse hover on each node for more details on a particular node. The dependency graph is visualized with the help of open source javascript library called dagre-d3[41]. By default, the library provides zoom in and zoom out functionality for the graph. We have also added a new feature to print node details to console when a user clicks a node.

5 Evaluation

In this chapter, various case studies are introduced which demonstrate the use of an extension. They have been chosen in order to illustrate the important features of CRI, therefore the contributions of this thesis. We have collected examples written using RxJS and BaconJS libraries from the internet. We have evaluated the extension against several RxJS and BaconJS applications and we present the results in this section. We focus on features provided by our extension and evaluate that feature with the applications. In the last section, we summarize and conclude the evaluation.

5.1 Operators

Operators play an important role in both libraries. Developers can use operators to transform, filter and for many other operations. To demonstrate how CRI helps developers to understand reactive applications, we have taken RxJS operators example code. This section also illustrates the evolution of dependency graph for the given example.

5.1.1 RxJS - Operators

In our example, we have selected *map*, *filter* and *last* operators from RxJS library. One can find use case of other operators in RxJS official documentation[1]. Dependency graph generated by CRI after executing the program is shown in figure 5.1. In the following example 5.1, *source* is an observable with values 1,2,3,4,5. In line 3, *map* operator is applied, which maps each values from an observable and adds 10 to each value and new **MapObservable** is created. At line 5, *filter* operator filters the even values out of **MapObservable**. The *last* operator at line 7 receives values 12 and 14 after applying filter operator and emits value 14 which the last value in observable. At line 9, subscriber *subscribe* is subscribing to *example* observable. Thus, *subscribe* will receive value 14 and is printed to console in line 10.

```
1 var source = Rx.Observable.from([1, 2, 3, 4, 5]);
2 // apply map, filter and last operator
3 var example = source.map(function (val) {
4   return val + 10;
5 }).filter(function (num) {
6   return num % 2 === 0;
7 }).last();
8 //output: "Last to pass test: 14"
```

[1] http://reactivex.io/documentation/operators.html

```
var subscribe = example.subscribe(function (val) {
  return console.log("Last value: " + val);
});
```

Listing 5.1: RxJS - Operators example

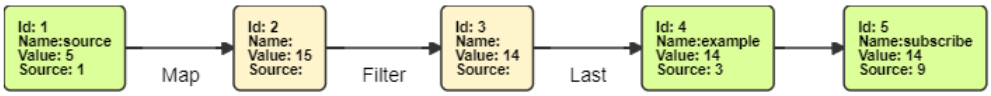


Figure 5.1: Dependency graph - RxJS Operators example

As we said earlier, CRI helps the developer to understand the flow of reactive applications. For listing 5.1, evolution of dependency graph can be depicted as shown in figure 5.2. Developer can use the provided slider to navigate to and forth to visualize evolution of graph. In the first 3 steps, node *source* is created and how the program runs internally after applying map and filter operator is depicted. In the last step, subscriber *subscribe* is created which is subscribed to node *example*. Thus, connection between *example* and *subscribe* exists.

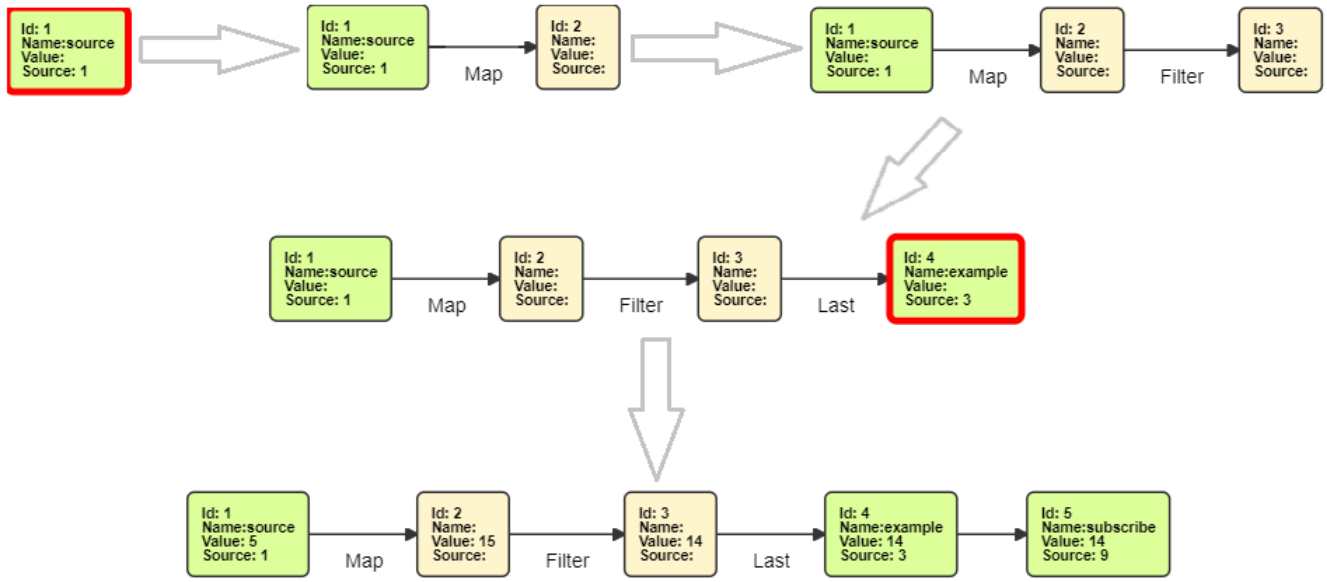


Figure 5.2: Dependency graph evolution - RxJS Operators example

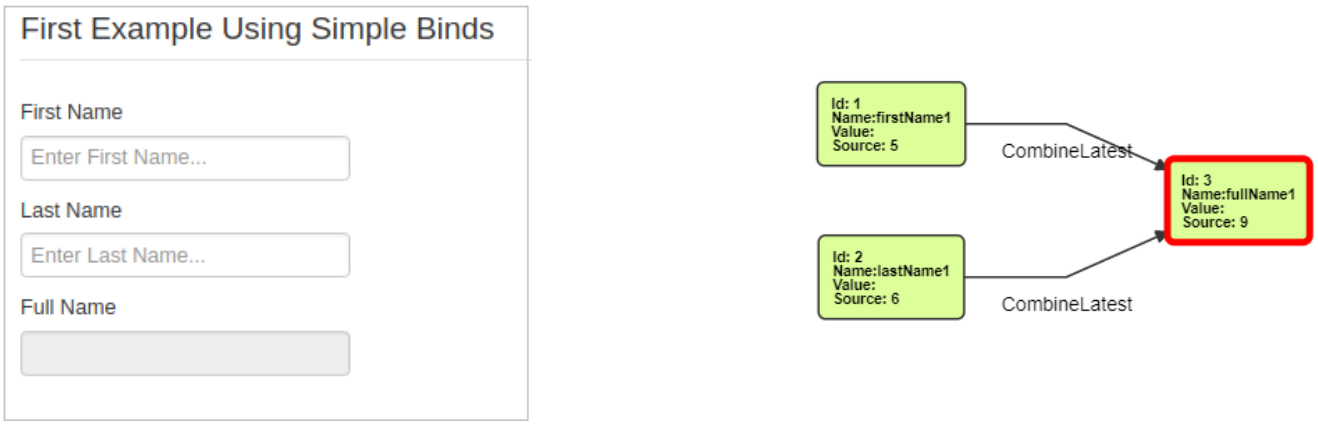
5.2 RxJS - Subjects

Listing 5.2 shows relevant javascript code for this example. In current example, user enters data in two HTML input fields: firstname and lastname. Output of the application is the combination of both names. The UI of an application looks as shown in figure 5.3a and dependency graph is shown in figure 5.3b.

```
// Create simple bindings for first and last name
```

```
var firstName1 = new Rx.BehaviorSubject('');
var lastName1 = new Rx.BehaviorSubject('');

// Create first and last name composite
var fullName1 = firstName1.combineLatest(lastName1, function (first, last) {
  return first + ' ' + last;
});

// Subscribe to them all
var fn1 = document.querySelector('#firstName1');
firstName1.subscribe(function (text) { fn1.value = text });

var ln1 = document.querySelector('#lastName1');
lastName1.subscribe(function (text) { ln1.value = text });

var full1 = document.querySelector('#fullName1');
fullName1.subscribe(function (text) { full1.value = text });

// Create two way bindings for both first name and last name
Rx.Observable.fromEvent(fn1, 'keyup')
.subscribe(function (e) { firstName1.next(e.target.value); })

Rx.Observable.fromEvent(ln1, 'keyup')
.subscribe(function (e) { lastName1.next(e.target.value); })
```

Listing 5.2: RxJS - Databinding example



(a) UI **(b)** Dependency graph

Figure 5.3: Simple Data-binding example - RxJS

5.3 RxJS - Animation Test

This is a simple animation example in which the user moves the mouse in the given area and the mouse trail with colorful boxes is animated. Relevant code for the example is listed in the listing 5.3. This kind of example is hard to implement with vanilla javascript. Thankfully RxJS does all the work for user and emits mouse pointer values in the form of data streams. UI for the example is depicted in figure 5.4 and dependency graph generated is shown in figure 5.5.

```
$(function () {
  Rx.Observable.prototype.movingWindow = function(size, selector, onShift) {
    var source1 = this;
    return Rx.Observable.create(function (o) {
      var arr = [];
      return source1.subscribe(
      function (x) {
        var item = selector(x);
        arr.push(item);
        if (arr.length > size) {
          var i = arr.shift();
          onShift(i);
        }
      },
      function (e) { o.onError(e); }
      );
    })
  }
  // Drawing area
  var canvas = $('#drawing');
  var source = Rx.Observable.fromEvent(canvas, 'mousemove')
  .movingWindow(
  25,
  function (x) {
    var b = new Box([x.clientX, x.clientY], canvas);
    b.showBox();
    return b;
  },
  function (b) {
    b.hideBox();
  }
  );
  var sourceSubscriber = source.subscribe();
});
```

Listing 5.3: RxJS - Animation Test example

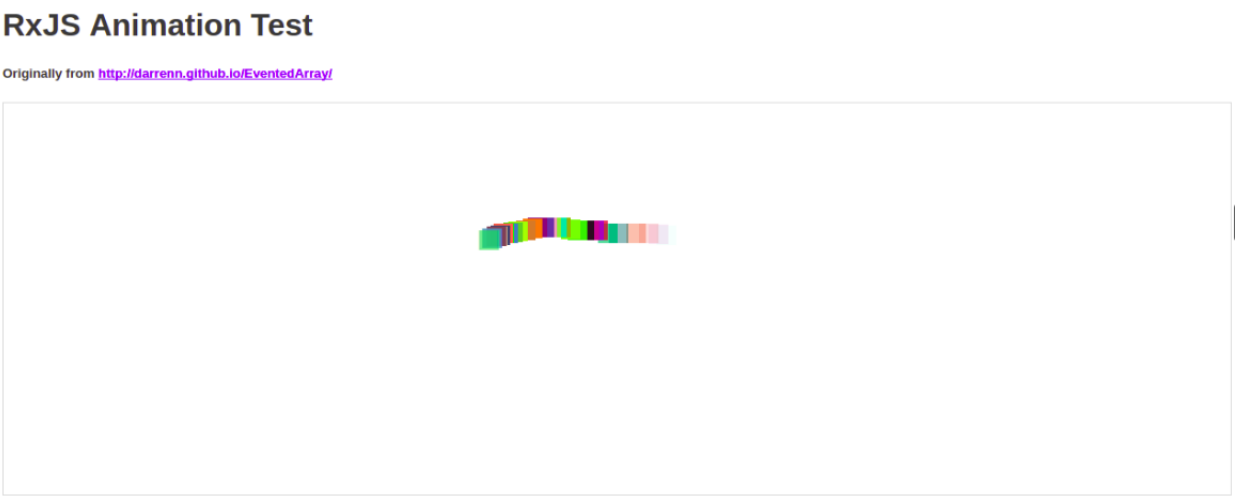


Figure 5.4: UI - Animation Test

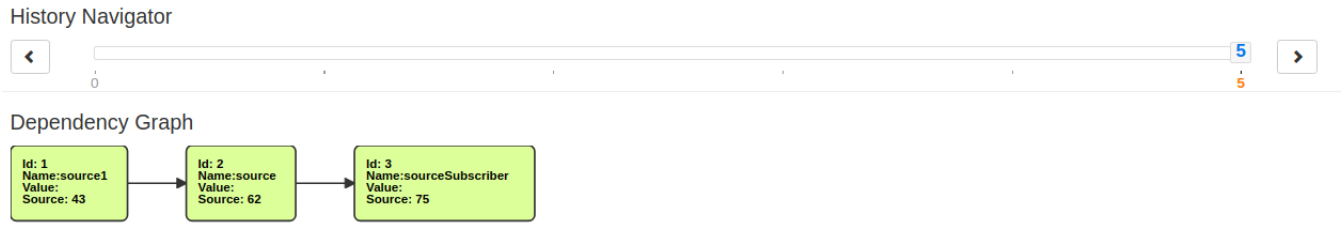


Figure 5.5: Dependency graph - Animation Test

5.4 RxJS - Drawing example

This is one of the advanced example from the previous examples. Figure 5.6 shows the options available for the user. Here, the user has seven different options to draw an image. User can set outer-radius, inner-radius, different color fillings to an image. As soon as user modifies the value from the available options, the same is reflected in the image. The relevant dependency graph is shown in figure 5.7.

```
1  var Observable = Rx.Observable;
2  var fromEvent = Observable.fromEvent;
3
4  var points$ = fromEvent(points, 'input', function (e) {
5    return e.target.value;
6  }).startWith(points.value);
7
8  var outerRadius$ = fromEvent(outerRadius, 'input', function (e) {
9    return e.target.value;
10 }).startWith(outerRadius.value).distinctUntilChanged();
11
12 var innerRadius$ = fromEvent(innerRadius, 'input', function (e) {
13   return e.target.value;
```

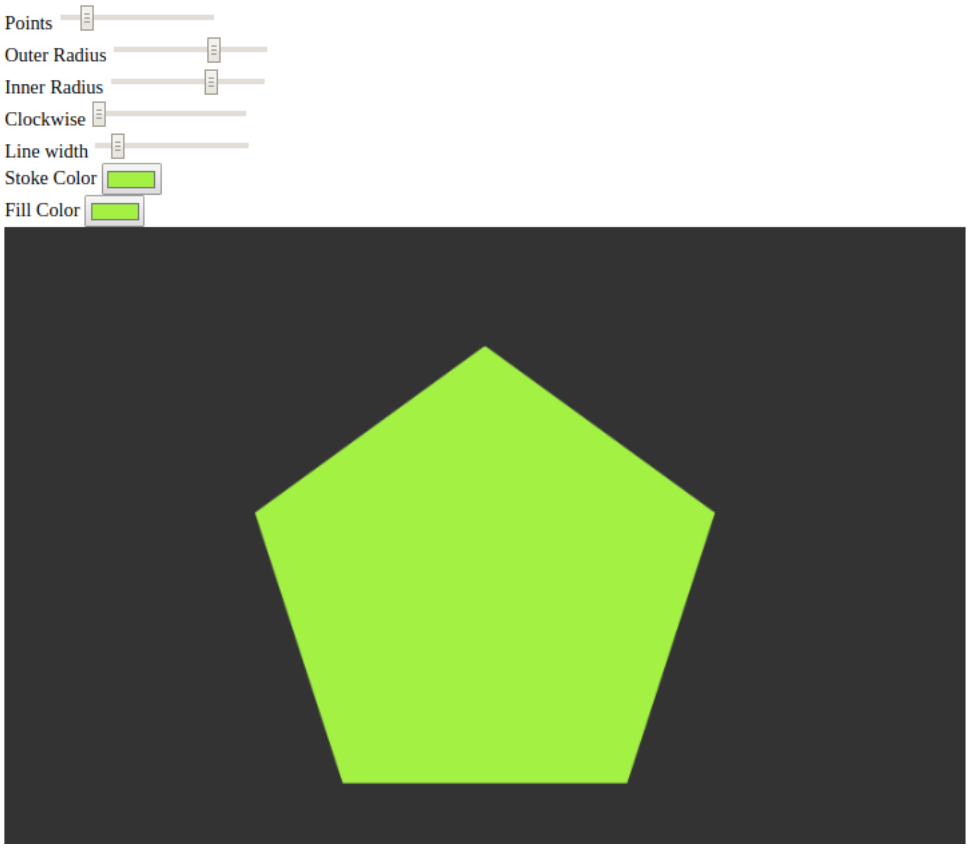


Figure 5.6: UI - Drawing example

```
14 }).startWith(innerRadius.value);
15
16 var angle$ = fromEvent(angle, 'input', function (e) {
17   return e.target.value;
18 }).startWith(angle.value);
19
20 var lineWidth$ = fromEvent(lineWidth, 'input', function (e) {
21   return e.target.value;
22 }).startWith(lineWidth.value);
23
24 var strokeColor$ = fromEvent(strokeColor, 'input', function (e) {
25   return e.target.value;
26 }).startWith(strokeColor.value);
27
28 var fillColor$ = fromEvent(fillColor, 'input', function (e) {
29   return e.target.value;
30 }).startWith(fillColor.value);
31
32 Rx.Observable.combineLatest(points$, outerRadius$, innerRadius$, angle$,
       strokeColor$, fillColor$, lineWidth$).subscribe(function (values) {
33   draw.apply(null, values);
34 });
```

Listing 5.4: RxJS - Drawing example

Listing 5.4 shows relevant RxJS code for the example. It looks complicated at the first glance. But with the help of dependency graph, user can understand which part of the code depends on other part of the code.

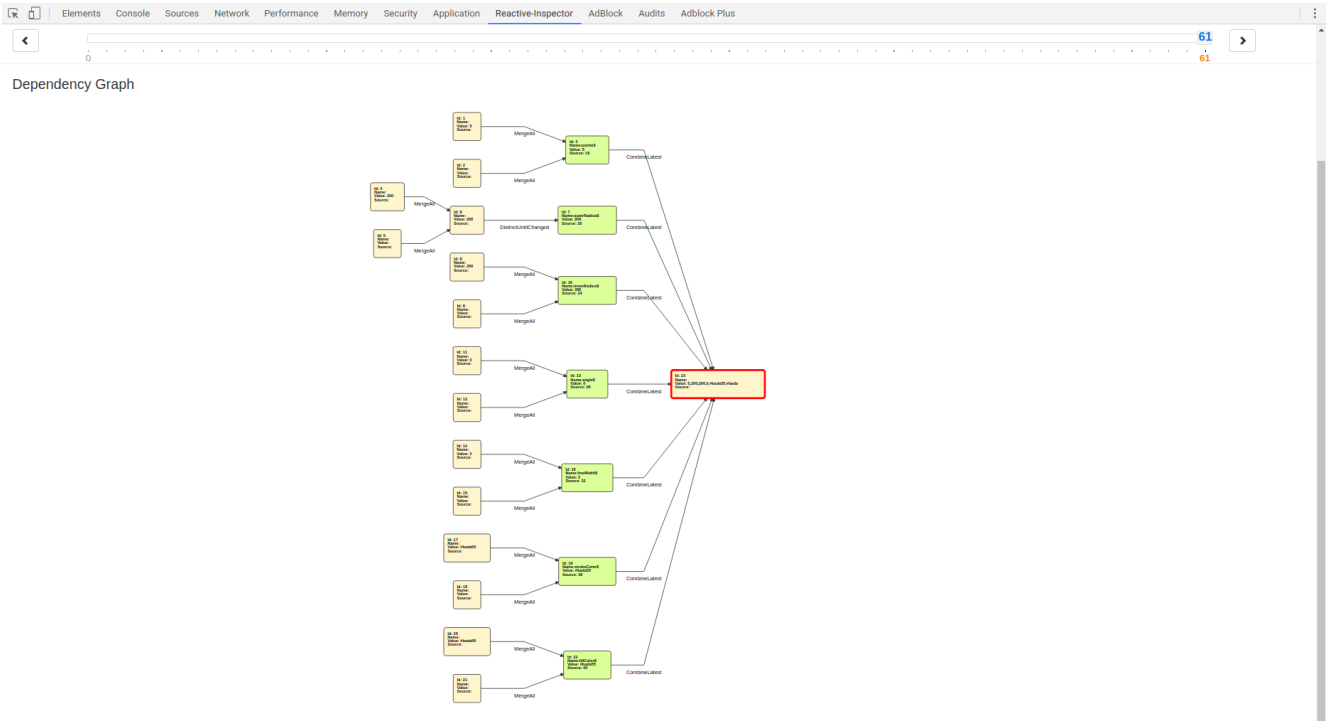


Figure 5.7: Dependency graph - Drawing example

We will evaluate breakpoint feature of CRI extension against this example. In large applications, it is important for the user to know at what point of the program bug exists. Breakpoint feature helps user to find out by specifying conditions. In our example, we will set up a breakpoint when node with ID 19 is created. We select *nodeCreated[nodeId]* from the dropdown available, which is discussed in 4.4 and set value to *nodeCreated[19]*. After setting up breakpoint, the user can see all the breakpoints as shown in figure 5.8. Figure 5.8 and 5.9 shows that when a program hits the breakpoint, it halts the execution and goes into debugger mode.

5.5 RxJS - Stopwatch example

Let us have a look at another typical example from RxJS applications. This application acts as a stopwatch with initial time of 90 seconds. When a user clicks on the timer, counter starts. When user clicks on running timer, it is paused. Listing 5.5 shows relevant RxJS code for the application. And figure 5.10 shows dependency graph generated for the application. In this example, there are 17 operators in total. Among them, 9 unique operators used.

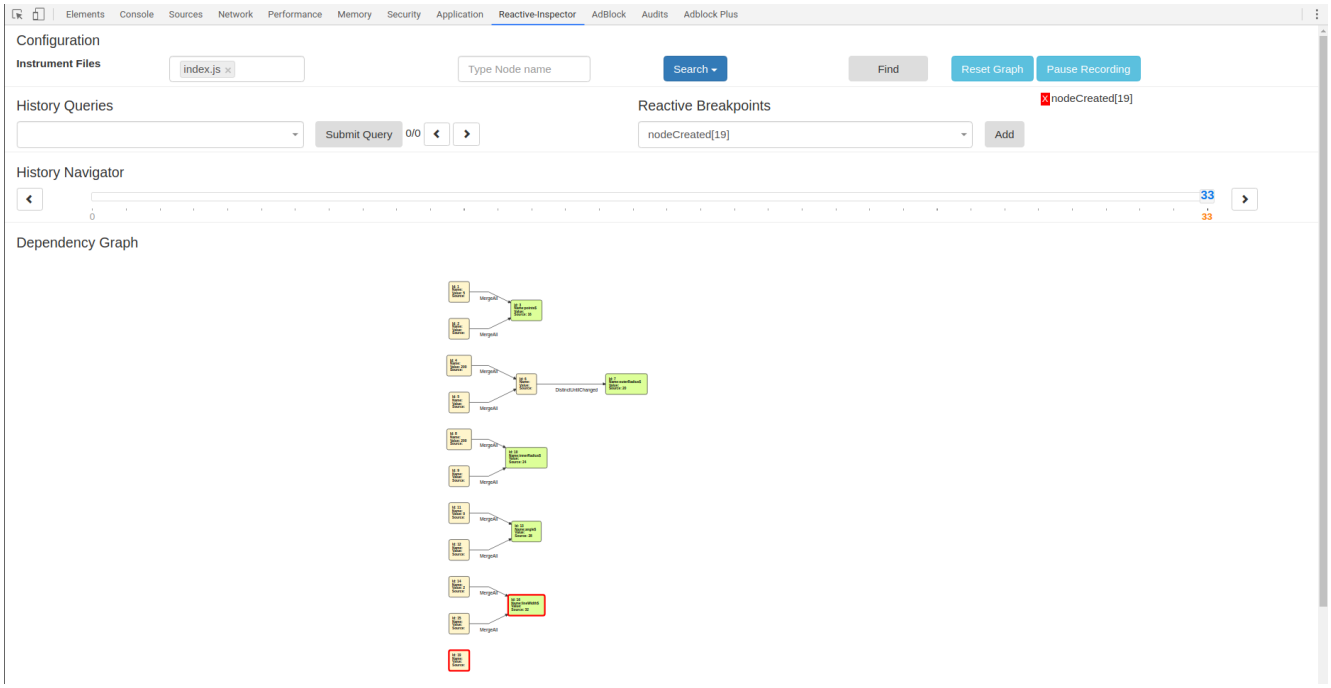


Figure 5.8: Setup Breakpoint - Drawing example

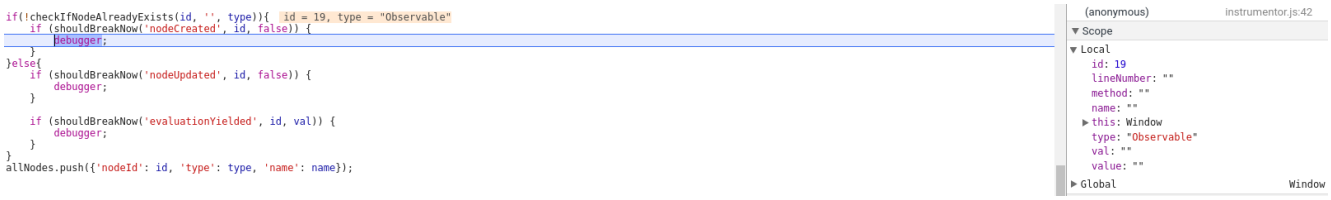


Figure 5.9: After Breakpoint is hit

```
// DOM elements

var interval$ = Rx.Observable.interval(1000);
var click$ = Rx.Observable.fromEvent(document, 'click');
var dblclick$ = Rx.Observable.fromEvent(document, 'dblclick');
var toggleOrReset$ = Rx.Observable.merge(click$.mapTo(function (isRunning) {
  return !isRunning;
}), dblclick$.mapTo(function () {
  return false;
})).startWith(false).scan(function (isRunning, toggleOrFalse) {
  return toggleOrFalse(isRunning);
}).do(function (isRunning) {
  return console.log('Running:', isRunning);
}).share();
var start$ = toggleOrReset$.filter(function (isRunning) {
  return isRunning;
});
var stop$ = toggleOrReset$.filter(function (isRunning) {
```

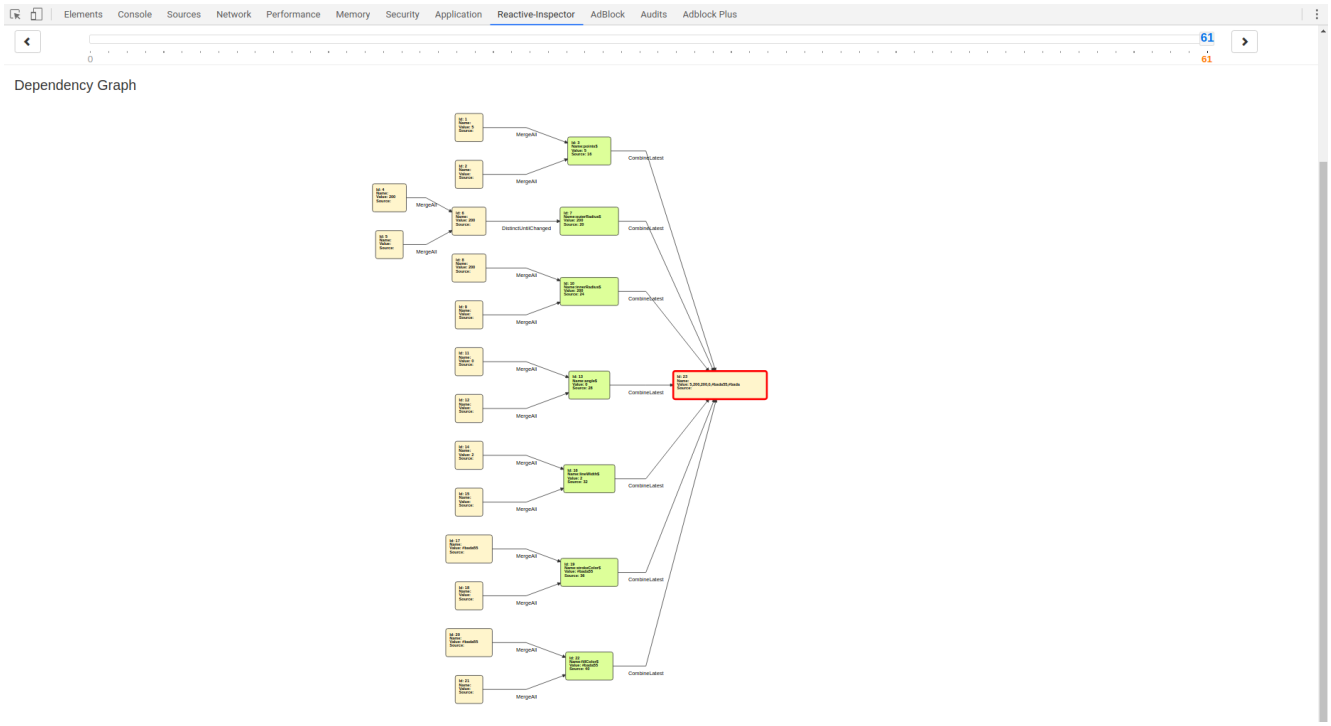


Figure 5.10: Dependency graph - Drawing example

```
19    return !isRunning;
20  });
21  var incOrDecOrReset$ = Rx.Observable.merge(interval$.takeUntil(Rx.Observable.merge
        (stop$, dblclick$)).mapTo(incOrDec), dblclick$.mapTo(reset));
22  start$.switchMapTo(incOrDecOrReset$).startWith(initialValue).scan(function (
        seconds, incOrDecOrReset) {
23    return incOrDecOrReset(seconds);
24  }).takeWhile(stillCan).map(toTime).subscribe(render);
```

Listing 5.5: RxJS - Stopwatch example

We will evaluate CRI features such as search, find dependents and dependencies of a given node against this application. Similarly, user can use these features for all the RxJS applications. For example, we will search node named *toggleOrReset$* in the dependency graph. Figure 5.11 depicts the result of the search. This features makes navigation in large dependency graphs much comfortable.

Dependency highlighting feature helps the user to find all the dependents and dependencies of a given node much easier. Figure 5.12 shows all the dependent nodes and figure 5.13 highlights all the dependency nodes of given node *toggleOrReset$*.

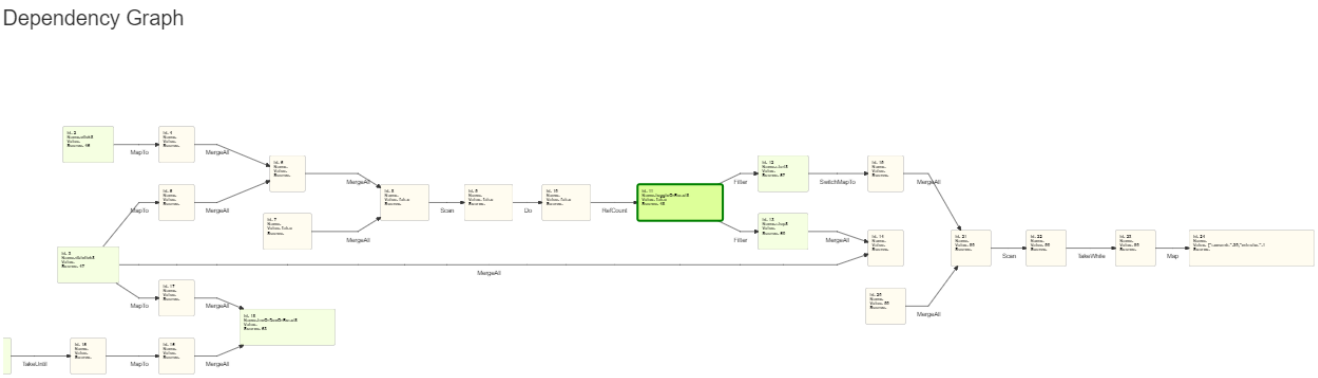


Figure 5.11: Search feature

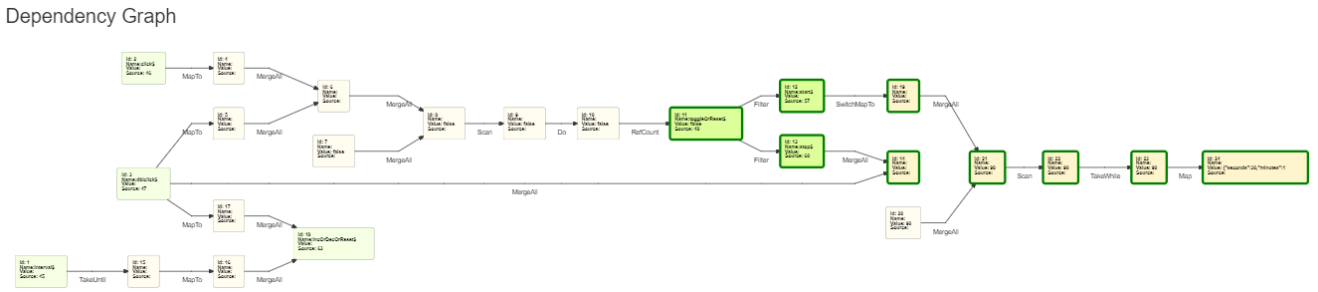


Figure 5.12: Highlighting Dependent nodes

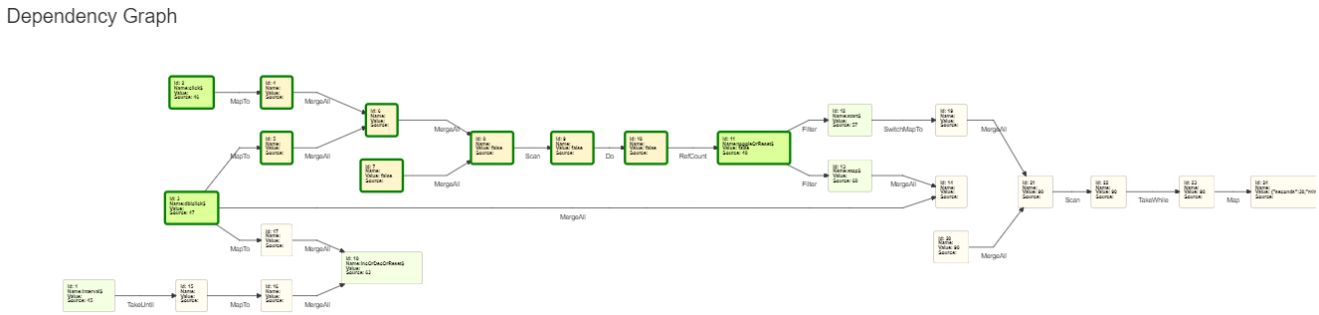


Figure 5.13: Highlighting dependencies of a node

5.6 BaconJS - Drag-n-Drop

Let us look into another interesting application called “Drag-n-Drop”. When uploading files to server via browser, some websites provide an option to drag and drop the files. Our application imitates such feature but not with files. In this application, the user can drag contents from one box and drop in another box. For example, in the figure 5.14, last box contains both **A** and **B** values. User can drop **B** in box B but not in A. Similarly, **A** can be dropped in box A only. Listing 5.6, shows relevant code for the example. Figure 5.15, depicts final view of the dependency graph.

```
1 var homeBin, drag, drop, this$ = this;
```

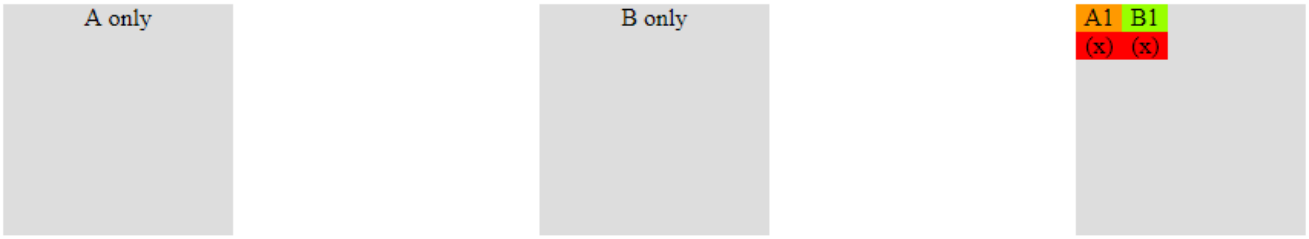


Figure 5.14: BaconJS - Drag-n-Drop UI

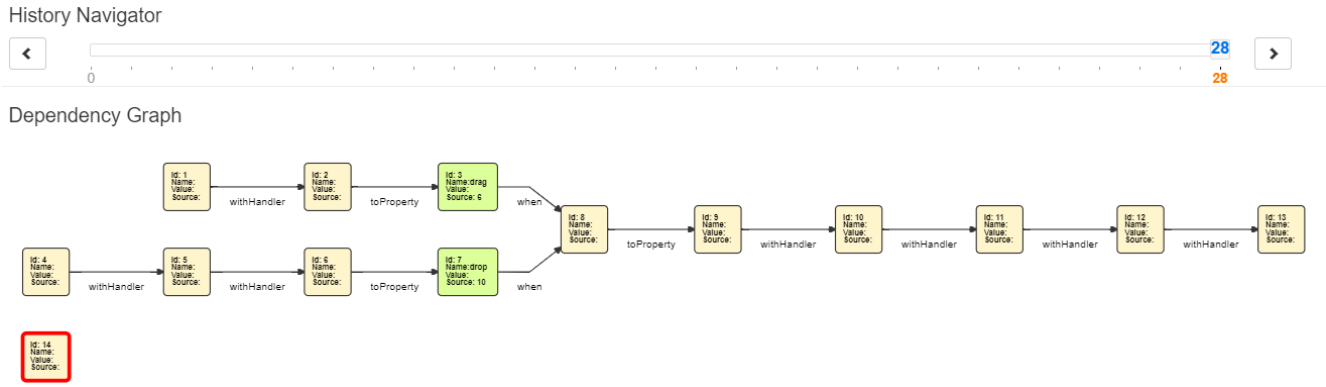


Figure 5.15: BaconJS - Drag-n-Drop Dependency graph

```
$.event.props.push('dataTransfer');
homeBin = $('#home-bin');
drag = $(document).asEventStream('dragstart', '.item').doAction(function (e) {
  e.dataTransfer.effectAllowed = 'move';
  return e.dataTransfer.setData('tmp', '');
  }).toProperty();
drop = $('.drag-bin').asEventStream('dragover drop').doAction(function (it) {
  return it.preventDefault();
  }).filter(function (it) {
    return it.type === 'drop';
}).toProperty();
drag.combine(drop, function (a, b) {
return [a, b];
}).filter(function (arg$) {
var a, b;
a = arg$[0], b = arg$[1];
return a.timeStamp < b.timeStamp;
}).map(function (arg$) {
var a, b;
a = arg$[0], b = arg$[1];
return {
item: a.target,
dest: b.target
};
}).filter(function (arg$) {
```

```
var item, dest;
item = arg$.item, dest = arg$.dest;
return item.dataset.type === dest.dataset.type || dest.dataset.type == null;
}).assign(function (arg$) {
var item, dest;
item = arg$.item, dest = arg$.dest;
dest.appendChild(
item);
});
$('.item .remove').asEventStream('click').assign(function (e) {
homeBin.append(
$(e.target).closest('.item'));
});
```

Listing 5.6: BaconJS - Drag-n-Drop Example

We will now evaluate *History Queries* feature against this example. Use case of History queries is described in the section 4.4. After dragging **B** to box B, we ran the query *nodeUpdated[drop]*. We found 2 stages where node *drop* is updated. Changes after the query can be seen as shown in figure 5.16. Node *drop* is updated at the step 14 and 81.

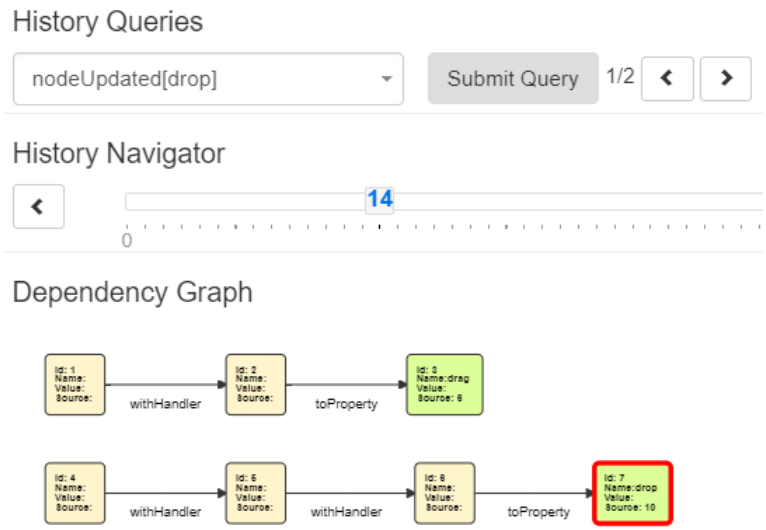


Figure 5.16: History Queries example

5.7 RxJS - Wikipedia updates

Listing 5.7 shows a short example from RxJS applications that creates websocket connection to third party server. Server sends different type of notifications which are then received by websocket connection. The data received is then converted to data stream named *updateStream*. The target application filters the *updateStream* into three streams named *newUserStream*, *editStream* and *updateCount*. Finally, values from each stream are used to draw a graph and updates

other information to target web page. Dependency graph of target application is shown in the figure 5.18 and UI is shown in figure 5.17. Splitting of one stream into three streams is easily understood with the help of dependency graph. Thus, helping user to quickly understand how the application is working.

```
1  // Create our websocket to get wiki updates
2  var ws = new window.WebSocket("ws://wiki-update-sockets.herokuapp.com/");
3
4  var messageStream = fromEvent(ws, 'message');
5
6  var updateStream = messageStream.map(function(event) {
7  var dataString = event.data;
8  return JSON.parse(dataString);
9  });
10
11 // Filter the update stream for newuser events
12 var newUserStream = updateStream.filter(function(update) {
13 return update.type === "newuser";
14 });
15 newUserStream.subscribe(function() {
16 var format = d3.time.format("%X");
17 updateNewUser(["New user at: " + format(new Date())]);
18 });
19
20 // Filter the update stream for unspecified events, which we're taking to mean
21 // edits in this case
22 var editStream = updateStream.filter(function(update) {
23 return update.type === "unspecified";
24 });
25 editStream.subscribe(function(results) {
26 updateEditText(["Last edit: " + results.content]);
27 });
28
29 // Calculate the rate of updates over time
30 var updateCount = updateStream.scan(function(value) {
31 return ++value;
32 }, 0);
33
34 var sampledUpdates = updateCount.sample(Rx.Observable.interval(samplingTime));
35 var totalUpdatesBeforeLastSample = 0;
36 sampledUpdates.subscribe(function(value) {
37 updatesOverTime.push({
38 x: new Date(),
39 y:(value - totalUpdatesBeforeLastSample) /
40 (samplingTime / 1000)
41 });
```

```
42 if (updatesOverTime.length > maxNumberOfDataPoints)  {
43 updatesOverTime.shift();
44 }
45 totalUpdatesBeforeLastSample = value;
46 update(updatesOverTime);
47 });
```

Listing 5.7: RxJS - Live updates from Wikipedia

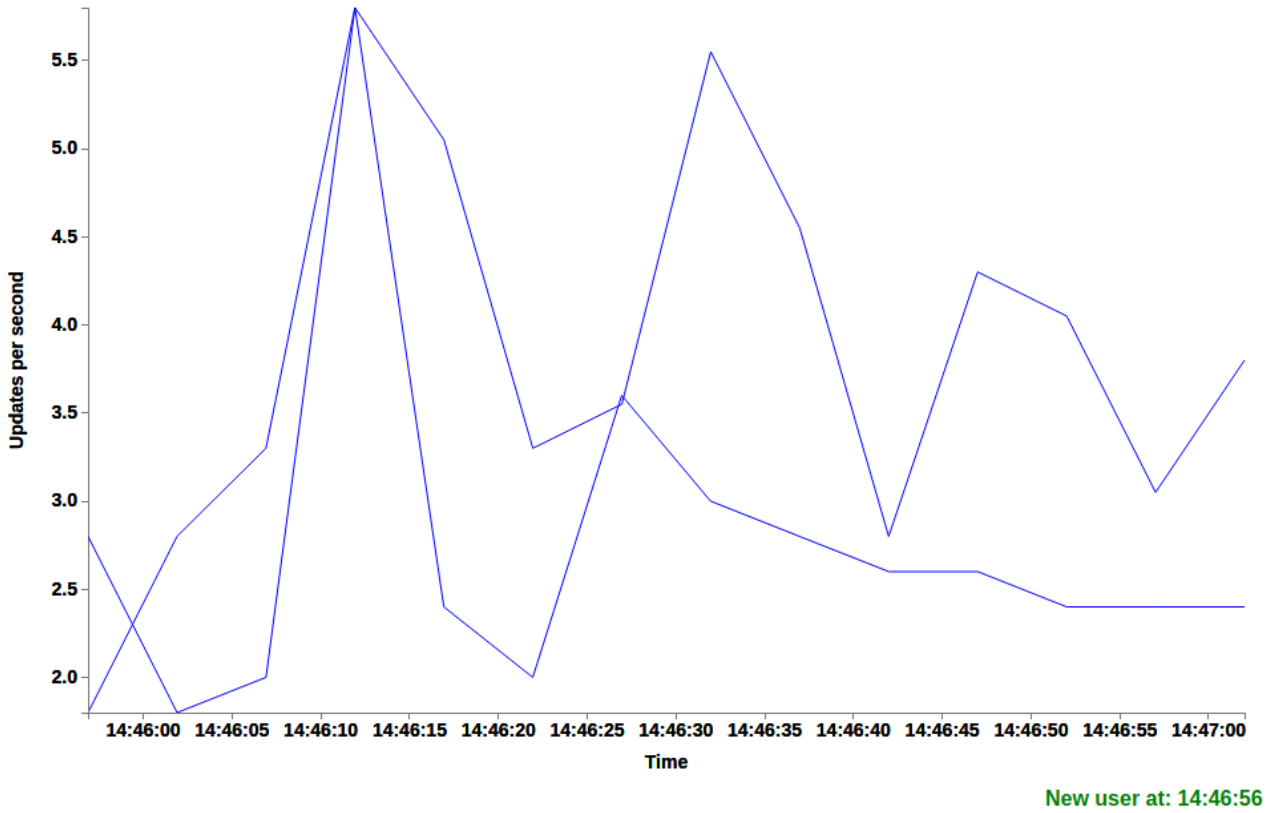


Figure 5.17: UI - Wikipedia updates

5.8 Existing design vs Current design

In this section we will discuss the improvisations made in this thesis from previous version of CRI. Listing 5.8, shows an example which counts the length of text input from the UI 5.19. In line 10, we are subscribing to *source* observable. Thus, every values emitted by *source* is received by the subscriber and set using function *setHtml* at line 5.

```
1 var source = Rx.Observable.fromEvent($toCount, 'keyup')
2 .map(function (e) { return 'length: ' + e.target.value.length; })
3 .distinctUntilChanged();
4
5 function setHtml(text) {
```

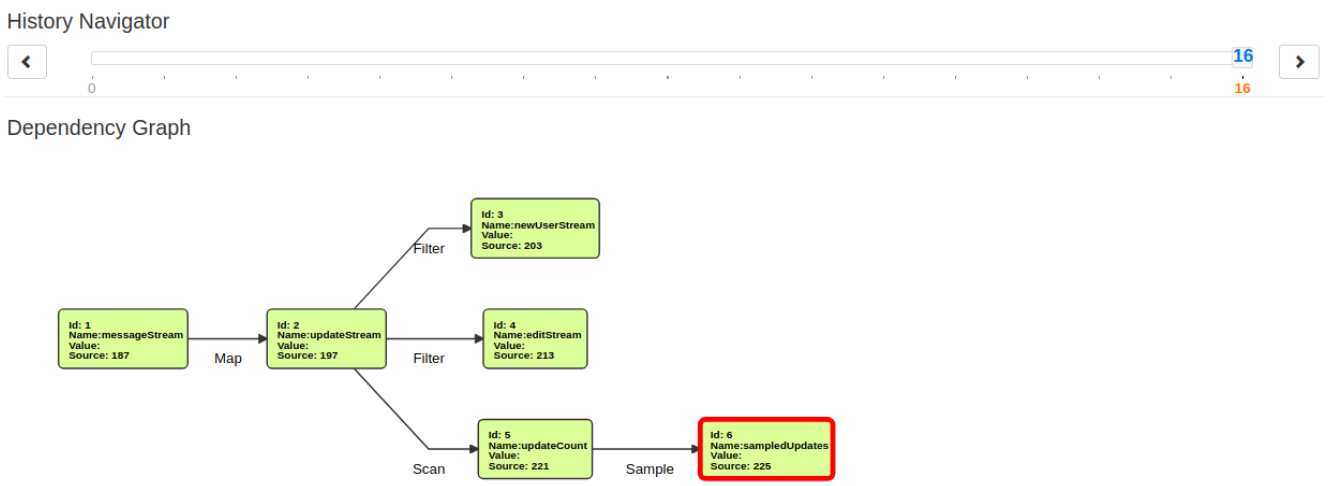


Figure 5.18: Dependency graph - Wikipedia updates

```
6   console.log(text);
7   this.innerHTML = text;
8 }
9
10 source.subscribe(setHtml.bind($result));
```

Listing 5.8: RxJS - Letter Count

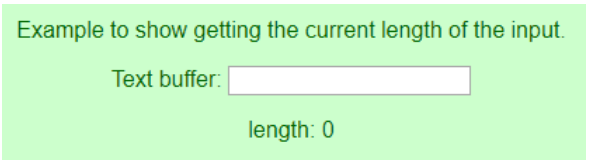


Figure 5.19: RxJS - Letter Count example

In previous version of CRI developed, the author is subscribing to each observable again to catch all the events using the code listed in listing 5.9. Here, *obs* represents an observable from the target application. Each value emitted is then logged to dependency graph using the method *sendToDevTools* at line 2. In current version, we have improved it and overridden RxJS library functions to catch all the events. Listing 5.10, shows relevant code. *Rx.Subscriber.prototype.next* method catches all the values emitted by all the observables from the target application. Thus, providing more control over all the emitted values and make necessary modifications.

```
1 obs.subscribe(function (x) {
2   sendObjectToDevTools({
3     content: {
4       'nodeId': obs.id,
5       'nodeType': obsType,
6       'nodeMethod': '',
7       'nodeRef': '',
8       'nodeValue': x,
```

```
      'sourceCodeLine': ''
    }, action: "saveNode", destination: "panel"
  });
}
```

Listing 5.9: Previous implementation

```
Rx.Subscriber.prototype.next = function (value) {
  var self = this;
  sendObjectToDevTools({
    content: {
    'nodeId': self._id,
    'nodeType': self.obsType,
    'nodeMethod': '',
    'nodeRef': '',
    'nodeValue': value,
    'sourceCodeLine': ''
    }, action: "saveNode", destination: "panel"
  });
 }
```

Listing 5.10: Current implementation

The disadvantage of previous implementation is, it logs the same values repeatedly because each observable is subscribed twice: once in target application and once in the CRI extension. This works fine for smaller application but there is huge communication overhead and excessive resource consumption. After typing the letters *TU* in the text-box from figure 5.19, total number of message communication in previous version of CRI is 36 and current version is 16. Resultant dependency graph for each version of CRI is shown in figures 5.20 and 5.21. There are 20 extra message communications, which consumes resources to communicate. The numbers vary to large extent in bigger applications. And current version of CRI provide more detailed information regarding what is going on in the target application. For example, look at the node with id 1, in both the figures 5.20 and 5.21. As the user types, CRI catches the events and maps them to display meaningful data to help the developers.

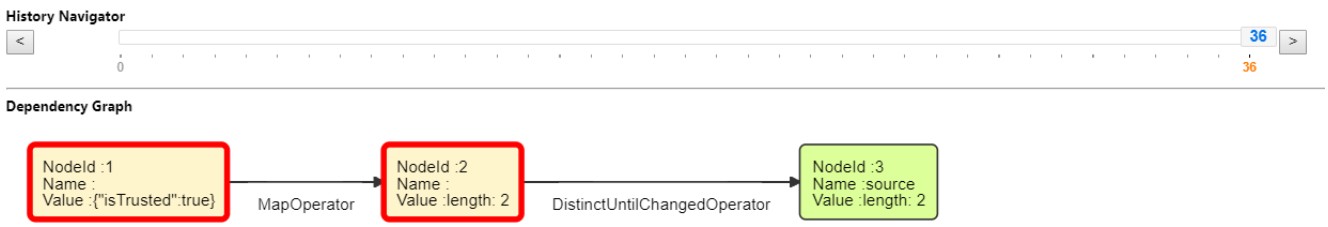


Figure 5.20: CRI - Previous version

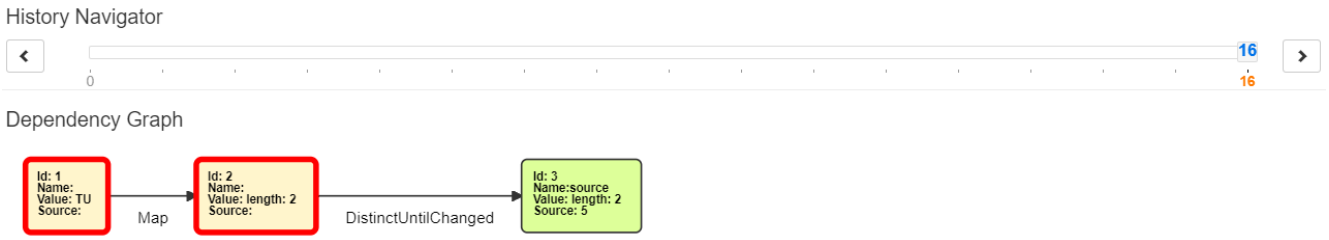


Figure 5.21: CRI - Current version

5.9 Profiling Time - Library calls from the User space

In this section we evaluate time taken by overridden RxJS library function calls for the applications discussed in this chapter earlier. Google Chrome provides developers the performance API - Google Chrome DevTools **Javascript Profiler**[11]. We used this profiling feature to evaluate performance of CRI extension. Javascript profiler provides more detailed information on which javascript functions were called and how long each took. This feature from Chrome can be accessed by navigating to DevTools and select the panel "Javascript Profiler". Then, user can click on start button to start the recording and upon stopping, the user can see the snapshot of the functions that were called and the time each function took. The figures 5.22 and 5.23 shows the time taken by functions without CRI and with CRI used respectively. Our primary focus is on **Self-time** view. Self-time is the time on specific function and the **Total-time** is the time executing sub function calls. In CRI, we are calling original library calls in our overridden functions, **Total-time** would include the time spent in the library activities. We can see that, time spent in the functions that we have overridden is minimal in comparison.

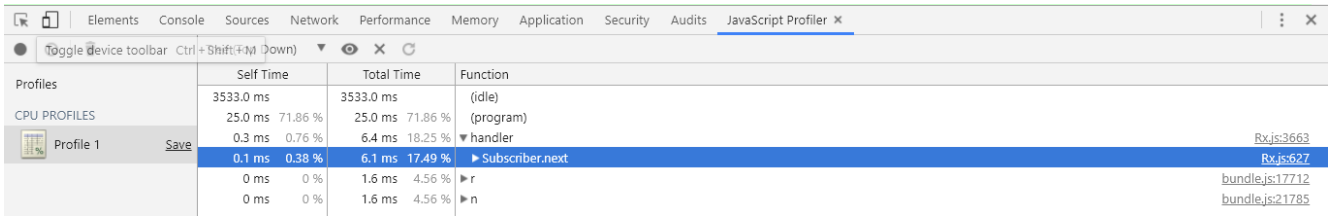


Figure 5.22: Javascript profiling - without CRI

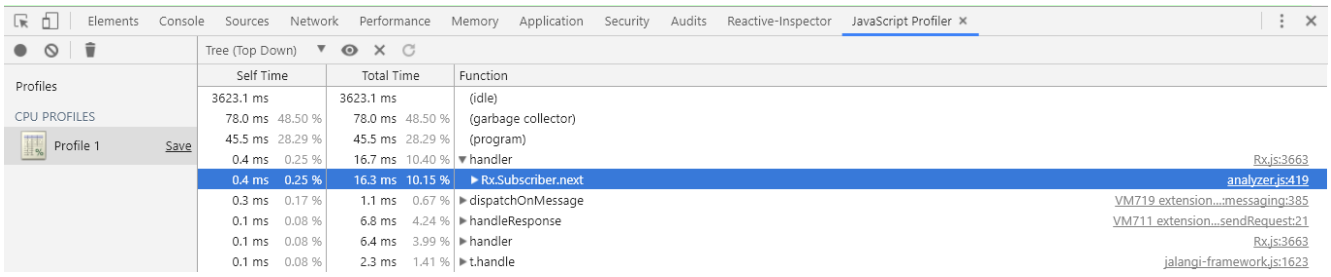


Figure 5.23: Javascript profiling - with CRI

5.10 Summary of Evaluation

The preceding evaluation illustrated with the help of various examples what the contributions of this thesis are. We have evaluated few other RxJS and BaconJS applications from the web and found that our extension works fine and support most of the operators from RxJS library. Additionally, we showed how visualization can help in getting good overview of reactive system and understand its internals at a glance. One main drawback of our extension is that it supports applications with javascript version upto ECMAScript 5.1[15] due to limitations of Jalangi framework[49]. Sometimes target applications have event streams which are continuously emitting values after every short interval of time, can cause heavy message passing and memory consuming. We are confident that the approach used in CRI extension works properly and helps developers to understand and debug reactive applications.

6 Conclusion and Future work

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

6.1 Final Remarks

The developed system is based on an already existing Chrome plugin developed at Technical University Darmstadt, which already provides the functionality to visualize dependency graph, navigating through history. The contribution of this thesis is to extend this functionalities to develop an advanced debugger and to support various RxJS operators.

The system developed in this thesis allows user to search for a particular node, to gain an understanding of its dependents and dependencies. Additionally, advanced methods to navigate the history has been developed. As shown in the evaluation, advanced debugging mechanisms for reactive programming have the potential to ease the development of reactive systems. Many ideas for further improvements are introduced in the last section.

6.2 Future Work

We believe that the CRI extension can very well be improved in the future. One aspect would be to try to instrument the javascript code using other methods such as ESPRIMA[29] due to limitations of Jalangi framework to support latest version of javascript. A survey on, how CRI helps the developers in real time to understand the reactive applications and find bugs in them, would provide further scope for development.

Another aspect would be to support other browsers such as Firefox and Safari. Also it would nice idea to integrate with NodeJS[4] ecosystem to debug the applications. A great help would be to improve the debugger in order to make it a so-called “edit-and-debug”. This would give the developer the possibility to change values in dependency graph and see the output on the fly. The plugin could also provide great assistance with performance analysis. It would be a very helpful to visualize the number of times a node is evaluated. This would help developers to find the bottleneck in their code.

List of Figures

Listings

Bibliography

[1] W. Abbas: Chrome Reactive Inspector. https://github.com/allprojects/chrome-reactive-inspector. last accessed: 23-08-2017.

[2] V. Authors: Callback Hell. http://callbackhell.com/. last accessed: 26-08-2017.

[3] V. Authors: ECMAScript. http://es6-features.org/. last accessed: 23-08-2017.

[4] V. Authors: NodeJS. https://nodejs.org/en/. last accessed: 27-09-2017.

[5] V. Authors: The reactive manifesto. http://www.reactivemanifesto.org/. last accessed: 23-08-2017.

[6] Bacon: Bacon.js. https://baconjs.github.io/.. last accessed: 31-08-2017.

[7] 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.

[8] H. Banken: RxFiddle. https://github.com/hermanbanken/RxFiddle. last accessed: 10-09-2017.

[9] S. Blackheath: Why (Functional) Reactive Programming? http://blog.reactiveprogramming.org/wp-uploads/2013/12/FRP4.pdf. last accessed: 26-08-2017.

[10] S. Bosnic; I. Papp; S. Novak: "The development of hybrid mobile applications with Apache Cordova". In: *2016 24th Telecommunications Forum (TELFOR)*. 2016, pp. 1–4. DOI: 10.1109/TELFOR.2016.7818919.

[11] G. Chrome: Javascript Profiler. https://developer.chrome.com/devtools/docs/profiles. last accessed: 27-09-2017.

[12] 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: https://doi.org/10.1007/11693024_20.

[13] C. Doblander; T. Parsch; H.-A. Jacobsen: "Geospatial Event Analytics Leveraging Reactive Programming". In: *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*. DEBS '15. Oslo, Norway: ACM, 2015, pp. 324–325. ISBN: 978-1-4503-3286-6. DOI: 10.1145/2675743.2776757. URL: http://doi.acm.org/10.1145/2675743.2776757.

[14] J. Drechsler et al.: “Distributed REScala: An Update Algorithm for Distributed Reactive Programming”. In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA ’14. Portland, Oregon, USA: ACM, 2014, pp. 361–376. ISBN: 978-1-4503-2585-1. DOI: 10.1145/2660193.2660240. URL: http://doi.acm.org/10.1145/2660193.2660240.

[15] ECMAScript: ECMAScript 5.1. https://www.ecma-international.org/ecma-262/5.1/. last accessed: 27-09-2017.

[16] J. Forsyth: RxVision. https://github.com/jaredly/rxvision. last accessed: 10-09-2017.

[17] J. Forsyth: RxVision playground. https://jaredforsyth.com/rxvision/examples/playground/. last accessed: 10-09-2017.

[18] J. J. Garrett: Ajax: A New Approach to Web Applications. http://adaptivepath.org/ideas/ajax-new-approach-web-applications/. last accessed: 26-08-2017.

[19] D. K. Gifford; J. M. Lucassen: “Integrating Functional and Imperative Programming”. In: *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*. LFP ’86. Cambridge, Massachusetts, USA: ACM, 1986, pp. 28–38. ISBN: 0-89791-200-4. DOI: 10.1145/319838.319848. URL: http://doi.acm.org/10.1145/319838.319848.

[20] Google: AngularJS. https://angularjs.org/. last accessed: 31-08-2017.

[21] Google: Chrome Developer Tools. https://developer.chrome.com/devtools. last accessed: 23-08-2017.

[22] Google: Chrome Storage. https://developer.chrome.com/apps/storage. last accessed: 10-09-2017.

[23] Google: Content Scripts. https://developer.chrome.com/extensions/content_scripts. last accessed: 08-09-2017.

[24] Google: Extending DevTools. https://developer.chrome.com/extensions/devtools. last accessed: 23-08-2017.

[25] Google: Manifest file. https://developer.chrome.com/extensions/manifest. last accessed: 23-08-2017.

[26] P. N. Hilfinger: GNU Debugger. http://people.eecs.berkeley.edu/~jrs/61bf98/reader/ucb/gdb.pdf. last accessed: 31-08-2017.

[27] R. Hoffman: Data processing system and method for debugging a JavaScript program. US Patent 6,061,518. 2000. URL: https://www.google.com/patents/US6061518.

[28] V. Huber: How Is Reactive Different From Procedural Programming? http://insights.dice.com/2014/01/13/how-is-reactive-different-from-procedural-programming/. last accessed: 26-08-2017.

[29] JQuery: Eprima. http://esprima.org/index.html. last accessed: 27-09-2017.

[30] F. Knüssel: BaconJs Blog. https://medium.com/@fknussel/event-streams-vs-properties-e55b53be8f42. last accessed: 31-08-2017.

[31] D. M. Kristaly et al.: “Web 2.0 Technologies in Web Application Development”. In: *Proceedings of the 1st International Conference on PErvasive Technologies Related to Assistive Environments*. PETRA ’08. Athens, Greece: ACM, 2008, 65:1–65:4. ISBN: 978-1-60558-067-8. DOI: 10.1145/1389586.1389663. URL: http://doi.acm.org/10.1145/1389586.1389663.

[32] I. Maier; T. Rompf; M. Odersky: “Deprecating the Observer Pattern”. In: (Jan. 2010).

[33] 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.

[34] 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.

[35] Microsoft: LINQ. https://msdn.microsoft.com/en-us/library/bb308959.aspx?f=255&MSPPError=-2147217396. last accessed: 06-09-2017.

[36] Microsoft: ReactiveX. http://reactivex.io/intro.html. last accessed: 26-08-2017.

[37] Microsoft: ReactiveX Subjects. http://reactivex.io/documentation/subject.html. last accessed: 26-08-2017.

[38] Mozilla: Javascript Object Prototype. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/prototype. last accessed: 10-09-2017.

[39] Mozilla: Mozilla Firefox. https://www.mozilla.org/en-US/firefox/new/. last accessed: 10-09-2017.

[40] Oracle: Oracle Debugger. http://docs.oracle.com/javase/7/docs/technotes/tools/windows/jdb.html. last accessed: 31-08-2017.

[41] C. Pettitt: dagre-d3. https://github.com/cpettitt/dagre-d3. last accessed: 14-09-2017.

[42] ReactiveX: Observable. http://reactivex.io/rxjs/class/es6/Observable.js~Observable.html. last accessed: 31-08-2017.

[43] ReactiveX: RxJava. https://github.com/ReactiveX/RxJava/. last accessed: 06-09-2017.

[44] ReactiveX: RxJs. https://github.com/ReactiveX/rxjs. last accessed: 06-09-2017.

[45] G. Salvaneschi: Reactive Inspector. https://guidosalva.github.io/reactive-inspector/. last accessed: 26-08-2017.

[46] 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.

[47] G. Salvaneschi; M. Mezini: “Debugging Reactive Programming with Reactive Inspector”. In: *Proceedings of the 38th International Conference on Software Engineering Companion*. ICSE ’16. Austin, Texas: ACM, 2016, pp. 728–730. ISBN: 978-1-4503-4205-6. DOI: 10.1145/2889160.2893174. URL: http://doi.acm.org/10.1145/2889160.2893174.

[48] G. Salvaneschi et al.: “An Empirical Study on Program Comprehension with Reactive Programming”. In: *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: ACM, 2014, pp. 564–575. ISBN: 978-1-4503-3056-5. DOI: 10.1145/2635868.2635895. URL: http://doi.acm.org/10.1145/2635868.2635895.

[49] Samsung: Jalangi2 Supported ECMAScript versions. https://github.com/Samsung/jalangi2#supported-ecmascript-versions. last accessed: 27-09-2017.

[50] C. Schuster; C. Flanagan: “Reactive Programming with Reactive Variables”. In: *Companion Proceedings of the 15th International Conference on Modularity*. MODULARITY Companion 2016. Málaga, Spain: ACM, 2016, pp. 29–33. ISBN: 978-1-4503-4033-5. DOI: 10.1145/2892664.2892666. URL: http://doi.acm.org/10.1145/2892664.2892666.

[51] 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.

[52] A. Staltz: How to debug RxJS code. https://staltz.com/how-to-debug-rxjs-code.html. last accessed: 07-09-2017.

[53] A. Staltz: The introduction to Reactive Programming you’ve been missing. https://gist.github.com/staltz/868e7e9bc2a7b8c1f754. last accessed: 26-08-2017.

[54] 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.

[55] I. Teo: Understanding the Observer Pattern. https://www.sitepoint.com/understanding-the-observer-pattern//. last accessed: 26-08-2017.

[56] A. University: How to debug RxJS. http://blog.angular-university.io/debug-rxjs/. last accessed: 10-09-2017.

[57] Wikipedia: Futures and promises. https://en.wikipedia.org/wiki/Futures_and_promises. last accessed: 26-08-2017.