Advancing the Chrome Reactive Inspector

Bachelor-Thesis von Benedikt Gross
Tag der Einreichung:

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

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Reactive Programming Technology

Advancing the Chrome Reactive Inspector

Vorgelegte Bachelor-Thesis von Benedikt Gross

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

Tag der Einreichung:

Erklärung zur Abschlussarbeit gemäß §23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Benedikt Gross, die vorliegende Master-Thesis / Bachelor-Thesis ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs.2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung überein.

Darmstadt, den 26. Feburar 2018 Benedikt Gross

Abstract

Since its introduction in 1997, reactive programming has become increasingly popular. Today there is an implementation of the paradigm in most popular programming languages. However, the development tool support, including Integrated Development Environments and debuggers, is still very limited. Traditional debuggers are designed to debug imperative and sequential code and are therefore not suitable to use with reactive applications. Although some development tools have already been developed for specific languages over the years, there are currently very few which target reactive libraries and frameworks for JavaScript. This is a rather surprising fact considering that the popularity of JavaScript [tio18].

In this thesis, we significantly improve the Chrome Reactive Inspector - one of the tools that provide support for JavaScript - and advance it with regards to being usable in a production environment. The Chrome Reactive Inspector provides many features to help users understand how dependencies between entities in a reactive application are formed at runtime and find logical errors within this applications source code. We implement means to connect the abstract representation with corresponding source code and we improve the User Interface to provide a better user experience. We also increase the runtime performance, which helps cope with resource demanding applications.

Contents

1 Introduction

The Chrome Reactive Inspector is an advanced debugging tool for Web applications using reactive programming with the libraries RxJS or BaconJS in JavaScript. The tool provides many features similar to those of traditional debuggers, such as breakpoints, tailored especially to the needs of users debugging reactive programming systems for these two JavaScript libraries mentioned above. This chapter gives a short introduction to reactive programming in general and the Chrome Reactive Inspector in particular.

1.1 Background

The reactive programming paradigm evolved from the Observer Design Pattern and therefore supports the publishing of changes to an object to a list of subscribers. While the Observer Design Pattern only supports returning one value at a time, in reactive programming it is possible for an object to return multiple values over time. Reactive programming provides means to specify dependencies between its entities in a declarative way. It is designed to handle event and data streams. This includes application User Interfaces and message passing in distributed system. Reactive programming is difficult to debug without an abstract representation of dependencies and streams. While there are already many implementations of the reactive programming paradigm for JavaScript, there are currently only two development tools designed to handle debugging in applications that use reactive programming in JavaScript: RxFiddle and the Chrome Reactive Inspector. RxFiddle supports the reactive library RxJS and the Chrome Reactive Inspector supports the libraries RxJS and BaconJS. Both tools enables advantageous debugging for reactive JavaScript applications without the need to modify the source code. The Chrome Reactive Inspector is a Google Chrome extension based on the extension Reactive Inspector for the Scala IDE for Eclipse. Being a Google Chrome extension provides the tool with platform independence, presumed a version of Google Chrome that supports extensions runs on the platform. The Chrome Reactive Inspector displays an abstract representation of the inspected application in form of a dependency graph. It also provides the user with the option to examine the creation of that graph in addition to tracking value updates traveling along the chain of dependencies at any time during the application's execution. In contrast, RxFiddle uses a marble diagram as the abstract representation.

1.2 Motivation

The Chrome Reactive Inspector is one of only two tools that support developers with debugging reactive JavaScript applications. Advancing it serves the purpose of improving debugging

experience for developers and reducing the need to modify the source code to enable a less sophisticated approach. Our goal is to increase usability of the Chrome Reactive Inspector and to advance its features so that it will eventually be usable in a production environment without limitation. The first step is to merge the previous work on the Chrome Reactive Inspector by Waqas Abbas and Pradeep Baradur. We then target the need for more detailed information about individual nodes of the dependency graph. We further aim at improving the performance and User Interface limitations of the Chrome Reactive Inspector when dealing with specific types of Web applications in order to cover a wider range of supported applications. In addition we examine the extension's User Interface in general with regards to a sound design, the required learning time and, eventually, their speed performing debugging tasks.

1.3 Our Contribution

In the last two sections, we presented a brief overview of reactive programming, introduced the Chrome Reactive Inspector, its origin and the scope of this thesis. In summary, we make the following contributions:

- Merging previous work on the Chrome Reactive Inspector
- Reworking the User Interface, including the reduction of cognitive load on the user, to better support them in performing debugging tasks
- Providing additional details to the dependency graph by connecting nodes with their corresponding lines of code
- Reducing computational resource consumption when working with applications that contain rapidly updated observables
- Providing a baseline for additional development targeting excessively created observables

1.4 Outline

In chapter 2 we explain reactive programming in general and the two JavaScript libraries RxJS and BaconJS in detail. We also examine existing tools for debugging reactive systems and why traditional debuggers are not suitable for this task. In addition, we present the previous work on the Chrome Reactive Inspector as well as its main competitor RxFiddle. Our contributions and the reasoning behind choosing specific approaches over other possible solutions are explained in chapter 3. Chapter 4 contains the verification of our claims and implementations as well as the results of the evaluation of the most important features in the Chrome Reactive Inspector for a set of test applications. In chapter 5 we summarize the result of this thesis and also point out possible targets for further development to increase the applicability of the Chrome Reactive Inspector.

2 State of the Art

In this chapter, we further explain the theory of reactive programming and approaches in debugging applications that use this pattern. We also examine existing debugging tools that target reactive programming frameworks and libraries as well as reactive programming libraries for JavaScript (JS). We discuss the Chrome Reactive Inspector (CRI) and work targeting the CRI prior to this thesis in detail in addition to its main competitor RxFiddle.

2.1 Implementation of reactive systems

Although there are many implementations of reactive systems for a multitude of programming languages, most of them are designed similarly and share the same building blocks. This section introduces (functional) reactive programming in general, and later the JavaScript libraries RxJS and BaconJS as specific implementations of the paradigm in detail. We also point out the main differences between the two libraries, their history, and their relevance by examining some statistics of the respective source code repositories hosted on GitHub [NA18] [bac18b].

2.1.1 Reactive programming

A reactive programming framework or library removes a lot of boilerplate code the developer usually needs to implement to propagate changes and events throughout a software system. It advances the well known Observer Design Pattern and overcomes several shortcomings of that approach. In the Observer Design Pattern, a *subject* propagates changes to its state to a list of previously registered observers [WH00]. A *subject* only returns one value, i.e. its state, while an *observable* (also called *behavior*) in reactive programming can provide multiple values over time as a stream. In addition to *observables*, which provide varying values over time, a reactive programming implementation usually includes a construct called *events* or *event streams* [bac18a], which handles discrete events over time.

What we refer to as an *observable* actually consists of three parts - a *producer* that produces values over time, the actual *observable* that stores these values, and the *observer* that notifies all *subscribers* (sometimes also called *consumers*). The notification of the *subscribers* is specific to *push-based* systems where the values are pushed through the reactive system when a new value arrives. This stands in contrast to a pull-based system in which values are only pulled on demand when they are requested by a *subscriber* of an *observable* or any of its descendants (i.e. other *observables* that depend upon it). *Observables* are categorized as either *hot* or *cold* [BR18]. A *cold observable* will create a new *producer* each time a *subscriber* subscribes to its *observer*. The

producer will be destroyed once the *subscriber* unsubscribes, i.e. the resources it uses are freed and it signals its own termination. This is referred to as the unicast approach. All *subscribers* to a *hot observable*, on the other hand, will (usually) share one *producer*. This is referred to as the multicast approach.

The reactive programming paradigm is declarative in its nature. This limits side-effects and is usually implemented with many *pure* functions, which serve as *operators*, that are used to transform, combine or change the behavior of *observables* and *event streams*. The *operators* return a new *observable* that depends upon the *observable* on which the *operator* was used. Using these operators, the developer declares dependencies between observables, which creates more comprehensible program code and structure [Sal+18, Why Reactive Programming?] than imperative programming style would produce.

2.1.2 RxJS and BaconJS

There are many JavaScript implementations of the reactive programming paradigm such as Cycle [cyc18], Kefir [kef18] or Most [cuj18]. An extensive and maintained list of reactive frameworks and libraries for JavaScript can be found at [Her18]. This section is focused on ReactiveX for JavaScript (RxJS) [rea18a] and Bacon (BaconJS) [bac18a] because these are the libraries the Chrome Reactive Inspector currently supports.

Rx.js

Reactive Extensions for JavaScript or RxJS [rea18a] is the JavaScript implementation of ReactiveX. ReactiveX implementations also exist for a number of other programming languages, such as Java (RxJava), Scala(RxScala) or C#(Rx.NET). In extension to the basic concepts of reactive programming, RxJS implements *subjects* and *schedulers*. A *subject* represents the implementation of an *event stream* and is the only option to realize a multicast *observable* since *observables* are *cold observables* in RxJS. *Subjects* can be both *observer* and *observable* at the same time. To convert a *cold observable* into a *hot* one, the functions *share* or *publish* can be used. They use a *subject* internally to track the shared *producer*. A *subject* will be terminated once the last *subscriber* unsubscribes. *Schedulers* are centralized dispatchers which are used to control concurrency and allow influencing the scheduling behavior when asynchronous functions such as *setTimeout*[rea18b] are called. *Observers* raise special events if the *observable* is completed or reaches an error state. The *observable* will not be reusable afterwards, even if it is a *subject*, which is a main difference to the later discussed BaconJS. RxJS provides several utility functions to convert values, arrays or events into observables [Bar17]. A basic example of RxJS code to show the general structure can be seen in listing 2.1 (Source: [rea18b] under manual/overview.html). The latest stable version of RxJS is version 5.5.6 with version 6.0.0-alpha already under active development. RxJS is licensed under the Apache 2.0 license and the source

code is publicly available on GitHub [NA18].

```
1 var button = document.querySelector('button');
2 Rx.Observable.fromEvent(button, 'click')
3   .throttleTime(1000)
4   .scan(count => count + 1, 0)
5   .subscribe(count => console.log(`Clicked ${count} times`));
```

Listing 2.1: Example of RxJS code.

Bacon.js

The basic concepts of BaconJS [bac18a] are *properties* - which represent the *observables* - and *event streams* that handle distinct events. In contrast to RxJS, BaconJS's *properties* are always *hot observables*. In addition, an error in an *event stream* or *property* will not cause them to terminate. Errors are, therefore, handled as any other value, which provides more fine-grained control for the developer. However, the *endOnError* function can be used if the termination is desired. Another advantage of BaconJS is the *spy* function, which can be used to observe the internal workings and, therefore, greatly reduces the required effort to create complementary tooling like the CRI or RxFiddle. It can also be used to easily create extensive logging without modifying the rest of the reactive application source code. According to the author(s), BaconJS was developed because they got frustrated with RxJS as its source code was not publicly available at the time and the documentation was minimal. They also claim that BaconJS has a more consistent and glitch-free stream/property behavior [bac18b]. However, they explain that RxJS has less overhead and consequently a better performance.

```
1 $("#username input").asEventStream("keyup")
2   .map(function(event) {
3       return $(event.target).val();
4     })
5   .toProperty("")
```

Listing 2.2: Example of BaconJS code.

BaconJS is still under active development, with the latest stable version being 2.0.0. It is licensed under the MIT license and the source code is publicly available on GitHub [bac18b].

While BaconJS is a newer reactive programming library and was developed to handle some shortcomings of RxJS, according to the GitHub statistics RxJS has still a much larger community. RxJS has 387 watches, 10540 stars, 991 forks and 185 contributors for the currently active repository, while BaconJS has 156 watches, 5839 stars, 328 forks and 83 contributors in total (last accessed: 31.1.2018 18:22). In addition, since RxJS was started earlier and due to its similarity to ReactiveX implementations in other languages, a number of tools and utility libraries exist exclusively for RxJS - including RxFiddle.

2.1.3 Debugging Reactive Code

Debugging is an important part of any form of programming. Virtually all of the most commonly used IDEs offer a precise detection of lexical and syntactic errors, such as misspelled keywords or missing control characters. They further apply the concept of breakpoints, are probably the most useful feature for in-depth code and state inspection at a precise point during the application's execution. In addition to simple breakpoints, modern IDEs also support conditional breakpoints, step by step execution, event logging, stack trace inspection and many other tools to debug code at runtime. Due to the declarative nature of reactive programming, traditional breakpoints and similar runtime debugging features cannot be used as easily as with imperative programming. For some IDEs the most basic form of declarative programming, chained function calls (also called Method chaining [Aut18f]), provide an obstacle for a user because the IDE's breakpoints are line- and not character-position-based. But even if the IDE can handle breakpoints inside anonymous functions for the usual navigation features like *Step Over* and *Step Into*, it is not trivial to balance between providing the necessary fine-grained stepping and having too many steps for the developer to iterate. For example one of the most advanced IDEs there is, Visual Studio (2017) for .NET, will step over the whole batch of chained functions if the developer uses *Step Over*, but will step into the first anonymous function in the chain if the developer uses *Step Into*. The developer can then use *Step Over* to enumerate the anonymous functions passed to the chained functions (see listing 2.3). This is a very specialized and desirable behavior that will match the intent of the developer most times when using the .Net Framework built-in *.NET Language-Integrated Query Expressions* (LINQ), but this behavior will break as soon as a custom function is added to the chain (*DoCustom* in the code example). Now the *Step Into* on line 1 will step into *DoCustom* for each entry in the list and will then step out of the batch of chained functions to line 10. If a similar code example in BaconJS is debugged with the step-by-step debugging feature of Google Chrome, it will actually step through the BaconJS library code even if the developer just uses *Step Over*. As BaconJS or other reactive programming libraries are not part of native JavaScript, Google Chrome has no means to determine if the user wants to step through the library code or not. This example shows that debugging declarative code is still a hard task for modern IDEs, especially if Out-of-Order execution (LINQ Expressions are executed Out-of-Order) is added. The IDE cannot always correctly interpret the developer's intentions when debugging declarative code with step-by-step execution. The developer might want to debug the custom chained function or they might want to iterate the anonymous functions passed as arguments (like "s => s.ToUpper()" line 2 and "s => s" line 8 in listing 2.3) to the chained functions. They might also want to step through framework functions such as *Select*, *Where*, or *OrderBy* in .NET themselves, which is made possible by the framework debugging option in Visual Studio 2017. For big collections or, in the case of reactive programming, observables with many submitted values, this problem is also increased if there are too many single executions of

one or more anonymous functions for the developer to step through. Another example for this is a breakpoint inside an anonymous function that is passed to a declarative function (like*Where* in C# or *map* in BaconJS), which halts the application for each element in the collection or data stream.

```
1   var lst = new List<string>{"test1","test2"};
2   var result = lst.Select(s => s.ToUpper())
3    .Where(s =>
4      {
5        var b = s.StartsWith("t");
6        return !b && s.StartsWith("T");
7      })
8    .OrderBy(s => s)
9    .DoCustom()
10   .ToList();
```

Listing 2.3: Simple example of .NET LINQ in C# to show the steps the Visual Studio 2017 for .NET debugger takes while debugging step-by-step.

This shows that a traditional debugger is not suitable for the use with declarative programming in general and reactive programming in particular. While the Visual Studio debugger is still somewhat useful to debug LINQ in .Net, because multiple specialized behaviors have been implemented over the years and LINQ expressions will most likely not cover the entire application and are often used to execute short tasks inside an imperative program flow, observables in a reactive application are usually much more intertwined and make up the general flow of the program; making a traditional debugger more of an obstacle.

As described in [msd18] reverting to the most basic debugging technique sometimes called *printf-debugging* (or in the RxJS terms *do-debugging*), to generate debug outputs or traces by directly modifying the source code, generally provides better results than using advanced features of a traditional debugger. Some shortcomings of this basic technique can be mitigated by using sophisticated tools like shown in [shi18] in the section "The Reactive Log" where the log of an application is visualized in a dependency graph containing code pieces as nodes. Do-debugging still requires modification of the original source code to trace the programs execution path and pollutes it with statements that do not contribute to the program logic itself, making the code harder to read. [SM16] describe the main issue to be missing abstractions and a "mismatch in the mental model", because traditional debuggers operate mainly on the runtime stack information and "do not consider any other abstraction within the running code" [Abb17]. Traditional debuggers, therefore, make it very hard for the developer to detect flaws in dependencies between reactive entities.

In the programming language JavaScript (JS) it is, unlike in compiled languages, by design harder to detect errors in the written code, because it is an interpreted script language and the JS code is not compiled before execution. The general advantage of a script language being

able to change the code at runtime and omitting the computation time needed to compile an application is shadowed by the fact that many bugs can only be found at runtime as well. In addition, Type-safe languages can detect many errors even before compilation by verifying that the used types are compatible. This makes reactive applications written in JavaScript even harder to debug than their counterparts in other languages.

Currently, there are few tools to help the developer debug reactive applications and apart from RxFiddle (described in detail in section 2.2.2) none of them, as far as we known, support debugging reactive JavaScript applications. One of those tools is [shi18] as mentioned above. Another is the Reactive Inspector for the Scala language [Sal+18] started and maintained by Prof. Dr. Guido Salvaneschi working in the Software Technology Group of the Technical University of Darmstadt, Germany. The Reactive Inspector is an extension of the Scala IDE for Eclipse. Many features and the basic concepts that are implemented in the *Chrome Reactive Inspector (CRI)* originate from the Reactive Inspector. This includes the representation of observables and their dependencies in a dependency graph called *reactive tree*, the option to query the history of this graph, or the option to search for a specific node in the current graph. They are described in details in section 2.2.1. One main feature of the Reactive Inspector which is currently not implemented for the CRI is called *Tree Outline*. Here the user can quickly jump to different areas of the dependency graph. This is useful for applications with large graphs.

2.2 Previous work on the CRI and its main competitor

In this section, we will cover the previous work on the Chrome Reactive Inspector as well as RxFiddle which is the main competitor for the CRI because both have a similar goal to help developers debug reactive systems in JavaScript with abstract concepts that go beyond traditional debugging.

2.2.1 The Chrome Reactive Inspector

The Chrome Reactive Inspector is based on the Reactive Inspector for Scala [Sal+18] and was developed by the Software Technology Group at the TU-Darmstadt Germany supervised by Prof. Dr. Guido Salvaneschi. It is implemented as a Google Chrome(Chrome) extension that extends the Chrome Developer Tools (DevTools) with another panel that inspects and instruments a target reactive Web application that is using RxJS or BaconJS. Prior to this thesis there have already been two Master theses targeting the CRI that provide the base theory and source code of this thesis.

Master Thesis by Waqas Abbas

[Abb17] The main goal of the Master Thesis by Waqas Abbas was to develop and implement a

concept for a Chrome extension based on the Reactive Inspector to debug reactive Web applications in JavaScript. One major difficulty was to find a viable solution for intercepting calls to the reactive libraries in order to retrieve the necessary information about observables and their dependencies that are used to build a dependency graph (called dependency tree in [Sal+18]) as well as to gather additional details like the variable name that directly correspond to observables and providing them with context by instrumenting the source code. An in-browser solution taken from a demo page [Gon16a] of Jalangi was used to instrument and analyze the source code directly. The main Jalangi framework [Gon16b] is currently not usable solely within a browser. Abbas also implemented several features to help the developer navigate and examine the dependency graph and its history. In this first version of the CRI, the user could already submit history queries that search the history of the graph for specific nodes or events like node creation, update or the creation of a dependency. In addition, the user could add *reactive breakpoints* - breakpoints that pause the program execution when a specific event occurs - or search for a node in a large dependency graph. At the time the CRI supported BaconJS[bac18a] and parts of RxJS[rea18a]. Abbas also added the first batch of small Web applications that could be used as test applications to manually verify features of the CRI.

Master Thesis by Pradeep Baradur

[Bar17] The main goal of the Master Thesis by Pradeep Baradur was to add full support for RxJS to the CRI, especially all *operators* and *subjects*. Since RxJS does not provide a uniform way to intercept all calls to the library like the BaconJS's *spy* function, this is a hard problem. For the actual implementation details see [Bar17] or the *rx-interception.js* file in the current version of the CRI. Baradur also reworked loading of the CRI's content scripts and instrumentation to only occur once the CRI DevTools panel is opened for an inspected Web application. This reduces the load on the browser if the CRI is not currently used. He also extended the search and history query features to allow searching for dependents or dependencies of a node. The focus of his thesis in comparison to Abbas's shifted from being mainly a proof of concept to advancing the CRI to be more reliable and ultimately being usable in a production environment. This included rearranging the UI to be more consistent and improving reliability in general as well as adding additional test applications.

Merging previous efforts

Since Waqas Abbas and Pradeep Baradur's works on the CRI partly overlapped and where developed in separate repositories, uniting both works was the first task for the thesis. Due to being developed separately after Baradur's Thesis started, the repositories shared no common history and due to many refactorings and renames in both projects as they advanced, merging the code base was not possible with automated tools and had to be done manually. Tracing the correlation between different changes was not trivial - in part because both theses were focused on

advancing the project and less on maintainability and modularization. For this thesis, we took the source code of Pradeep Baradur's Thesis as a base and added features that Waqas Abbas developed after Baradur's Thesis had started. The main reason for this is that Baradur removed many bugs and race conditions that happened in some rarer cases of code execution order, in his goal to make the CRI more reliable, which were outside of the scope of Abbas's work.

2.2.1.1 Exceptional properties of the CRI implementation

The special method that is used to intercept the loading of a Web application by dynamically reloading the JavaScripts to allow direct access and replace specified scripts with an instrumented version has several implications that break the usual *isolated world* [Aut18b] of Google Chrome extensions and their content scripts. The loading of the inspected HTML is stopped, the HTML file is scanned for references to JavaScript files that are then loaded and, on demand, instrumented. The scripts are then executed within the context of the content script that does the instrumentation. This is necessary so that both the inspected pages own scripts as well as the content scripts of the CRI which are responsible for the recording of the inner workings of the used reactive system share the same RxJS or BaconJS instance. But this approach also has some downsides, for example, a global variable in one of the inspected pages scripts may also be used in a script of the CRI could cause unforeseen errors - e.g. common libraries like jQuery [Fou18] that provide global objects will cause version conflicts. This also implies a potential security risk, because a Chrome extension runs in a *trusted* context and has access to many Chrome APIs (Application programming interfaces) that are not accessible from a normal Web application - the CRI could, therefore, be manipulated through shared global variables to execute malicious code by an inspected page. Another implication of this approach is that the CRI needs to filter all referenced JS files of the inspected page to exclude all RxJS or BaconJS scripts from the loading process to use its own versions. This will lead to errors if the versions do not match exactly. Dynamically loading the scripts that are referenced by the inspected pages HTML also cannot handle any module loader without special interception logic. As module loaders are widely used in modern Web applications and are even part of the ECMA6 [Aut18d] standard as the *import* keyword, this currently imposes a significant limitation on the number of Web applications the CRI can be used with. We provide more detail and discuss some of the possible options in chapter 5.

In addition to the special circumstances due to the dynamically loading of scripts in the shared context, the CRI also uses a Demo [Gon16a] in-browser version of the Jalangi [Gon16b] that does not provide the full features of Jalangi. Instead, it only contains features used in the Demo - for example, the source code position information of functions and variables do not contain the scripts file name. Jalangi itself contains hundreds of analysis customization options that are

not included in the in-browser Demo. Another obstacle presented by using the in-browser Demo script is that it is only available in the *minified* version that is hard to read and debug.

2.2.2 RxFiddle

RxFiddle [Ban18b] is a tool developed as part of a Master Thesis by Herman Banken at the Delft University of Technology that displays interactive marble diagrams for RxJS Web applications. RxFiddle instruments the used RxJS library to collect information about the inner workings of the library. It is available as a Web application at [Ban18b] and the source code is publicly available at [Ban18a] under the MIT license. RxFiddle is the main competitor of the CRI as it is the only other tool (that we are aware of) that tackles debugging for reactive systems in JavaScript. In RxFiddle the user can run JavaScript code visible on the left (1) and select a node from a simplistic dependency graph (2) to view a marble diagram showing details of its events, value streams (3), the interaction of operators and anonymous functions (4) passed to these operators.

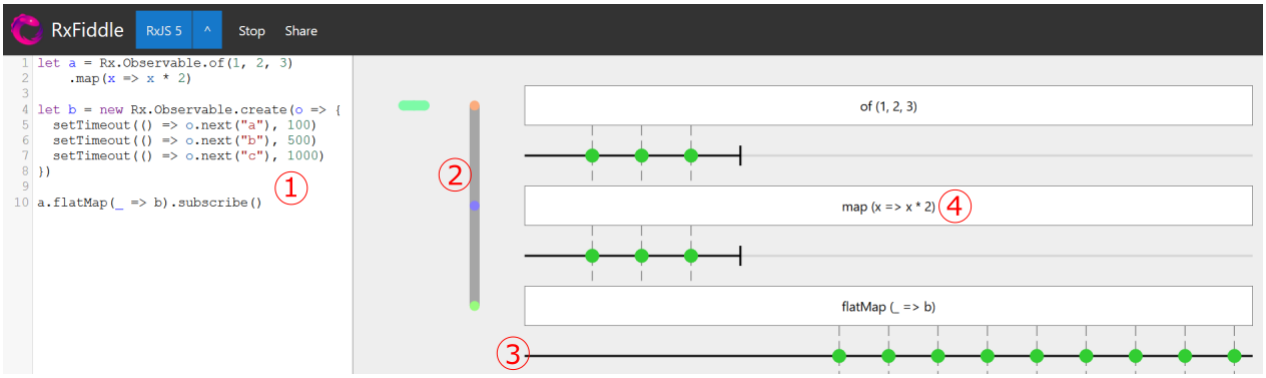


Figure 2.1.: RxFiddle Demo [Ban18b]

In the future, the author plans to support multiple additional collectors for RxScala, RxJava, RxSwift and/or RxNet [Ban18c], which would record and transmit the necessary data on these platforms to RxFiddle. This would enable the user to use the same debugging environment which would be especially beneficial for applications that consist of a mixture of platforms e.g. a Web application with a RxJS front-end and RxNet back-end on the Web server. One of the limitations of RxFiddle is that there is no direct linking of the abstract representation to the actual source code like the CRI provides with variable or method names if available for a node. This forces the developer to keep the reactive and declarative mindset but increases the time it takes to track down the responsible code that caused an issue that is detected in the abstract view. For example, the variables *a* and *b* do not appear anywhere in the abstract view (right side) of RxFiddle in figure 2.1. The right side of the variable assignment of *a* is present ("of(1, 2, 3)") and easy to find, but the assigned value to *b* is listed as "Observable" which is hard to connect if multiple observables without direct dependencies exist. Although it needs to be mentioned that

the CRI cannot handle multiple variables having the same variable name yet, even if they are in different scopes. RxFiddle also has trouble handling large applications properly according to the author [Ban18a, Issue 6]. The data collection works fine - the tool is just not equipped to display and navigate large dependency graphs yet. Another limiting factor is the performance while rendering the graph or marble diagram. The author proposes several options to tackle this limitation like *priority ranking*; for details see [Ban18a, Issue 6]. Although displaying large applications in one graph is hard for the CRI as well, RxFiddle requires more spaces of the User Interface (UI) for a single reactive operator and it currently has no option to zoom the graph of marble diagram. A large dependency graph for a single point in time used in the CRI, where some descriptive details of the nodes are always visible, is easier to examine than a large marble diagram that shows all values over time as marbles, but only shows node information in tooltips that need to be opened first. A similar issue for RxFiddle is the displaying of rapidly updating observables like an observable created by the RxJS *interval* function. Since each value update creates a new marble for the observable, the view becomes encumbered fast. As of this moment, the CRI also generates one or more steps for each update (if the node is not explicitly excluded). It is, however, still easily possible to examine node values in the first or last few steps of the history (jump to start or end and use the next/previous buttons). The CRI can also query the result of a recording which provides means to cope with large dependency graphs or Histories. In RxFiddle the limiting factor for fine-grained examination of nodes or values is the available space and the over-plotting of marbles. Details of nodes or marbles can only be viewed if the user hovers over them to open the tooltip. For a performance comparison between the newest version of the CRI and RxFiddle regarding *interval observables* see section 4.1.3 since increasing the performance of the CRI with this kind of observables was a goal of this thesis - further discussed in section 3.3.

3 Contribution

In this chapter, we discuss the features and changes introduced in this thesis as well as other implementation options and why we chose our approaches over the others. In the following sections, in lack of a better term, observable is used as a term to denote an actual RxJS observable but also as a placeholder for any reactive entity from RxJS or BaconJS depending on the context.

3.1 Advancing the User Interface

The User Interface (UI) is an important part of every application. A good UI helps the user focus on the task at hand by displaying the necessary information required for that task. For a website, this task may simply be to gather information about the subject of the website itself. An IDE usually provides a number of tools to reduce the manual work of the user by automating parts of the development process. There are many factors that need to be taken into account when developing a new UI or changing an existing one. A UI should be designed to match an applications specific use-cases while also keeping some familiar elements to guide the user. The main reason for including familiar elements is simplicity for the user: If the UI elements look familiar and match a user's expectations, it is much easier for the user to handle the UI as there are not as many features a user needs to learn how to use. In fact, it is also important to consider some of the characteristics of human cognition which we will not discuss in detail due to the limited scope of this thesis.

Reducing the cognitive load on the user

The main goal of the changes to the UI we implemented in this thesis is to reduce the cognitive load on the user. The user should not need to actively look for information in the UI; Instead, the UI should present its functions and information in an easily accessible manner. This way, the user spends more attention on their actual task and less attention on finding the information. The first change we introduced to reduce the cognitive load on the user is to move some input elements to a menu that is displayed at the top of the CRI's UI. Nearly every desktop application and many websites (e.g. http://reactivex.io/rxjs/, Windows File Explorer, Eclipse, Microsoft Paint, Adobe PDF Reader and many more) use a menu at the top of the UI. This adds structure and also removes many buttons from the normal working space which can then be used to display more relevant information. The user knows that the buttons are in the menu and opens it on demand. Except for the space required by the menu itself, the buttons hidden in the menu bar do not require space or attention by the user at other times. We examined our own usage

frequency, i.e. how often we use a certain input element, and estimated the tasks that include the input element to determine which elements should be part of the main UI and which should be hidden in a menu (or closeable panel for inputs other than buttons). We found that we rarely use the *Reset*, *Download* and *Pause/Resume* buttons. The latter was used most among them and since the CRI does not actually have many buttons yet, we decided to place it directly in the menu bar instead of moving it into the submenu where we placed the other rarely used buttons. The *Pause/Resume* button is also the only one of those buttons used in a normal workflow, i.e. in the special case of working with rapidly updated observables (see section 3.3) it is used regularly to counter performance issues or focus on a specific part of the execution. The other buttons of the UI are each part of a specific feature of the CRI and cannot be moved away from other elements which belong to that feature. These features from left top to bottom right are the *instrumented files list*, *history queries* and the navigation through their results, *reactive breakpoints*, *search*, *history navigation* and the *dependency graph*. The dependency graph and history navigation are closely tied together and represent the main part of the CRI. They are the working area where the focus of the UI should be. The other features are complementary to them and support the user in examining their content. Although *history queries* and *reactive breakpoints* are not needed to effectively use the CRI, they can help the user in almost any task. *Reactive breakpoints* may even be mandatory to discover issues with an application that otherwise could not be detected without a much greater effort - similar to normal Breakpoints in traditional IDEs. *Reactive breakpoints* are currently exclusive to the CRI, while the dependency graph with its history is just an alternative to the marble diagrams used in RxFiddle to abstractly represent observables and dependencies. In conclusion, *history queries* and *reactive breakpoints* should be a permanent part of the UI. We, therefore, decided to keep them where they were positioned in the CRI2 - above the history navigation. The search feature is not as important in most tasks. Searching and highlighting a specific node, its dependencies or dependents, is only useful for applications with large dependency graphs; In small graphs it is easier to find nodes manually. Therefore, the search feature could be moved to a closable panel that opens on demand similar to the search implementation in most IDEs where a user presses CTRL F or clicks the button in the menu to open the search bar. The reason we did not move the feature to a closable panel yet is simply that the area of the UI where the history query and *reactive breakpoints* are placed provides enough space, with standard screen resolutions for desktop computers, to keep the search there as well. With the current UI, this space is otherwise left empty. However, as soon as these parts of the UI change or another feature is introduced, the search should be moved to a closable panel. The remaining feature is the *instrumented files list*. It is used as a scoping feature to select which files should be instrumented. For small to medium-sized applications the list of files will normally include all relevant JS files and will not be changed at all. It is, therefore, reasonable to move it from the main UI to free the space that could then be used to increase the size of the dependency graph's canvas. After examining the

outcome of this change we found, however, that in case the user switches to another application that requires other files to be instrumented or forgot to include a JS file in the list, it is really difficult to detect the reason why the dependency graph is not showing the expected result. If the Instrument Files list is moved from the main UI, the user may lose track of the fact that not all JS files are automatically instrumented or, if they decided to instrument only a subset of JS files as a filtering measure, that this filter is still in place. It is also easier for new users to comprehend the need to select files for instrumentation if the list is part of the main UI. This could be mitigated by adding a dialog informing the user at the first start of the CRI, but for this reason, in addition to the reasons mentioned above, we decided the benefits outweigh the downside of a small part of the UI being occupied by a rarely changed input. Therefore, we kept the *instrumented files list* at the top of the UI and also added a red border around the text input in case the list is empty to help make new users aware of the missing input.

To further decrease the cognitive load when working with the CRI, we removed any text from nodes and their tooltips that does not carry any value. The less text is displayed in the UI, the less a user needs to read to find the information they require. This is most significant for the *Method* field in the tooltips since most observables do not correspond to a method and the field, therefore, is empty for most nodes. The *Value* field is usually set, but in case no value is available, there is no need to keep the label for the field. No information is lost if the field label for *Value* is simply omitted in that case. It is, however, important to display values that explicitly denote an *empty* value like *undefined*, *null*, or, in case of a value that carries positive numbers if it is set, -1. The *Name* field is also not set for every node. Figure 3.1 shows an example of nodes from the CRI2 with all fields as well as nodes from the current version. Another important aspect to consider is if to display a field directly on the node or rather in

(a) CRI version 2.0 **(b)** CRI version 3.0

Figure 3.1.: A sample of nodes before and after the changes introduced by this thesis.

the tooltip. The same principle that applies to general elements of the UI applies here as well. Only values for which it is necessary to always have them present, without the need to open every tooltip, should be displayed as part of the nodes. We evaluated for each field where they should be located regarding this criteria. The *Name* is important to identify relevant nodes, including those depending or dependent on such a *named* node. The *Id* is required to add *reactive breakpoints* and to identify nodes without a name from one step to another in case the nodes are repositioned due to a new edge being added. The *Value* is usually the property that

lets developers see if the JS code of the application works as intended. This can be achieved for example by checking if an observable fulfills the expectation of having a specific value at a specific point in the execution. The developer is also able to observe a value passing along the chain of dependencies. For these reasons, the fields *Name*, *Id*, and *Value*, as long as they have a value, should always be displayed as part of the node. The *Type* should be hidden in the tooltip, because often what type of observable the library uses internally does not matter to the user. For example in most scenarios for RxJS, there is no difference between a *FromEventObservable* and a basic *Observable*. The information should, however, be still present in the tooltip and not omitted completely because some types of observables do behave differently than others such as RxJS's *subjects* in contrast to its *observables*. The source code position labeled *Source* is also just used in few special tasks. One example is the user wanting to find the source code corresponding to a node in the graph to modify that code or inspect it further. The *Source* should, therefore, be displayed in the tooltip rather than the nodes themselves. The field *Number of Updates* is introduced by this thesis as a way to detect bottlenecks in the performance of the inspected application and to help identify nodes that cause the generation of a huge amount of steps in the CRI through excessive value updates. It is based on the Time Profiling feature of the Reactive Inspector [Sal+18], but for now, only provides the total number of updates on a node instead of detailed timing and value evaluation information. It is only useful for the purposes mentioned above and, therefore, should not be displayed on the nodes themselves.

An important cognitive task that a user needs to do often while using the CRI is to track changes between steps in the dependency graph's history. To track these changes is necessary to understand how the graph is constructed or to track value updates traveling along a chain of dependent observables. It is, however, not always trivial since the rendering engine *D3.js* [Bos18] used by the CRI sometimes moves nodes and edges around to better fit the available space. To help the user with this task, the previous version of the CRI already highlighted the nodes that were changed for each step in which a node was changed. We extended this feature to also highlight edges that were updated or added to further decrease the manual searching process. Another feature we added to the UI is a button that hides the tool section of the UI - everything except the menu, history navigation, and dependency graph - to allow the user to reduce distractions and free up additional space if they want to focus on the manual examination of the graph and its history. The button is located at the top right corner and has a small arrow icon instead of a text label to reduce its required space in the UI. The option to hide the tool section is especially useful when working with a small screen or a large dependency graph. Another button we added to the UI, as part of the menu, was a button to reload the inspected page and, therefore, the dependency graph as well. We discovered that reloading was already part of the usual workflow by pressing CTRL F5 and decided to add a button that allows users to achieve this without being aware of the shortcut. An example workflow in which a reload can be used is to create a *dependencyCreated* or *nodeCreated* breakpoint for a node which is

already displayed in the dependency graph, and then reloading the application to actually hit the breakpoint. As figure 3.1 shows, we also changed the colors used in the CRI UI. The node which was changed in the currently viewed step was highlighted by using a red border in the previous version of the CRI. The color red is a signal color normally used to show that immediate attention is required or to show that an error has occurred. In traffic signs and lights the color red is used to signal cars to stop or to be aware of dangers. Therefore, we decided to remove it from the normal workflow of the CRI. Highlighting the current node is important for the user to immediately identify the change between steps, but it does not require the user's full attention the whole time. It also neither represents an error, therefore the increased cognitive load on the user is unnecessary. The combination of red and green (*named* nodes are colored green) is also problematic for some users with a fairly common vision deficiency, although as the green is a very bright shaded they should still be able to distinguishable the border on a highlighted *named* node from the node itself. Because of these reasons we decided to use blue with light orange as the highlighting that has less contrast between the two shades but still enough to not void the highlighting. To this purpose, we chose a shade of blue and selected the colors for nodes, edges and highlighting from suggestions provided by a color scheme designer [Sta17]. We chose to keep the text color black because it is the most used color for text in general and is easy to read on most colored backgrounds. The nodes that do not correspond to a named variable were left without a background to further reduce the cognitive load. The brain is trained to ignore empty spaces, therefore, making the text and named nodes appear more important.

Another change we introduced to the UI of the CRI was the automatic reloading of the dependency graph if the *instrumented files list* is changed. This does not reduce the cognitive load on the user but removes the redundant click on the reload button. It is not very useful to change the list of instrumented files without restarting the inspected application and the recording. If the reloading is not triggered automatically, the user will even be confused that their changes to the instrumented files list do not take effect immediately.

3.2 Connecting abstract graph with JavaScript code

Nodes in the dependency graph that correspond to observables that are directly assigned to variables provide developers with some form of context. A *named* node can be used as an anchor to identify the node itself as well as other nodes up and down the chain of dependencies. It helps the user to separate parts of the dependency graph that are relevant to the piece of code they want to examine and parts that are currently not relevant. Without any orientation, it is very hard to interpret a shown dependency graph. The user can still try to identify nodes by their values or the values of their neighbors, but these values may be missing at certain stages of the application. This all boils down to the issue of connecting the abstract representation of the applications observables provided by the dependency graph with the actual JS source code

responsible for the creation of those observables. If an error is detected by examining nodes in the dependency graph, the user still needs to track down which lines of code are responsible for the issue in order to change them. As mentioned above, *named* nodes solve this issue in a way, but not all observables created during runtime are assigned to variables. In fact, it is not at all desirable, although this is possible to achieve by editing the source code. To assign the result of each function in a chain of functions to a variable negates most of the benefits of using chained functions in the first place. If the variables are not used by multiple lines of code and are just introduced to support the debugging tool, most of all the disadvantages of *do-debugging* apply here as well. We, therefore, evaluated several possibilities to provide nodes that do not have a corresponding variable in the source code with additional details that help provide some context to the user. A very basic form of context was already present in the previous versions of the CRI in form of the source code position information. This information was, however, not very accurate in case of chained functions because only the line-based start of the source code position information provided by Jalangi was used. In case of chained functions, the source code position actually spans a block that may consist of multiple lines of source code and is character-based. In addition, the information provided by the Jalangi Demo analysis does not contain the name of the JS file to which the position belongs. Another downside of this approach is that to look up the source code in another tool apart from the CRI and switching back to the CRI to check the position of the next node is very cumbersome to execute for more than a few nodes. The developer should be able to easily switch between source code inspection and examining the dependency graph because not all issues can be found in either of them when used alone. For example, if there is an issue in an anonymous function, a user cannot find this issue directly in the dependency graph. They may only see the result in node values that do not match the values that they expect with the original intention of the code in mind. The main difficulty to achieve the connection between the two views on the application by showing actual source code instead of just the position information is that the JS code which is used during runtime is instrumented with Jalangi to enable a detailed analysis. Jalangi instrumented code, however, is very hard to read as is shown in listing 3.1.

```
1 $sonWallet = J$.W(593, '$sonWallet', J$.F(585, J$.I(typeof $ === 'undefined'
      ? $ = J$.R(569, '$', undefined, true, true) : $ = J$.R(569, '$', $, true,
       true)), false)(J$.T(577, '#wallet-son', 21, false)), J$.I(typeof
      $sonWallet === 'undefined' ? undefined : $sonWallet), true, true);
2 $fatherWallet = J$.W(625, '$fatherWallet', J$.F(617, J$.I(typeof $ === '
      undefined' ? $ = J$.R(601, '$', undefined, true, true) : $ = J$.R(601, '$
      ', $, true, true)), false)(J$.T(609, '#wallet-father', 21, false)), J$.I(
      typeof $fatherWallet === 'undefined' ? undefined : $fatherWallet), true,
      true);
```

Listing 3.1: Example of RxJS code.

For example, "J$.W" signals the analysis engine that a write operation is executed when the function is called. This makes it unfeasible to directly show the instrumented code to the user. In addition to being very hard to read, the instrumentation of the source code is an implementation detail of the CRI and should not be exposed. Additionally, the code position retrieved from the Jalangi analysis corresponds to the position in the original source code without instrumentation. Note that the JS code that is used during runtime is also dynamically loaded and, therefore, cannot be navigated to out of the box with Chrome DevTools navigation API. Although this is mitigated by adding "//# sourceURL=" with a custom URL to the bottom of a script. Since showing the Jalangi instrumented source code is not useful for a normal user, we decided to tie the custom source URL to a new option called *CRI Developer Mode* which is located in the options page of the CRI. The different approaches discussed in the following section are designed to enable a user to use both the abstract representation and the source code in tandem.

3.2.1 Evaluated possibilities of displaying source code

Using the *chrome.debugger* API

One possible option is to use the *chrome.debugger* API to open and navigate to the corresponding position of a node. To navigate to a line number in a file if the actual file URL, which can differ for dynamically loaded scripts due to many factors, is known is easily possible. For the reasons explained above, however, navigating to the *instrumented* code is not desirable. A non-instrumented version of the source code seems to exist in Chrome's *Source* windows as a dynamic script, but it is not actually used at runtime. This approach provides a user with syntax highlighting through Chrome and other features like search with regular expressions. They are also able to inspect the whole code of the application if required, not just the piece of code corresponding to a single node. The downside of this approach is that the user needs to leave the UI of the CRI to access the source information. For a single node, this is not an issue, but it is really cumbersome to inspect the source information of multiple nodes, even if the navigation to the position is automated. In addition, it is not possible to highlight the code block that corresponds to the inspected node, which would be especially useful for observables corresponding to the middle of a block of chained functions. The start and end of the block would have to be manually searched for by the user. Another downside that most likely confuses the user is that some features of Chrome's DevTools work as expected, but others do not and there is no way for the CRI to display this disparity. Due to the fact that the code shown to a user is not actually being executed, all features specific to runtime interaction like breakpoints, direct code evaluation or watches will not be usable. A user familiar with Chrome's debugging capabilities will most likely expect them to work properly. A modification of this approach is to display the code in another IDE than Chrome. As some modern IDE's provide a featured to navigate to a function's declaration (in WebStorm called "Go to declaration") which is more scalable than us-

ing a *string* based search for enterprise-sized applications. Although this feature is not available in Chrome, having the user install another IDE and use it in parallel with Chrome will increase the dependencies of the CRI and the development effort coupled with that dependency while still suffering from all the downsides of this approach explained above.

Using an integrated lightweight IDE

Another possibility is to use a file viewer that is opened, for example, as a closable panel to show the full source code. Depending on the implementation this approach provides full customization options of the code presentation to the user. In theory, advanced syntax highlighting could be applied, i.e. syntax highlighting based on the usage and structure of functions like WebStorm uses to detect classes and other patterns. It is also possible to highlight observables in the source code while the user hovers over the nodes in the dependency graph, or even permanently coloring the source code based on the abstract representation. The main problem with this approach is the development effort required to achieve a viable implementation. There is no library available, that we are aware of, which provides the needed functionality and customization options. To develop such a library ourselves requires nearly the same amount of development effort needed to implement a traditional IDE for JS since it basically is a lightweight JS IDE displayed in a panel of the CRI. Another issue with this solution is, in addition to the required development effort, that nodes which are close in the dependency graph because they share a dependency do not necessarily correspond to lines of code that are located in close proximity to each other. The code may even be split over several JS files in extreme cases which requires additional view options by any lightweight IDE apart from just showing one JS file.

Mirroring interactions to a non-instrumented copy of the application

An approach that is possible in theory, but will not withstand a detailed evaluation is to mirror any interaction to the inspected application to a copy of it. The idea is to have one application with instrumented JS files, and one where the JS files are not instrumented. This would enable the CRI to use another approach like using the *chrome.debugger* API to display the actual source code used at runtime while retrieving analysis data from the instrumented mirror application. It enables the user to benefit from every feature of Chrome or another IDE when debugging JS. The problems introduced by copying the entire application and mirroring interactions to it, however, go far beyond the doubled resource consumption. One major issue is the actual recording of user interactions to the original application. There are many tools like Katalon Studio [LLC18] using Selenium [Aut18i] that can be used to record and replay interactions to a Web application. These tools are often used to automate the testing of Web applications. Most of these tools are, therefore, designed to ignore certain aspects of interactions like the positioning of elements to make the tests more robust against small changes. Although there are some tools that use pixel positions to record and replay interactions. We could, however, find not a

single tool that records mouse movements and replays them exactly. It is possible to develop a solution ourselves, but the mentioned tools solve many issues such as always replaying the interactions with the exact same screen resolution. This is especially important if the layout of the applications UI reacts to different screen resolutions. Another issue with replaying interactions to the copied website are security restrictions imposed by Chrome (and other modern browsers) itself. Chrome restricts specific interactions when they are performed by Web applications or Chrome extensions instead of a (human) user. This is to protect users from attackers using custom extensions or Web applications. An example of these restricted interactions is the using of Chrome's context menu or the operating system's clipboard. The reasoning to restrict both are similar and are due to the fact that these features are usable to breach boundaries otherwise monitored by Chrome itself or in case of the clipboard by the operating system. Since these interactions are restricted, it is impossible to replay them on the mirror Web application. This voids all other attempts to replay other types of interaction because the two applications will desynchronize eventually. The danger of the two applications becoming desynchronized is also created by observables or JS code which are time sensitive. For example, an observable or function which is scheduled to be executed or updated every few milliseconds and has any side effect, such as increasing a value every time the execution happens, needs to be mirrored to the copy exactly to maintain matching values. This also includes any JS code that is execution-order dependent. The same execution order must be maintained, which is especially hard for asynchronous code even though JS's runtime is event-based [Aut18a] and does not actually execute code in parallel. In summary, any code dependent on the order of execution or anything timing related needs to be mimicked precisely to keep the mirror in sync with the original application. This cannot be guaranteed for all possible applications in a normal runtime environment.

Source Code Tooltips

The solution we decided to implement is an extension to the already present tooltips for nodes in the dependency graph. The tooltip for a node, if the user hovers over it with the mouse pointer, changes to a short snippet of source code which opens if the user presses the CTRL key. The displayed source code corresponds to the code where the observable represented by the node originates from in addition to a few lines of code before and after the relevant code as orientation. The details of the implementation are explained in the next section. This approach, however, has some limitations. If the node belongs to an observable at the end of a long chain of functions or an anonymous function is used which covers several lines, the code piece may actually be too long to show in a tooltip. If the code is not wrapped properly the code will also be hard to read in a tooltip due to the limited space. In addition, it is also not possible to use any kind of tool such as a search feature inside a tooltip. As the tooltips are relatively short, this issue is not too serious because the code snippet can quickly be read altogether. Another limitation of this approach is that it is not possible to view the full source code. However, since

the user cannot change the source code in the CRI either, they have to switch to another tool at some point in the debugging process anyway. For now, the Source Code Tooltips do not provide syntax highlighting and, if added in the future, it is important to note that syntax highlighting of short snippets is not very effective in JavaScript. Since JavaScript is a dynamically typed language, in order to correctly highlight language elements beyond keywords, it is necessary to consider how the elements are used in other areas of the source code. The usefulness of Source Code Tooltips is also greatly decreased if a function reference with an identifier instead of an anonymous function is passed to an *observable operator*. Some other patterns like using callbacks that are not realized as anonymous functions or functions that create other functions also have the same issue. An example of this is shown in listing 3.2. The source code corresponding to the *named* node "fatherWalletValue" ranges from line 4 to 5. If the function declaration is not directly placed nearby and, therefore, shown in the tooltip as well, it is not possible to view the implementation of "createFunction" in the UI of the CRI.

```
1 function createFunction(add) {
2   return (value) => value + add;
3 }
4 let fatherWalletValue = sonWalletValue
5   .map(createFunction(10));
```

Listing 3.2: Example of using a creation function in RxJS.

The greatest benefit of using Source Code Tooltips is that they compliment the dependency graph very well. A user does not have to leave the CRI to view the corresponding source code and the abstract representation is still the focus of the CRI. The tooltips solve the issue of providing additional details for nodes that do not have a corresponding variable name. In fact, they provide additional information for these nodes as well. Using tooltips also has the advantage that they are easy and fast to open and close again. Therefore, the user is able to inspect several tooltips in short sequence. This helps them to find and remember nodes without a corresponding variable name between multiple uses of the CRI by other means than remembering the Ids. The Ids will also change if the user modifies the source code in a way that adds new nodes to the dependency graph or changes the order of existing ones. Another benefit of using Source Code Tooltips is, that we are able to highlight the exact source code block responsible for a node even within a chain of functions. This is similar to the options provided by using a custom lightweight IDE. In contrast to using another *full-fledged* IDE, however, using tooltips inside the CRI does not introduce new dependencies. Another very important feature of using tooltips is that they do not require additional space in the UI. Using a panel that opens on demand as a pop-up or as part of the UI would result in a resize and movement of other UI elements or in case of a pop-up overlap parts of the UI until it is closed manually. Instead, the normal tooltips open when a user hovers over a node and the tooltip will be replaced when they hold the CTRL key. If they release the key or move the mouse away from the node, the tooltip will respectively change back to the

original one or close automatically. This fits well in the already existing workflow of inspecting several nodes with the mouse pointer and is in our opinion very intuitive to use. There was, however, no User Study conducted as part of this thesis to verify that this is true for the majority of users. We chose this approach based on the benefits it provides, even though there are some downsides which will require additional development effort to overcome in the future.

3.2.2 Implemented Solution - Source Code Tooltips

In this section, we explain the implementation of the Source Code Tooltips in detail. To retrieve the non-instrumented source code necessary to show in the tooltips, we store them in a dictionary before executing the instrumentation. To provide the source code corresponding to a specific node in the dependency graph, we first extended the Jalangi library used in this thesis. As explained in chapter 2, the used Jalangi library is not actually the full Jalangi framework, but just an in-browser version developed for a single demo. It, therefore, does not support many features of the actual Jalangi framework. One of the missing features is the required JS filenames that are not included in the position information provided by the demo Jalangi. We, therefore, decided to implement an ad-hoc solution to retrieve the filename. There is no way to determine the source JS file inside the Jalangi analysis because the Jalangi sandbox object is shared by all JS files and the information is not submitted as a parameter to any of the analysis methods. The only solution with minimal changes to the used Jalangi library, which is desirable if the library is ever updated, is to replace the J$ object with a shallow copy of the original J$ object. The copy provides the required information to the analysis and then calls the original. The "W" function of Jalangi is responsible to intercept variable assignments and is used to link nodes to a corresponding variable name. We extended this function and any function responsible for passing the recorded information to the analysis engine to accept the filename as an additional parameter. This replacement is done for every JS file with the corresponding filename. However, since the Jalangi object is quite large, we decided to exclude inline JS scripts directly embedded within the HTML. Inline scripts are often very short and cannot be taken out of their context in the HTML easily. Otherwise, it would be possible to collect every inline script and consolidate them in one artificial script that is then instrumented and modified with the replacement J$ object. Therefore, one copy for every inline script is necessary. The shallow copy function of JQuery we used duplicates only references and not data, but for large objects with many functions, this still has a performance impact if done repeatedly. There is, however, not actually a downside to omitting the inline scripts from the Jalangi extension. We simply know that any node which stems from an inline script has, if available, a source position information which does not specify a filename. The filename of these nodes is labeled as *html*.
With the implementation so far, only nodes with a corresponding name will have a Source Code Tooltip, because only variable assignments were reasonable to capture in the previous versions

of the CRI. We, therefore, extended the analysis engine (located in jalangi-analysis.js) to also intercept function invocations. This enables the recording of source code position information for functions and their parameters. We only check the *return* value of the intercepted functions, if they are an observable, because arguments of functions either are not observables, or they are created by a call to the reactive library and should already have been recorded. To retrieve the filename information as well we extended the corresponding functions of the J$ object similar to the extension of the variable assignment interceptions. With this implementation, it is possible that an observable is recorded multiple times for the same node, for example, if a node is assigned to a variable inside a function and then returned as the result. In this cases, it is important to always use the position information intercepted from the variable assignment. The reason for this is that if the node in the dependency graph has a name from a variable, the Source Code Tooltip should always show the assignment of that variable in order to not confuse the user. Note that since we retrieve the source code position information from the Jalangi analysis, the Source Code Tooltips will not work for nodes that stem from JS files that are not selected for the instrumentation.

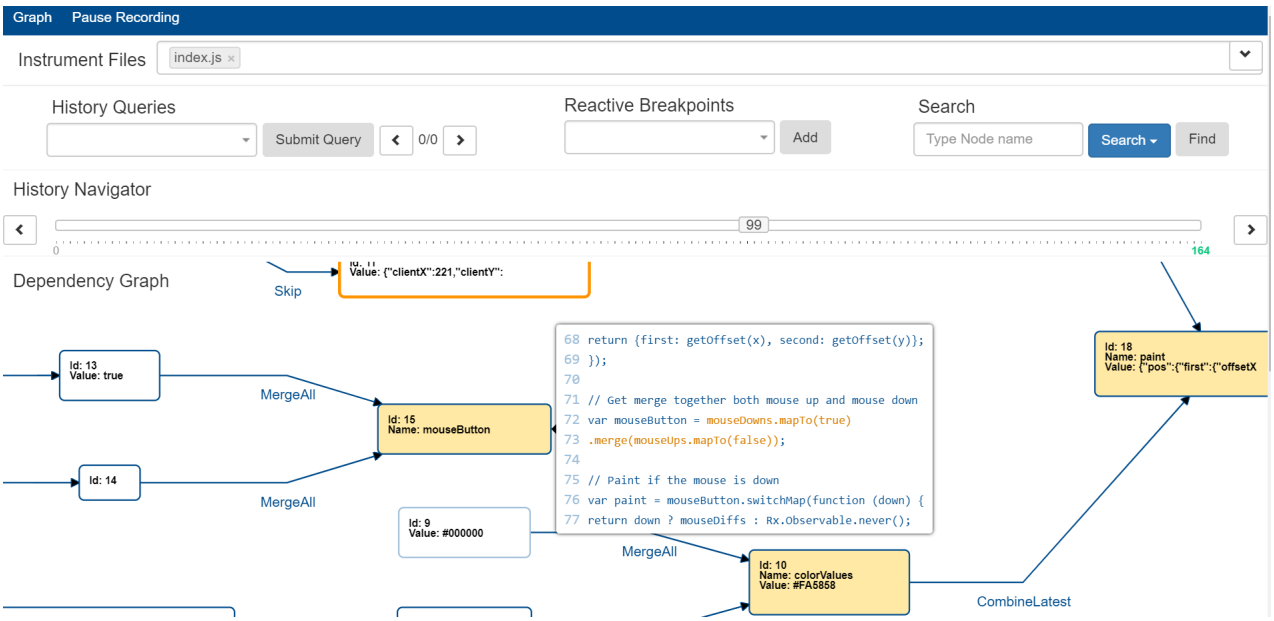


Figure 3.2.: A sample of CRI3 UI with opened Source Code Tooltip.

We made several additions to the UI of the CRI in order to show the Source Code Tooltips in the way described above. The class TooltipManager (located in tooltips.js) handles the creation and displaying of tooltips for all nodes in the dependency graph. It adds the UI events used to switch out the tooltips and requests the code needed to construct the Source Code Tooltips from the instrumentation script if the tooltip was not opened before in this step for that specific node. The used tooltip library tipsyJS [Fra18] which was used in previous versions of the CRI is no longer actively developed. We still decided to continue using it, because it provides a simple

way to swap out tooltips as it accepts functions for any of the options passed as parameters to the library. However, we had to modify the library to include a fix for a rendering issue that was already submitted as a pull request to the GitHub repository of that library but not accepted due to not being actively developed anymore. In addition, to enable the capturing of keyboard events for the dependency graph it is necessary to explicitly make the containing HTML element focusable. This is done by adding a *tabindex* attribute. Since the container also needs to have the keyboard focus in order to receive the keyboard events, the container is automatically focused when the user moves the mouse pointer over the dependency graph. The mouse events used to open or close a Source Code Tooltip of a node are registered when the user hovers over the node and removed when the mouse leaves the area of the node. We also chose to give the user a visual clue if a node has a Source Code Tooltip. This prevents users having to probe each node if a Source Code Tooltip is available. The border of nodes that cannot be linked to a source code position is, therefore, displayed faded. We chose this specific visual representation because having no Source Code Tooltip for a node means the node does not provide any context information and the normal tooltip will most likely also not provide any valuable information. One could argue that the Location information could now be omitted from the normal tooltips as well, but it is currently the only way for the user to see from which JS file the node originated since Source Code Tooltips currently do not show the filename. However, this may be changed in the future. An example of the CRI UI with an opened Source Code Tooltip can be seen in figure 3.2.

3.3 Rapidly updated Observables

In this section, we discuss the difficulties when handling applications with rapidly updated observables and our approach to increasing the CRI's ability to cope with them. If the value of one or more observable is changed rapidly, each change will result in a new step in the dependency graph's history. Observables that are, for example, created from timers, mouse movement or a network connection submitting messages, change their value very often in a short period of time. The previous version of the CRI is not able to handle rapidly updated observables well. For a detailed performance comparison, see chapter 4. However, there were two features added to the CRI that enable the user to still use the CRI with rapidly updated observables to some extent. One feature is the option to pause the recording and resume it at a later time. This feature is useful for applications which run some time before the limits of the CRI are exceeded and allow the user to click the *Pause Recording* button before that happens. The user is then able to examine the recording up to this point without any issues related to performance. This is due to the fact that user interaction is slow enough for the CRI to handle, while steps may be generated by the thousands in the same period of time if the recording is not paused. The other feature that was added is the ability to exclude certain nodes from the

recording process, although this is cumbersome to do. The user needs to detect and remember the Ids of nodes which are being updated rapidly and then add these Ids to a list located on the options page. This ad-hoc solution can still be useful in certain cases, but in others, the full exclusion of a node will cause errors in the recording process or show a disrupted dependency graph. In the following, we explain the increased demand for resources, the steps implemented to increase performance in this thesis and the limitations of the UI in handling these type of observables in general.

3.3.1 Difficulties and previous approach

Not only does the generation of hundreds of steps decrease the expressiveness of each individual step, with some exceptions, but it is also very demanding on computational resources. Messages between content scripts, background scripts, and the extensions panel have to be transmitted in addition to saving the dependency graph's history. This also increases the number of rendering operations of the graph itself which requires a lot of size and position computations. If the computational effort exceeds the capabilities of Chrome and the underlying system, the CRI and sometimes even Chrome itself will stop working. There is a threshold regarding the minimum time span between individual updates that if undercut leads to the CRI crashing at some point in the execution. The reason for this is the accumulation of requests if more requests are submitted, for example to the Chrome message or storage API, than requests are processed. Over time the resource demand on scheduling requests and managing the JavaScript runtime's event queue increases beyond the available computation power. In this case, the UI becomes unresponsive and Chrome will eventually terminate the extension. In the remainder of this thesis, we will refer to this type of performance issue as *request buildup*. The steps that are often still significant to a user belong to the start of the history. They usually contain some form of setup phase, were observables and their dependencies are created, but there are no values submitted to them yet. An example of this is visible in many of the test applications, as value updates will not occur until the user starts using some form of input element. These steps can still be examined and provide valuable information about the created observables that is not value-update related. This setup phase, however, is not present in every Web application. If the application does not have a setup phase, the history becomes hard to examine. Even if there is a setup phase some parts may be implemented with something like the *Lazy Loading* Pattern [Aut18e] in which the creation of observables will just happen on demand. There may also be no setup at all. It is possible to create observables dynamically - i.e. the type of dependencies used are a result of some form of computation or configuration - which will not necessarily happen at the start of an application. Another case in which the history becomes hard to examine for a specific task is if the application has multiple busy phases. Such an application may wait for some message from an external component that results in many computations and updated observables. Since

the history does not provide an actual timeline and will show steps in a linear sequence these busy phases become very hard to identify and distinguish. Another difficulty that stems from the *Pause/Resume Recording* features is that pausing the recording leads to a broken history. There is no visual feedback in the history navigation that signals the user a pause has happened at some point in the history. If the recording is paused and later resumed, the steps before and after this gap are not sequential anymore, even though the history indicates the contrary. After the recording was paused, the dependency graph will actually show a false image of the variables as their value may differ from the actual variables in the application. This difference only persists until every node is updated once again, but if an observable or dependency is created during the pause, it will not be visible at all. The later is somewhat intentional for excessively, dynamically created observables discussed in section 3.4. But this still leaves the user to remember at which step they paused the recording or they will get confused if they examine the dependency graph while stepping over the gap.

3.3.2 Reworking the dependency graph History

To reduce the computational resources required to record the dependency graph's history and implement a more sophisticated storing mechanism, we reworked the previous implementation. The history records every change to the dependency graphs as a new step. While previous versions of the CRI stored the whole graph for each step inside an array for simplicity. This means the required memory will increase proportionally with the number of steps. However, the overall memory consumption will not exceed the limits of the environment except in extreme cases, because modern computer systems normally have more than 1GB of RAM available. We decided to use Delta Encoding to reduce the required computations to store succeeding steps. The term Delta Encoding simply describes that instead of storing the full *state* of some entity, for example an object, only the difference to the previous state is stored. This difference is called the Delta and in the case of the CRI, the *state* is the dependency graph in a specific step of the history. The Delta represents the change to the dependency graph from one step to the next. This approach is very useful to reduce the size of data that needs to be stored for any stream of data where the Delta between elements in the stream is marginally smaller than an element itself. It is used with great success in video encoding, because the difference between succeeding frames is usually very small, but depending on the resolution of the video, each frame can contain a large amount of information. As the first step towards implementing Delta Encoding, we modularized the history and introduced *Paging* to it. Although the term *Paging* may be misleading as it usually refers to memory management of the operating system [Aut18h], we use it to describe a software engineering pattern as it best describes our approach. This pattern is commonly used to handle large streams of data. What we refer to as *Paging* is the implementation of a (software) cache to the history that loads a subset of succeeding steps. The

history will only store a fixed size of steps at once. As soon as the size of this cache is exceeded, a part of the loaded steps is written to the local disk using the Chrome storage API and removed from the cache. The cache is in fact better referred to as a Page, because its content always consists of sequential steps while a usual (software) cache used in programming may contain elements in a random order. If a step is requested that is not currently loaded in the Page, the requested step is loaded along its surrounding steps from the Chrome storage API. The requested step is loaded as the middle of the Page, if possible, to prevent the need to immediately load another step from the local storage, if the user clicks next or previous. This approach keeps the memory consumption of CRI fairly constant and independent of the number of steps in the history, as the maximum number of loaded steps is the size of the Page. We built the Delta Encoding on top of the Paging, to keep the benefits of both approaches. As mentioned earlier Delta Encoding reduces the data stored to the changes between succeeding steps. These changes are also the relevant information a user of the CRI examines when working with it. Therefore, storing these changes and their types actually provides more useful data and makes the changes easier to query. With the previous approach, the changes between steps needed to be calculated or logged in addition to the history as it was for the history query feature. For the same reason we implemented Delta Encoding ourselves instead of using a JSON Patch [Dha18] based library like the one developed by *Joachim Wester et al.* [al18]. The library calculates Deltas itself which consumes additional resources. Delta Encoding also enabled an easy and more robust way to detect the current changed node, and later edge, instead of storing the highlighting in the graph itself. The current change in a step is simply the node or edge affected by the last Delta. With the simplest form of Delta Encoding, a specific step of the history would be created by loading the very first step and then applying all changes stored as Deltas in sequential order on the dependency graph stored in that first step. To reduce the computational cost needed to reconstruct a dependency graph for a certain step, we implement an algorithm similar to the one explained at [Aut17] without B-frames. We call the I-frames *Base Stages*, which store a full graph, and P-frames *Delta Stages*, which only encode the Delta to the previous step. The size of the Delta Window can be configured in code and is currently set to 100, meaning each *Base Stage* will be followed by 100 *Delta Stages* before a new *Base Stage* is created. The implementation is located in *history.js*. Paging is used to load a subset of continuous *BaseStages* all at once and store the rest with the Chrome storage API. This is important in order to cope with the user stepping forward and backwards repeatedly over the threshold between two *BaseStages*, which would be very resource demanding if only one *Base Stage* would be loaded at once. Increasing the Page size also reduces the number of storage and load commands submitted to the local Chrome storage using the API. The Deltas are used on the graph level and, therefore, always contain the full node or edge that was changed or added, in contrast to value-based Deltas. As not all changes to the graph are easily reversible their Deltas always belong to the proceeding *Base Stage* and not to the succeeding *Base Stage*. This is also the reason we did not implement

B-frames as used in the original algorithm. To keep the *Delta Stages* that succeed a *Base Stage* linked to it, we store them as part of the *Base Stage* themselves. If a new *Delta Stage* is added or requested that is in the forward direction in regard to the sequential order of the steps, without crossing over any *Base Stage* boundaries, all changes are directly applied to the currently loaded dependency graph. If, however, the direction is backwards, the new graph is constructed by emptying the graph, loading the *Base Stage* and then applying all Deltas up to the requested stage. The same logic is used if a random step is requested that is not part of the current Base Stage. This means that loading steps in the forward direction, is always faster than accessing steps at random or backwards. As noted above the forward direction is crucial for the recording process and has a great impact on the performance of the CRI when handling applications with rapidly updated observables. The used algorithm and implementation keep this use case fast and simple. On the other hand, stepping back and the random access of steps requires additional computation in comparison to the previous implementation of the history. However, they are only ever triggered by the user themselves which greatly lowers their performance requirements. The *GraphManager* class handles drawing of the displayed dependency graph in the UI and request the loading of a different step from the history when necessary. It also implements the highlighting of the current node or edge. The *History* class handles the actual data, caching and requesting loading of steps from the storage when necessary. It also manages when to switch to the next *Base Stage* instead of adding another *Delta Stage*. The shared, singleton object *stageStorage* provides an Object-Oriented interface, that loads and stores *Base Stages* with their *Delta Stages*, to the key-value based Chrome storage API. *stageStorage* also schedules request internally to ensure that they are handled sequentially and in the correct order. Guaranteeing that stages finish saving before the loading operation is executed. For this reason, the loading operation is asynchronous and uses a callback that is invoked when the loading is done. To achieve the synchronicity of these operations, *stageStorage* uses a queue internally. If an operation finishes storing or loading it will invoke the next operation in the queue.

3.3.3 Additional performance improvements

To further increase the performance of recording new steps to enable using the CRI with faster updated observables, we inspected the resource consumption with memory analysis as well as CPU profiling tools. A comparison of these inspections for the CRI3 and the previous version of the CRI is shown in chapter 4. We discovered, that for applications with rapidly updated observables, a huge amount of CPU resources were spent on computations responsible for rendering positions and sizes of the Scalable Vector Graphics (SVG) HTML element used to display the dependency graph. A rendering operation was triggered for each new step in the history. Throttling these rendering operations and UI updates greatly increased the performance of the

CRI. Since humans are not able to detect and comprehend changes to the UI beyond a specific update rate, decreasing the number of rendering operations will not even be noticed by a user. In fact, we increased the interval between rendering operations even further. The goal was to use an interval compromising between delay that a user notices and computational resources used to render the graph in that interval. According to an article by Thomas Burger [Bur18] the average human reaction time to visual input is 250ms. Since the CRI does not rely on real-time interaction, they claim that the upper delay limit for applications falling into that category is 500ms before the performance of the user is affected. Although the recording process of the CRI needs to fulfill real-time requirements to a certain degree, most UI interactions of the user happen after the recording as they examine the history. If a requested result is displayed with a delay of 100ms, the user will experience it as an immediate result. Delays above that will be noticed, but will not annoy most users if they are shorter than one second. In order to increase the performance of the CRI while not introducing any nuances for the user with applications which the extension can easily handle without the throttling, we chose 230ms as a compromise. This is significantly lower than 500ms, but since some Web applications will change node values in intervals shorter than that, we decided to use 230ms as it provided the most fluent experience. In the future, it is desirable to add a self-regulating mechanism that increases or decreases the delay on demand of the inspected Web application. The lower boundary for the delay should still be 230ms as not to waste computational resources. However, for applications that require extensive resources like the test application *RxJS mario*, there is no fixed upper boundary for the delay. Steps are created so fast that the user is not able to gather any useful information. The displayed dependency graph and especially the history will only be useful to examine when the recording is stopped. Therefore, up to that point in time, the UI is not important and the delay at which rendering is triggered should be increased accordingly. It is to be noted, however, that as explained by Thomas Burger in the article [Bur18], the user's performance executing a task will suffer from the delay until the recording is paused. Beyond decreasing UI rendering operation, there is also the option to exclude some nodes from the recording introduced in the CRI1. This is effective if the application only has a few rapidly updated observables. For most applications with these type of observables, however, usually many observables have dependencies to one or more of the rapidly updated observables without filtering the submitted values. This leads to chains of rapidly updated observables that all need to be ignored to drastically reduce the number of generated steps in the history. At this time, the CRI does not provide any tools that supports the user with excluding observables. Some of the options in this regard are discussed in chapter 5 and may be the target of further development.

3.4 Excessively created Observables

This section discusses the origins of excessively created observables as well as some of the possible solutions to handle them, however, the implementation of these solutions is outside the scope of this thesis. In addition to observables that are rapidly updated, another type of observable is difficult to handle for the CRI and similar tools. If new observables are created excessively, for example as part of a loop, the dependency graph is flooded with new nodes belonging to those observables. An example where this issue would occur is a Web application that uses networking to connect to a list of clients. The network clients each create an observable and each message submitted to one of them is gathered in a single observable that depends on them. If many clients connect or reconnect to the Web application, the dependency graph becomes very large and hard to examine. We refer to excessively created observables as a special type of an observable for simplicity, although technically the term describes a group of observables that are created from the same source code. Since these observables stem from the same lines of source code, the nodes will all share their detailed information such as the source position information or name, except for their value. In addition to many long-lasting observables, the same issue occurs if observables are created temporarily, for example, as part of a function that performs some type of calculation. Since the CRI does not detect if observables are no longer used, the dependency graph shows nodes even after they were deleted by the garbage collector of the JS runtime. Another issue for these type of observables is that if their nodes have a *name*, the current implementation of the CRI's history query and search feature cannot distinguish them properly because the node identification for some of the queries are based solely on the *name* and does not consider the scope the name, i.e. the variable, belongs to. There is no simple way to exclude or handle this type of observables, as they have numerous sources and behaviors that need different solutions. All solutions have in common that they increase the complexity of the recording process.

3.4.1 Increased recording complexity

To exclude nodes from the recording of rapidly updated observables the user is able to specify Ids of the nodes to ignore. However, this approach does not provide a solution for excessively created observables, since instead of updating existing nodes, new nodes are created. For *named* nodes the exclusion is possible using the name, but it is not possible for any other nodes. For this reason, the CRI needs to provide rules that specify certain characteristics by which newly created nodes are filtered. In contrast to ignoring the value updates and still creating the nodes, this type of observables needs to be excluded from the recording process entirely. Yet, some visual representation of these nodes is required in the dependency graph. Detection of nodes that belong to excessively created observables needs to handle many subcategories of these

observables. One possible solution is to replace the group of nodes with a single pseudo node that merges all nodes of the group. This is not suitable for all categories of excessively created observables and is only applicable if all outgoing edges have the same target and ingoing edges have the same source. For cases in which these differ, the edges may be displayed with a sophisticated visual graph optimization such as edge bundling [Zho+13]. In this approach of merging a group of nodes, the values of the merged nodes are displayed on the pseudo node. These values cannot be omitted in general even though they differ for every observable in the group and one group may consist of hundreds of nodes, depending on the inspected Web application. To display these values a scalable UI element is needed that allows the user to inspect any of the values independent of the group size. The detection of nodes to merge into a group is fairly robust when using the source position, although there may be some other cases in which nodes should be merged. As we mentioned earlier there are several possible options to handle these type of observables, but all have in common that the recording process increases in complexity. If the user should be able to reverse the merging of nodes on demand or a rule-based exclusion is chosen as the approach to handle this issue, the recording becomes customizable. To enable the user to keep the customization over more than one debugging session, it needs to be persisted. Since the customization is also highly specific to a single inspected Web application, some form of session or project management is required. In contrast to the current implementation of excluding nodes via the options page, the loading and storing of this persistent customization should be integrated into the menu of the main UI, because it directly corresponds to the currently displayed dependency graph.

3.5 The Chrome Reactive Inspector - A growing project

At the beginning only one developer, Waqas Abbas was working on the project who was later joined by Pradeep Baradur. They worked independently but parallelly for some time and both extended the CRI as discussed in section 2.2.1. Their work on the CRI was strictly focused on implementing the basic extension itself and then extending it with many features. As the project grew and their work later was merged as the first task of this thesis, however, the focus moved to the CRI being a full-fledged project that introduces new maintainability and extensibility requirements. In this section, we describe our approach to increasing development velocity and to the new requirements by reorganizing the file structure and increasing modularity.

3.5.1 Increasing velocity

In the CRI1 the JS files of the extension were mostly organized into directories with files which implement similar or related functionality. For example, all files containing the code for the recording of nodes from the reactive libraries were located in the *event-sniffer* directory. As

we started working on the project, we discovered that this organization is not suitable at the top level, because it most likely confuses any developer working on the project. The reason for this is the different contexts in which JS files are executed in a Chrome extension. The *window* object representing the global scope of a JS file can be different to the one used in other files. The *window* object in a content script is shared among all content scripts and provides access to the Document Object Model (DOM) of the inspected Web application. However, the *window* object in JS files that run in the context of the DevTools panel of the CRI is specific to that context and provides access to the DOM of the CRI's panel. Background scripts also have their own shared context. In addition, scripts running in these different contexts have access to other subsets of the Chrome API. If the JS files of the extension are organized as in the previous version of the CRI, JS files located in the same directory may run in different contexts. To remove this confusion, we separated the JS files at the top level by their context. Background scripts are located in the *background* directory while content scripts are located in the *content-scripts* directory. The scripts that run in the context of the DevTools panel are located in the *devtools* directory. All external libraries are located in the *libraries* directory because some libraries are used in multiple contexts. At any level below the top level separation by context, the scripts are still organized into directories with files sharing similar or related functionality. This reduces the risk for a developer to lose track of the *window* object and access to the Chrome API available in a script. A large part of the documentation for the previous versions of the CRI was included in the theses of Waqas Abbas and Pradeep Baradur, however, the comments directly providing documentation within the source code were sparse. A developer starting to work on the project had to check the external documentation to comprehend the source code, increasing the time required to do so. Therefore, to further increase development velocity we also added additional source code documentation to reduce the time needed to comprehend the existing code. We measured the number of single- and multi-line comments for both the Chrome Reactive Inspector version 2 (CRI2) created by Pradeep Baradur as part of his thesis and the current version of the Chrome Reactive Inspector, version 3 (CRI3), to verify that the number increased. The source code of the CRI2 contains 231 single- and 32 multi-line comments while the source of the CRI3 contains 271 single- and 52 multi-line comments. These numbers include TODO comments, found mostly in the CRI3, and commented code, found mostly in the CRI2. Only JS files developed in the CRI were included in the measurements. As the project grew and more and more features were added, some JS files became larger than 500 lines of code, for example, *panel.js*. In mind of software engineering aspects like separation of concerns and modularity, long files with multiple independent tasks are not desirable. In addition, the nature of JS as a script language without class definitions (before ECMA6) tempts developers to neglect encapsulation and use many global variables and functions. To increase modularity and encapsulation we used the JS closure pattern, in order to help track down and correct errors or extend existing code by enclosing the influence of a change or limiting the code that needs to be

inspected for a specific error. A closure prevents variables to be accessed from the outside and makes them effectively private. This works because variables are bound to a function context in JS. In the example seen in listing 3.3, a self-executing function that returns a function is used to implement the closure pattern. In the example, the variable *counter* is not accessible from outside the self-executing function. It is also possible to use closures to create pseudo-classes in JS that can be instantiated with the *new* keyword. In that case, the self-executing function returns a constructor function that in turn return an object. In addition to both these construct, we used closures to implement singleton objects. An example for this is the *stageStorage.js* file. Using closures prevents the pollution of the global window object (for a specific context) while also documenting that these functions belong together. In addition, they limit the interface to other components of the application to a set of explicitly public variables and functions.

```
1 var add = (function () {
2   var counter = 0;
3   return function () {return counter += 1;}
4 })();
```

Listing 3.3: Example of RxJS code.

In total, we added nine new classes. The only JS scripts that are still not using a closure to encapsulate their internal logic are the files *background-communication*, *options*, and *panel*. The first two are the only scripts inside their respective context. The *panel* script ties all features together and previously used and provided many globally accessed variables. This increased the development effort to separate individual components and dependencies beyond the scope of this thesis as this was not the main focus. Nonetheless, several efforts have been made to move some components to their own JS files and to reduce the number of global variables. In the future, the code responsible for the search and history query feature is to be extracted too. Ultimately the *panel* should only be responsible to instantiate other classes and delegate events and tasks to them. To further increase modularity and provide developers with a robust and tested pattern for separation of concerns in application with a UI, a common pattern like Model View Controller (MVC) or Model View ViewModel (MVVM) should be used for the CRI DevTools panel. Code metrics are usually used to verify that modularity and separation of concerns are adhered, while still avoiding high coupling of these modules. However, most code metrics that are usually used in Object-Oriented (OO) programming, such as coupling and cohesion [Aut18c], cannot be calculated for a dynamically typed language like JavaScript. Instead of these metrics that provide robust results for OO programming, we used more ad-hoc and less robust metrics. In detail, we used the total number of lines, the number of files and the average number of lines per file to provide insight on the extent to which we increased modularity. See the Appendix A for the exact results.

3.5.2 Build process

The CRI provides a packed Chrome extension file that is deployed as the stable version. This packed extension contains by default all files in the chosen subdirectory, including files that are only relevant for developers of the CRI like the *.gitignore* file in addition to IDE project files that may even expose sensual information about developers or their development systems. The suggested approach is to implement a custom build process that creates a new directory which only contains files needed to run the extension [mik14]. We implemented the build process in JS using *NodeJS* [Aut18g] in the *build_process* project. The reason to use a JS build script in place of any other sophisticated build language is to provide the developers with a familiar language and familiar tools. In the future, this build process may also be used to reduce the size of the CRI's JS files by minifying them.

4 Evaluation

In this chapter, we evaluate the contributions of this thesis and the general state of the project. On that account, we first examine the changes to the UI, the Source Code Tooltips, and performance improvements. We then verify the most important features of the CRI by checking a set of specifications for each test application and discussing the result.

All performance measurements and tests in this chapter were performed with Chrome Version 64.0.3282.140 (Official Build) 64-bit, WebStorm 2017 3.4 and Windows 10 Home 64-bit version 1709 build 16299.192. The used device has 8GB RAM, Intel i7 7500 2.7GHz 4CPUs Processor with integrated Intel(R) HD Graphics 620. Each test application was hosted on the integrated Web server of WebStorm on the local machine. No other applications were running during the execution of the tests, but background processes, services, and other unpredictable factors may still have influenced the results of the performance measurements.

4.1 Evaluation of Improvements

This section examines the improvements introduced by this thesis and their implementations. We inspect the changes made to the UI as well as the Source Code Tooltips and their applicability regarding the test applications. At last, we examine the performance improvements introduced by this thesis in comparison with the previous version of the CRI.

4.1.1 Changes to the User Interface

Most of the changes to the User Interface are, by their nature, hard to evaluate in regards to their usability improvements. The experienced usability of a UI can be highly subjective. Consistent evaluation results are often achieved by collecting and analyzing empirical data, for example through Clickstream analysis [Tan09]. Since the CRI is not yet used in active production systems by a large number of users, a User Study would be the usual approach but would exceed the scope of this thesis. However, some changes are still verifiable as improvements. For example, the text per node in the dependency graph was reduced without losing any information. This is clearly a general improvement of usability because the change did not affect any other part of the UI and removed obsolete text. In addition to highlighting a node that was updated in the currently viewed step of the dependency graph history, edge highlighting now provides additional information for the user. This helps track down differences between steps which were not as easily detectable before while not affecting any other aspects of the UI. Furthermore, there is now an option - the *hide* button - to hide parts of the UI which are not necessary to

examine the dependency graph. Although the *hide* button as a new UI element might introduce minor usability issues, it can still be considered an objective improvement - as can the edge highlighting.

4.1.2 Inspecting Source Code Tooltips

The Source Code Tooltips that connect nodes in the dependency graph with the JS code they originate from are created whenever the user hovers over a node and presses the CTRL key. Due to their on-demand nature, their performance impact on the time-critical computations of the CRI like the recording process can be neglected. The original JS files are stored in a variable in a content script of the CRI that is queried for the lines corresponding to a node if requested. Storing a copy of the whole JS file in addition to the instrumented version, however, should not exceed the memory (i.e. Random Access Memory) of any modern computer that runs Google Chrome. We will examine the robustness of this feature in regards to its accuracy in a later section as part of the test application evaluation. To investigate and verify the usefulness of Source Code Tooltips we examined each test application and calculated the percentage of nodes that gain additional context through this feature. As a *named* node already has some form of context that helps the user to identify it and its dependents, we counted *named* nodes separately. For the exact circumstances and inputs used for each test application see section 4.2. Over all test applications, there were 586 nodes in total. Of these, 440 nodes had a Source Code Tooltip which equals to approximately 75% of all nodes. Only 199, approximately 33.9% of all nodes are *named*. This means that the introduction of Source Code Tooltips provided additional means to identify a node to the user for 35.1% of the total number of nodes. The remaining 146 nodes that cannot yet be linked to specific positions in the source code in part consist of nodes that can easily be interpreted by examining the nodes that depend on them. In the RxJS test application *canvas-painting* for example, the node with Id 6 is a *FromEventObservable* created by "Rx.Observable.fromEvent(colorchar,"click")" and is not yet detected by the Jalangi analysis.

4.1.3 Scrutinizing Performance with rapidly updated Observables

In this section, we evaluate the performance improvements resulting from the reworked graph history and the less frequent UI rendering in detail. For this purpose, we developed a new test application for RxJS which is used as a simple benchmark on how well the CRI - and in the last part of this section RxFiddle - copes with rapidly updated observables. The application uses an *interval observable* to generate one update every five milliseconds over a period of five seconds. Listing 4.1 shows the observable responsible for the updates. The full source code including the termination after five seconds is presented in Appendix A.

```
1  var intervalObservable = Rx.Observable.interval(5)
2    .timestamp()
3    .bufferCount(2, 1)
4    .map(function (w) {
5        return w[1].timestamp - w[0].timestamp;
6      })
7    .share();
```

Listing 4.1: Example of RxJS code.

To exclude the initial setup of the CRI from the gathered performance data we create a *reactive breakpoint* for the *nodeCreated* event of node one ("nodeCreated[1]"), ran the application until it paused at the *reactive breakpoint*, started the performance recording and then continued execution. We used the Chrome-JavaScript Profiler to record a CPU profile that includes the time spent per function. The Memory tab of Chrome DevTools was used to inspect the memory usage. The duration of the recordings varies from the actual execution time of the test application in the respective case because they were started and ended manually.

CPU Profile

The label "(program)" in the recording indicates native code execution of Chrome. The mea-

Self Time		Total Time		Function
187860.5 ms		187860.5 ms		(idle)
65573.0 ms	41.53 %	65573.0 ms	41.53 %	▶ getBBox
20463.7 ms	12.96 %	27166.9 ms	17.21 %	▶ buildFragment
11525.4 ms	7.30 %	11525.4 ms	7.30 %	▶ removeChild
10575.6 ms	6.70 %	10575.6 ms	6.70 %	(program)
5693.5 ms	3.61 %	5693.5 ms	3.61 %	▶ getBoundingClientRect
4696.5 ms	2.97 %	5007.3 ms	3.17 %	▶ merge
3792.4 ms	2.40 %	22513.8 ms	14.26 %	▶ remove
3167.0 ms	2.01 %	3167.0 ms	2.01 %	▶ getElementsByTagName
2949.2 ms	1.87 %	2949.2 ms	1.87 %	▶ RegExp: <(?!area|br|col|er
2544.7 ms	1.61 %	2544.7 ms	1.61 %	(garbage collector)
2139.2 ms	1.35 %	2155.9 ms	1.37 %	▶ slice

(a) Chrome Reactive Inspector version 2.0

Self Time		Total Time		Function
23188.9 ms		23188.9 ms		(idle)
1697.3 ms	38.75 %	1697.3 ms	38.75 %	(program)
408.3 ms	9.32 %	615.0 ms	14.04 %	▶ ja
330.9 ms	7.56 %	330.9 ms	7.56 %	▶ getBBox
198.1 ms	4.52 %	198.1 ms	4.52 %	▶ removeChild
157.7 ms	3.60 %	157.7 ms	3.60 %	▶ GetNextId
148.7 ms	3.39 %	148.7 ms	3.39 %	(garbage collector)
122.2 ms	2.79 %	122.2 ms	2.79 %	▶ getBoundingClientRect
65.9 ms	1.50 %	65.9 ms	1.50 %	▶ replace
59.2 ms	1.35 %	59.2 ms	1.35 %	▶ getElementsByTagName
56.6 ms	1.29 %	59.7 ms	1.36 %	▶ merge
55.6 ms	1.27 %	1709.2 ms	39.02 %	▶ Apply

(b) Chrome Reactive Inspector version 3.0

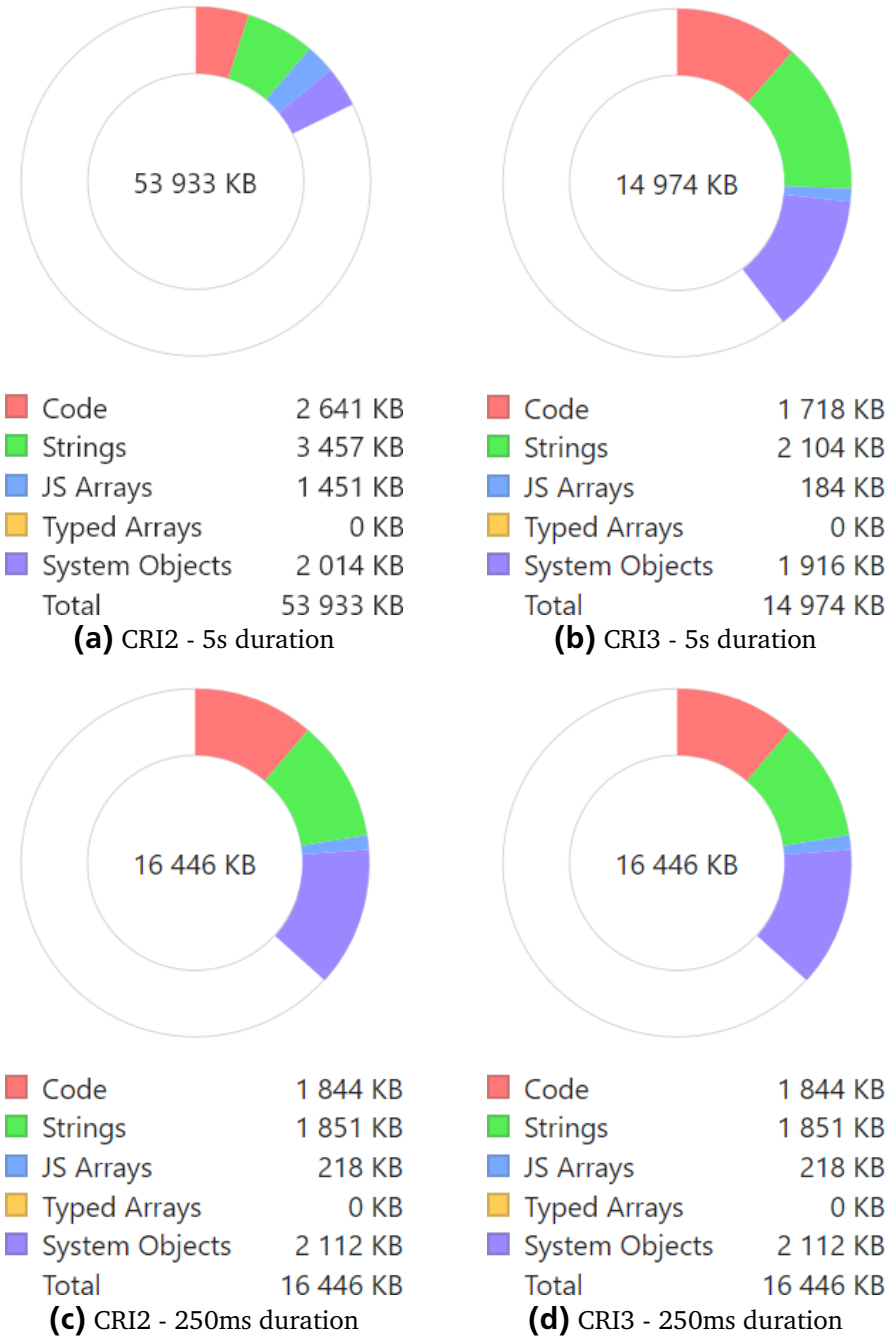
Figure 4.1.: CPU Profiles recorded by Google Chrome's JavaScript Profiler.

sured execution time for this label is far less accurate than for any other label because it is not apparent how this execution time is calculated. As a result, the execution time can simply be increased by stopping the recording at a later point in time due to the limitations of the recording tool. For the CRI2 the execution of *PerformanceTest* took approximately 154.0 seconds with an overall recording interval of 187.9 seconds. To differentiate between execution time and recording time we selected an area with significantly higher CPU load that should correspond reasonably well to the actual execution time. The largest percentage of *self time*, i.e. the time

spent executing code directly in a function itself, that was spent in a single function directly corresponds to excessive UI updates. These updates happen at least once per created step in the history since the slider is updated and triggers the rendering of a new dependency graph for that step. *getBB* (getBoundingBox) calculates the size of nodes in the graph, whereas *buildFragment*'s impact stems from the *domManip* function of jQuery.
For the CRI3 the execution took approximately 5.5 seconds with an overall recording interval of 23.2 seconds. It is visible that still a sizable amount of time is spent inside UI related computations especially *getBB*. The most time of execution not related to Chromes native code is spent inside the *ja* function of jQuery that cannot easily be tracked down to corresponding CRI code. Overall, most of the time is spent executing Chromes native code, but as described earlier this can have numerous reasons. Due to the kind of changes made between the CRI2 and the CRI3, however, we are able to account at least part of that execution time to storage operations using the Chrome Storage API. As the CRI3 was approximately 28 times faster than the CRI2, the performance improvements regarding rapidly updated observables implemented in the cause of this thesis were effective. It is, however, worth noting that the CRI2 is barely able to handle this test without crashing, i.e. the UI becomes temporarily unresponsive. This probably increases the performance differences between the two versions of the CRI more than would be the case with a test that both CRI versions would handle well - i.e. without the UI becoming temporarily unresponsive. We chose to keep the test as-is to underline the impact of *request buildup* and the impact of load exceeding the extension's capacity. As of now the CRI3 also has a specific capacity of updates it is able to handle without crashing. This capacity is much higher than that of the CRI2, mostly due to the throttling of rendering operations. In the future, the CRI should be extended to automatically detect if rendering computations accumulate beyond its limit and increase or decrease the throttle interval for UI updates accordingly. Since all steps are recorded even if the rendering is not able to keep up, it is a viable approach to increase the throttle time in order to keep the extension from crashing. The user is then able to pause the recording anytime and examine the steps in details whereas a crash will render the CRI useless to the user.

Memory Profiling

To reveal the impact of a large history on the memory consumption of the CRI we executed *PerformanceTest* with a limit of 250ms for both versions of the CRI in addition to the execution we used for CPU profiling with a limit of five seconds. For the test execution with the full duration (results visible in figures 4.2a and 4.2b) the CRI2 used 52.7MB in total while the CRI3 used only 14.6MB. The most notable difference in a specific category is the size of the memory used by JS arrays which include the stored dependency graph. In the test with reduced execution time (250ms), the CRI2 used approximately 16.1MB of memory while the CRI3 used 12.6MB of memory. Both versions of the CRI have a similar memory consumption for the test with reduced execution time. The CRI2, however, shows the dependency graph with a noticeable delay



(a) CRI2 - 5s duration **(b)** CRI3 - 5s duration

(c) CRI2 - 250ms duration **(d)** CRI3 - 250ms duration

Figure 4.2.: Memory Profiles of the CRI during the execution of *PerformanceTest* recorded and displayed by Google Chrome's Memory DevTool.

which is still less than five seconds. Since the CRI3's history implements a stream like approach through paging, the memory usage once the size of a single *page* is exceeded is fairly constant and does not increase with the number of steps. As the CRI2's history does not support paging the memory usage increases approximately proportional to the number of steps in the history. However, since modern computers usually have more than 1GB of memory, even the higher memory consumption of the CRI2 does not have any influence on the overall performance detectable by the user. We made no distinction between the CRI3 with Delta Encoding and Paging

and the CRI3 with only Paging, because this examination shows that the memory usage is far too low to impose any limitation on the overall performance of the CRI. As there were no significant changes to the recording process of RxJS and BaconJS in the CRI, we did not execute separate performance measurements for a test application using BaconJS.

Comparison to RxFiddle

To qualify the discussion in section 2.2.2 on performance with rapidly updated observables, we also examined the performance of RxFiddle with the test application introduced above and compared it to the CRI3. Although RxFiddle does not have performance issues with *PerformanceTest* executed in Google Chrome, i.e. not exceeding a delay of one second when hovering over a node before the tooltip is shown, if executed with Firefox (57.0.4 (64-bit)) the browser becomes unresponsive for a few seconds. The overall memory consumption of RxFiddle during the test was 52.5MB and the execution took approximately 5.5 seconds - similar to the CRI3. As mentioned earlier, this magnitude of memory consumption is not a limiting factor for the overall performance. The CRI generates approximately 2700 steps during the test execution while RxFiddle generates approximately one thousand values for each operator in the chain of *intervalObservable* which is displayed in figure 4.3. Examining these results any further does not provide additional insight since the tools are based on completely different technologies. RxFiddle uses TypeScript and runs as a Web page while the CRI does not use TypeScript, any form of bundling, or minifying and runs as a Chrome extension.

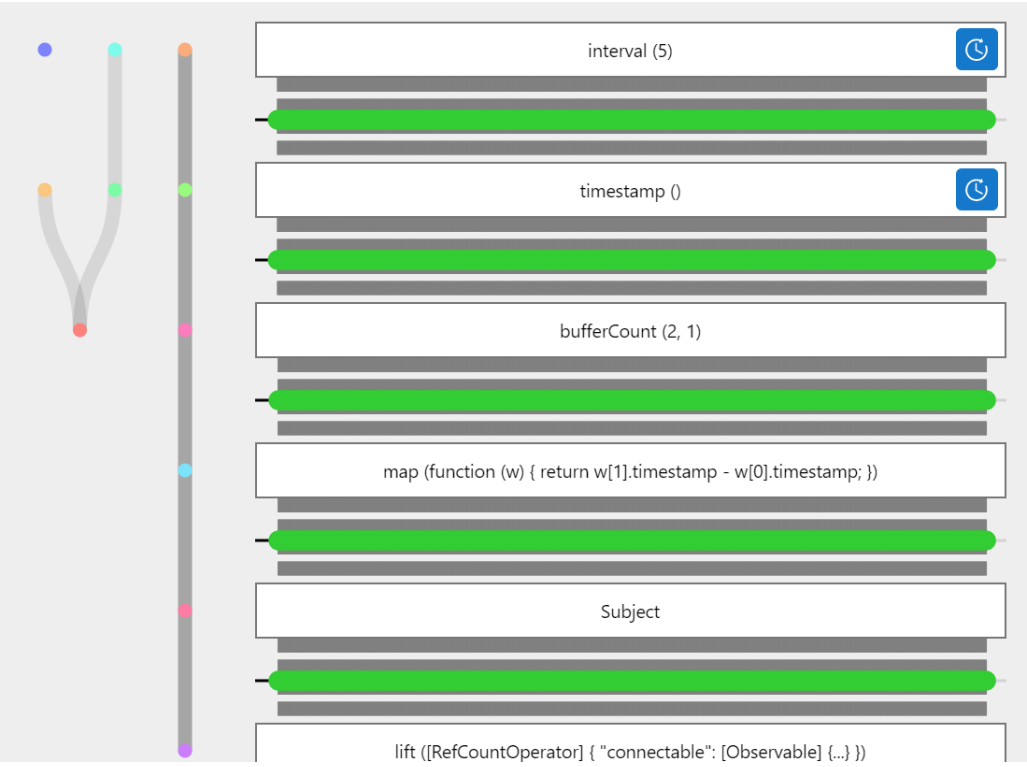


Figure 4.3.: Test application PerformanceTest inspected with RxFiddle

4.2 Reviewing the Test Applications

To establish the current state of the most important features of the CRI we compiled a set of specifications which we validated for each of the test applications. They were designed to provide a baseline for the robustness of the current CRI implementation. Note that we also included specifications targeting features not introduced by this thesis as a means of manual regression testing for the most important aspects of the CRI. Since most of these specifications cannot be tested for each node, step and/or, case in reasonable time, we specified the exact tests we used to approximate the results. These test probes should provide a reasonably precise evaluation which is easy to reproduce in order to verify our results.

Spec1 The dependency graph is shown and all observables that are assigned to a named variable are displayed distinctively. (Test: Up to the first five *named* observables in the code are displayed with an orange background.)

Spec2 The history of the dependency graph can be navigated. It is possible to navigate to the previous, succeeding or any random step in the history. (Test: Jump to the median step; click next five times; click previous five times.)

Spec3 For each node, no space is occupied by fields that hold no value in the node itself or its tooltip.

Spec4 The ids of the nodes start at one and are continuous. If the test application is started again with the exact same inputs, each node has the same Id as in the last execution.

Spec5 The Source Code Tooltips show and highlight the corresponding lines of code for each node with a Source Code Tooltip. (Test: If possible, choose two nodes: one that corresponds to the middle of a chain of reactive function calls and one that corresponds to the end of a chain of reactive function calls. For both nodes check if the highlighting is correct.)

Spec6 The node or edge that was updated in a step is highlighted. (Test: Check this behavior for the first ten steps in the history.)

Spec7 The history queries can be used to search for a specific event in the history. (Test: Queries for *evaluationYielded* and *nodeUpdated* find the corresponding steps for the first *named* node.)

Spec8 The graph can be searched, the matching node(s) is/are highlighted and if the search is reset the original coloring is restored. (Test: Search for the dependencies and dependents of the first named node and then reset the search.)

Spec9 *Reactive breakpoints* can be used to pause the debugger when a specific event occurs. (Test: *a)* A breakpoint set for the first node created will break at step one. *b)* A breakpoint for the

first node updated will break before the value is updated in the original observable. Note: The value will already be updated in the CRI UI.)

We excluded some of the test applications that were used in the previous works targeting the CRI from this evaluation, either because they no longer work due to outside influences or because they did not provide any additional value. The results of this evaluation are visible in tables 4.1 and 4.2. For the following statistics, we interpreted *ambiguity* between nodes with the same name, labeled with *Am*, as if the check failed because the CRI is not able to handle them properly yet. *N.A.* specifies that the check was not applicable, for example, because there were no *named* nodes in the dependency graph and the check is dependent on *named* nodes. We, therefore, treated *N.A.* as a success. Due to specification number 9 having two parts that can fail or succeed individually we treated their results separately in the following statistic. Overall 388 of 420 single checks (10 per test application) were successful. The specifications number 2 up to number 6 in addition to number 9 *b)* verified for all test applications. The 32 checks that failed have four distinct characteristics. All checks labeled with *ambiguity* are the result of the CRI not being able to handle multiple nodes with the same name. This happens for example if a named variable is created within a loop. The history query and search feature will not be able to correctly distinguish between the ambiguous nodes. Any operation that requires a name will only find the node last added to the dependency graph and only show the result for that node. Specification 9 *a)* failed for all RxJS applications where the node with Id 1 is recorded by the RxJS interception and not by the Jalangi analysis beforehand. It seems the RxJS interception records nodes out of order in certain circumstances. We verified that this behavior already existed in the previous versions of the CRI2 by executing the same check with the *RxJS son-father-wallet* test application with the CRI2. The failed checks for specification number 8 denote fails to identify every dependent or dependency. This seems to correspond to observables being dynamically created but the check will not fail for every dynamically created observable. In case of the check for the *RxJS crop* application, this behavior is not consistent with previous versions of the CRI and stems from changes introduced in the CRI3, however, the issue in the *RxJS stopwatch* application was already present in the CRI2. In the new test application *RxJS InlineScriptTest*, which was added to verify that the CRI is working with JS code directly embedded within HTML, the CRI does not detect every *named* observable. The reason is that the extension is currently not able to detect and intercept JS code inside HTML attributes. The failed check for specification number 1 in the *BaconJS blog_url* test application is the result of BaconJS interception not being able to detect the used observables correctly. The *named* observable is not detected and in addition, each change to the used observable in the JS code creates a new node in the dependency graph. This needs to be counted as a complete breakdown of the recording for this test application and is consistent with previous versions of the CRI. The remaining failed check for specification number 1 in the *RxJS alphabetinvasion* application also denotes a full breakdown of the recording. It seems the current implementation is not able

Application	Spec 1	Spec 2	Spec 3	Spec 4	Spec 5	Spec 6	Spec 7	Spec 8	Spec 9
alphabetinvasion	✗	✓	✓	✓	✓	✓	N.A.	N.A.	$✗^{3}$✓
mario	✓	✓	✓	✓	✓	✓	✓	✓	$✗^{3}$✓
PerformanceTest	✓	✓	✓	✓	✓	✓	✓	✓	$✗^{3}$✓
animated-image	✓	✓	✓	✓	✓	✓	✓	✓	$✗^{3}$✓
animationtest	✓	✓	✓	✓	✓	✓	✓	✓	$✗^{3}$✓
backpressure	✓	✓	✓	✓	✓	✓	✓	✓	$✗^{3}$✓
crop	✓	✓	✓	✓	✓	✓	✓	✗	$✗^{3}$✓
creaditcard-drag	✓	✓	✓	✓	✓	✓	✓	✗	✓
draw-with-combine-latest	✓	✓	✓	✓	✓	✓	✓	✓	$✗^{3}$✓[1]
earthquake	✓	✓	✓	✓	✓	✓	N.A., ✓	✓	$✗^{3}$✓[1]
simple-databinding	✓	✓	✓	✓	✓[2]	✓	✓	✓	✓
stopwatch	✓	✓	✓	✓	✓	✓	✓	✗	✓
subjects-examples	✓	✓	✓	✓	✓[2]	✓	✓	✓	✓
wiki-updates	✓	✓	✓	✓	✓	✓	✓	✓	✓
smart-counter	✓	✓	✓	✓	✓	✓	✓	✗	$✗^{3}$✓
state-store	✓	✓	✓	✓	✓	✓	✓	✓	$✗^{3}$✓
letter-counter	✓	✓	✓	✓	✓	✓	✓	✓	$✗^{3}$✓
son-father-wallet	✓	✓	✓	✓	✓	✓	✓	✓	$✗^{3}$✓
movie-search	✓	✓	✓	✓	✓	✓	✓	✓	✓
follow-the-mouse	✓	✓	✓	✓	✓	✓	✓	✓	✓
drag-and-drop	✓	✓	✓	✓	✓	✓	✓	✗	✓
canvas-painting	✓	✓	✓	✓	✓	✓	✓	✓	$✗^{3}$✓
twitter-follow-box	✓	✓	✓	✓	✓	✓	✓	✓	✓
rest-api-call	✓	✓	✓	✓	✓	✓	✓	✓	$✗^{3}$✓
image-sampler	✓	✓	✓	✓	✓	✓	✓	✓	$✗^{3}$✓
InlineScriptTest	✗	✓	✓	✓	✓	✓	✓	✗	$✗^{3}$✓

Table 4.1.: Result of checking the specifications for the RxJS test applications

to detect the observables correctly or, in most cases, at all. This behavior is also consistent with previous versions of the CRI and is the result of the programming patterns used in this application. The observables which were not recorded correctly are never assigned and no reactive operator or function is used on them except for *subscribe* or *unsubscribe*. Example code that shows the described pattern is visible in listing 4.2. The observable created from the call to "Rx.Observable.timer(750)" is never recorded.

```
1 Rx.Observable.timer(750).subscribe(function () {
2   self.playfield.removeChild(enemy);
3 });
```

Listing 4.2: Extract of RxJS AlphabetInvasion test application.

For the test application *RxJS mario* we added a *reactive breakpoint* on the last dependency that is created because the excessive use of timers and rapidly updated observables still causes the CRI to crash after a few seconds. This is probably the result of *request buildup* due to a large number of calls to the communication API of Chrome, but it is hard to provide a sufficient proof because Chrome itself will stop responding as well.

Application	Spec 1	Spec 2	Spec 3	Spec 4	Spec 5	Spec 6	Spec 7	Spec 8	Spec 9
timer	✓	✓	✓	✓	✓	✓	✓	✓	✓
blog_url	✗	✓	✓	✓	✓	✓	N.A.	N.A.	✓
comb-lock	✓	✓	✓	✓	✓	✓	Am	Am	✓
dragdrop	✓	✓	✓	✓	✓	✓	N.A.,✓	✓	✓
events	N.A.	✓	✓	✓	✓[2]	✓	N.A.	N.A.	✓
operators	✓	✓	✓	✓	✓	✓	✓	✗	✓
son-father-wallet split file test	✓	✓	✓	✓	✓	✓	Am	Am	✓
operators-and-events	✓	✓	✓	✓	✓	✓	✓	✓	✓
son-father-wallet	✓	✓	✓	✓	✓	✓	✓	✓	✓
up-down-counter	✓	✓	✓	✓	✓	✓	✓	✓	✓
form-validation	✓	✓	✓	✓	✓	✓	✓	✓	✓
movie-search	✓	✓	✓	✓	✓	✓	✓	✗	✓
bar-chart	✓	✓	✓	✓	✓	✓	✓	✓	✓
websocket-wikipedia	✓	✓	✓	✓	✓	✓	✓	✓	✓
multiselect-card	✓	✓	✓	✓	✓	✓	✓	✓	✓
true-false-logger	✓	✓	✓	✓	✓	✓	✓	✓	✓
drawing	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 4.2.: Result of checking the specifications for the BaconJS test applications

4.2.1 Summary

We examined the set of specifications for each of the test applications and found several issues. The problems with ambiguous variable names and out-of-order recording of nodes, issues that should be resolved at some point, do not break the overall usefulness of the CRI. This is because the dependency graph can still be examined and other features work as well. In contrast, the issues with recording in the test applications *RxJS alphabetinvasion* and *BaconJS blog_url* cause the CRI to display an incomplete dependency graph. This greatly reduces the usefulness of the extension with these applications. All issues found should be the target of further development to increase the robustness of the CRI, but issues in recording observables should clearly be prioritized.

1 The node with Id 2 was used because the node with Id 1 is never updated

2 The middle of a reactive function chain could not be examined because no function chain was long enough.

3 The breakpoint pauses program execution at the right step, but since nodes are added out of order the node with Id 2 is already added to the dependency graph

5 Conclusion and Future

In this chapter, we summarize the contributions and results of this thesis and provide some insight on some possibilities for new features that follow along the path set by this thesis.

5.1 Conclusion

In this thesis, we made several contributions to the goal of advancing the CRI. The UI has been updated to direct the users focus in addition to speeding up tasks performed by a user. We also reduced the cognitive load presented by the UI itself. An objective verification of this using multiple usability metrics like a User Study and Clickstream analysis is still missing, but most changes have clear benefits. We further strengthened the connection between the abstract representation and the source code with the introduction of Source Code Tooltips. The evaluation showed the potential information gain added by this feature. Along with its simplicity and accessibility, this makes Source Code Tooltips a valuable addition to the CRI. However, even though many improvements have been made to cope with resource demanding applications that contain, for example, rapidly updated observables, there is still some development effort required to remove the current limitations as seen in the RxJS test application *mario*. We also proposed several approaches to better support excessively created observables. One of these approaches is required to increase the CRI's robustness when handling any arbitrary application. However, as we have explained, the extension is still usable to some degree with most applications that excessively create observable except in very severe cases. As part of this thesis, we also improved extensibility and maintainability of the extensions source code. However, the evaluation of the test applications and features of the CRI, including features introduced by this thesis, has shown that some of these features still need to be improved with further development.

Nonetheless, the Chrome Reactive Inspector supports many features that help to cope with issues encountered by reactive debuggers in general. Including some that are currently not present in its main competitor RxFiddle.

5.2 Future

In this section, we discuss targets for future development that were not implemented as part of this thesis. We focus on possible additions to the CRI which would further increase the range of supported applications. As TypeScript and ECMA2015 compliant JavaScript (ES6) have become very popular as they overcome some limitations of ECMA5 compliant JavaScript (ES5), supporting these technologies would broaden the range of application the CRI supports

by far. However, the used Jalangi Demo library is not able to handle either. Especially due to this reason, it would be beneficial to replace Jalangi with another JS code analysis library that works in-browser by design. Although it would also be possible to add a dedicated second library explicitly to instrument and analyze TypeScript and ES6 to the CRI because it is unlikely that there is a tool that is able to handle all three technologies. Many Web applications that use ES6 in their development actually *transpile* it to ES5 because ES6 is not fully supported by all browsers yet. This means, that the CRI could be used on the *transpiled* JS code, but it renders the Source Code Tooltips useless since the code shown is not the one the user is working with.

Another advancement for the CRI would be supporting the loading of scripts from within other scripts. This has become a usual practice[1] for many developers, since its introduction, and is referred to as module loading. There are several popular libraries that include module loading functionality. These include NodeJS [Aut18g], requireJS [req18], as well as, the *import* keyword used in ES6 and TypeScript. Since the CRI scans an inspected page for HTML *script* tags to determine if the recording should run, any script using a module loader to load scripts that use RxJS or BaconJS cannot be recorded properly. Scripts loaded by module loaders are also excluded from the instrumentation process. To create the dependency graph the CRI records internal processes of RxJS or BaconJS. For this reason, the CRI needs access to the *Rx* or *Bacon* object used in the inspected applications scripts. To enable the use of the CRI with applications using module loaders, the tool would need to intercept calls to the respective module loader and load the files itself before submitting them to the application. This way, the loaded scripts could also be replaced with an instrumented version. The breach of the *isolated world* Chrome extension content scripts normally run in as explained in section 2.2.1.1, enables the developer to have total access to the pages JS. Therefore, to solve the problem of module loaders, the developer could overwrite the respective loading function or keyword and provide a custom implementation that submits the scripts to the CRI before actually loading them.

To further improve the CRI's ability to handle applications using rapidly updated observables, the current node exclusion feature should be extended. Currently, the feature only allows the user to exclude nodes by Id from the extension Option page. This is very cumbersome to do for many nodes as the user repeatedly needs to leave the context of the CRI DevTools panel. As discussed in section 3.3.3 usually more than one node needs to be excluded. In addition excluding a node completely may distort the dependency graph due to missing edges. A more sophisticated approach would be to only exclude value updates beyond the first for a node because this would keep the dependency graph complete. It would also ease the use of the exclusion feature if the user would be able to exclude nodes by a direct interaction with that node, for example, via a right click and a context menu. To improve usability if a chain of observables each are rapidly updated it would also be helpful to provide an option to exclude

[1] RequireJS is ranked place 14 in the top used JavaScript libraries and functions according to https://trends.builtwith.com/javascript/RequireJS - Week beginning Jan 01nd 2018 queried on 2018-01-03

all dependents or ancestors in the dependency chain from the recording as well. As the exclusion of a node is always specific to a single inspected application, providing persistent customization for a single application as discussed in section 3.4 would also improve the user experience.

Since the presented solutions for handling excessively created observables in section 3.4 were not implemented as part of this thesis, they are also a target for further development so as to increase the range of applications the CRI supports.

List of Figures

Listings

List of Tables

Bibliography

[Abb17] Waqas Abbas: “A Debugging Tool for Javascript Reactive Libraries”. Master-Thesis. TU-Darmstadt Germany, June 5, 2017.

[al18] Joachim Wester et al.: JSON Patch. Feb. 20, 2018. URL: https://github.com/Starcounter-Jack/JSON-Patch.

[Aut17] Various Authors: Video compression picture types. Dec. 3, 2017. URL: https://en.wikipedia.org/wiki/Video_compression_picture_types.

[Aut18a] Various Authors: Concurrency model and Event Loop. Feb. 15, 2018. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop.

[Aut18b] Various Authors: Content Scripts. Feb. 3, 2018. URL: https://developer.chrome.com/extensions/content_scripts.

[Aut18c] Various Authors: Coupling. Feb. 19, 2018. URL: https://en.wikipedia.org/wiki/Coupling_(computer_programming).

[Aut18d] Various Authors: import. Feb. 4, 2018. URL: https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Statements/import.

[Aut18e] Various Authors: Lazy Loading. Feb. 16, 2018. URL: https://en.wikipedia.org/wiki/Lazy_loading.

[Aut18f] Various Authors: Method chaining. Feb. 20, 2018. URL: https://en.wikipedia.org/wiki/Method_chaining.

[Aut18g] Various Authors: Node.js. Feb. 20, 2018. URL: https://nodejs.org/en/.

[Aut18h] Various Authors: Paging. Feb. 20, 2018. URL: https://en.wikipedia.org/wiki/Paging.

[Aut18i] Various Authors: Selenium. Feb. 15, 2018. URL: http://www.seleniumhq.org/.

[bac18a] baconjs.github.io: Bacon.js. Jan. 21, 2018. URL: https://baconjs.github.io/.

[bac18b] baconjs.github.io: Bacon.js Repository. Jan. 21, 2018. URL: https://github.com/baconjs/bacon.js.

[Ban18a] Herman J Banken: hermanbanken/RxFiddle. Feb. 2, 2018. URL: https://github.com/hermanbanken/RxFiddle.

[Ban18b] Herman J Banken: RxFiddle. Feb. 2, 2018. URL: https://rxfiddle.net/.

[Ban18c] Herman J Banken: RxFiddle Tutorials Page. Feb. 2, 2018. URL: https://rxfiddle.net/tutorials.html.

[Bar17] Pradeep Baradur: “A Debugger for Reactive Javascript Libraries”. Master-Thesis. TU-Darmstadt Germany, Oct. 3, 2017.

[Bos18] Mike Bostock: D3 Data-Driven Documents. Feb. 14, 2018. URL: https://d3js.org/.

[BR18] Software Engineer at Google Ben Lesh; also working on RxWorkshop.com: Hot vs Cold Observables. His own views, not necessarily the companies. Google. Feb. 3, 2018. URL: https://medium.com/@benlesh/hot-vs-cold-observables-f8094ed53339.

[Bur18] Thomas Burger: How Fast is Realtime? Human Perception and Technology. Feb. 20, 2018. URL: https://www.pubnub.com/blog/2015-02-09-how-fast-is-realtime-human-perception-and-technology/.

[cuj18] github.com cujojs: Monadic streams for reactive programming. Feb. 3, 2018. URL: https://github.com/cujojs/most.

[cyc18] cycle.js.org: Cycle.js. Feb. 3, 2018. URL: https://cycle.js.org/.

[Dha18] Dharmafly: JSON Patch. Feb. 20, 2018. URL: http://jsonpatch.com/.

[Fou18] The jQuery Foundation: jQuery JavaScript library. Feb. 4, 2018. URL: https://jquery.com/.

[Fra18] Jason Frame: Facebook-style tooltips plugin for jQuery. Feb. 16, 2018. URL: https://github.com/jaz303/tipsy.

[Gon16a] Liang Gong: Interactive Demo of Jalangi. Electric Engineering and Computer Science. University of California, Berkeley. Nov. 22, 2016. URL: http://people.eecs.berkeley.edu/~gongliang13/jalangi_ff/demo_integrated.htm.

[Gon16b] Liang Gong: Jalangi - A Dynamic Analysis Framework for JavaScript. Electric Engineering and Computer Science. University of California, Berkeley. Nov. 22, 2016. URL: http://people.eecs.berkeley.edu/~gongliang13/jalangi_ff/index.html.

[Her18] Christoph Hermann: Awesome FRP JS. Feb. 3, 2018. URL: https://github.com/stoeffel/awesome-frp-js.

[kef18] kefirjs.github.io: Kefir.js. Feb. 3, 2018. URL: https://kefirjs.github.io/kefir/.

[LLC18] Katalon LLC.: Katalon Studio. Feb. 15, 2018. URL: https://www.katalon.com/.

[mik14] miket@chromium.org: Chromium Issue 314360. Feb. 10, 2014. URL: https://bugs.chromium.org/p/chromium/issues/detail?id=314360#c10.

[msd18] msdn.microsoft.com: Testing and Debugging Observable Sequences. Feb. 2, 2018. URL: https://msdn.microsoft.com/en-us/library/hh242967(v=vs.103).aspx.

[NA18] N.A.: ReactiveX rxjs. Jan. 31, 2018. URL: https://github.com/reactivex/rxjs.

[rea18a] reactivex.io: ReactiveX. Jan. 21, 2018. URL: http://reactivex.io/.

[rea18b] reactivex.io: RxJS. Feb. 3, 2018. URL: http://reactivex.io/rxjs/.

[req18] requirejs.org: require.js A JavaScript module loader. Feb. 22, 2018. URL: http://requirejs.org/.

[Sal+18] Guido Salvaneschi et al.: Reactive Inspector. Jan. 21, 2018. URL: https://guidosalva.github.io/reactive-inspector/.

[shi18] shiny.rstudio.com: Debugging Shiny applications. Feb. 2, 2018. URL: https://shiny.rstudio.com/articles/debugging.html.

[SM16] Guido Salvaneschi; Mira Mezini: "Debugging for Reactive Programming". In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. Austin, Texas: ACM, 2016, pp. 796–807. ISBN: 978-1-4503-3900-1. DOI: 10.1145/2884781.2884815. URL: http://doi.acm.org/10.1145/2884781.2884815.

[Sta17] Petr Staníček: Paletton. Dec. 5, 2017. URL: http://paletton.com/.

[Tan09] David Taniguchi: Click stream analysis. US Patent 7,587,486. Sept. 8, 2009.

[tio18] tiobe.com: TIOBE Index for february 2018. Feb. 21, 2018. URL: https://www.tiobe.com/tiobe-index/.

[WH00] Zhanyong Wan; Paul Hudak: "Functional Reactive Programming from First Principles". In: *SIGPLAN Not.* 35.5 (May 2000), pp. 242–252. ISSN: 0362-1340. DOI: 10.1145/358438.349331. URL: http://doi.acm.org/10.1145/358438.349331.

[Zho+13] Hong Zhou et al.: "Edge bundling in information visualization". In: *Tsinghua Science and Technology* 18.2 (2013), pp. 145–156.

Appendices

A APPENDIX

A.1 Ad-hoc code metrics for CRI2 and CRI3

The lines of code include comments, lines containing only brackets and empty lines. CRI2 has 8 own JS files with 2353 total and 294 average lines of code per file. The scripts *background*, *devtools* and *content-script-end* are empty and were ignored for these measurements. CRI3 has 19 own JS files with 3580 total and 188 average lines of code per file.

A.2 Source code of performance test.

```
1  var intervalObservable = Rx.Observable.interval(5)
2  .timestamp()
3  .bufferCount(2, 1)
4  .map(function (w) {
5  return w[1].timestamp - w[0].timestamp;
6  })
7  .share();
8
9  var pauser = new Rx.Subject();
10 var pausable = pauser.switchMap(function (paused) {
11 return paused ? Rx.Observable.never() : intervalObservable;
12 });
13 // generates approximately 1000 steps
14 setTimeout(function () {
15 pauser.next(true);
16 }, 5000);
17
18 pausable.subscribe(function (value) {
19 console.log("Value: " + value);
20 });
21 pauser.next(false);
```

Listing A.1: RxJS code of PerformanceTest