Debugging Data Flows in Reactive Programs

Herman Banken
Delft University of Technology
Delft, The Netherlands
hermanb@ch.tudelft.nl

Erik Meijer
Delft University of Technology
Delft, The Netherlands
h.j.m.meijer@tudelft.nl

Georgios Gousios
Delft University of Technology
Delft, The Netherlands
g.gousios@tudelft.nl

ABSTRACT

Reactive Programming is a style of programming that provides developers with a set of abstractions that facilitate event handling and stream processing. Traditional debug tools lack support for Reactive Programming, leading developers to fallback to the most rudimentary debug tool available: logging to the console.

In this paper, we present the design and implementation of RxFiddle, a visualization and debugging tool targeted to Rx, the most popular form of Reactive Programming. RxFiddle visualizes the dependencies and structure of the data flow, as well as the data inside the flow. We evaluate RxFiddle with an experiment involving 111 developers. The results show that RxFiddle can help developers finish debugging tasks faster than with traditional debugging tools.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; *Data flow languages*; Software maintenance tools;

KEYWORDS

reactive programming, debugging, visualization, program comprehension

ACM Reference format:
Herman Banken, Erik Meijer, and Georgios Gousios. 2018. Debugging Data Flows in Reactive Programs. In *Proceedings of ICSE '18: 40th International Conference on Software Engineering, Gothenburg, Sweden, May 27–June 3, 2018 (ICSE '18),* 13 pages.
https://doi.org/10.1145/3180155.3180156

1 INTRODUCTION

Software often needs to respond to external events and express computations as data flows. Traditionally, handling asynchronous events was done using the *Observer design pattern* [23] (in object-oriented environmets) or *callback functions* [22] (when the host language supports higher-order functions). Using these patterns, the system consuming the data does not have to block waiting for new data to arrive, but instead it yields control until new data is available. While these patterns decouple the data producer from the consumers, they typically lead to dynamic registration, side effects on the consumer side, and inversion of control [17, 46].

Reactive Programming (RP) is an alternative to these patterns for event driven computation. RP defines event streams as lazy collections and provides operators that allow developers to deal with the complications of asynchronous event handling. RP started in academia in the form of Functional Reactive Programming (FRP) [15, 18, 19, 35, 39], but in recent years the use of RP has exploded. Languages such as Elm [14] and libraries such as Reactor [27], Akka [9] and Rx [38] are being used by companies like Netflix, Microsoft and Google, to build highly responsive and scalable systems. Front-end libraries like Angular[1], that use RP in their foundations, are used by many large sites (9.1% of Quantcast Top 10k websites[2]). A group of developers and companies has standardized "Reactive Programming" in the form of the Reactive Manifesto [8].

While reactive programs offer more declarative and concise syntax for composing streams, RP does not work well with traditional interactive debuggers, shipped with most IDEs [48]. RP borrows from Functional Programming (FP) for its abstractions, its laziness and its use of "pure" functions. Those features contribute to a control flow that is hidden inside the RP implementation library and lead to non-linear execution of user code. This results in non-useful stack traces, while breakpoints do not help either, as relevant variables are frequently out of scope. Furthermore, using a low level debugger makes it harder to interact with the high level abstractions that RP provides. Compared to imperative programming, there is limited knowledge on how to efficiently debug reactive programs. Traditional imperative program debugging practices [5] do not apply to RP [48].

In this work, we address the issue of RP debugging by designing and implementing a high level debugger called RxFiddle for a popular version of RP, namely Reactive Extensions (Rx). RxFiddle (1) provides an overview of the dependencies in the data flow, (2) enables introspection of both the data flow and the timing of individual events, and (3) enables developers to trace values back through the data flow. To guide our design, we conducted interviews among professional developers. After building RxFiddle, we validated it with a user experiment involving over 100 developers. We found that RxFiddle can help developers comprehend and debug RP data flows faster.

To steer the research, we formulate the following research questions:

RQ1 How do developers debug RP?
Before designing tools it is important to understand the practices they must support along with the problems in the current state of the art [50]. For this, we performed an extensive

ICSE '18, May 27–June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180156

[1] https://angular.io/
[2] https://trends.builtwith.com/, accessed 2017-06-20

analysis of the literature (both scientific and practitioner-oriented) and conducted interviews with RP practitioners.

RQ2 How can we design a tool that helps developers debug RP?
By examining the results of RQ1, the limitations of traditional debuggers and the opportunities that RP programs offer in terms of structure and explicit dependencies between data flows, we design a novel RP debugger. We validate the design's feasibility by providing an implementation for the popular JavaScript RP library RxJS.

RQ3 Can our specialized RP debugger speed up comprehension & debugging?
To validate our design and examine whether specialized tooling can improve the debugging experience, we measure the speed and correctness of comprehension with an open experiment.

2 BACKGROUND: REACTIVE PROGRAMMING AND RX

RP is a declarative programming paradigm for working with streams of input data. According to a definition of reactivity[3] a reactive program must interact with the environment "at a speed which is determined by the environment". Conceptually, when a reactive program is run, it sets up a data processing pipeline and waits until input arrives, i.e., when the environment changes. Reactive Programming languages and libraries provide developers with a set of abstractions and methods to create such programs.

Many RP implementations share a notion of a collection that abstracts over *time*, in contrast to *space* like standard collections. This collection comes in different flavors, such as Observable (Rx [38]), Signal (Elm [14]), Signal/Event (REScala [47]) or Behavior/Event (FRP [18]). The implementations differ in the precise semantics of their collections, their execution model (push/pull), and the set of available operators. In this paper, we focus on the Rx formulation, but our work is applicable to other RP implementations to some extent.

Understanding how we derive our visualization requires a minimal understanding of the internals of Rx. Rx introduces two basic types, *Observable* and *Observer*. Observables define the data flow and produce the data while Observers receive the data, possibly moving the data further down the stream. Figure 1a shows a very basic example of an "in situ" data flow in Rx. Initially, an Observable is created, here using the static `of()` method, then dependent Observables are created using the `map()` and `filter()` methods on the Observable instance. Finally we `subscribe()` to start the data flow and send the data to the console.

Assembly. It is important to note that Observables are lazy; initially they only specify a blueprint of the desired data flow. Creating this specification is called the *assembly* phase. In Figure 1a, the assembly phase consists of the calls to `of()`, `map()` and `filter()`, creating respectively Observables o_1, o_2 and o_3 (Figure 1b).

Subscription. When the `subscribe()` method of an Observable is called, the data flow is prepared by recursively subscribing "up" the stream: every subscribe call creates an *Observer*, that is passed to the input Observable, which again subscribes an Observer to its input Observable, until finally the root Observables are subscribed to. We call this the *subscription* phase. In Figure 1a, inside the single `subscribe()` call, the Observer object s_1 is created, and passed to o_3, which in turn will recursively subscribe to o_2 with a new Observer s_2 with destination s_1, until the full chain is subscribed (Figure 1b).

Runtime. After the root Observables are subscribed to, they can start emitting data. This is the *runtime* phase. Depending on the nature of the Observable, this might attach event listeners to UI elements, open network connections or start iterating over in memory data. Events are pushed to s_3, to s_2 and finally to s_1 which calls `console.log()` in Figure 1a.

Rx identifies three types of events that can occur during the runtime phase: *next*, *error* and *complete* events. *next* events contain the next value in the flow, an *error* event signifies an unsuccessful termination to a stream, while a *complete* event denotes the successful termination of the stream. There are restrictions on their order: an Observable may first emit an unlimited amount of *next* events, and then either an *error* or a *complete* event. Observables do not need to emit any *next* events, and do not need to terminate.

More complex programs feature operators that merge Observables[4], split Observables[5] or handle higher-order Observables[6], resulting in more complex graphs. An example of a higher-order Observable operation (`flatMap()`) is shown in Figure 1d. While merging and splitting happens on an Observable level (the `source` property still points to one or more dependencies), higher-order Observable flattening only manifests within Observer structures (there is no reference between the Observables). Figure 1e shows this with an `inner` Observable that is subscribed twice (for both values 2 and 3, value 1 is skipped), resulting in two identical data flows over o_1. The data flow through $s_{4,n}$ and s_{4_m} is pushed into s_1, flattening the data flow.

Marble Diagram. The term *Marble Diagram* comes from the shape of the glyphs in the images used to explain Rx in the official documentation. An example is shown in Figure 1c. The diagrams contain one or more timelines containing the events that enter and leave Observables. Next events are typically represented with a circle, error events with a cross and complete event with a vertical line. From the diagram developers can understand how operators work by inspecting the difference between the timelines, where events might be skipped, added, transformed or delayed. Mapping time on the x-axis provides insight that is missing when inspecting only a single time slice.

3 RESEARCH DESIGN

To answer our research questions, we employ a three-phase Sequential Exploratory Strategy, one of the mixed methods research approaches [13, 28]. First, we interview professional developers and review available documentation (RQ1) to form a understanding about current debugging practices. Second, we apply this understanding to design a debugger and implement it to test its feasibility (RQ2). Finally, we validate the debugger using an experiment (RQ3).

[3] "Reactive programs [..] maintain a continuous interaction with their environment, at a speed which is determined by the environment, not the program itself." [6]

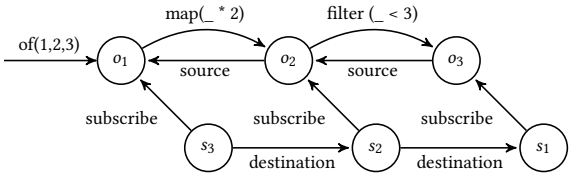
[4] `concat()`, `merge()`, `combineLatest()`, and `zip()`

[5] `partition()`, or through multicasting with `share()` or `publish()`

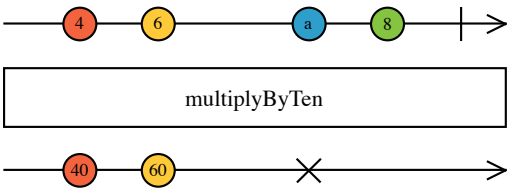
[6] `flatMap()`, `mergeMap()`, and `concatMap()`

```
Observable.of(1, 2, 3)
  .map(x => x * 2)
  .filter(x => x < 3)
  .subscribe(console.log)
```

(a) Rx code example



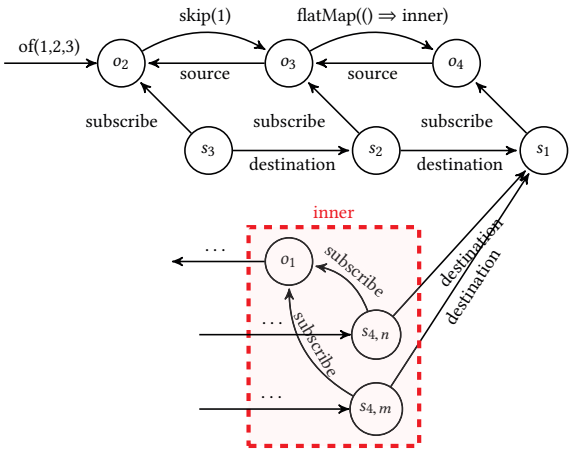
(b) Rx graph example



(c) Marble Diagram

```
let inner = Rx.Observable.of("A", "B")
let outer = Rx.Observable.of(1, 2, 3)
  .skip(1)
  .flatMap(() => inner)
  .subscribe()
```

(d) Higher-order flatMap operation



(e) Higher-order Rx graph example

Figure 1: Samples of Rx Observables

4 RQ1: RP DEBUGGING PRACTICES

To validate the need for better tools we must first understand how existing tools are used (RQ1). For this, we interview developers, as we want to explore and understand how they use existing tools and techniques to debug Rx code. The questions are semi-structured. We first establish a general understanding of the experience of the subjects. We then ask several open questions regarding their use of RP, how subjects debug RP and test RP. Table 1 lists the questions used as a guideline for the interviews.

Five developers with professional programming experience ranging from 4 to 12 years were interviewed. The first four developers (D1-D4) work in Company A, which builds reactive systems [8] using various RP solutions. Developer experience with Rx ranges from a month to over a year. The fifth developer (D5) works in Company B, and is concerned with building and maintaining a large scale distributed server application, that uses Rx to handle asynchronous events.

4.1 Interviews

In the following paragraphs we discuss the results of Q6-Q10 in detail. Not every subject answered each question in the same detail, so we discuss the answers that provide meaningful insights in the current practice.

Testing. Of the 4 subjects of Company A, none performed tests specifically for Rx logic. *"Just running the application"*, is enough according to D3, saying that they only test the business logic in their application and consider the Rx code as "glue"' which either works or not. In contrast, D5 and his team at Company B extensively test their application using the Rx library's built-in test facilities like "marble tests" and the `TestScheduler` [44]. Using tests, the subject *confirms his beliefs about the behavior* of the chain of operators, while tests are also helpful when refactoring code.

Debugging. All subjects independently mention using temporary `printf()` debugging statements (printing messages to the system output, e.g. with `console.log()` in JavaScript). Subjects use `printf()` debugging to *"add more context"* (D1) to their debug sessions. Printing which values flow through the flow allows them to *"quickly reason what happens"* (D3). Breakpoints are only used when the cost of recompilation is high, for example when TypeScript is used instead of Javascript: developers prefer to attach their debugger to a running program session rather than inserting `printf()` statements and restarting the session.

Often, it is difficult to use existing debuggers to inspect the life cycle of Observables (`subscribe()` and `dispose()`), as the corresponding code lives within the Rx library. Debugging inside the Rx library was described as *"painful"* by D2, when using the Node.js debugger to step through the inners of Rx. Alternative solutions used by our subjects are (1) creating a *custom* `debug()` *operator* which prints these life cycle events (D5), and (2) creating custom Observables (with `Observable.create()`) that override the default lifecycle methods with facilities to print life cycle events (D2, D5). While `printf()` debugging and breakpoints are useful in various degrees when executing a single Observable chain, these methods both become considerably more difficult and *"overview is easily lost"* when executing multiple chains concurrently (D3, D5).

	Question
Undestanding the subjects	
Q1	Explain your (professional) experience.
Q2	Assess your experience on a scale from beginner to expert.
Q3	Explain your (professional) reactive programming experience.
Q4	Assess your RP experience on a scale from beginner to expert.
Q5	Have you ever refactored or reworked RP code?
Content questions	
Q6	How do you test or verify the workings of Rx code?
Q7	How do you debug Rx code?
Q8	How do you use documentation on Rx?
Q9	What difficulties do you experience with RP?
Q10	What is your general approach to understand a piece of Rx?

Table 1: Interview questions

Documentation. Subjects give different reasons to consult the documentation, but the most common reason is to *"find an operator for what I need"* (D1). They feel that there might be an operator that precisely matches their needs, however knowing all operators by heart is not common (the JavaScript Rx Observable API has 28 static methods and 114 instance methods), therefore subjects sometimes end up doing an extensive search for some specific operator. Another reason to visit the documentation is to *comprehend how operators in existing code work*. For this, subjects use the Marble Diagrams at RxMarbles.com [36] (D2, D5), the RxJS 4 documentation on GitHub (D2, D5), the RxJS 5 documentation at ReactiveX.io [44] (D1, D4, D5) and the online book IntroToRx.com [10] (D4). D1 specifically mentions the need for more examples in the documentation.

Difficulties experienced. The IDE does not help with developing Rx (D2, D4); according to D4 *"Rx is more about timing than about types"*, and *"... you miss some sort of indication that the output is what you expect"*. It is not always clear what happens when you execute a piece of code, *"mostly due to Observables sometimes being lazy"* (D2). Flows are clear and comprehensible in the scope of a single class or function, but for application-wide flows it becomes unclear (D3, D4 and D5). D3 mostly used RxScala and mentions that creating micro services helps in this regard. D1 mentions that *"you need to know a lot as a starting [RxJS] developer"*, giving the example of the many ways to cleanup. D1 used both logging while analyzing existing code and learning to overcome inexperience.

Understanding. Subjects first *look at which operators are used*, then they *reason about what types and values might flow through the stream* (D2, D3, D4 and D5), using various methods. By analyzing the variable names D2 forms an expectation of the resulting value types, then reasoning backwards, to see how this data is derived. *Running the code*, is used when possible by D5, to observe the outcome of the stream, as this "shows the intentions of the original developer". If it remains unclear how the data is transformed, the subject injects a debug() operator or looks up operators in the documentation.

4.2 Analysis of Literature

Developers can learn Rx through several sources, such as the official documentation at ReactiveX.io, books, online courses, and blog posts. We gathered resources to be analyzed by selecting 4 popular books about Rx, and complement this with the official documentations and an article by a core contributor of RxJS. All reviewed resources either mention debugging briefly and suggest using the do() operator for `printf()` debugging, or teach the developer `printf()` debugging via code samples.

The RxJS 4 documentation [3] and two books [20, 42] propose the use of the do() operator for debugging. Esposito and Ciceri [20] further explain how to best format the log statements and introduce ways to limit the logging by modifying the Observable through means of throttling and sampling. The RxJava book [42] also contains tips to use the various do-operators to integrate with existing metric tools. To our knowledge the only article [37] addressing issues of debugging Rx is by Staltz, one of the contributors of RxJS, noting that conventional debuggers are not suitable for the higher level of abstraction of Observables. Staltz proposes three ways to debug Rx: (1) tracing to the console, (2) manually drawing the dependency graph, and (3) manually drawing Marble Diagrams.

We analyzed a set of 13 books about RxJS, which was created by selecting 69 books matching "RxJS" from the O'Reilly Safari catalogue [2], and further reducing the set by filtering on the terms "debug" and "debugger". While, none of the remaining books had a chapter about debugging, many of these books use `printf()` debugging in their code samples. Notably, Blackheath suggests [7], in a "Future Directions" chapter, that special debuggers could provide a graphical representation of FRP state over time and would allow debugging without stepping into the FRP engine.

4.3 Overview of practices

The available literature matches the results of the interviews: `printf()` debugging is both commonly advised and used. While the conventional debugger works in some cases, this is mostly the case for the procedural logic that interleaves Rx logic. Rx-specific debuggers are suggested, but not implemented. We found that developers use `printf()` debugging to learn the behavior of Observables, behavior meaning both their values flowing through and their (one or many) subscriptions.

Overall, we identified four overarching practices when debugging Rx code:

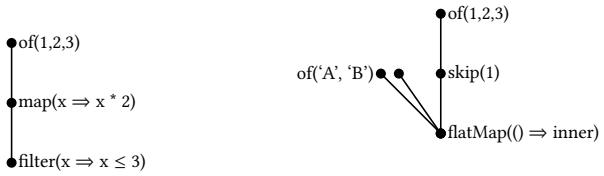
(1) Gaining high-level overview of the reactive structure.
(2) Understanding dependencies between Observables.
(3) Finding bugs and issues in reactive behavior.
(4) Comprehending behavior of operators in existing code.

5 RQ2: DEBUGGER DESIGN

In this section, we describe the design of a visualizer for the ReactiveX (Rx) family of RP libraries to answer RQ2. Given the findings of RQ1, the requirements for our visualizer are:

REQ1 *Provide an overview of Observable flows.* This overview should support practices 1 and 2, by graphically representing the relations between all Observables and their interactions.

REQ2 *Provide detailed view inside the data flow.* This view should support practices 3 and 4 by giving access to both data flow and life-cycle events and should be able to show the behavior of an operator visually.



(a) DFG of Figure 1b **(b) DFG of Figure 1e**

Figure 2: Simplified DFGs corresponding to examples in Figure 1

To meet those requirements, we propose a visualizer consisting of two parts: (1) a Data Flow Graph and (2) a Dynamic Marble Diagram. The data flow graph satisfies REQ1 by providing a high-level overview and by showing how different flows are created, combined and used. The dynamic marble diagram offers a more in-depth look into a single data flow, by showing the contents (in terms of values and subscriptions) of the flows. Developers can use it to learn the behaviors and the interplay of operators.

5.1 Data Flow Graph

Simplified graphs. When running an RP program, Observables are created that depend on other Observables (their *source*) and Observers are created to send their values to a defined set of Observers (their *destination*). Figure 1b shows these relations in a graph. For the simplest of programs, the relations between the Observables ($O = o_1, o_2, o_3$) and those between Observers ($S = s_1, s_2, s_3$) share an equally shaped sub-graph after a reversal of the Observer edges. To provide more overview, we process the graph to merge the two Observable and Observer sequences together, simplifying it in the process, resulting in a *Data Flow Graph* (DFG) as in Figure 2a. To do so, we retain only the Observer subgraph nodes, complementing them with the metadata of the corresponding Observable nodes. Higher-order relations are retained, as shown in Figure 2. Figure 3B shows the DFG in practice.

Layout. Layout is used to add an extra layer of information to the graph. If multiple subscriptions on the same Observable are created, multiple flows are kept in the graph and they are bundled together in the resulting layout. Using it, developers can find related flows. They can also identify possible performance optimizations; for example, when they see Observables to be reused often, they can introduce the `share()` operator to optimize subscriptions.

Our layout engine is based on StoryFlow [33]. StoryFlow was initially introduced to visually describe complex storylines involving multiple characters and interactions between them in a way that minimizes storyline crossings.[7] Whereas StoryFlow clusters on physical character location, we cluster flows per Observable. Furthermore, StoryFlow supports interactivity in various layout stages of which we use the algorithms for *straightening* and *dragging*. A selected flow is thus highlighted, straightened and positioned at the right in order to align with the Marble Diagram.

[7] An example visualization of the Lord of the Rings character storylines can be found here: http://www.shixialiu.com/publications/storyflow/index.html

Color. Observables can be reused, so coloring the nodes can be used to identify the same Observable in multiple places in the graph. For example, in Figure 1e the `inner` Observable is reused twice, which we denote visually by applying the same color to its two occurrences in the DFG.

5.2 Dynamic Marble Diagrams

We extend the original notion of the Marble Diagram by introducing animation; our dynamic marble diagrams update live when new events occur and are stacked to show the data in the complete flow. This allows developers to trace a value back through a flow, *an operation which is impossible using a classic debugger*. Handcrafted marble diagrams can use custom shapes and colors to represent events, but for the generic debugger we use only three shapes: next-events are a green dot, errors are a black cross and completes are a vertical line, as shown in Figure 3C. For our generic debugger, it is unfeasible to automatically decide which properties (content, shape and color) to apply to events, as the amount of events and distinguishing features might be unbounded. Instead the event values are shown upon hovering the mouse cursor on the marble.

5.3 Architecture

To support the visualization, we design a debugger architecture consisting of two components: a host instrumentation and a visualizer. By splitting the instrumentation from the visualization, the debugger can be used for the complete Rx family of libraries by only reimplementing the first component.

The **Host instrumentation** instruments the Rx library to emit useful execution events. Depending on the language and platform, specific instrumentation is required. What the instrumentation does is wrap calls to functions that i) create or modify the DFG, and ii) introduce events to Observers. The instrumentation uses an operational protocol consisting of 4 functions to drive the debugger interface.

The **Visualizer** takes the output of the host instrumentation, the initial graph, and simplifies it into a Data Flow Graph. Then it lays out the Data Flow Graph and creates the debugger's User Interface. By separating the visualizer, we can export generated graphs and visualize them post mortem, for example for documentation purposes.

The components can run in their own environment. The instrumentation must run inside the host language, while the Visualizer can use a different language and platform.

5.4 Implementation

To validate our design and to provide an implementation to the developer community, we created RxFiddle.net. The RxFiddle project is a reference implementation of our reactive debugger design. Besides the visualizer, the website also contains a code editor for JavaScript code with sharing functionality, for developers to share snippets with their peers, as shown in Figure 3A. In this section we will explain different parts of the implementation. For RxFiddle, we initially focused on RxJS (JavaScript).

Instrumentation. With JavaScript being a dynamic language, we use a combination of prototype patching and Proxies [1] to instrument the RxJS library: the Observable and Observer prototypes

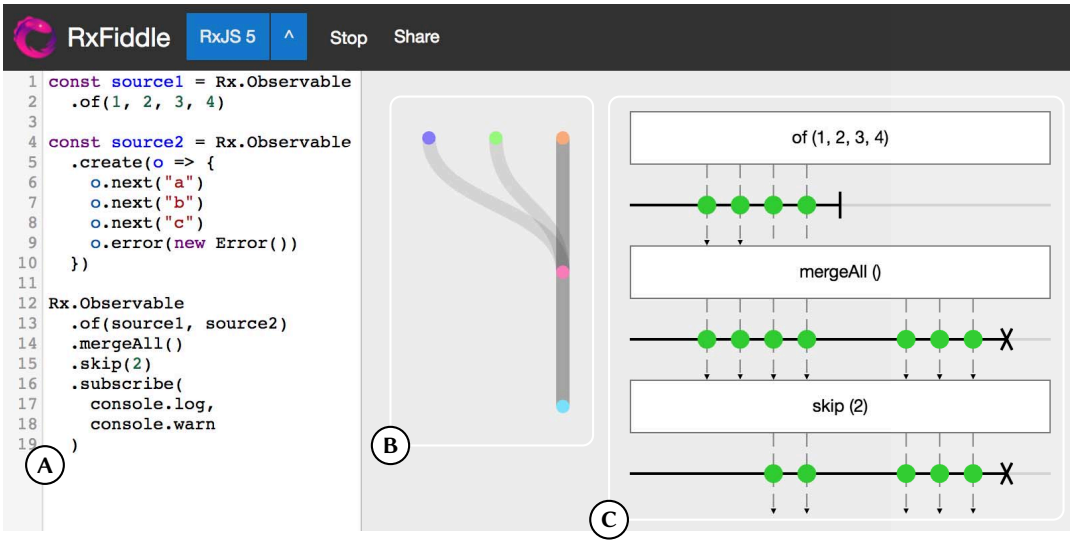


Figure 3: Screenshot of RxFiddle.net, showing the Code Editor (A), the DFG (B) and the Dynamic Marble Diagram (C)

are patched to return Proxies wrapping the API method calls. The instrumentation passes every method entry and method exit to the Linking step.

Linking. We distinguish between method calls from the different phases (Section 2). From the assembly phase, we detect when Observables are used as target or arguments of a call or as return value, and create a graph node for each detected Observable. We add an edge between the call target and call arguments and returned Observables, denoting the *source* relation. Also, we tag the returned Observable with the call frame information (time, method name, arguments). In the subscription phase, we detect calls to `subscribe()`: the destination Observers are passed as arguments, so we create the graph nodes and save the relation as an edge. In the runtime phase, we detect `next`, `error` and `complete` calls on Observers and add these as meta data to the Observer nodes.

Graph Loggers. From the Linking step the graph mutations are streamed to the environment of the visualizer, where the graph is rebuilt. Depending on the host language, a different protocol is used: RxFiddle's code editor executes the code in a Worker [1] and transmits events over the `postMessage` [1] protocol, while RxFiddle for Node.js transmits over WebSockets. Being able to support multiple protocols, extends the possible use cases; our prototype implements a code editor for trivial programs, a Node.js plugin for server applications, and Chrome DevTool extensions[8] for web applications.

Visualizer. The visualizer receives the current state in the form of a graph from the Logger. It then uses the Observers in the graph to create the DFG. To layout the DFG using StoryFlow, we first rank the graph using depth first search, remove slack [24] and reverse edges, in order to create a directed acyclic graph. We then add dummy nodes to replace long edges with edges spanning a single rank. Finally, we order and align the nodes in the ranks assigning coordinates for the visualization. It is important that layouting is fast, as it runs every time the DFG is changed. To render the Marble Diagrams, the flow *to* and *from* the selected Observer is gathered, by recursively traversing the graph in the direction of the edges.

[8]https://developer.chrome.com/extensions/devtools

6 RQ3: EVALUATION

In this section, we evaluate our debugger to assess the efficacy of our approach. To do so, we use an experiment, in which we control for the debugger facilities that subjects use. The "control" group is provided a classic web development environment, while the "treatment" group uses RxFiddle.

Ko et al. [31] describes two commonly used measures for experiments regarding tools in Software Engineering: *success* on task, and *time* on task. The goal of our experiment is to measure the *time* required to solve programming problems correctly. If our reasoning for RQ2 is right and our debugger design lends itself for RP, we expect to see that the group using RxFiddle can more quickly reason about the reactive code at hand and can trace bugs faster. We do not use success or correctness as a measure for the experiment, as we expect both groups to be able to complete the tasks correctly: while the current debugging situation is non-optimal, it is still used in practice, indicating that it works at least to some extend. The construct of time also matches debugging better; developers need to continue debugging *until* they find an explanation or a solution to their problem, while assumptions can be tested and corrected.

We measure the time from the moment the participant received the question until the correct answer is given. Participants use either the built-in Chrome Browser debugger (group *Console*) or the RxFiddle debugger (group *RxFiddle*). This single alternative Console debugger together with the experiment UI (which acts as a small IDE) offers all the debugging capabilities subjects of our preliminary interviews (RQ1) reported to use.

The experiment consists of a questionnaire, a warm-up task and four programming tasks, all available in a single in-browser application, of which the source code is available at [4]. The questionnaire

contains questions regarding age, experience in several programming languages and several reactive programming frameworks. We use this self estimation as a measurement of skill instead of a pretest, since it is a faster and better estimator [21, 30, 49]. The warm-up program is placed in the same environment as the programming problems and contains several tasks designed to let the participants use every control of this test environment. The first two programming problems requires the participants to obtain an understanding about the behavior of the program and report the findings. The last two programming problems contain a program with a bug. The participants are asked to find the event that leads to the bug in the third problem and to identify and textually propose a solution in the fourth problem. The first two problems are synthetic examples of two simple data flows, taken and adapted from the Rx documentation, while the latter two are carefully constructed to match the documented use of Rx operators and contain some mocked (otherwise remote) service which behaves like a real world example. In T3, an error in an external service propagates through the Rx stream. In T4, concurrent requests lead to out-of-order processing of responses.

We use a between-subjects design for our setup. While this complicates the results — subjects have different experience and skills — we can not use a within-subjects design as it would be impossible to control for the learning effect incurred when asking subjects to perform survey questions with and without the tool. This also allows us to restrict the amount of tasks to incorporate in the experiment, requiring less time from our busy subjects. In the experiment environment, subjects can answer the question and then hit "Submit"; alternatively they can "Pass" if they do not know the answer.

6.1 Context

The experiment was run both in a offline and in an online setting. The offline experiment was conducted at a Dutch software engineering company. Subjects are developers with several years of programming experience, ranging from little to several years of experience with RP. As we did not try to measure the effect of learning a new tool, we explained RxFiddle in the introductory talk and added the warm-up question to get every participant to a minimum amount of knowledge about the debugger at hand.

The online experiment was announced by the authors on Twitter, and consequently retweeted by several core contributors to RP libraries, and via various other communication channels, such as Rx-related Slack and Gitter topics. Subjects to the online experiment took the test at their own preferred location and have possibly very different backgrounds. We created several short video tutorials and included these in the online experiment to introduce the participants to the debug tool available to them and the tasks they needed to fulfill. The introductory talk given to the offline subjects was used as the script for the videos, in an attempt to get all participants to the same minimum level of understanding.

6.2 Results

The online experiment was performed outside of our control, and some participants quit the experiment prematurely. In total we had 111 subjects (13 offline, 98 online) starting the survey, of those 98 completed the preliminary questionnaire, and 89, 74, 67, and 58 subjects started respectively T1, T2, T3 and T4. All of the subjects in the offline setting started all tasks. Figure 4b shows the outcome of the tasks; in the remainder of this section we consider only the outcomes marked as "Correct".

Overall. Figure 4b shows the time until the correct answer was given per task. Here, we consider the combined results from the offline experiment and the online experiment. We make no assumptions about the underlying distribution, so we perform a non-parametric Wilcoxon Mann-Whitney U test (H_0*: times for the Console group and RxFiddle group are drawn from the same population*) to see if the differences are significant, and a Cliff's delta test for ordinal data to determine the effect size. The results are shown in Figure 4a.

For task T3, we can reject H_0 with high significance ($p < 0.05$), *the RxFiddle group is faster.* For the tasks T1, T2 and T4 we can not reject H_0 ($p > 0.05$), meaning the *RxFiddle group and Console group perform or could perform equally.*

Control for experience. To investigate this further, we split the results for different groups of subjects. When we control for the self-assessed Rx experience, we see bigger differences for all tasks for groups with more experience, as shown in Figure 4c and Figure 4d (we split at the median; `exp_rx` > "Beginner"-level). Still, for tasks T1, T2, and T4 we can not reject H_0, but the results are more significant comparing only experienced subjects.

7 DISCUSSION

We now discuss our main findings, how RxFiddle resolves the debugging problem of Rx, and contrast our design to other design choices and possibilities of future work.

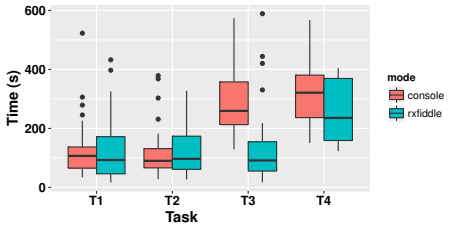
7.1 Main results

Quick and dirty debugging. Through interviews and literature we establish that current debugging practices for RP consist mostly of `printf()` debugging. The shortcomings of this method were evident from the interviews: it works reliably only for synchronous execution or when small amounts of events being logged, otherwise the overview is lost. Furthermore, the time-context of events and dependency-context of flows are not available using this method. We attribute the prevalence of `printf()` debugging to this *"quick and dirty"* method being available in every language and on every platform, without a viable alternative.

Improved context: being complete, disposing doubts. With our design and complementary implementation, we show that our abstract model of RP is suitable for visualization on two levels: overview and detail. At the overview level, we complement the dependencies visible in source code with a graph of the resulting structure, showing the run-time effect of certain operators on the reactive structure. At the detail level, we add the time context, by showing previous values on a horizontal time line, and the dependency context, by showing input and output flows above and below the flow of interest. While the results of our evaluation could be observed as a negative, RxFiddle is a new tool, where subjects have only just been exposed to the tool and received only a short training. We expect that by designing a debugger model so close to the actual abstractions, our debugger works especially well for users with some

	console	rxfiddle	W	p-value	Cliff's δ
T1	34	36	559	0.540	0.0866
T2	32	31	517	0.780	-0.0424
T3	23	28	96	$6.19e^{-6}$	0.702
T4	13	12	60	0.347	0.231

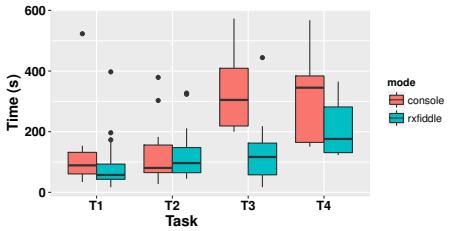
(a) Results comparing the Console and RxFiddle groups



(b) Time until correct answer per task, overall

	console	rxfiddle	W	p-value	Cliff's δ
T1	16	17	105	0.276	0.228
T2	14	13	99	0.720	-0.0879
T3	10	11	10	$7.88e^{-4}$	0.818
T4	8	7	13	$9.34e^{-2}$	0.536

(c) Results comparing the Console and RxFiddle groups, with Rx experience above "Beginner"-level.



(d) Time until correct answer per task, for subjects with more than "Beginner"-level of experience with Rx.

Figure 4: Experiment results, overall (top row) and experienced developers only (bottom row).

knowledge of these abstractions; while only T3 shows better performance with high significancy, we observe slightly better results when controlling for experience. Future research might investigate the effect of experience in more detail, including the use of more complicated tasks, with larger samples.

In the presented research, we did not perform tests with subjects using their own code. However, during piloting and after the release of RxFiddle, we received positive feedback regarding the completeness of the visualization. As one user put it, *"by using RxFiddle when learning and understanding what RxJS does in our project, I have a feeling of improved control over our Observables, Subscriptions and the reactive parts of our app"*. Specifically the life-cycle events, which are generally hard to debug using `printf()` debugging, are more clear: *"Initially we were reluctant to manually subscribe, but after seeing that 'complete' often triggers a 'dispose', the team became more confident to sometimes use* `subscribe()` *directly"*. Future research might address this evaluation aspect by designing experiments specifically using code owned by the users.

7.2 Implications

The developers using Rx in practice now have an alternative to `printf()` debugging. Developers can try RxFiddle on their codebase to better understand the reactive behavior of their application, and potentially detect and verify (performance) bugs they are not aware of. At least one example of this has already occurred in practice: one of our interview subjects reported a bug[9] in the `groupBy()` implementation of RxJS, which resulted in retention of subscriptions, increased memory usage and finally led to an out-of-memory exception. The subject detected the bug in practice and required extensive amount of debugging involving the Node.js debugger to trace down; the same bug is immediately obvious in RxFiddle when examining the life-cycle events using the visualization.

[9]https://github.com/ReactiveX/rxjs/issues/2661

Contributors of RP libraries could use tools like the RxFiddle visualizer in documentation to provide executable samples, which would allow for a better learning experience, and at the same time would introduce novice developers to other ways of debugging than `printf()` debugging.

7.3 Limitations and Future Work

Multiple inputs and outputs. If we compare our debugger visualization to the visualization of learning tools, like RxMarbles [36] or RxViz [41], the main difference is that those tools show all input and output Observables of a single operator concurrently, while RxFiddle shows one input and output Observable per Marble Diagram, part of a single full flow (a path through the graph). The choice to show a full flow allows developers to trace events from the start until the end of the flow, but restricts us in showing only a single ancestor flow per node at each vertical position, as adding a third dimension would clutter the (currently 2D) visualization. For future research, it would be interesting to compare (1) the different ways Observable streams can be combined in Marble Diagrams and (2) which visualization elements can be added to explicitly show *causality* and *lineage* for events and show *durations* for subscriptions.

Edge visualization. In our graph visualization, the edges represent the dependencies and the path of the events. Nodes with multiple incoming edges *merge* the events, however users could falsely think that all event data ends up in the outgoing path: besides data flows, Rx also uses Observables *for timing*, as durations (`window()`), as stop conditions (`takeUntil()`), or as toggles (`pausable()`). Different visual representations for joining paths could be explored to distinguish between using Observables for data or for timing.

Graph scalability. Debugging large reactive systems over longer periods of time can result in significantly larger Observable graphs

and Marble Diagrams than currently evaluated. During tests of RxFiddle with larger applications like RxFiddle itself and an existing Angular application, the graph became too large to render in real time. Besides rendering performance, a potentially even bigger issue is with communicating large graphs to the developer. We propose several extensions to RxFiddle to remedy this issue: (1) pruning the graph of old flows to show only the active flows, (2) bundling flows that have the same structure and only rendering a single instance offering a picker into the flow of interest, (3) collapsing certain parts of the graph that are local to one source file or function, (4) adding search functionality to quickly identify flows by operator or data values, (5) support navigation between code & graph.

Marble Diagram scalability. Our experience shows that while Marble Diagrams are useful for small to medium amount of events (< 20), both better performance and better functionality could be achieved by providing a different interface for high volume flows. Above a certain threshold of events, this high volume interface could be the default, offering features like (1) filtering, (2) watch expressions (to look deeper into the event's value), and advanced features like (3) histograms & (4) Fast Fourier Transform (FFT) views. Moreover, manually examining these distinct events could take a long time; a debugger could leverage the run-time information about the events that actually occur, to provide a UI. Advanced features like histograms could help the filtering process, while FFT could offer new ways to optimize the application by doing smarter windowing, buffering and sampling later on in the chain.

Breakpoints. Placing traditional breakpoints in a reactive program stops the system from being reactive, and therefore can change the behavior of the system. Breakpoints can be used by developers in two ways: i) to modify the application state by interacting with the variables in scope, and ii) to notify them of an event occurrence. While the first is arguably not desirable for reactive systems, the notification property might be a good addition to RxFiddle. BIGDEBUG [26], a debugging solution for systems like Spark [52], introduces *simulated breakpoints* for this purpose. When a simulated breakpoint is reached, the execution resumes immediately and the required lineage information of the breakpoint is collected in a new independent process. Implementing this for RxFiddle is a matter of creating the right UI as the required lineage data is already available.

Other RP implementations. RxFiddle is specific to Rx, but the debugger design is applicable to other RP implementations. The visualization should work for every RP collection abstracting over time, and would be directly applicable to languages such as REScala, and various JavaScript RP implementations. Future work could investigate whether the debugger protocol can be generalized such that other RP semantics can be captured too, for example by providing extension points for the language specific features.

8 THREATS TO VALIDITY

External validity. For the interviews we selected 5 professional developers that were both available and worked on projects involving RxJS. The online experiment was open to anyone who wanted to participate, and shared publicly. These recruitment channels pose a threat to generalizability: different practices might exist in different companies, different developer communities and for different RP implementations & languages. Future work is needed on validating the debugger in these different contexts.

Our code samples for the tasks are based on documentation samples and common use cases for Rx; RxFiddle might perform differently on real-world code, especially when the developer is familiar with the project or domain. The experiment consists of 2 small and 2 medium tasks; for larger tasks the effect of using the debugger could be bigger and therefore be better measurable. Still, we chose for these smaller tasks: in the limited time of the subjects they could answer only so many questions.

Construct validity. We measure the time between the moment a question is displayed and the moment its correct answer is submitted. Even though our questions and code samples are short and were designed to be read quickly, still some variation is introduced by different reading speeds of subjects. A setup where the question and code can be read before the time is started can remedy this threat; but introduces the problem of *planning* when given unlimited time [31]: subjects can start planning their solution before the time starts. Furthermore, subjects might have different strategies to validate their (potentially correct) assumptions before submitting, ranging from going over the answer once more, to immediately testing the answer by submitting it. However, explicitly stating that invalid answers do not lead to penalty might introduce more guessing behavior. Future studies could use longer tasks, with preparation time to read the sample software at hand, with a wizard-like experiment interface presenting one short question at a time.

Internal validity. As a result of the recruitment method of the experiment, a mixed group of developers took part, attracting even those without Rx experience. To reduce the variation in experience that this introduces, we separately examined the results of more experienced developers.

At the time of the experiment RxFiddle was already available online for use, and furthermore some of the experiment subjects had already used RxFiddle during piloting. We mitigate this issue partially by providing a instruction video at the start of the experiment, however subjects with extensive experience with RxFiddle might bias the results.

The *subject-expectancy effect* [31] poses a validity concern, since subjects who expect a certain outcome, may behave in a way that ensures it. Our subjects had the opportunity to learn the context of the experiment and thus could be more motivated to use RxFiddle than using the traditional debugger. Our online experiment captures motivation to some extend as drop-out (defined as quiting, before having started all tasks) happens; the approximately equal drop-out in both groups (RxFiddle 56.3%, Console 63.4%), suggests no significant motivational differences. Future studies could offer subjects external motivation (e.g. by ranking contenders and gamification [16] of the experiment, or organizing a raffle among top contenders), to limit the threats introduced by motivation.

9 RELATED WORK

RP Debugging. REScala [47] is an RP library for Scala, based on Scala.React. Recently a debugger model was created for REScala,

called "RP Debugging" [48], featuring a dependency graph visualization, breakpoints, a query language and performance monitoring. The debugger fully integrates with the Eclipse IDE and the Scala debugger facilities, creating a (Scala) developer experience and a feature RxFiddle currently does not offer: reactive breakpoints. Our debugger design supports multiple languages, and works outside of the IDE, in the browser environment and/or connecting to a production system. Rx has different reactive semantics and arguably a more powerful, but also more extensive API, which includes operators acting in the time domain (`delay()`, etc.). Therefore, we argue that seeing values in a flow over time is very valuable; RP Debugging shows the latest values at the selected time.

RP Visualization. RxMarbles [36] visualizes single Rx operators, for the purpose of learning and comprehension. Users can drag to modify (only) the timing of events and instantly see the changes reflected in the output. By using specific precoded inputs and timings the essence of the operator is made clear. In RxViz [41], Moroshko takes a similar approach, but provides a code editor instead of prepared inputs, and visualizes the output of the stream. RxMarbles does not support higher-order streams, while RxViz subscribes to the one outer and multiple inner streams when it detects a higher-order stream, showing them concurrently. In contrast to our work, these tools are not debuggers: focus is on teaching the behavior of single operators or stream outputs, instead of full programs.

Omniscient Debugging. Omniscient debuggers [43] trace, store and query all events in a program execution. When storing vasts amount of program execution information, performance and efficiency becomes a problem and research in omniscient debuggers focuses on this specifically. We also trace events of the entire execution, however in contrast to omniscient debuggers we only store trace events regarding RP data flows. The RP semantics allow us to create future optimizations, for example retaining only the active flow structure, while the flow's data is kept in a rolling buffer.

Dynamic Analysis. The study of program execution is called "dynamic analysis" [12]. In many cases, dynamic analysis involves a *post mortem* analysis, where first the program is run, collecting an execution trace, and then the trace data is analyzed to create a visualization. Derived visualizations, like class and instance interaction graphs, function invocation histories [32], invocation views and sequence diagrams [11] show the possibility to use trace information for debugging. Arguably, on-line analysis is more useful for debugging than the standard post mortem analysis. Reiss, in reference [45], mentions the compromises that have to be made to make an on-line analysis: reduced tracing is required to not slow down the system (known as the observer effect) and fast analysis is required to lower the cost of getting to the visualization, to not discourage the users. In our design, we handle the same compromises as they are relevant for RP debugging too, and our JavaScript trace implementation bears resemblance to that of Program Visualiser [32].

Understanding Debugging. Debugging for general purpose languages revolves around attaching a debugger, stepping through the code, attaching code or data breakpoints, navigating along different calls in the call stack and examining variables and results of expressions [51]. However, existing research, measuring how these different tasks are part of the developers work day, found that while developers spend much time on comprehending code, they do not spend much time inside the IDE's debugger [40]. Beller et al. [5] found that only 23% of their subjects actively use the IDE's debugger, with the most common action being adding breakpoints, followed by stepping through code. The automated tooling of these studies did not measure different kinds of debugging other than using the IDE provided tools, however Beller's survey indicates that 71% also uses `printf()` statements for debugging. No indication was given of any RP language and libraries used by the subjects in the study, but the observation that `printf()` debugging is common, matches our experience with debugging reactive programs.

Debugging for Program Comprehension. Developers need to both comprehend and debug code almost daily. Initially, comprehension was seen as a distinct step programmers had to make prior to being able to debug programs [29]. This distinction is criticized by Gilmore: "debugging [is] a design activity" [25], part of creating and comprehending programs. Maalej et al. [34] interviewed professional developers and found that developers require runtime information to understand a program, and that debugging is frequently used to gather this runtime information. This supports our view that debugging is not only used for fault localization, but also for comprehension.

10 CONCLUSIONS

Through analysing the current RP debugging practices, this work shows the prevalent method for RP debugging is `printf()` debugging. To provide a better alternative, we present an RP debugger design and its implementation for the RxJS libray (RxFiddle), which enables developers to: (1) gain a high-level overview of the reactive data flow structure and dependencies, and (2) investigate the values and life-cycle of a specific data flow, at run-time.

Through an experiment, we show that RxFiddle is an viable alternative for traditional debugging and in some cases outperforms traditional debugging in terms of time spent. There are several promising directions for improving our design. Specifically scalability could be improved and different edge visualizations could be explored, to improve the usability of the tool. Furthermore, by leveraging already captured meta data about timing of events, even more insight could be provided. At the implementation level, we plan to extend RxFiddle to other members of the Rx-family of libraries.

In this paper, we make the following concrete contributions:

(1) A design of a generic RP debugger, initally tuned for the Rx RP variant
(2) The implementation of the debugger for RxJS, and the service RxFiddle.net

In the month after the release of RxFiddle.net the site was visited by 784 people from 57 different countries. The debugger was already used by 53 developers, excluding the use inside of the experiment. During that same period 42846 interactions with the visualizations of the debugger have been recorded, such as selecting Observables or inspecting values by hovering the mouse over the event.

The debugger and the platform are open source and are available online at [4].

REFERENCES

[1] [n. d.]. Mozilla Developer Network. ([n. d.]). http://mdn.io/Proxy, http://mdn.io/Worker and http://mdn.io/postMessage.

[2] [n. d.]. O'Reilly Safari Books Online. ([n. d.]). http://www.safaribooksonline.com

[3] [n. d.]. RxJS 4 Documentation. ([n. d.]). https://github.com/Reactive-Extensions/RxJS/blob/master/doc/gettingstarted/testing.md

[4] Herman J Banken. 2017. RxFiddle, release 1.0.4. http://github.com/hermanbanken/RxFiddle. (July 2017). https://doi.org/10.5281/zenodo.814981

[5] Moritz Beller, Niels Spruit, Diomidis Spinellis, and Andy Zaidman. 2018. On the Dichotomy of Debugging Behavior Among Programmers. In *40th International Conference on Software Engineering, ICSE 2018, Gothenborg, Sweden*. Open Access version: https://pure.tudelft.nl/portal/files/38319543/paper.pdf.

[6] G Berry. 1989. *Real time programming: special purpose or general purpose languages*. Technical Report 1065. INRIA.

[7] Stephen Blackheath and Anthony Jones. 2016. *Functional Reactive Programming*. Manning, New York.

[8] Jonas Bonér, Dave Farley, Roland Kuhn, and Martin Thompson. 2014. The reactive manifesto. (2014). http://www.reactivemanifesto.org/pdf/the-reactive-manifesto-2.0.pdf

[9] Jonas Bonér, Viktor Klang, and Roland Kuhn. 2010. Akka Library. (2010). http://akka.io

[10] Lee Campbell. 2012. Introduction to Rx. (2012). http://www.introtorx.com

[11] Bas Cornelissen, Andy Zaidman, Danny Holten, Leon Moonen, Arie van Deursen, and Jarke J van Wijk. 2008. Execution trace analysis through massive sequence and circular bundle views. *Journal of Systems and Software* 81, 12 (2008), 2252–2268.

[12] Bas Cornelissen, Andy Zaidman, Arie Van Deursen, Leon Moonen, and Rainer Koschke. 2009. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering* 35, 5 (2009), 684–702.

[13] John W Creswell. 2013. *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage publications, Thousand Oaks, CA, USA.

[14] Evan Czaplicki. 2012. Elm: Concurrent FRP for functional GUI. (2012).

[15] Evan Czaplicki and Stephen Chong. 2013. Asynchronous Functional Reactive Programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, Article Czaplicki:2013:AFR:2491956.2462161, 12 pages. https://doi.org/10.1145/2491956.2462161

[16] Darina Dicheva, Christo Dichev, Gennady Agre, and Galia Angelova. 2015. Gamification in education: A systematic mapping study. *Educational Technology & Society* 18, 3 (2015), 75–88.

[17] Jonathan Edwards. 2009. Coherent Reaction. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, New York, NY, USA, Article 1640058, 8 pages. https://doi.org/10.1145/1639950.1640058

[18] Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*. ACM, New York, NY, USA, 263–273. https://doi.org/10.1145/258948.258973

[19] Conal M. Elliott. 2009. Push-pull Functional Reactive Programming. In *Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell (Haskell '09)*. ACM, New York, NY, USA, 25–36. https://doi.org/10.1145/1596638.1596643

[20] Antonio Esposito and Michael Ciceri. 2016. *Reactive Programming for. NET Developers*. Packt Publishing Ltd, Birmingham, UK.

[21] Janet Feigenspan, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. 2012. Measuring programming experience. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*. IEEE, IEEE, New York, NY, USA, 73–82. https://doi.org/10.1109/ICPC.2012.6240511

[22] Keheliya Gallaba, Ali Mesbah, and Ivan Beschastnikh. 2015. Don't Call Us, We'll Call You: Characterizing Callbacks in Javascript. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, New York, NY, USA, 1–10. https://doi.org/10.1109/ESEM.2015.7321196

[23] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[24] Emden R Gansner, Eleftherios Koutsofios, Stephen C North, and K-P Vo. 1993. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering* 19, 3 (1993), 214–230.

[25] David J Gilmore. 1991. Models of debugging. *Acta psychologica* 78, 1 (1991), 151–172.

[26] Muhammad Ali Gulzar, Matteo Interlandi, Seunghyun Yoo, Sai Deep Tetali, Tyson Condie, Todd Millstein, and Miryung Kim. 2016. BigDebug: Debugging Primitives for Interactive Big Data Processing in Spark. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, Article 2884813, 12 pages. https://doi.org/10.1145/2884781.2884813

[27] Felipe Gutierrez. 2017. *Reactive Messaging*. Apress, Berkeley, CA, 163–178. https://doi.org/10.1007/978-1-4842-1224-0_10

[28] William E Hanson, John W Creswell, Vicki L Plano Clark, Kelly S Petska, and J David Creswell. 2005. Mixed methods research designs in counseling psychology. *Journal of counseling psychology* 52, 2 (2005), 224.

[29] Irvin R Katz and John R Anderson. 1987. Debugging: An analysis of bug-location strategies. *Human-Computer Interaction* 3, 4 (1987), 351–399.

[30] Sebastian Kleinschmager and Stefan Hanenberg. 2011. How to Rate Programming Skills in Programming Experiments?: A Preliminary, Exploratory, Study Based on University Marks, Pretests, and Self-estimation. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU '11)*. ACM, New York, NY, USA, Article Kleinschmager:2011:RPS:2089155.2089161, 10 pages. https://doi.org/10.1145/2089155.2089161

[31] Andrew J Ko, Thomas D Latoza, and Margaret M Burnett. 2015. A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering* 20, 1 (2015), 110–141.

[32] Danny B Lange, Yuichi Nakamura, et al. 1995. Program Explorer: A Program Visualizer for C++.. In *Proceedings of the USENIX Conference on Object-Oriented Technologies on USENIX Conference on Object-Oriented Technologies (COOTS) (COOTS'95)*. USENIX Association, Berkeley, CA, USA, Article Lange:1995:PEP:1268098.1268103, 1 pages. http://dl.acm.org/citation.cfm?id=1268098.1268103

[33] Shixia Liu, Yingcai Wu, Enxun Wei, Mengchen Liu, and Yang Liu. 2013. Storyflow: Tracking the evolution of stories. *IEEE Transactions on Visualization and Computer Graphics* 19, 12 (2013), 2436–2445.

[34] Walid Maalej, Rebecca Tiarks, Tobias Roehm, and Rainer Koschke. 2014. On the comprehension of program comprehension. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23, 4 (2014), 31.

[35] Ingo Maier, Tiark Rompf, and Martin Odersky. 2010. *Deprecating the observer pattern*. Technical Report. EPFL.

[36] A. Medeiros. 2014. RxMarbles. (2014). http://rxmarbles.com

[37] A. Medeiros. 2015. How to debug RxJS code. (2015). http://staltz.com/how-to-debug-rxjs-code.html Online; Accessed September 2016.

[38] E Meijer. 2010. Subject/Observer is dual to iterator. In *FIT: Fun Ideas and Thoughts at the Conference on Programming Language Design and Implementation*. ACM, New York, NY, USA, 1–2.

[39] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. 2009. Flapjax: A Programming Language for Ajax Applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, New York, NY, USA, 1–20. https://doi.org/10.1145/1640089.1640091

[40] Roberto Minelli, Andrea Mocci and, and Michele Lanza. 2015. I Know What You Did Last Summer: An Investigation of How Developers Spend Their Time. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension (ICPC '15)*. IEEE Press, Piscataway, NJ, USA, Article 2820289, 11 pages. http://dl.acm.org/citation.cfm?id=2820282.2820289

[41] M. Moroshko. 2017. RxViz: Animated playground for Rx Observables. (2017). https://github.com/moroshko/rxviz

[42] Tomasz Nurkiewicz and Ben Christensen. 2016. *Reactive Programming with RxJava*. O'Reilly Media, CA, USA.

[43] Guillaume Pothier and Éric Tanter. 2009. Back to the future: Omniscient debugging. *IEEE software* 26, 6 (Nov 2009), 78–85. https://doi.org/10.1109/MS.2009.169

[44] ReactiveX 2014. ReactiveX.io. (2014). http://reactivex.io/

[45] Steven P. Reiss. 2006. Visualizing Program Execution Using User Abstractions. In *Proceedings of the 2006 ACM Symposium on Software Visualization (SoftVis '06)*. ACM, New York, NY, USA, Article Reiss:2006:VPE:1148493.1148512, 10 pages. https://doi.org/10.1145/1148493.1148512

[46] Guido Salvaneschi, Sven Amann, Sebastian Proksch, and Mira Mezini. 2014. An Empirical Study on Program Comprehension with Reactive Programming. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, Article 2635895, 12 pages. https://doi.org/10.1145/2635868.2635895

[47] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. 2014. REScala: Bridging Between Object-oriented and Functional Style in Reactive Applications. In *Proceedings of the 13th International Conference on Modularity (MODULARITY '14)*. ACM, New York, NY, USA, Article Salvaneschi:2014:RBO:2577080.2577083, 12 pages. https://doi.org/10.1145/2577080.2577083

[48] Guido Salvaneschi and Mira Mezini. 2016. Debugging for reactive programming. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, Article Salvaneschi:2016:DRP:2884781.2884815, 12 pages. https://doi.org/10.1145/2884781.2884815

[49] Janet Siegmund, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. 2014. Measuring and Modeling Programming Experience. *Empirical Softw. Eng.* 19, 5 (Oct. 2014), 1299–1334. https://doi.org/10.1007/s10664-013-9286-4

[50] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. 2010. An Examination of Software Engineering Work Practices. In *CASCON First Decade High Impact Papers (CASCON '10)*. IBM Corp., Riverton, NJ, USA, Article 1925815,

15 pages. https://doi.org/10.1145/1925805.1925815

[51] Diomidis Spinellis. 2016. *Effective Debugging: 66 Specific Ways to Debug Software and Systems* (1st ed.). Addison-Wesley Professional, Boston, MA, USA.

[52] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, Berkeley, CA, USA, Article 2228301, 1 pages. http://dl.acm.org/citation.cfm?id=2228298.2228301