# Debugging Support for Reactive Programming

## Feasibility of a Ready-to-hand Debugger for RxJS

Anonymous Author(s)


## ABSTRACT

Debugging reactive data-flow-oriented applications is a cumbersome task. Unfortunately, modern development environments provide only suitable tools to debug control-flow-oriented programs. As a result, software engineers utilizing RxJS, a popular library for reactive programming in JavaScript, use inapt debugging tools, utilities outside of their accustomed IDE, or antiquated debugging practices like manual print statements. This paper presents two contributions to reactive debugging: (i) Operator log points, a novel debugging utility for reactive programming, make manual print statements obsolete. We implement them for RxJS as an extension for Microsoft Visual Studio Code. By doing so, we integrate the utility with the workflow of software engineers seamlessly, thus (ii) proof the feasibility of a ready-to-hand debugging utility for reactive programming by existence.



## CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; *Data flow languages*; *Software maintenance tools*; • **Human-centered computing** → *Human computer interaction (HCI)*; *User centered design*.

## KEYWORDS

reactive programming, reactive debugging, human computer interaction, user centered design

**ACM Reference Format:**
Anonymous Author(s). 2022. Debugging Support for Reactive Programming: Feasibility of a Ready-to-hand Debugger for RxJS. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2022).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn


## 1 INTRODUCTION

When software engineers look at the source code of an existing application, they want to understand how the program was implemented technically. They do this either because they want to get themselves acquainted with a new code base they never worked with before (e.g., during onboarding of a new team member) or, more often, because someone reported an unexpected application behavior (e.g., the program crashed). Inspecting source code at runtime is commonly known as "debugging" [1]. Layman et al. [6] formalized an iterative process model (see Figure 1) by dividing the broader task of debugging into three steps: The engineer uses (i) gathered context information to build a hypothesis on what the problem at hand might be. They then (ii) instrument the program using appropriate techniques to prove their hypothesis. Eventually, they (iii) test the instrumented program. If the outcome proves the hypothesis to be correct, the process ends. Otherwise, the engineer uses gained insight as input for the next iteration.

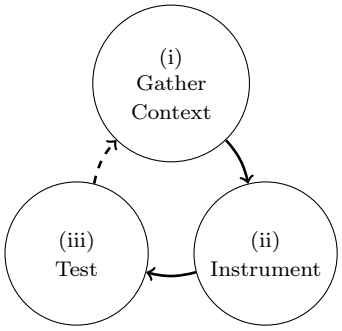


**Figure 1: Iterative Debugging Process after Layman et al.: Gather context to formalize hypothesis, instrument and test system to prove hypothesis, resulting in a new iteration or a confirmed hypothesis.**

The most basic debugging technique for instrumentation and testing is manually adding print statements to the source code: They generate execution logs when placed across the program's code and allow to reconstruct its runtime behavior. However, the number of generated log entries increases, the required amount of work to analyze the logs gets out of hand quickly. This is why specialized debugging utilities provide tools to interact with a program at runtime: After interrupting program execution with a breakpoint, they allow engineers to inspect stack frames, inspect and modify variables, step through successive source code statements, or resume program execution eventually. These utilities work best with imperative or control-flow-oriented programming languages since they interact with statements and stack frames of the debugged program.

Modern IDEs enable software engineers to debug programs, no matter what programming language they are implemented with, using one generalized user interface (UI). The result is a unified user experience (UX) where debugging support is only a click away.


*ISSTA 2022, 18-22 July, 2022, Daejeon, South Korea*

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

However, by adopting control-flow-oriented debugging utilities into their workflows, software engineers face a new problem when working with reactive programming (RP). Salvaneschi et al. [16] described this shortcoming of traditional debuggers when confronted with RP and coined the concept of *RP Debugging*. Later, Banken et al. [3] proposed a solution for debugging RxJS RP programs in an external visualizer utility.

Alabor et al. [2] examined the RP debugging habits of software engineers in an observational study. They replicated the observation by Salvaneschi et al. and observed that even engineers aware of RP debugging tools did not use them. Instead, these engineers used manual print statements.

Within this paper, we are going to present two contributions to the field of RP debugging:

1. *Operator log points* are a novel utility for debugging RP programs. They make manual print statements obsolete by providing specialized log points for RP applications.
2. By implementing operator log points in *RxJS Debugging for Visual Studio Code*, an extension for Microsoft Visual Studio Code[1] (vscode), we provide a proof by existence for the feasibility of a ready-to-hand RP debugging utility. Software engineers can debug RxJS programs without learning new UX patterns or additional setup effort.

Before we do a deep-dive on the functionality of operator log points in Section 4, we present an example for the primary challenge of RP debugging in Section 2 and discuss related work in Section 3. Next, we give an overview of performed usability inspections and validations in Section 5. Finally, we consider threats to validity regarding the usability tests in Section 6 and introduce topics for future work in Section 7.

## 2 RP DEBUGGING: THE HARD WAY

A primary characteristic of RP is the paradigm shift away from imperatively formulated, control-flow-oriented code (see Listing 1) to declarative, data-flow-focused source code [16]. Instead of instructing the computer how to do what, i.e., one step after another, we use RP abstractions to describe the transformation of a continuous flow of data.

RxJS implements reactive sources with *Observables*. An observable generates five types of life cycle events: Once a consumer (i) *subscribes* to an observable, the observable starts to (ii) *emit* values, (iii) *completes* (e.g., when a network request has been completed), fails with an (iv) *error*, or may get (v) *unsubscribed*. Engineers use *Operators* to transform these events on their way through the data-flow graph. An operator modifies values, composes other observables, or changes how life cycle events get forwarded (e.g., catch an error and emit an empty value instead). Listing 2 shows

[1] https://code.visualstudio.com

```
1 import reportValue from './reporter';
2
3 for (let i = 0; i < 5; i++) {
4   if (i < 4) {
5     reportValue(i * 2);
6   }
7 }
```

**Listing 1: Basic example of imperative-style/control-flow-oriented programming in JavaScript: Multiply integers between 0 and 4 for every value that is smaller than 4 and call reportValue with the result.**

```
1 import reportValue from './reporter';
2 import { of } from 'rxjs';
3 import { filter, map } from 'rxjs/operators';
4
5 of(0, 1, 2, 3, 4).pipe( // Observable with ints 0..4
6   filter (i => i < 4),   // Operator omitting 4
7   map(i => i * 2),       // Operator multiplying by 2
8 ).subscribe(reportValue)
```

**Listing 2: Basic RP example implemented with RxJS in JavaScript: Generate a data-flow of integers from 0 to 4, skip values equal or larger then 4, multiply these values by 2 and call reportValue with each resulting value.**

an example of a source observable, two operators, and one consumer.

Traditional debuggers reach their limitations when facing data-flow-oriented code: While we can navigate through the successive iterations of the *for* loop in Listing 1 using the step controls of the debugger, this is not possible for the transformations described in Listing 2. Assuming we set a breakpoint within the lambda function passed to *filter* on Line 6, stepping over to the next statement will not lead to the lambda of *map* on Line 7 as one might expect. Instead, the debugger continues in the internal implementations of *filter*, part of the RxJS RP runtime. With a deeper understanding of the difference between control- and data-flow-oriented programming, this might look plausible. However, previous research [2,3,16] revealed that software engineers expect different behavior from the debugging tools they have at hand. As a direct consequence, engineers fall back to the problematic debugging technique of adding manual print statements, as exemplified in Listing 3 on the next page.

## 3 RELATED WORK

Salvaneschi et al. [16] identified the divergence between a control-flow-oriented debugger's expected and actual behavior as one of their key motivations for RP debugging. The stack-based runtime model of control-flow-oriented debuggers does not match the software engineers' data-flow-oriented mental model of the program they are debugging. Because the debugger has a "lack of abstraction," it cannot interpret

```
1  import reportValue from './reporter';
2  import { of } from 'rxjs';
3  import { filter, map, tap } from 'rxjs/operators';
4
5  of(0, 1, 2, 3, 4).pipe(
6    tap(console.log),      // <-- Print Statement
7     filter (i => i < 4),
8    tap(console.log),      // <-- Print Statement
9    map(i => i * 2),
10   tap(console.log),      // <-- Print Statement
11 ).subscribe(reportValue)
```

**Listing 3: Manually added print statements on Lines 6, 8 and 10 to debug a data-flow implemented with RxJS in JavaScript.**

high-level RP abstractions and works on the low-level implementations of the RP runtime extension instead. Salvaneschi et al. proposed *Reactive Inspector* [15], the first specialized RP debugging solution for RP programs implemented with REScala, an RP extension for the Scala programming language. Integrated with the Eclipse IDE, the utility provides a wide range of RP debugging functionalities like the visualization of data-flow graphs and the information that traverses through them. Reactive breakpoints allow to interrupt program execution once a graph node reevaluates its value.

Since then, RP has gained more traction across various fields of software engineering. With a shared vision on how to surface RP abstractions at the API level, *ReactiveX*[2] consolidated numerous projects under one open-source organization. Together, its members provide RP extensions for many of today's mainstream programming languages like Java, C#, and Swift. For the development of JavaScript-based applications, software engineers can rely on RxJS[3]. Angular by Google is one of the more popular adopters of this library and uses RxJS to model asynchronous operations like fetching data in web frontend applications.

Two years after Salvaneschi et al. proposed RP Debugging, Banken et al. [3] showed that debugging RxJS-based RP programs is quite similar to REScala-based ones. They were able to categorize the debugging motivations of their study participants into four main, overarching themes. These directly correlate with the debugging issues identified by Salvaneschi et al. earlier, as we show in Table 1.

Banken et al. provided a debugger in the form of an isolated visualizer: *RxFiddle*. The browser-based application visualizes the runtime behavior of an RxJS program in two dimensions: A central (i) data-flow graph shows which elements in the graph interact with each other, and a dynamic (ii) marble diagram[4] represents the processed values over time.

[2]http://reactivex.io/
[3]https://rxjs.dev
[4]Marble diagrams are a visualization technique used throughout the ReactiveX community to graphically describe the behavior of observable-based data-flow graphs. A marble represents a life cycle event, e.g., an emitted value. Multiple marbles are arranged on a thread from left to right, indicating the point in time when the respective life cycle event happened. See https://rxmarbles.com/ for examples.

**Table 1: Correlation of debugging issues identified/-solved by Salvaneschi et al. with overarching debugging motivations by Banken et al.**

| Salvaneschi et al. | Banken et al. |
|---|---|
| Missing dependencies | Understanding dependencies between observables |
| Bugs in signal expressions | Finding bugs and issues in reactive behavior |
| Understanding RP programs | Comprehending behavior of operators in existing code<br>Gaining high-level overview of the reactive structure |
| Performance Bugs | - |
| Memory and Time Leaks | - |

Both Salvaneschi et al. and Banken et al. suggested technical architectures for RP debugging systems. Both suggestions can be summarized as distributed systems consisting of two main components: The (i) RP runtime is instrumented to produce debugging-relevant events (e.g., value emitted or graph node created). These events get processed by the (ii) debugger, which provides a UI to inspect the RP program's state.

Another two years after Banken et al. published their work, Alabor et al. [2] examined the state of RxJS RP debugging. Software engineers still struggled to use appropriate tools to debug RxJS programs according to the interviews they conducted. The authors performed an observational study and found instances of engineers who knew about RP-specific debugging tools but abstained from using them during the experiment. They credited this circumstance to the fact that the IDEs of their subjects did not provide suitable RP debugging utilities ready-to-hand.

Alabor et al. conclude that knowing the correct RP debugging utility (e.g., *RxFiddle*) is not enough. The barrier to using such utilities must be minimized. I.e., RP debugging utilities must be fully integrated into the IDE to live up to their full potential, so using them is ideally only an engineer's keypress away and adheres to accustomed, known UX patterns.

# 4 AN RXJS DEBUGGER READY-TO-HAND

We translated these findings into the central principle for the design of our RP debugger for RxJS: *Ready-to-hand*. Software engineers should always have the proper debugging tool available, no matter what program they are currently working on. Further, this tool should integrate with the engineer's workflow seamlessly.

## 4.1 Operator Log Points

Operator log points combine the concept of log points as known from control-flow-oriented debuggers with live *probes*,

formerly proposed by McDirmid [7][5] for RP programs. They display life cycle events produced by an RxJS operator directly within the source code editor.

Possible operator log points are suggested ready-to-hand through an icon annotation within the code editor, next to the respective operator. While the software engineer instruments the source code to prove their debugging hypothesis, they can enable a log point by hovering the mouse pointer over its associated annotation and selecting the *Add Operator Log Point* action (see Figure 2). When ready to test their hypothesis, the engineer starts the RxJS program using the built-in JavaScript debugger; no extra effort is required. Once the program is running, each enabled operator log point displays the life cycle events together with the source code that produced them. Engineers are free to enable or disable additional log points during the debugging session; the life cycle event display will adapt accordingly.

Once finished debugging, the software engineer stops the program. Contrary to manual print statements, no clean-up work is necessary afterward since operator log points do not require any code modifications.

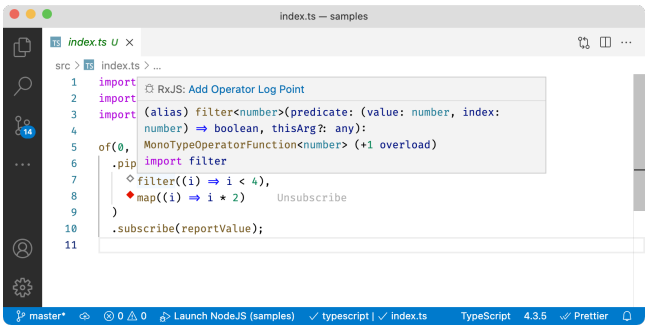


**Figure 2: *RxJS Debugging for vscode* used to debug code from Listing 2. A diamond icon indicates operator log points: A grey outline represents a suggested log point (Line 7), a filled, red diamond an enabled log point (Line 8). The source code editor shows life cycle events at the end of the respective line (Line 8, "Unsubscribe"). Log points are managed by hovering the respective icon and selecting the appropriate action.**

## 4.2 Suggesting a Log Point

Log points for operators are automatically suggested while the software engineer edits the source code of an RxJS program. To interpret the programs code semantically, the debugger extension leverages on the TypeScript[6] programming language toolchain.

We use the TypeScript parser to continuously evaluate source code, which results in an abstract syntax tree (AST). Along with the semantical structure of the program, the AST contains type and positional information for every parsed token. The extension processes the type information to detect all present RxJS operator functions. For every operator function found, the positional information allows to annotate the relevant source code in the editor with an icon.

## 4.3 Architecture

The technical architecture of *RxJS Debugging for vscode* (see Figure 4) is a refined version of the system proposed by Banken et al. [3].

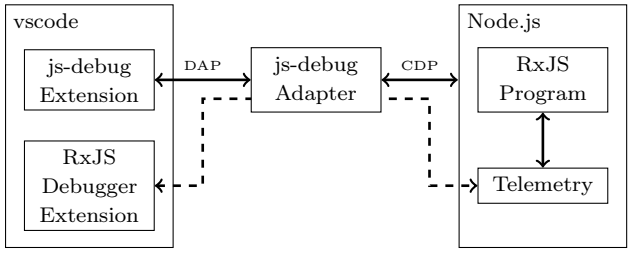


**Figure 3: The *Telemetry* component instruments the *RxJS program* (right). The *RxJS Debugger Extension* runs inside of the vscode process. The two components communicate with each other by reusing the CDP communication channel established by the generic vscode JavaScript debugger called *js-debug*.**

JavaScript virtual machines (VM) like V8 (used in Google Chrome or Node.js) or SpiderMonkey (used in Mozilla Firefox) implement the Chrome DevTools Protocol (CDP)[7]. Debugging tools like vscode's built-in JavaScript debugger use CDP to connect and debug JavaScript programs. RxFiddle by Banken et al. [3] uses WebSockets to exchange relevant data. We leverage the CDP connection established by the vscode's JavaScript debugger, making the system more robust since we do not need to maintain an additional channel for debugger communication.

# 5 USABILITY INSPECTION AND VALIDATION

We followed a User-Centered Design (UCD) [5] approach in three iterations to conceptualize and implement our debugging utility. The relevant methods we applied helped us to keep our efforts aligned with our main goal: To establish a debugging utility that is ready to hand and does not requiry any extra learning or setup procedures.

After sketching a rough proof of concept (PoC) in the first step, we performed a cognitive walkthrough [17] to validate our idea of replacing manual print statements with operator log points. The resulting data helped us to build a prototype of the extension. Next, we used this prototype to conduct a moderated remote usability test with three subjects. This allowed us to uncover pitfalls in the UX concept and find

[5] As a matter of fact, operator log points were originally called *operator probes*, but got renamed after initial confusion with our test users.
[6] TypeScript is a strongly typed programming language that compiles to JavaScript https://www.typescriptlang.org/
[7] https://chromedevtools.github.io/devtools-protocol/

**Table 2: Cognitive walkthrough action sequence with eight steps.**

| Step | Task |
|---|---|
| 1 | Open File |
| 2 | Navigate to Operator |
| 3 | Open Code Actions |
| 4 | Create Operator Log Point |
| 5 | Open Operator Log Point Monitor |
| 6 | Launch Application |
| 7 | Interact with Application |
| 8 | Interpret Runtime Behavior |

misconceptions early in the development process. Finally, we used the results of these sessions for further refinement. We completed the first minor version of the RxJS RP debugger, which we released to the Visual Studio Marketplace in May 2021[8].

We used the test cases created by Alabor et al. [2] for both the cognitive walkthrough and the remote usability test.

## 5.1 Cognitive Walkthrough

We concluded the first iteration of our development process with a PoC demonstrating the basic concept of operator log points.

Looking for an informal, expert-driven usability inspection method [8], we found the cognitive walkthrough [17] to be a good fit in this early stage of development. We prepared the profile of a typical user for the RP debugger as input to the inspection. Based on this profile and the debugging process by Layman et al. [6], we created the action sequence available in Table 2. We performed the walkthrough using the *Problem 1* web application by Alabor et al. [2].

The cognitive walkthrough revealed six usability issues, as summarized in Table 3. The full inspection report, including the complete user profile, is available on Github[9].

## 5.2 Moderated Remote Usability Test

After the initial validation using the cognitive walkthrough, we completed the development of the refined prototype, ready to test with real users.

*5.2.1 Study Design.* "Think aloud" tests for high functionality systems benefit from at least five test subjects or more [9]. The feature spectrum of the RP debugger prototype is small; hence the probability of finding major usability issues with a smaller subject population is high. Therefore, we decided to work with three individual subjects for our study.

Participants, recruited via Twitter, were required to have worked with RxJS during the past year and use vscode as their primary IDE. We sent out a PDF containing a short briefing and a prototype description a week before the actual test session. The briefing contained information about software requirements (Zoom, Node.js, npm/Yarn, and vscode) and details on what the subjects might encounter during their test session. Here, we emphasized the importance of "think aloud" [4,11], the practice of continuously verbalizing thoughts without reasoning about them.

[8] An anonymized excerpt of the extensions Marketplace presence is available in the supplementary material.
[9] The report is available in the supplementary material.

**Table 3: UX issues identified using cognitive walkthrough inspection.**

| Step | Issue |
|---|---|
| 3 | The user might know code actions, indicated through the yellow light bulb icon, for providing refactoring and quick fix options. It is questionable if they would expect operator log point options in here as well. |
| 4 | When enabling an operator log point, the user does not get any confirmation that this action was successful. Exception: The list of enabled operator log points in the debugging view is visible. |
| 5 | The monitoring pane, showing logs for enabled operator log points, must be opened manually. The user might not be aware of this after enabling a log point. |
| 5 | The monitoring pane is empty initially. Users might not know what to do next after opening it. |
| 7 | The user might not interact with the RP program in the opened default browser in order to get live feedback in the monitoring pane. |
| 7 | The opened default browser might overlay the monitor pane in vscode. Because of this, the user might miss on the live trace of values and life cycle events. |

*5.2.2 Study Execution.* At the start of a test session, we provided each participant with a ZIP file[10] containing the *Problem 2* web application by Alabor et al. [2] and the packaged version of the debugger extension prototype[11]. While the subject prepared their development environment, we started the video, screen, and audio recording with their consent. Also, we gave a scripted introduction to the code base they just received.

The participants had 25 minutes to resolve as many bugs as possible using the debugger prototype. Rather than tracking each subject's success rate of fixed defects, we emphasized detecting usability issues in their workflow instead.

*5.2.3 Study Evaluation.* One participant could not get the prototype extension up and running on their system, which means we had only two valid data sets for further evaluation after study execution. We categorized the observed usability issues by debugging process phase (i.e., gather context, instrument hypothesis, and test hypothesis) and task (e.g., "Setup Environment," "Manage Log Points," or "Interpret Log"). From a total of 10 issues, we observed four being a

[10] *This link might reveal the author(s) identity/identities https://github.com/ANONYMOUS*
[11] *This link might reveal the author(s) identity/identities https://github.com/ANONYMOUS*

**Table 4: Major UX issues observed during usability test sessions.**

| Phase | Task | Issue |
|---|---|---|
| Instru. | Setup | Participant starts the application in debugging mode, even though they have started it before. |
| Instru. | Manage | Participant unable to find log point list in debugging view. |
| Test | Interpret | Participant has difficulties to make a connection from a log point to the generated log entry. |
| Test | Interpret | Participant interprets logged value as the "input" of the instrumented operator. |

problem for both remaining study subjects. Thus we prioritized them as "major." The full usability issue report is available on Github[12]. Table 4 presents the four major issues.

## 5.3 Utilization

*5.3.1 Application of Results.* We applied the results from the cognitive walkthrough and the usability tests to refine and complete the RxJS RP debugger presented in Section 4. For example, both the PoC and the prototype had an extra view for displaying the output of a log point, visually disconnecting them from each other. We classified this circumstance as prone to confuse the user during the walkthrough but did not change the prototype yet. The usability tests with real subjects confirmed our suspicion, however. Because of this, we changed the UI for the final, current version and introduced the inline display for log point output directly in the code editor. Another example of an improvement is how the debugger suggests operator log points: The subjects were unaware that suggested log points were available via the code action menu, even though this is an established UX pattern in vscode. Therefore, we removed the suggestions from this menu and introduced the diamond-shaped indicator icon, which is always visible.

*5.3.2 Concept Verification.* The applied inspection and verification methods, in combination with the practical implementation of the debugger, deliver the existence proof for the feasibility of a ready-to-hand RP debugging utility. Even though the usability test revealed four major usability issues, we successfully verified that operator log points resolve the problems previously identified by Alabor et al. [2].

# 6 THREATS TO VALIDITY

The results of the usability test are subject to the following threats and limitations:

## 6.1 Internal Validity

We performed the usability test in an uncontrolled, remote environment, and all participants used their own computers and software installations. The downside of this is the early failure of one subject, which could not get the prototype extension running on their system resulting in an invalid data set. Even though we could have prevented this situation in a controlled lab environment, we consciously decided to take this risk and, in turn, get more realistic results from users working in the context of their accustomed development environment.

## 6.2 External Validity

Due to the circumstance that one study participant could not set up the prototype extension, we ended up having only two valid data sets after the remote usability test. Two test subjects should have allowed us to find around 50% of all usability issues present [10]. Because the two remaining subjects share four of 10 issues, we are confident that we identified the most critical usability problems nonetheless.

## 6.3 Construct Validity

We carefully moderated the test session once test subjects fell silent for more than 10 seconds and reminded them to "think aloud." Even though the participants told us that "speaking to themselves" created an unfamiliar environment for them, we expect the moderation techniques used [4] to minimize any influences on the results.

# 7 FUTURE WORK

There are several ways how future work can contribute to the efforts presented in this paper.

## 7.1 Field Test

Version 0.1.2 of *RxJS Debugging for vscode* can debug RxJS programs running in the Node.js JavaScript VM. The major release 1.0.0 generalizes this solution further and brings operator log points to RxJS applications running in web browsers. Thus, we expect installations of the debugger to increase further since more software engineers can benefit from its features.

We see the opportunity for a comprehensive field test on how engineers use the novel RP debugger once its next iteration is available. Usage statistics provided through the planned analytics reporting module will prove helpful in these regards.

## 7.2 Visualizer Component

Banken et al. [3] proposed visualization techniques for RxJS data-flow graphs in *RxFiddle*. The debugging utility we presented in this paper benefits from the integration of such a visualizer. The graphical representation of an observable graph helps novice engineers to understand RxJS concepts better, and experienced engineers get a new angle on the composition of multiple observables when debugging.

## 7.3 Record and Replay

A software engineer can record the behavior of a RP program and replay that data independently as many times as they

[12]The report is available in the supplementary material.

wish later [12]. Such a function would allow two things: During debugging, the engineer can rerun a recorded failure scenario without depending on external systems like remote APIs. Further, recorded data might be used for regression testing to verify that a modified program still works as expected [13].

### 7.4 Time Travel Debugging

Contrary to regular control-flow-oriented debuggers, omniscient [14], or *time travel* debuggers cannot only step forward but also backward in time. This is because they rely on recorded data rather than a currently running program. Once there is a way to record, store and replay debugging data as suggested before, time travel debugging is a possible next step. Software engineers can then manually navigate through recorded data and observe how individual system parts react to the stimuli.

## 8 CONCLUSION

We presented *operator log points* as a novel debugging utility for programs implemented using reactive programming in this paper. With *RxJS Debugging for vscode*, we demonstrated how operator log points replace manual print statements for RxJS-based programs. We developed the debugger using a user-centered design process facilitating usability inspection and validation methods, which allowed us to identify and resolve four major usability issues. In addition, we successfully verified that the proposed utility fulfills the requirement of readiness-to-hand, i.e., that it integrates seamlessly with software engineers' daily workflows and does not require additional learning or setup effort.

## REFERENCES


[1] 1990. *IEEE standard glossary of software engineering terminology.* IEEE. DOI:https://doi.org/10.1109/IEEESTD.1990.101064

[2] Manuel Alabor and Markus Stolze. 2020. Debugging of RxJS-based applications. In *Proceedings of the 7th ACM SIGPLAN international workshop on reactive and event-based languages and systems* (REBLS 2020), Association for Computing Machinery, 15–24. DOI:https://doi.org/10.1145/3427763.3428313

[3] Herman Banken, Erik Meijer, and Georgios Gousios. 2018. Debugging data flows in reactive programs. In *Proceedings of the 40th international conference on software engineering*, ACM, 752–763. DOI:https://doi.org/10.1145/3180155.3180156

[4] T. Boren and J. Ramey. 2000. Thinking aloud: Reconciling theory and practice. *IEEE Transactions on Professional Communication* 43, 3 (September 2000), 261–278. DOI:https://doi.org/10.1109/47.867942

[5] Kim Goodwin. 2009. *Designing for the digital age: How to create human-centered products and services.* Wiley Pub.

[6] Lucas Layman, Madeline Diep, Meiyappan Nagappan, Janice Singer, Robert Deline, and Gina Venolia. 2013. Debugging revisited: Toward understanding the debugging needs of contemporary software developers. In *2013 ACM / IEEE international symposium on empirical software engineering and measurement*, IEEE, 383–392. DOI:https://doi.org/10.1109/ESEM.2013.43

[7] Sean McDirmid. 2013. Usable live programming. In *Proceedings of the 2013 ACM international symposium on new ideas, new paradigms, and reflections on programming & software - onward! '13*, ACM Press, 53–62. DOI:https://doi.org/10.1145/2509578.2509585

[8] Jakob Nielsen. 1994. Usability inspection methods. In *Conference companion on human factors in computing systems*, 413–414.

[9] Jakob Nielsen. 1994. Estimating the number of subjects needed for a thinking aloud test. *International Journal of Human-Computer Studies* 41, 3 (September 1994), 385–397. DOI:https://doi.org/10.1006/ijhc.1994.1065

[10] Jakob Nielsen and Thomas K. Landauer. 1993. A mathematical model of the finding of usability problems. In *Proceedings of the SIGCHI conference on human factors in computing systems - CHI '93*, ACM Press, 206–213. DOI:https://doi.org/10.1145/169059.169166

[11] Mie Nørgaard and Kasper Hornbæk. 2006. What do usability evaluators do in practice?: An explorative study of think-aloud testing. In *Proceedings of the 6th ACM conference on designing interactive systems - DIS '06*, ACM Press, 209. DOI:https://doi.org/10.1145/1142405.1142439

[12] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering record and replay for deployability: Extended technical report. *arXiv:1705.05937 [cs]* (May 2017). Retrieved from http://arxiv.org/abs/1705.05937

[13] Ivan Perez and Henrik Nilsson. 2017. Testing and debugging functional reactive programming. *Proceedings of the ACM on Programming Languages* 1, ICFP (August 2017), 1–27. DOI:https://doi.org/10.1145/3110246

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

[15] 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 Press, 25–36. DOI:https://doi.org/10.1145/2577080.2577083

[16] Guido Salvaneschi and Mira Mezini. 2016. Debugging for reactive programming. In *Proceedings of the 38th international conference on software engineering - ICSE '16*, ACM Press, 796–807. DOI:https://doi.org/10.1145/2884781.2884815

[17] Cathleen Wharton, John Rieman, Lewis Clayton, and Peter Polson. 1994. *The cognitive walkthrough: A practitioner's guide*. Institute of Cognitive Science, University of Colorado.