Code Coverage Criteria for Asynchronous Programs

by

Mohammad Ganji

B.Sc., University of Tehran, 2020

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

© Mohammad Ganji 2023
SIMON FRASER UNIVERSITY
Spring 2023

Copyright in this work is held by the author. Please ensure that any reproduction or re-use is done in accordance with the relevant national copyright legislation.

Declaration of Committee

Name: **Mohammad Ganji**

Degree: **Master of Science (Computing Science)**

Thesis title: **Code Coverage Criteria for Asynchronous Programs**

Committee: **Chair:** Manolis Savva
Assistant Professor, Computing Science

Saba Alimadadi
Supervisor
Assistant Professor, Computing Science

Steven Ko
Committee Member
Associate Professor, Computing Science

William N. Sumner
Examiner
Associate Professor, Computing Science

Ethics Statement

The author, whose name appears on the title page of this work, has obtained, for the research described in this work, either:

a. human research ethics approval from the Simon Fraser University Office of Research Ethics

or

b. advance approval of the animal care protocol from the University Animal Care Committee of Simon Fraser University

or has conducted the research

c. as a co-investigator, collaborator, or research assistant in a research project approved in advance.

A copy of the approval letter has been filed with the Theses Office of the University Library at the time of submission of this thesis or project.

The original application for approval and letter of approval are filed with the relevant offices. Inquiries may be directed to those authorities.

Simon Fraser University Library
Burnaby, British Columbia, Canada

Update Spring 2016

Abstract

In recent years, asynchronous programming has gained significantly in popularity. Asynchronous software often exhibits complex and error-prone behaviors and should therefore be tested thoroughly. Code coverage has been the most popular metric to assess test suite quality. However, traditional code coverage criteria are not sufficient as a measure of test adequacy for asynchronous applications. In particular, they do not adequately reflect completion, interactions, and error handling of asynchronous operations. This research proposes novel test adequacy criteria for measuring: (i) eventual completion of asynchronous operations in terms of both successful and exceptional execution, (ii) registration of reactions for handling both possible outcomes, and (iii) execution of said reactions through tests. We present JScope, a code coverage tool for automatically measuring these criteria in JavaScript applications and implement it as an interactive plug-in for Visual Studio Code. An evaluation of JScope on 20 JavaScript applications shows that the proposed code coverage criteria can help improve assessment of test adequacy, complementing traditional criteria. Furthermore, an investigation of 15 real GitHub issues concerned with asynchrony demonstrates that the new criteria can help reveal faulty asynchronous behaviors that are untested and are deemed covered by traditional coverage criteria. We also report on a controlled experiment with 12 participants to investigate the usefulness of JScope in realistic settings, demonstrating that it is effective in improving programmers' ability to assess test adequacy and detect untested behavior, and that it can be helpful for debugging.

Keywords: coverage, dynamic analysis, asynchronous JavaScript

Dedication

To my loving mom and dad, whose excitement for my achievements exceeds my own, and without whom my research journey would have remained a distant dream.

Acknowledgements

I like to express my deepest gratitude to Dr. Saba Alimadadi, my supervisor, whose unwavering support and guidance have been invaluable throughout my career. I extend my appreciation to Dr. Frank Tip for his active collaboration and supervision. I am also grateful to Sadjad Tavakoli, my dear colleague and friend, for contributing to a more pleasant work environment in the lab.

Table of Contents

Declaration of Committee **ii**

Ethics Statement **iii**

Abstract **iv**

Dedication **v**

Acknowledgements **vi**

Table of Contents **vii**

List of Tables **x**

List of Figures **xi**

1 Introduction **1**

1.1 Asynchronous Programming 1

1.2 Testing Asynchronous Code 2

1.3 Asynchronous Coverage Criteria - An Overview 2

1.4 Contributions 3

1.5 Publications 4

1.6 Dissertation Outline 4

2 Background **5**

2.1 Code Coverage Criteria 5

2.2 Asynchronous Programming 5

2.3 Promises and async/await in JavaScript 6

3 Motivation and Challenges **9**

3.1 Unhandled Exceptions 9

3.1.1 Example 1 (Running Example) 10

3.2 Pending Asynchronous Operations 10

3.2.1 Example 2 11

4 Asynchronous Coverage Criteria **13**
4.1 Events and Traces 13
4.2 Coverage Criteria for Promise-Based Code 14
4.3 async/await 16
4.4 Feasibility of Asynchronous Coverage Criteria 17

5 Approach **18**
5.1 Instrumentation and Trace Collection 18
5.2 Measuring Asynchronous Coverage 19
5.3 Visualizing the Asynchronous Coverage 20
5.4 Implementation 21

6 Evaluation **23**
6.1 Asynchronous Coverage 23
6.1.1 Experimental Design and Procedure 23
6.1.2 Results and Discussion 23
6.2 Asynchronous Coverage and Test Effectiveness 26
6.2.1 Experimental Design and Procedure 26
6.2.2 Results and Discussion 27
6.3 Usefulness of JScope to Developers 30
6.3.1 Experimental Design and Procedure 31
6.3.2 Results and Discussion 32
6.4 Performance 34
6.5 Threats to Validity 35

7 Related Work **36**
7.1 Code Coverage Criteria 36
7.2 Program Analysis for JavaScript 37
7.3 Visualization 37
7.4 User Studies 38
7.5 Mutation Testing 38

8 Conclusion and Future Work **39**
8.1 Conclusion 39
8.2 Future Work 39

Bibliography **41**

Appendix A Supplamantary Data File: JScope Source Code **49**

Appendix B Supplamantary Data File: User Study Materials **50**

Appendix C Consent Form **51**

List of Tables

Table 4.1 Trace events for asynchronous operations. 13

Table 6.1 Summary of different coverage metrics reported by JScope and traditional coverage. 24

Table 6.2 correlation coefficients for asynchronous and traditional coverage criteria. 25

Table 6.3 Asynchrony-related JavaScript issues from Github. 26

Table 6.4 Tasks used in the user study. 32

Table 6.5 Performance overhead of JScope; The numbers show an average of five executions 34

List of Figures

Figure 1.1 High level overview of JScope 3

Figure 3.1 Implementation of `RepoService.remove`. The plus and minus signs indicate the changes in the commit related to the fix. 10

Figure 3.2 Implementation of async function `visibility` from Odoo's website builder. The plus and minus signs indicate the changes in the commit related to the fix. 11

Figure 5.1 JScope coverage results for CLA Assistant. The open editor shows `RepoService.remove` in repo.js. 21

Figure 6.1 Statement coverage vs JScope results, before and after the bug fix. Green marks beside line numbers show covered statements reported by Istanbul. Overlaying highlights are generated by JScope, where yellow and red indicate promises with inadequate async coverage. . 28

Figure 6.2 Simplified implementation of the function `buildProxyReq` in express-http-proxy before the fix. P1, P2, P3, and P4 show promises detected by JScope. 29

Chapter 1

Introduction

1.1 Asynchronous Programming

The importance of responsive and efficient programming has become paramount in today's digital landscape. With the continuous rise in demand for high-performing and interactive applications, developers must find ways to optimize their code and minimize its impact on user experience. Asynchronous programming is a solution that has gained traction in recent years to address these challenges [50, 38].

In contrast to synchronous programming, where code is executed line-by-line in a blocking fashion, asynchronous programming allows non-blocking and out-of-order execution of code. This means that the program can continue to process user input or execute other tasks while it awaits the completion of a long-running operation. This enables real-time user interactions, efficient client-server communications, and non-blocking I/O, all essential for modern web development.

Asynchronous programming is present in various programming environments such as mobile development and embedded systems [30, 25]. It is also implemented and used by many popular languages, such as Java, Python, and C# [8, 7, 6]. However, it is most recognized for its utility in JavaScript. JavaScript, the most widely used programming language according to GitHub's yearly survey [4], is single-threaded. Asynchronous execution of potentially long-running tasks is the reason that seamless user interactions and responsiveness of JavaScript applications is possible.

Traditionally, JavaScript has relied on callbacks and event-based programming to enable asynchrony. However, callbacks are notorious for creating code that is difficult to read and debug, often leading to what is commonly known as *callback hell* [11, 37, 47].

Promises [31, Section 27.2] and `async/await` [31, Section 15.6] have been introduced into ECMAScript in recent years as improved alternatives to callbacks for implementing asynchrony. These methods have become popular in writing asynchronous code, overtaking the use of error-prone callbacks. They have proven to be so effective that other languages like C++, Python, Java, Dart, and C# have added similar syntaxes to their languages

[5, 6, 7, 8, 12]. Despite the improved syntaxes, inherent non-determinism and non-sequential execution of asynchronous code impairs developers' comprehension and debugging abilities. Understanding the flow of asynchronous execution and identifying and fixing faults caused by JavaScript promises and `async/await` remain still challenging endeavors for developers regardless of their popularity [51, 18, 77, 83].

1.2 Testing Asynchronous Code

Software testing is a crucial part of the software development process, and developers often rely on an application's tests to identify faults and verify its behavior. They often use code coverage criteria such as statement and function coverage to assess the adequacy of their tests and to identify and address shortcomings in existing tests [85]. Although a high coverage does not guarantee a better bug finding capability, it is a necessary first step, as tests are not able to identify bugs in unexercised code [43].

Testing asynchronous code poses unique challenges due to its non-deterministic nature and out-of-order execution. Programs may involve multiple asynchronous operations executed concurrently, or throw exceptions during the execution of asynchronous code, making it difficult to ensure that the application behaves correctly. Traditional coverage criteria are unable to examine various scenarios of exercising asynchronous code in terms of eventual completion of asynchronous operations, their interactions, and their error handling. Further, developers may fail to devise proper reactions for possible outcomes of asynchronous events, leading to missing code that cannot be discovered by traditional criteria. Despite the importance of testing asynchronous programs and the severity of the issues that occur in such programs, there are currently no code coverage criteria that target the adequacy of tests with regard to exploring scenarios that occur in asynchronous code.

1.3 Asynchronous Coverage Criteria - An Overview

In this dissertation, we introduce a new set of coverage criteria for assessing the adequacy of tests in exercising the asynchronous behavior of JavaScript applications. Our proposed criteria quantify the adequacy of tests in covering eventual successful or exceptional completion of asynchronous operations, registration of respective reactions for continuing the flow of execution based on the operations' outcome, and sufficient execution of chains of reactions by the application's tests. These criteria target the semantics of Promises and `async/await`, as the most popular mechanisms for supporting asynchrony in JavaScript. Our metrics are meant to be complementary to existing coverage metrics, such as statement and function coverage.

Figure 1.1 Our approach for automatic calculation of asynchronous coverage starts by a non-intrusive instrumentation of the code. Analysis of the collected data allows our algorithm to calculate three types of coverage criteria for a given application, namely *settlement*

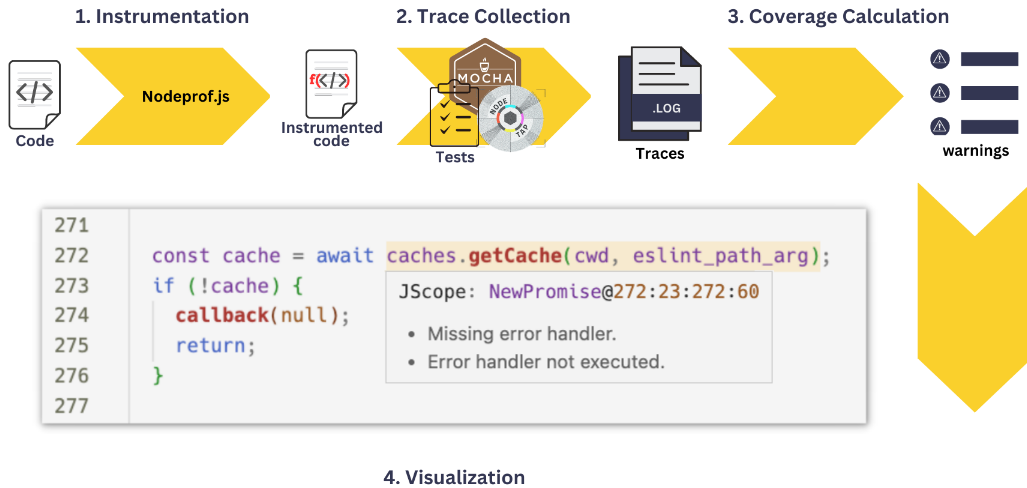

Figure 1.1: High level overview of JScope

coverage, *registration coverage*, and *execution coverage*. Our approach presents the results as a textual report, as well as an interactive visualization embedded in a development environment. We implement our approach in an open-source plugin for Visual Studio Code named JScope.

1.4 Contributions

Our work makes the following key contributions.

- We introduce a novel set of coverage criteria for assessing test adequacy of asynchronous code. These criteria quantify the degree to which key scenarios are exercised in asynchronous code, including successful/exceptional completion of asynchronous operations, and registration and execution of reactions.

- We propose an approach for measuring said coverage criteria that intercepts and analyzes asynchronous interactions. In addition to the coverage report, our approach visualizes the coverage by overlaying a set of visual cues and warnings on the code to guide programmers towards the tests' insufficiencies.

- We implement our approach in an open-source interactive VS Code extension named JScope, which we will make publicly available after the anonymous review process.

- We report on three different experiments to evaluate our approach. The results from comparing traditional and asynchronous coverage reports from 20 Node.js applica-

tions show that our approach is effective in identifying test insufficiencies with respect to asynchronous code and can complement traditional coverage metrics. We also find that JScope's warnings can lead to finding real bugs in parts of code despite being completely covered by traditional coverage criteria. Finally, we show through a user study that JScope improves the accuracy of programmers in their testing and debugging activities in realistic setting by 28% on average.

1.5 Publications

This dissertation also includes our paper currently under submission at ESEC/FSE conference. The paper below is the result of contributions of our collaborator and advisor from North Eastern University Frank Tip, my supervisor Saba Alimadadi, and myself.

- ESEC/FSE 2023 (Under Submission) – Code Coverage Criteria for Asynchronous Programs. Mohammad Ganji, Saba Alimadadi, Frank Tip

1.6 Dissertation Outline

The contents of this dissertation are organized as follows: Chapter 2 presents a background on code coverage criteria, asynchrnous programming, and asynchrony in JavaScript. Chapter 3 contains two real problems as motivational examples for this research. Chapter 4 and 5 describe our proposed coverage criteria for asynchronous code, and how we use JScope to measure these criteria. Chapter 6 details our evaluation results. Chapter 7 discusses related work and the scope of our study. Finally, chapter 8 concludes our work

Chapter 2

Background

2.1 Code Coverage Criteria

Code coverage criteria are commonly used metrics for evaluating the quality of the test suites. They measure the extent to which the source code program has been executed during testing.

There are several methods for measuring code coverage, including but not limited to statement coverage, branch coverage, and function coverage. Statement coverage measures the number of lines of code execrised by the tests, while branch coverage measures the number of branches (e.g., if/else statements) taken during testing. Function coverage measures the number of functions or methods executed during testing.

Code coverage criteria can provide insight into the thoroughness of testing. They help identify areas of the code that have not been tested, allowing developers to focus their testing efforts on these areas and improve the overall quality of their code.

Despite its widespread use, there are challenges associated with code coverage criteria. For instance, achieving high code coverage does not necessarily guarantee the absence of bugs in the software, and low code coverage does not necessarily indicate a high number of bugs. Additionally, code coverage can be difficult to measure in some types of software, such as multi-threaded, concurrent, and asynchronous systems.

2.2 Asynchronous Programming

Asynchronous programming is a paradigm that enables multiple tasks to be executed in parallel, without blocking the execution of other tasks. It allows tasks to be executed independently, and uses callbacks or other forms of event-based notifications to make their results available. This allows applications to continue processing other tasks while waiting for the result of an asynchronous operation.

As an example, consider a web page that needs to fetch data from a server. In a synchronous programming model, the page would freeze until the data has been received from

the server. However, in an asynchronous programming model, the data fetch operation would be executed asynchronously, allowing the page to continue processing other tasks (such as receiving user input, or updating the graphical user interface) while waiting for the data to arrive. Once the data has been received, a callback function would be invoked to handle the data and update the page.

Asynchronous programming is commonly used in JavaScript to handle user interactions, network requests, and other I/O-bound operations. Its inherent asynchrony, has made JavaScript a widely used language for developing dynamic and responsive applications.

To handle the asynchronous nature of the language, developers often turn to two commonly used techniques: promises and `async/await`. These techniques play a crucial role in effective JavaScript programming and are indispensable tools for building performant and scalable applications. The following section includes more detail about the implementation and utilities of Promises and `async/await`.

2.3 Promises and async/await in JavaScript

This section provides a brief review of promises [31, Section 27.2], and `async/await` [31, Section 15.6], two features for asynchronous programming that were added to JavaScript in recent years.

Creating promises A *promise* represents the value of an asynchronous computation, and is in one of three states: pending, fulfilled, or rejected. The state of a promise can change at most once: from pending to fulfilled, or from pending to rejected. We will say that a promise is *settled* if its state is fulfilled or rejected. Promises are created by invoking the `Promise` constructor, and are initially in the pending state. Promises come equipped with two methods, `resolve` and `reject`, for fulfilling or rejecting the promise with a particular value, respectively. For example, the following code assigns a promise to a variable `p1` that is either fulfilled with the value `"hello"` or rejected with an `Error` object.

```
1 const p1 = new Promise( (resolve, reject ) => {
2   if (Math.random() > 0.5) { resolve("hello"); }
3   else { reject (new Error('oops')); }
4 });
```

Promises can also be constructed using the functions `Promise. resolve` and `Promise .reject`. Each of these functions takes a single argument, i.e., the value that the promise should be fulfilled or rejected with. The following example creates a promise that is fulfilled with the value `3`:

```
5 const p2 = Promise.resolve(3);
```

Synchronization functions such as `Promise.all` and `Promise.race` are other ways to create promises. They wait on a set of promises to be settled in any order, returning a single

promise upon invocation. We will explain these functions in more detail near the end of this section.

Registering reactions on promises The `then` and `catch` methods enable programmers to register *reactions* on promises, i.e., functions that are executed asynchronously when a promise is fulfilled or rejected. The value returned by a reaction is wrapped in another promise, thus enabling programmers to *chain* asynchronous computations and propagate errors. For example, the following code fragment shows the creation of a promise chain that starts with `p1`:

```
6 p1.then( function f1(v) { console.log(v + " world"); } )
7 .catch( f3(err) { console.log("error occurred: " + err); }
```

If `p1` was fulfilled with the value `"hello"`, the reaction that is registered by calling `then` on `p1` on line 6 concatenates that value with another string `" world"` and prints it to the console, Line 7 registers a reject reaction on the promise that is created by calling `then` on line 6. It prints an error message if any of the previous promises in the chain is rejected. Therefore, the above code snippet will either print `"hello world"` or `"error occurred: oops"`.

Linking promises If the `resolve` associated with the `Promise` constructor is invoked with an argument that is a promise, or when a reaction that is registered by calling `then` or `catch` returns a value that is a promise (called p), the former promise p' becomes *linked* with p. If p is resolved with a value v, then p' is resolved with v as well, and if p is rejected with a value e, then so is p', and if p remains pending, so does p'. The following example:

```
8 const p3 = Promise.resolve("hello")
9 const p4 = Promise.resolve("there")
10 p3.then( () => p4 ) // establish link with p4
11   .then( (v) => console.log(v) ) // prints "there"
```

creates promises `p3` and `p4`. Given that `p3` is fulfilled, its reaction is executed and returns `p4`, so `p4` and the promise returned by `p3.then()` on line 10 become linked. Since `p4` resolves to `"there"`, the promise returned by `p3.then()` on line 10 resolves to `"there"` as well, causing the reaction registered on line 11 to execute and print this value.

Synchronization Several functions are provided for synchronizing asynchronous computations. `Promise.all` provides a mechanism to wait for a set of promises to be settled in any order; it takes as input an array of promises $p_1, \cdots, p_n$ and returns a promise p' that is resolved with an array $[v_1, \cdots, v_n]$, if each p_i is fulfilled with a value v_i. If any p_i is rejected, then p' is immediately rejected with the same value, regardless of what happens to other promises p_j $(j \neq i)$. `Promise.any`, `Promise.race` and `Promise.allSettled` are other synchronization functions with similar structure but different behavior.

Similarly, `Promise.allSettled` takes an array of promises $p_1, \cdots, p_n$ as an argument and fulfills when all of the $p_1, \cdots, p_n$ settle, i.e. fulfill or reject. Its value will be an array containing values or errors of the settled $p_1, \cdots, p_n$.

`Promise.any` takes an array of promises $p_1, \cdots, p_n$ as an argument and fulfills as soon as any of the p_i fulfills, with the value that p_i was fulfilled with, or it rejects if all of the p_i are rejected, with an array of rejection reasons. Finally, `Promise.race` takes an array of promises $p_1, \cdots, p_n$ as an argument and returns a promise that fulfills or rejects as soon as one of these p_i fulfills or rejects, with the same value that p_i was fulfilled or rejected with.

`async`/`await` JavaScript's `async`/`await` feature provides a syntactic enhancement on top of promises. A function declared as `async` returns a promise that is fulfilled with the function's return value. Inside an `async` function, `await`-expressions may be used to wait for a promise to be settled. If an awaited promise p is fulfilled with value v, then an expression `await` p evaluates to v; if it is rejected with a value `err`, `err` is thrown as an exception that can be caught using `try/catch` . Consider the following code fragment:

```
12 async function f(){
13   try {
14     let v = await p;
15     /* 1 */
16   } catch(e){ /* 2 */ }
```

`p` is an expression that evaluates to a promise. The execution of the code fragment `/* 1 */` depends on fulfillment of `p`. So one may think of code fragment `/* 1 */` as a fulfill reaction associated with `p`, and similarly the code fragment `/* 2 */` as a reject reaction of `p`.

Chapter 3

Motivation and Challenges

We use this section to elaborate on some of the challenges in identifying parts of asynchronous code that despite being covered by the tests, are not tested "sufficiently" and thus may include bugs. We use real bug reports shown in Figure 3.1 and 3.2 to illustrate the challenging nature of locating bugs in asynchronous code. These challenges are intensified by developers' confidence in correctness of the code, when they have tests that exercise that piece of code. While existing coverage metrics may show full coverage of these code segments, these metrics are unable to examine the execution of scenarios specific to asynchronous code. As such, they will not identify inadequacies in testing asynchronous code.

3.1 Unhandled Exceptions

An asynchronous operation can eventually terminate successfully, or it may fail. While a successful completion is usually the desired outcome, the failures or exceptional cases should be tested thoroughly to verify the applications' robustness and error recovery. Exceptional scenarios are often not thoroughly tested by many applications, which can lead to bugs and unexpected behaviors during execution should an exception occur [18]. These operations are even equipped with extended exception handling in JavaScript. For instance, `await` expressions can be surrounded by `try/catch` for handling a failed completion of the async function. However, these rules are not enforced, and their proper usage or absence is not examined by current testing and coverage techniques. As such, there are many applications that do not have adequate exception handling in place and do not sufficiently test exceptional and failure cases in their asynchronous code. In the following example, we discuss how failure to properly handle the rejection of an asynchronous operation results in the whole system crashing. The bug occurs despite code coverage reports showing that the related part of the code was in fact covered.

```
17       remove: async (req) => {
18         const dbRepo = await repo.remove(req.args)
19         if (dbRepo && dbRepo.gist) {
20           try {
21 -           webhook.remove(req)
22 +           await webhook.remove(req)
23           } catch (error) { // handle the error } }
24         return dbRepo
25       }
```

Figure 3.1: Implementation of `RepoService.remove`. The plus and minus signs indicate the changes in the commit related to the fix.

3.1.1 Example 1 (Running Example)

CLA Assistant is a web service that streamlines the process of signing Contributor License Agreements (CLAs). [1] This project is built by SAP SE [2] developers and has more than 1000 stars. The code in Figure 3.1 shows the async function `RepoService.remove`, which is responsible for removing a repository from CLA Assistant (using `repo.remove` on line 18) and removing all of its webhooks (`webhook.remove`, line 21).

To handle unexpected errors, the call to `webhook.remove` is placed inside a `try/catch` (lines 20–23), which assures programmers of the robustness of this code segment. Programmers' confidence in verifying this code segment is reinforced by covering and exercising all its statements through the tests. Despite this, a bug was reported where an unhandled rejection in this method resulted in the hard shutdown of the service. Further investigation showed that while there is a `try/catch` in place to handle errors in removing webhooks, the developers failed to `await` the asynchronous `webhook.remove` method. Without an `await` statement, the program does not wait for the `async` function to complete its execution. In other words, The execution of `RepoService.remove` could end before `webhook.remove` rejected with an error asynchronously. The exception was thrown outside the scope of `RepoService.remove` and thus the `catch` clause could not have caught it, causing an unhandled rejection.

The fix adds an `await` before `webhook.remove` to make `RepoService.remove` wait until its completion (line 22).

3.2 Pending Asynchronous Operations

An asynchronous operation remains pending until it is "settled" successfully or through a failure, i.e., `resolved` or `rejected`. It is common to chain asynchronous operations to

[1] https://github.com/cla-assistant/cla-assistant

[2] https://sap.com

```
26  async function visibility (preview, widgetValue, params) {
27  -   await new Promise(resolve => {
28  +    await new Promise((resolve, reject) => {
29        this.trigger_up('action_demand', {
30          onSuccess: () => resolve(),
31  +               onFailure: () => reject(), // ADDED IN FIX.
32        }); });
33      this.trigger_up(' option_visibility_update ', {show});
34  }
```

Figure 3.2: Implementation of async function `visibility` from Odoo's website builder. The plus and minus signs indicate the changes in the commit related to the fix.

impose an ordering on their execution. In such cases, successful and exceptional completion of an asynchronous operation each trigger respective reactions, and the execution of the program continues. It is typically expected for all asynchronous operations to "settle." In cases where this does not happen, the appropriate reactions are not invoked, and the chain of execution is interrupted. The following example demonstrates a real bug where a pending asynchronous operation causes the program to freeze in a loading state, preventing the users from futher interactions with the system.

3.2.1 Example 2

Figure 3.2 shows changes related to a bug fix from Odoo, a suite of web based open source business apps, including Marketing, eCommerce, and Website Builder apps. [3] It has nearly 25K stars on GitHub and is forked over 16K times. The async function `visibility` is responsible for updating the visibility of a field inside a widget in the sidebar menu of the website builder. The execution of this method depends on the completion of a promise that notifies the parent widget to toggle its visibility (lines 27–32). The notification occurs through `trigger_up` on lines 29–32. The `trigger_up` method accepts `onSuccess` or `onFailure` callback arguments, and executes one of them according to the outcome of the event. Here, the `onSuccess` callback fulfills the promise by calling its `resolve` method (line 30). The `visibility` method then makes the field on the widget visible, allowing the user to interact with the editor (line 33).

The bug report indicates a scenario where a widget just gets stuck, with a spinner spinning forever. The issue occurs when the event fired by `trigger_up` ends with an exception. Hence, the `onSuccess` callback is not called to resolve the promise. As there is no reject reaction devised for an unsuccessful completion of the promise, it never settles. As the execution of the remaining part of the `visibility` method depends on the settlement of the

[3] https://github.com/odoo/odoo/pull/87123

promise, the pending promise prevents the execution of line 33. This causes the widget to get stuck in a loading state, making the application dysfunctional.

The fix `reject`s the promise upon failure of `trigger_up` (line 31), which settles the promise and allows the execution to continue.

Chapter 4

Asynchronous Coverage Criteria

Our goal is to define coverage criteria that reflect to what extent the possible asynchronous behaviors of an application are exercised. We introduce these criteria in terms of events in execution traces that pertain to the use of asynchronous features. We define three coverage criteria: *settlement coverage*, *reaction registration coverage*, and *reaction execution coverage*. These criteria target the completion of all asynchronous operations (successful and exceptional), registration of reactions for both outcomes of the operations, and the execution of said reactions, respectively. We begin by defining coverage notions for JavaScript applications that use promises, and will then explain informally how these notions extend to `async`/`await`. Finally, we will discuss the feasibility of these criteria.

4.1 Events and Traces

Table 4.1 defines the promise-related events that may occur during execution. Here, we assume that each promise that is created at run time has a unique *promise identifier (pid)*. Further, let $\mathcal{S}$ define the set of source locations where promises are created, including: (i) calls to the `Promise` constructor, (ii) calls to `Promise.resolve()` and `Promise.reject()` , (iii) calls to `then` , `catch` , and `finally` on promise objects, (iv) calls to `Promise.all`,

$Create(pid,\ loc)$	creation of promise *pid* at location *loc*
$Fulfilled(pid,\ \mathrm{loc})$	promise *pid* is fulfilled at location *loc*
$Rejected(pid,\ \mathrm{loc})$	promise *pid* is rejected at location *loc*
$Link(pid,\ pid',\ loc)$	promise *pid* becomes linked to promise *pid′* at location *loc*
$Reg_{fulfill}(pid,\ f,\ loc,\ [pid'])$	Register fulfill reaction f on promise *pid* at location *loc*, which may chain it to promise with id *pid′*
$Reg_{reject}(pid,\ f,\ loc,\ [pid'])$	Register reject reaction f on promise *pid* at location *loc*, which may chain it to promise with id *pid′*
$Exec_{fulfill}(pid,\ f,\ \mathrm{loc})$	execute fulfill reaction f on promise *pid* at location *loc*
$Exec_{reject}(pid,\ f,\ \mathrm{loc})$	execute reject reaction f on promise *pid* at location *loc*

Table 4.1: Trace events for asynchronous operations.

Promise.race, Promise.any, and Promise.allSettled, and (v) the end of execution of an async function (either normal or exceptional exit).

Create events occur when any of situations (i)-(v) occurs. *Link* events occur when the resolve function associated with a call to the Promise constructor or Promise.resolve are invoked with an argument that is itself a promise. A *Link* event is always immediately preceded by a *Create* event.

Fulfilled events occur when the resolve function associated with a Promise is invoked with an argument that is not a promise, and when a reaction returns a value that is not a promise. Likewise, *Rejected* events occur when the reject function associated with a Promise is invoked, and when a reaction throws an exception. Note that the trace only records *Fulfilled* and *Rejected* events for promises that are explicitly fulfilled or rejected (i.e., it does not contain such events for linked promises that are resolved or rejected due to being linked to another promise).

$Reg_{fulfill}$ events happen when then is used to register a fulfill-reaction on a promise, and Reg_{reject} events happen when catch or the second argument of then is used to register a reject-reaction. Lastly, $Exec_{fulfill}$ and $Exec_{reject}$ events happen when a previously registered fulfill-reaction or reject-reaction starts executing, respectively.

As an example, executing lines 1-7 in the code snippets given in Section 2.3 gives rise to the following trace if p1 was fulfilled.

```
35 Create(pid_p1, L1:L4)
36 Fulfilled(pid_p1, L2:L2)
37 Create(pid_p2, L5:L5)
38 Fulfilled(pid_p2, L5:L5) // promise. resolve is instantly fulfilled .
39 Create(pid_then, L6:L6) // call to then returns a promise
40 Reg_fulfill(pid_p1, f1, L6:L6, pid_then)
41 Create(pid_catch, L7:L7) // call to catch returns a promise
42 Reg_reject(pid_then, f2, L7:L7, pid_catch)
43 Exec_fulfill(pid_p1, f1, L6:L6)
44 Fulfilled(pid_then, L6:L6)
```

Calls to then and catch return promises. The $Reg_{fulfill}$ connects then to p1 and Reg_{reject} connects catch to then, creating a promise chain. Also, f2 is never executed because none of the other promises in the chain are rejected.

4.2 Coverage Criteria for Promise-Based Code

In the definitions that follow, we will use pid, pid', $\cdots$ to represent promise identifiers. Moreover, $f, f', \cdots$ will denote functions, and $loc, loc', \cdots$ will denote source locations. Definition 1 defines a trace as a sequence of trace events (see Table 4.1). We will use $\tau, \tau', \cdots$ to refer to execution traces.

Definition 1 (trace)**.** *A* trace *is an ordered sequence of trace events as specified in Table 4.1.*

For each promise *pid* that occurs in a trace τ, there is a unique trace element *Create*(*pid*, *loc*) corresponding to its creation. We define *loc*(*pid*) as the location *loc* that is referenced in this trace element. The first coverage criterion we define is *promise settlement coverage*. This measures the fraction of promises defined by an application that are settled (i.e., fulfilled or rejected). Here, we consider a promise *pid* originating from location *loc* to be fully covered if the trace contains both *Fulfilled* and *Rejected* events for *pid*, which requires location *loc* to be executed at least twice. Moreover, when a *Fulfilled* or *Rejected* event is observed for a promise *pid*, all promises directly or indirectly linked with *pid* are settled as well. To capture this, we first define $\mathcal{L}(pid,\ \tau)$ to denote the set of promises linked to *pid* in trace τ.

Definition 2 (linked promises)**.** *Let pid be the promise identifier for a promise. Then, the set of promises linked to pid in a trace* τ*, denoted by* $\mathcal{L}(pid,\ \tau)$*, is defined as:*

$$\mathcal{L}(pid,\ \tau) = \{\ pid' \mid\ pid' = pid\ \ or \\ \exists loc : Link(pid,\ pid',\ loc) \in \tau, pid' \in \mathcal{L}(pid,\ \tau)\ \}$$

Note that *pid* itself is also an element of $\mathcal{L}(pid,\ \tau)$.

Using Definition 2, we now define the notion of promise settlement coverage as stated in Definition 3. Informally, the definition computes the number of locations loc' of promises pid' that are linked to a promise *pid* for which a *Fulfilled* or a *Rejected* event occurs in the trace τ. It then divides the sum of these by $2 * |\mathcal{S}|$, where $\mathcal{S}$ is the number of locations where a promise is created.

Definition 3 (promise settlement coverage)**.** *Let program* $\mathcal{P}$ *create promises at locations in* $\mathcal{S}$*, and let* τ *be the trace for an execution of* $\mathcal{P}$*. We define the* promise settlement coverage *of* τ *as:*

$$\frac{\begin{array}{c}|\{\ loc' \mid Fulfilled(pid, loc) \in \tau,\ pid' \in \mathcal{L}(pid,\ \tau),\ loc' = loc(pid')\ \}| + \\ |\{\ loc' \mid Rejected(pid, loc),\ pid' \in \mathcal{L}(pid,\ \tau),\ loc' = loc(pid')\ \}|\end{array}}{2 * |\mathcal{S}|}$$

Next, Definition 5 measures the percentage of promises on which reactions are registered. Here, we consider a promise fully covered if both a fulfill reaction and a reject reaction are registered on it. Note that the rejection of a promise p may be handled at the end of a promise chain that includes p. To make this precise, Definition 4 defines the set of dependent promises *pid* that occur at the end of a chain of fulfill-reactions that starts at *pid*, i.e., $pid \rightsquigarrow pid'$.

Definition 4 (dependent promises)**.** *Let program* $\mathcal{P}$ *create promises at locations in* $\mathcal{S}$*, and let* τ *be the trace for an execution of* $\mathcal{P}$*. Then:*

$$pid \rightsquigarrow pid'\ \textbf{\textit{if}} \begin{cases} pid \equiv pid'\ or \\ pid \rightsquigarrow pid''\ \textbf{\textit{and}}\ Reg_{fulfill}(pid'', loc, f, pid') \end{cases}$$

Using Definition 4, Definition 5 below computes reaction registration coverage through the following steps: (i) compute the number of locations loc' where a $Reg_{fulfill}$ event occurs on a promise pid for which a *Create* event occurs in the trace, (ii) compute the number of locations loc' where a Reg_{reject} event occurs on a promise pid', where $pid \rightsquigarrow pid'$, and where a *Create* event for pid occurs in the trace, and (iii) compute the sum of these, and divide it by $2 * |\mathcal{S}|$.

Definition 5 (reaction registration coverage)**.** *Let program $\mathcal{P}$ create promises at locations in $\mathcal{S}$, and let τ be the trace for an execution of $\mathcal{P}$. We define the* reaction registration coverage *of τ as:*

$$\frac{|\{\, loc' \mid Create(pid, loc) \in \tau,\ Reg_{fulfill}(pid, f, loc', pid') \in \tau \,\}| + |\{\, loc' \mid Create(pid, loc) \in \tau, pid \rightsquigarrow pid',\ Reg_{reject}(pid', f, loc', pid'') \in \tau \,\}|}{2 * |\mathcal{S}|}$$

Lastly, we define the notion of *reaction execution coverage*, measuring the percentage of promises with executed reactions. This is expressed by Definition 6 below, which is similar to Definition 5, except that it checks for the presence of $Exec_{fulfill}$ and $Exec_{reject}$ events in the trace instead of $Reg_{fulfill}$ and Reg_{reject} events. Achieving full reaction execution coverage for a promise created at loc requires that loc is executed at least twice.

Definition 6 (reaction execution coverage)**.** *Let program $\mathcal{P}$ create promise at locations in $\mathcal{S}$, and let τ be the trace for an execution of $\mathcal{P}$. We define the* reaction execution coverage *of τ as:*

$$\frac{|\{\, loc' \mid Create(pid, loc) \in \tau,\ Exec_{fulfill}(pid, f, loc') \in \tau \,\}| + |\{\, loc' \mid Create(pid, loc) \in \tau, pid \rightsquigarrow pid',\ Exec_{reject}(pid', f, loc') \in \tau \,\}|}{2 * |\mathcal{S}|}$$

4.3 async/await

The semantics of JavaScript's `async`/`await` is defined in terms of promises, and provides more convenient syntax that is highly similar to that of sequential code. Our approach is to generate the events of Table 4.1 as discussed below.

An `async` function always returns a promise, thus upon calls to `async` functions a *Create* event is included in the trace. When an `async` function returns a value that is not a promise, a *Fulfilled* event is included in the trace to reflect its fulfillment. When an `async` function returns a value that is a promise, a *Link* event is emitted, and a *Rejected* event is emitted if an `async` function throws an exception that is not caught within its body.

For registration and execution of reactions, consider lines 12-16 in the code snippets from Section 2. Upon execution of this code, the trace will contain a $Reg_{fulfill}(pid_p$, await, L14:L14)

event at line 14. A $Reg_{reject}(pid_p$, try/catch, L13:L16) event is also added for `p`, as **`await`** `p` is located inside a `try/catch` . If the execution of the async function in the example above continues to `/* 1 */`, an $Exec_{fulfill}(pid_p$, await, L14:L14) event is included in the trace. Also, if `p` is rejected and `/* 2 */` is executed, the trace will contain a $Exec_{reject}(pid_p$, try/catch, L13:L16) event. Assuming these elements in a trace, the same coverage definitions apply as before.

4.4 Feasibility of Asynchronous Coverage Criteria

The proposed coverage criteria for asynchronous programs are similar to traditional coverage criteria in the sense that 100% coverage, while desirable, is not always attainable. For example, in a conditional statement **`if`** E **`then`** S_1 **`else`** S_2, if the condition E always evaluates to **`true`**, then the **`else`**-branch and all the statements in S_2 are unreachable, and branch coverage and statement coverage will be less than 100%. Analogously, in a code fragment p`.`**`then`**`(`f`)`, the promise created by the call to **`then`** will remain pending if p is never fulfilled causing promise settlement coverage to remain less than 100%, and reaction registration coverage and reaction execution coverage may remain below 100% for similar reasons. Note that the use of `Promise.resolve` and `Promise.reject` may give rise to promises that are always resolved [1] or rejected.

Promise chains also undermine the feasibility to achieving 100% registration coverage. Since chain operations (i.e. `then` and `catch`) return a promise themselves, there will always remain a promise at the end of any promise chain without any reactions registered to it. Further, it is generally accepted to only include error handling operations inside the callback of a `catch` function without any other logic. As a result, it may be impossible in some programs to write a test scenario that rejects the promises returned by `catch` . In 5.2 we discuss heuristics defined to remedy such cases in our implementation.

[1] Note, however, that `Promise.resolve` does not necessarily produce a promise that is always resolved: if the argument passed to `Promise.resolve` is itself a promise, then that promise becomes linked with the newly created promise (the linked promise may remain pending or be rejected). Similar scenarios arise for **`async`** functions.

Chapter 5

Approach

In this chapter, we describe our approach and our tool, JScope, for automatically measuring and visualizing asynchronous coverage criteria as defined in Section 4.2. We will use the term "async coverage" to refer to the results of settlement, reaction registration, and reaction execution coverage combined, as JScope calculates and reports them collectively. Our approach relies on the instrumentation of asynchronous behaviors of a JavaScript application on the fly. JScope executes the instrumented code through the application's test suite to collect execution traces. Next, it utilizes the traces to locate promises, their reactions, and relations between them such as chains as means to calculate async coverage. Finally, JScope presents the results and relevant warnings in terms of a textual report and an interactive visualization. JScope is embedded within the development environment of Visual Studio Code [1] and allows user interactions to discover more information about individual promises and relationships between them.

5.1 Instrumentation and Trace Collection

To automatically collect trace events described in Table 4.1 for a program, we instrument the behavior of JavaScript promises and `async` functions on the fly. Executing the instrumented code through running the program's test suite, we obtain a trace of events created as discussed in Section 4.1.

In Section 4.1 we observed an example of how we collect traces for promise chains. To furhter explain how we translate `await` statements into the traces defined in Table 4.1, let's consider our first motivating example from Figure 3.1. Initially, the application's test suite contains a test case T1 for `RepoService.remove`. T1 tests a scenario where a repository and its webhook are successfully removed. Figure 5.1 shows the collected trace from running T1 for the code fragment 3.1 after the fix.

```
45    Create(p1, L18:L18) // repo.remove is an async function and returns a promise
```

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

```
46      Reg_fulfill(p1, await, L18:L18)
47      Fulfilled(p1, L18:L18) // the async function repo.remove executes successfully
48      Exec_fulfill(p1, await, L18:L18) // Code beyond await is executed
49      Create(p2, L22:L22) // webhook.remove is an async function and returns a promise
50      Reg_fulfill(p2, await, L22:L22)
51      Reg_reject(p2, try/catch, L20:L23)
52      Fulfilled(p2, L22:L22) // webhook.remove ends successfully
53      Exec_fulfill(p2, await, L22:L22) // Code beyond the second await is executed
```

Trace Events related from execution of RepoService.remove through T1 in CLA Assistant.

The fix also adds a new test case T2 to the test suite to check for a case where the repository is removed, but call to `webhook.remove` fails with a rejected promise. Traces collected from running T2 are shown in Listing 5.1.

```
54      Create(p3, L18:L18)
55      Reg_fulfill(p3, await, L18:L18)
56      Fulfilled(p3, L18:L18)
57      Exec_fulfill(p3, await, L18:L18)
58      Create(p4, L22:L22)
59      Reg_fulfill(p4, await, L22:L22)
60      Reg_reject(p4, try/catch, L20:L23)
61      Rejected(p4, L22:L22) // webhook.remove is rejected in T2.
62      Exec_reject(p4, try/catch, L20:L23) // so the catch block is executed to handle the exception.
```

Trace Events related from execution of RepoService.remove through T2 in CLA Assistant.

As it appears in these examples, an `await` behind a promise is translated as a $Reg_{fulfill}$, similar to using a `then` . Similarly, a `try/catch` block is an alternative to using `catch` for handling exceptions. As such, it is translated to a Reg_{reject}.

5.2 Measuring Asynchronous Coverage

Initially, we analyze collected trace data to identify individual promises throughout the test suite. As promises can only be settled once, at least two test cases are required to achieve full async coverage for a promise. As such, we uniquely identify a promise based on its static creation location in the code. Multiple *Create* events with the same location across several test case executions in a test suite will be considered as the same promise. To improve identification of promises, we consider the call-sites of async functions and regular functions that return a promise as their identifying location, elaborated more in Section 5.4. We then integrate different execution paths corresponding to the same promise to locate its various settlements, registered reactions and exception handling mechanisms and execution of such reactions.

Next, we detect relations between promises such as promise chains and linked promises. By definition, a reject reaction at the end of a chain is capable of catching all exceptions caused by any promise in that chain. In order to have a more precise representation of

sufficient error handling, our algorithm propagates a reject reaction in a chain to all of its ancestor promises. Additionally, for promises returned by `catch` , we only require *Fulfilled* event, and the rest are considered covered. This implies that registering reactions for `catch` is optional, as ending chains with a `catch` is a generally accepted way of using promises. While there is no obligation for `catch` to be at the end of a chain, we apply this heuristic to make asynchronous coverage metrics more consistent with its usages in realistic scenarios. Similarly, to avoid unresolvable missing coverage warnings, $Reg_{fulfill}$ events are optional for `then` . Note that without these heuristics achieving 100% async coverage would be impossible, as there will always be one promise without any handlers at the end of any chain. It is also possible for a promise to be linked to another promise, meaning its eventual state will be entirely dependent on the fate of the other promise. Our algorithm also detects promise links by locating where a promise $p1$ is resolved with another promise $p2$, and applies all *Fulfilled* and *Rejected* events of $p2$ to $p1$ as well. Note that these heuristics are specific to the implementation of promises in JavaScript and are not behaviors associated with asynchrony. Features such as chaining or linking may not exist in a different language with asynchronous features. As such, we did not include these heuristics in the definition of our coverage criteria. Finally, we (1) calculate and visualize the async coverage for the application by combining async coverage from all promises, and (2) report a list of warnings for all promises' missing reactions.

For instance, the promise `p1` in trace 4.1 has four different trace events associated with it: *Fulfilled*, $Reg_{fulfill}$, Reg_{reject}, and $Exec_{fulfill}$. Although $Reg_{reject}(pid_{p2}$, f2, L7:L7, $pid_{p3})$ is not directly registered to `p1`, we infer the `p1`⇝`p2`⇝`p3` chain and assign the reject reaction to all related promises including `p1`. The execution covers only half of the possible settlements for `p1` resulting in a partial async coverage. A new test case rejecting `p1` is required for full async coverage.

Consider the code from Figure 3.1 as another example. Executing the full test suite combines the traces from T1 and T2, merging all promises that are created at the same locations. In our example, p1 and p3 are considered as a single unique promise, and so are p2 and p4. The traces from T1 result in a full registration coverage for p2, as both $Reg_{fulfill}$ and Reg_{reject} events are in place. It also includes *Fulfilled* and $Exec_{fulfill}$ for p2, leading to a partial settlement and execution coverage for this promise. The traces from T2 append *Rejected* and $Exec_{reject}$ events to p2, leading to its full async coverage.

5.3 Visualizing the Asynchronous Coverage

We designed an interactive visualization integrated in VSCode, a widely used development environment, based on feedbacks gathered from a preliminary user study we conducted. Users can invoke JScope on demand in VSCode through a button (Figure 5.1, A), which will begin to automatically calculate asynchronous coverage for the selected project. It then

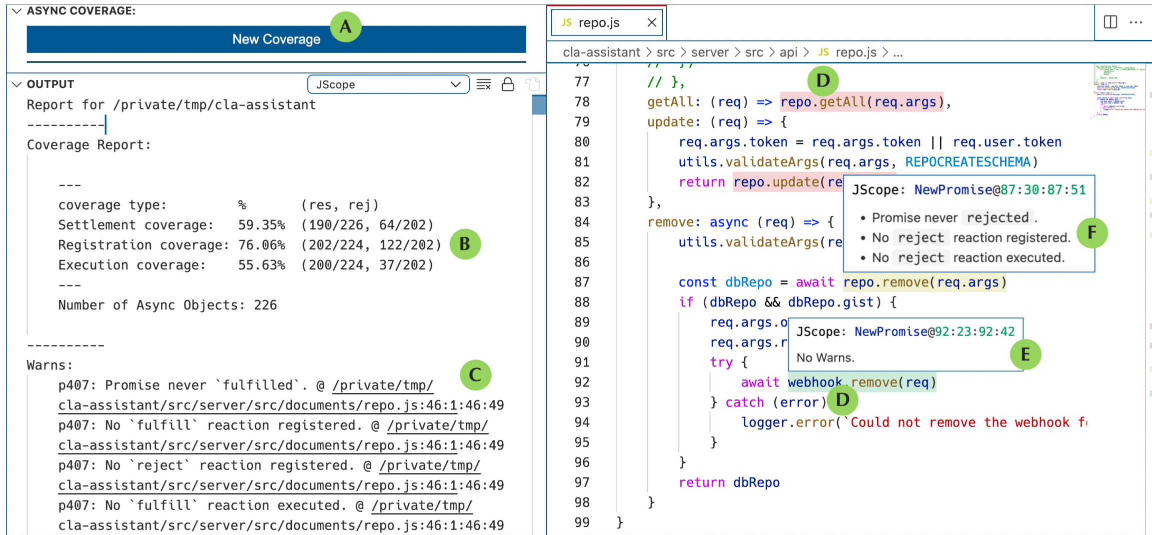

Figure 5.1: JScope coverage results for CLA Assistant. The open editor shows `RepoService.remove` in repo.js.

presents the results in terms of a textual report (Figure 5.1, B&C) and visual cues overlayed on the code (D–F in Figure 5.1).

We improved the JScope's usability and accesibility, in an iterative process using user feedback. We collected the feedback through a priliminary user study where participants performed coverage-related tasks using JScope.

JScope summarizes async coverage results in the *Coverage Overview* panel to help with overall understanding of async coverage. It consists of two parts, an async coverage summary (Figure 5.1, B), followed by a list of clickable warnings (Figure 5.1, B). We utilized hyperlinks to connect the warnings with the location of their respective promises in the code. This allows developers to seamlessly go back and forth between the warning and the relevent sections in the code. Additionally, we overlay relevant visual cues on the code in the editor. JScope highlights promises using a red-yellow-green "color spectrum" to determine their level of async coverage. (Figure 5.1, D) As such, the promise in line 82 is highlighted with red, indicating minimal async coverage. Similarly, the green and yellow highlights on line 92 and 87 indicate fully and partially covered promises, respectively. Users can obtain more details on the warnings on demand, by hovering the mouse over warning cues (Figure 5.1, E&F). By leveraging the integration of focus within the context [27], we help maintain programmers' mental model of the overall program while working with individual promises. It further helps prevent the users from being overwhelmed by the information.

5.4 Implementation

We used NodeProf.js [76] to instrument JavaScript programs. It is a newer and more lightweight instrumentation framework built on top of Jalangi [71] and further supports

the `async/await` syntax. JavaScript uses builtin functions to settle promises, i.e. calls to `resolve` and `reject` functions associated with a promise constructor cannot be intercepted using NodeProf.js. It also doesn't provide an API to access registered reactions to a promise. As such, we utilized JavaScript proxies [3] to modify the behavior of these builtin features in order to intercept the execution of callbacks used for settling promises and registering their reactions. Our dynamic analysis required additional functionalities for detection of exception handling for await statements and try/catch blocks. To this end, we extended the implementation of Nodeprof.js to support detection of try/catch blocks.

Using code location as an approximation to uniquely identify a promise compromises the precision of our results. It is possible for a function that returns a promise to be called in different locations, with different outcomes for each promise instance. In order to improve the accuracy of such cases and achieve consistency with the behavior of async functions, our implementation consideres call-sites of these functions as the creation location of the promise. Similar to an async function, if a regular function returns a promise, each of its call-sites are considered as a separate unique promise object. For instance, running JScope on the code below will result in two different promises created on lines 69 and 70.

```
63    function promisify(value) {
64        return new Promise((resolve, reject) => {
65            resolve(value);
66        });
67    }
68
69    const p1 = promisify(14);
70    const p2 = promisify(15);
```

We utilized programmatic APIs of Mocha [1] and Tap [2] testing frameworks for automatic execution of apps through their test suites and VSCode's extension development API to integrate JScope into its editor. JScope is available for download as an open-source application [9].

Chapter 6

Evaluation

We evaluate our approach by measuring the asynchronous coverage for 20 JavaScript applications, studying their correlations with traditional coverage criteria, and investigating bugs in asynchronously uncovered code. Further, we assess the usefulness of JScope for developers through a controlled experiment. We address the following research questions.

RQ1. Does having high traditional coverage imply adequate testing of asynchronous code?

RQ2. How can asynchronous coverage criteria facilitate identifying test inadequacies regarding faulty asynchronous code?

RQ3. How does using JScope help improve developers' performance in assessing test adequacy and debugging?

RQ4. What is the performance overhead of JScope?

6.1 Asynchronous Coverage

To answer **RQ1**, we ran JScope on 20 web applications, measured three types of asynchronous coverage criteria and studied their correlations with traditional coverage metrics.

6.1.1 Experimental Design and Procedure

We randomly selected 20 open-source JavaScript applications from GitHub that used promises and/or `async`/`await` and had a test suite. We ran JScope on the subjects by automatically exercising them through their test suites. We measured the results of the three asynchronous coverage metrics, and calculated statement, function, and branch coverage of benchmarks using Istanbul, [1] a popular JavaScript coverage tool. We then examined the possible correlations of our proposed asynchronous coverage criteria with these traditional criteria.

6.1.2 Results and Discussion

[1] `https://istanbul.js.org/`

Application			Objects	Traditional Coverage			Asychronous Coverage		
Name	**LOC**	**#Tests**	**#Promises**	**Statement(%)**	**Function(%)**	**Branch(%)**	**Settlement(%)**	**Registration(%)**	**Execution(%)**
1. Node Fetch	2475	392	12	97	100	94	74	68	59
2. CLA Assistant	20406	315	225	94	94	84	59	76	56
3. Minipass Fetch	1523	57	20	100	100	100	74	59	55
4. Cacache	1878	95	99	100	100	100	64	66	55
5. Github Action ...	485	42	10	100	100	100	100	100	100
6. Co	470	43	10	99	100	98	84	94	94
7. Delete Empty	272	20	8	91	100	80	47	77	46
8. JSON Schema ...	3070	256	34	88	88	78	87	94	88
9. Async Cache Dedupe	1476	120	13	100	100	100	56	83	57
10. Environment	4374	328	64	81	76	72	51	70	51
11. Socket Cluster Server	2044	72	52	82	70	70	62	50	41
12. Socket Cluster Client	10648	37	13	73	54	53	77	45	41
13. Minipass	840	131	10	100	100	100	87	50	20
14. Grant	2756	495	29	98	97	89	53	70	52
15. Express HTTP Proxy	798	106	57	96	97	87	70	100	80
16. Install	556	31	7	98	98	95	46	100	78
17. Cachegoose	224	27	8	91	92	79	43	80	30
18. Enquirer	10491	179	88	68	63	61	50	47	42
19. Avvio	5460	180	13	94	95	91	50	56	37
20. Matched	274	30	9	96	100	78	53	100	64
AVERAGE	3385	144	39	92	91	85	64	74	57

Table 6.1: Summary of different coverage metrics reported by JScope and traditional coverage.

	Statement	Function	Branch	Settlement	Registration	Execution
Settlement	0.20	0.10	0.26	1	0.11	0.48
Registration	0.49	0.56	0.35	0.11	1	0.79
Execution	0.31	0.33	0.29	0.48	0.79	1

Table 6.2: correlation coefficients for asynchronous and traditional coverage criteria.

The results are displayed in Table 6.1. The first four columns show an application's name, lines of code, number of tests, and number of promise objects observed in the analysis, respectively. The next three columns depict the results of traditional coverage criteria, i.e., statement, function, and branch coverage. The following three columns include the calculated asynchronous coverage metrics for each benchmark: settlement, registration, and execution coverage.

As an example, the Node Fetch application in row 1 contains 2475 lines of source code and 392 tests. 12 promise objects are created during the execution of Node Fetch through its test suite by JScope. The coverage of the application's code is 97%, 100%, and 94% in terms of statement, function, and branch coverage, respectively. However, the asynchronous coverage in terms of settlement, registration, and execution coverage is 74%, 68%, and 59%, respectively.

Overall, the benchmarks had relatively high traditional coverage scores, with an average of 92%, 91%, and 85% statement, function, and branch coverage, respectively. However, it can be seen that the settlement, registration, and execution coverage scores were much lower, with an average of 64%, 74%, and 57%, respectively. This means that, on average, the test suite of a typical JavaScript application a exercises 92% of the statements but about 65% of the expected outcomes of its promises and async functions. a may not even register over 25% of necessary reactions for async operations. Even fewer reactions are actually exercised through tests.

Next, we examined the potential **correlations** between asynchronous and traditional coverage criteria. We used the Kendall rank correlation coefficient, which does not assume a normal data distribution. Kendall coefficient is in the range of [-1, +1], where closeness to +1 and -1 shows a stronger correlation. Numbers closer to 0 indicate weaker relationships accordingly. The results, as depicted in Table 6.2 show no strong correlations between traditional and asynchronous coverage metrics. The lack of strong correlations supports our hypothesis that traditional coverage metrics are not necessarily equipped for indicating the sufficient execution of asynchronous scenarios through tests. In other words, covering more lines or functions does not imply covering more of the asynchronous behavior of an application.

	Commit	App	Category	Settlement	Registration	Execution	Branch
1.	#f56491a	express-http...	Unhandled Exception	63	96	74	87
2.	#d902776	cla-assistant	Unhandled Exception	58	75	55	84
3.	#8ff7de7	streamroller	Unhandled Exception	60	81	67	100
4.	#8e94a60	eslint_d.js	Unhandled Exception	70	65	65	89
5.	#6bcf8ca	checkfire	Unhandled Exception	40	55	40	-
6.	#fff6640	postgres	Unhandled Exception	71	83	60	91
7.	#2fc9693	haraka-...	Unhandled Exception	25	33	33	-
8.	#e5615da	ioredis	Unhandled Exception	76	69	55	88
9.	#146bb3b	install	Unhandled Exception	50	100	62	96
10.	#0dfff52	json-schema...	Unhandled Exception	80	91	81	88
11.	#cbcdfc6	socketcluster...	Unhandled Exception	63	50	43	70
12.	#dfbafbf	clamscan	Pending Operation	58	89	62	38
13.	#48a2ddf	cla-assistant	Broken Promise Chain	58	75	55	84
14.	#b0a86d4	avvio	Broken Promise Chain	38	58	38	90
15.	#68342f8	libnpmteam	Unnecessary Asynchrony	40	83	61	100

Table 6.3: Asynchrony-related JavaScript issues from Github.

Overall, while the high traditional coverage scores raise confidence in sufficient verification of the code, they are not equipped with identifying shortcomings of the tests in asynchronous scenarios. For instance, while 85% of the branches are exercised on average, only 57% of the expected reactions of asynchronous operations are invoked.

6.2 Asynchronous Coverage and Test Effectiveness

To address **RQ2**, We used JScope and Istanbul to examine both types of coverage for code snippets related to previously resolved issues on GitHub. We investigated (1) if traditional coverage criteria raise any warnings regarding inadequate testing of faulty asynchronous code and (2) if JScope could have helped discover these bugs during testing.

6.2.1 Experimental Design and Procedure

We randomly selected 15 issues from JavaScript projects on GitHub that relied on asynchrony for their execution, were accompanied by a supported test suite and used one of the testing frameworks supported by JScope. All these issues were fixed by the developers. We used the issue descriptions, commits, conversations, and related pull requests to confirm that the issues were valid and caused by an asynchronous part of the code. Next, we

collected traditional code coverage results using Istanbul to ensure for each program the statements relevant to the code causing these bugs were executed and thus covered by the tests before the fix. We then ran JScope on two versions of each project, one immediately before and one immediately after each bug fix. We used JScope's output to investigate the inadequacies of the tests in exercising the asynchronous behavior in code segments related to each bug.

6.2.2 Results and Discussion

Table 6.3 lists the 15 analyzed bugs and the changes in asynchronous and traditional coverage reports before and after the fix. Columns 1–3 show the commit pertaining to the bug fix, the application name, and the bug category, respectively. The next three columns display the calculated async coverage numbers before the fix. Finally, the last column displays the branch coverage before the fix. We included branch coverage as a representative of the basic traditional coverage metrics, i.e. statement, branch, and function coverage to demonstrate the general state of these applications in terms of their coverage before the fix. As priorly mentioned, all these applications have full traditional coverage for code segments relevant to the bug.

Overall, JScope showed insufficient coverage and relevant warnings for all bugs, addressing which could have helped detect and fix the bug before deployment. Traditional coverage, however, showed no sign of warning or insufficient testing for any of the bugs or their relevant code segments. While the async coverage scores generally increased after the fixes, they also decreased in a few cases. In those cases, the fix introduced new promises that were not fully covered and thus decreased the overall async coverage.

Next, we discuss the main categories of studied bugs and describe how JScope's reports and warnings could have benefited the bug finding process through two examples.

Unhandled Exceptions

Developers often neglect to test exceptional executions of asynchronous operations [18]. While current coverage criteria can indicate insufficient testing of conditions and branches, they are unable to detect insufficient testing of alternative scenarios for asynchronous operations, such as missing reactions for rejected asynchronous operations or missing error handling.

Example A. Eslint_d.js is an application that daemonizes ESLint [2] for higher performance and has >30k weekly downloads on the NPM registry (Table 6.3, row 4). It caches and reuses a single linter object to reduce overhead. Line 272 of the left code snippet in Figure 6.1-A shows how the async function `getCache` is invoked to asynchronously retrieve

[2] https://www.npmjs.com/package/eslint

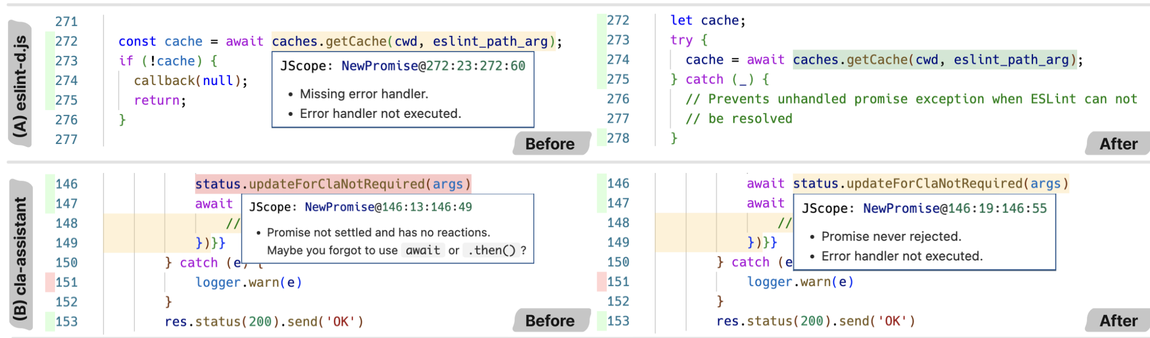

Figure 6.1: Statement coverage vs JScope results, before and after the bug fix. Green marks beside line numbers show covered statements reported by Istanbul. Overlaying highlights are generated by JScope, where yellow and red indicate promises with inadequate async coverage.

a cached ESLint linter object from a given path. The program, using **await**, waits until this promise fulfills. A bug was reported on this method despite the full coverage of this code segment by the tests, as depicted by the green markings by the line numbers. The bug report stated that the application crashes with an unhandled promise exception if the path given to `getCache` cannot be resolved. The proposed fix added a `try/catch` around the call to `getCache` to allow handling exceptions caused by the rejected promise and prevent further crashes (Figure 6.1-A, right snippet, lines 273–278). A corresponding test was also added to the test suite that programmatically simulates the exception and exercises the **catch** block (lines 275-278).

This bug had remained undetected in production for four months. However, running JScope on the faulty version of the code reported insufficient coverage in terms of a missing reject reaction for the promise returned by `getCache`, shown as the highlighted code on line 272 and the "Missing error handler" warning message box (Figure 6.1-A). Having had access to JScope during testing could have helped reveal this bug before production.

Our results in Table 6.3 showed multiple instances of unhandled exceptions, similarly missed by the applications' tests. Row 3 is an example where developers managed to achieve 100% statement coverage, while still failing to detect a missing reject reaction causing a crash. Consider our first motivating example from Section 3.1.1. Ambiguous reports mention the same issue two years before the fix. The issue persisted to a point where it had damaged the users' trust, with a user calling CLA Assistant a phishing tool. [3]

This group of bugs can get more complicated in practice, and thus harder to locate and address. Figure 6.2 is a simplified code fragment related to row 1. a middleware to proxy

[3] https://github.com/cla-assistant/cla-assistant/issues/[561,691, and 822]

```
71  function buildProxyReq(Container) {
72    return new Promise((resolve) => { // P1
73      Promise.all ([parseBody, createReqOptions]) // P2
74      .then((responseArray) => { // P3
75        resolve (Container)
76      }) });
77  }
78  buildProxyReq(container).catch(next); // P4
```

Figure 6.2: Simplified implementation of the function `buildProxyReq` in express-http-proxy before the fix. P1, P2, P3, and P4 show promises detected by JScope.

HTTP requests with 350k weekly downloads on NPM. [4] `P1`'s executor function constructs a promise chain `P2` $\rightsquigarrow$ `P3`, where `P3` fulfills `P1`. The developer is under the impression that with `catch` (P4) in place, exceptions caused by `buildProxyReq` will be properly handled. However, this code creates two separate chains `P1` $\rightsquigarrow$ `P4` and `P2` $\rightsquigarrow$ `P3`, allowing rejections in the second promise chain to crash the program. This application is also included in row 16 of the coverage summary table15. Despite full statement coverage for `buildProxyReq`, JScope shows warns pointing out to the accidental broken promise chain.

Broken Promise Chains

JavaScript programs will not wait for the completion of asynchronous operations, unless explicitly specified. In other words, the execution of operations that depend on the completion of a promise is reliant on properly chaining them through promise reactions or `await` statements. Developers can mistakenly break the chain of asynchronous operations by not awaiting their completion [51]. This may alter the flow of execution leading to undesired outcomes. Moreover, the outcome of the promise will not be used, and potential exceptions will not be caught, which can lead to a myriad of issues in programs. Our first motivating example displayed a case were this mistake led to the CLA Assistant application crashing, caused by an unhandled exception thrown by an un-awaited promise (Section 3.1.1).

Example B. Row 13 of Table 6.3 shows another issue in CLA Assistant. Repositories that use CLA Assistant may require contributors to sign a Contributor License Agreement (CLA) through CLA Assistant's web interface. When a user signs a CLA through CLA Assistant's web interface, `handleWebhook` is invoked (partially shown in Figure 6.1-B). Upon invocation of the async function `updateForCla\- NotRequired` (line 146), a promise is returned that asynchronously communicates the status update on the signature to GitHub servers. It then sends a confirmation to the user (line 153).

[4] `https://github.com/villadora/express-http-proxy/pull/274`

Users had reported issues where the web interface shows an updated status for a pull request, whereas on GitHub, the repository is still pending CLA Assistant's update. Two other preceding issues vaguely report the same bug with reporters unable to reproduce the error. [5]

JScope reported low async coverage for the promise on line 146 before the fix (Figure 6.1-B). The warning states that the promise has not settled and has no reactions, suggesting a fix through adding a `then` or `await` statement. This matches the fix provided by the developers for the original issue, which added an `await` before the call to `updateForClaNotRequired` to wait for the function's completion before sending a response to user (line 146). Note that after the fix, partial coverage for the promise on line 146 indicates that erroneous scenarios are not tested. This is also apparent with statement coverage showing untested `catch` block on line 151.

Pending Operations

If not explicitly settled, asynchronous operations remain pending, causing endless execution of programs or memory leaks. These cases often happen as a result of developers treating asynchronous code similar to synchronous code, such as incorrectly calling `return` inside the promise executor function to denote its completion instead of calling `resolve` as is the case in Table 6.3, row 12. For these cases, JScope reports missing fulfillment and low settlement coverage for the pending promise.

Unnecessary Asynchrony

Developers may complicate code by using promises where asynchrony is not required. They may also nest promises, causing unanticipated bugs as in Figure 6.2. While generally less severe, JScope warns about their missing rejection scenarios.

Overall, async coverage criteria can effectively expose test inadequacies related to asynchrony that are undetected by traditional coverage metrics. As such, JScope can help identify parts of code that contain asynchrony-related bugs in practice despite being covered by traditional coverage.

6.3 Usefulness of JScope to Developers

To address **RQ3**, we conducted a controlled user experiment to investigate the effectiveness of JScope in helping programmers identify and debug (un)covered JavaScript code.

[5] https://github.com/cla-assistant/cla-assistant/issues/[520, 697]

6.3.1 Experimental Design and Procedure

Our experiment had a "between-subject" design to avoid the carryover effect. We divided our participants into two groups: *control* and *experimental* groups. The experimental group had access to a simplified and web-based version of JScope results. Both groups had access to the code, as well as statement coverage results from Istanbul, loaded on our web-based user interface. The web-based user interface for experimental group contained JScope's visual cues and user interactions and a style similar to JScope for consistency.

Variables.

Our *Independent Variable* is the type of tool used, referred to as *Tool* from hereon, which is a nominal variable with two levels: *JScope* and *Istanbul*. We consider two continuous *Dependent Variables*, task completion duration (seconds) and accuracy (%), which represent the performance of programmers in completing the tasks.

Participants.

We recruited six male and six female participants, aged 21–35, consisting of 10 graduate students and two software engineers, with 1–5 years of experience in software development. We assigned them randomly to experimental and control groups. We balanced the expertise based on our participants' responses to a pre-questionnaire (Section 6.3.1).

Experimental Object.

We used a simplified `src/body.js` file from Fetch, [6] an application for implementing browsers' `window.fetch` in Node.js, which has a test suite and >25M weekly downloads. For the debugging task, we chose a fixed bug from Docusaurus, an application for building and deploying websites. [7] The unhandled reject reaction bug, covered by the tests, led to silent failure of the whole application.

Tasks.

We designed three tasks that pertained to test adequacy and quality assessment (Table 6.4). The first task was designed to assess the effectiveness of *Tool* in helping programmers identify both well-tested and insufficiently tested functions (T1.A & T1.B). T2 required the participants to locate all created promises (T2.A) and to identify those that were not sufficiently tested (T2.B). T3 was designed to investigate the usefulness of *Tool* in helping participants identify the root cause of the bug (T3.A) and propose a fix (T3.B).

[6] `https://github.com/node-fetch/node-fetch`

[7] `https://github.com/facebook/Docusaurus/issues/238`

Task	Description
T1.A	Identifying sufficiently tested functions
T1.B	Identifying less robust functions (i.e. not sufficiently tested)
T2.A	Locating all promises created during testing
T2.B	Identifying promises that are not properly tested
T3.A	Locating the root cause of a failure
T3.B	Finding the fix to the failure

Table 6.4: Tasks used in the user study.

Pre-study.

All participants filled a pre-questionnaire form prior to their session, indicating their demographic information and their experience in programming, JavaScript development, and testing, and self-assessed proficiency levels. We used this data to fairly balance the participants between groups.

Training.

The participants were given a refresher tutorial on main concepts of asynchronous JavaScript and coverage. The experimental group also received a tutorial on using JScope.

Tasks.

Next, the participants started performing the tasks (Table 6.4). The participants were allowed to interact with the code and the tools and write their answers on a Google Doc shared with the examiner. We measured the *duration* during the session by providing each task to the participants individually, which they returned after completing the task. To measure *accuracy*, we used pre-defined rubrics to mark the participants' responses after the session.

Post-study.

After the session, the participants responded to a post-questionnaire form with qualitative data on usefulness of the *Tool* used and its limitations.

6.3.2 Results and Discussion

We ran the Shapiro-Wilk normality test on the data, and since the distributions were not normal, we used Mann-Whitney U tests to analyze the results. The results showed a statistically significant difference (28% on average) on the total accuracy of responses for the experimental group using JScope (Mean=95%, STDDev=9%), compared to the control group (Mean=74%, STDDev=12%).

The results also showed the control group spent slightly less time in total (Mean=33:56, STDDev=4:35), compared to the experimental group (Mean=36:29, STDDev=5:01), although the difference was not statistically significant. The results of individual tasks showed that although the experimental group spent more time for completing T1 compared to the control group, they performed all other tasks faster (14%–33% on average). It was expected for the experimental group to spend more time on T1 due to the additional learning curve incurred by their infamiliarity with JScope, and they still achieved an average of 33% higher accuracy for T1. We do not find these results surprising, due to the following. First, the unfamiliarity of the participants with JScope and necessity of using both tools incurred a larger learning curve for experimental group, which affected the time in which they completed T1. Further, T1 required examining the whole code, regardless of whether or not asynchrony is involved. It is reasonable that JScope did not improve the duration, as the tasks were generic and not directly related to async coverage, and the participants in the experimental group The experimental group still achieved an average of 33% higher accuracy for T1, and the difference was statistically significant. Further analysis of the results showed that the control group believed they had successfully completed the task and thus terminated it earlier, while they had missed part of the necessary information and hence achieved lower accuracy scores. For the remaining tasks, the experimental group performed consistently faster than the control group, while achieving higher accuracy.

More Accurate Assessment of Test Effectiveness.

The tasks involved performing various activities including general function coverage to more specific promise coverage, for all of which JScope showed to improve the accuracy of the participants. We had hypothesized that JScope would be most useful for tasks directly involving asynchronous interactions.

For instance, T2 involved examining promises and async/await statements, where we expected JScope to be helpful. Using JScope helped the experimental group perform significantly better for T2. They completed this tasks 33% faster (p=0.02) and 30% more accurately (p=0.04) on average.

Debugging.

The effectiveness of tests is directly dependent on its bug finding capability. Coverage metrics do not directly attribute to identifying and fixing bugs. However, they can facilitate the process by guiding programmers towards the less verified portions of the code that may contain bugs. Using JScope helped experimental group in the debugging process by helping them achieve more accurate answers while spending less time locating the root cause of a failure (T3.A) and finding a fix (T3.B). The results were statistically significant for the accuracy of the proposed fix (T3.B) where experimental group achieved an average of 37% higher accuracy (p=0.03).

Application			Execution Time (s)	Instrumentation (s)	Slowdown
Name	**LOC**	**#Tests**	**Normal(Instrumented)**		
21. Node Fetch	2475	392	11(22)	35	2x
22. CLA Assistant	20406	315	5(69)	97	14x
23. Minipass Fetch	1523	57	4(32)	26	8x
24. Cacache	1878	95	3(61)	29	20x
25. Github Action ...	485	42	1(30)	41	30x
26. Co	470	43	1(6)	25	6x
27. Delete Empty	272	20	0.5(38)	30	76x
28. JSON Schema ...	3070	256	10(218)	36	22x
29. Async Cache Dedupe	1476	120	21(32)	23	2x
30. Environment	4374	328	28(2501)	93	89x
31. Socket Cluster Server	2044	72	19(53)	30	3x
32. Socket Cluster Client	10648	37	9(62)	32	7x
33. Minipass	840	131	3(36)	25	12x
34. Grant	2756	495	5(364)	59	73x
35. Express HTTP Proxy	798	106	27(58)	43	2x
36. Install	556	31	0.08(118)	25	1475x
37. Cachegoose	224	27	2(34)	48	17x
38. Enquirer	10491	179	0.4(98)	40	245x
39. Avvio	5460	180	6(56)	25	9x
40. Matched	274	30	0.06(8)	29	133x
MEDIAN	1523	101	4.5(54.5)	31	15.5x

Table 6.5: Performance overhead of JScope; The numbers show an average of five executions

participants feedback.

Overall, the experimental group found JScope useful. In particular, they liked the overview of the coverage report, interactions with the overlayed visual cues, and the warning messages that guided them towards missing functionality or tests.

Overall, participants using JScope performed 28% more accurately in testing and debugging asynchronous code.

6.4 Performance

We measured the performance of JScope in terms of its overhead of instrumentation and test suite execution time by averaging five executions of each test suite, with and without JScope. Table 6.5 presents our performance analysis for the applications in Table 6.1. The results indicate a median of 31 seconds of instrumentation (23–97 seconds). The slowdown factor for execution of the instrumented code generally ranges 2x–100x (median: 15.5x). This slowdown is similar to other instrumentation-based dynamic analyses for JavaScript [18, 77, 39].

6.5 Threats to Validity

Our study participants, benchmark projects, or issues, may not be a proper representation of the real world. We tried mitigating this by randomly selecting participants who met the minimum experience requirements and projects of different sizes from different domains that had the prerequisites for using JScope.

To mitigate the examiner's bias in our user study, we delegated the timekeeping to the participants, allowing them to decide the start and end time of each task by handing them the tasks separately and asking them to return it afterwards. We defined a detailed rubric for grading the accuracy of the results before conducting our study to address the same threat in measuring participants' accuracy. We attempted to mitigate the impact of expertise level in our study by classifying participants based on their responses to our pre-questionnaire.

Regarding the completeness of our results, Our approach might miss the cases where the code is not covered, since dynamic analysis is not complete. However, our manual investigation of JScope results for projects in Table 6.1 and Table 6.3 indicates that the results for promises that are executed by the test suites are predominantly accurate.

To balance the training between both groups of our study, Both groups received a tutorial on main concepts such as testing and coverage, JavaScript development, and asynchrony in JavaScript. The control group then received a tutorial on Istanbul. The experimental group were presented with a brief training on the visualization and usability of the tool, which they were seeing for the first time. The tutorial and the accompanying narration were carefully crafted to provide a consistent introduction to the tool, without providing further knowledge on the semantics of JavaScript, the object application, and the tasks used in the study. They had only a few minutes to familiarize themselves with the tool. We believe the lack of experience of the experimental group could affect the results against JScope, as in reality, our users would have more experience and/or have access to more support with the tool. Furthermore, this type of introduction of the tool, as our independent variable, is also commonly practiced in the research community, more thoroughly discussed in Chapter 7.

Finally, we published JScope and the user study data to allow reproducibility of our experiments [9, 10].

Chapter 7

Related Work

7.1 Code Coverage Criteria

While being the most prominent test quality assessment technique [86], code coverage criteria have always been under scrutiny about their effectiveness [42, 35, 43, 41]. Hemmati investigated the effectiveness of code coverage criteria, which he refers to as control-flow coverage, by showing their inadequacy to detect bugs when used in isolation [41]. His results suggest that combining these coverage criteria with data-flow coverage metrics increases the bug detection capability substantially. Inozemtseva et al. also conducted a large scale study on effectiveness of code coverage criteria [43]. Their study resulted in a low correlation between code coverage criteria and test effectiveness, suggesting that code coverage criteria alone are not a good indicator of the effectiveness of the test suites. A more recent study on code coverage for JavaScript applications indicates that a majority of uncovered parts of code in JavaScript are event-dependent and asynchronous callbacks [35]. The results of this study support our motivation that asynchronous JavaScript is prone to a wider range of bugs and testing asynchrony in JavaScript is a challenging endeavour.

Generic nature of traditional coverage criteria has lead to emergence of various domain-specific coverage criteria, covering a wide range of domains in programming including but not limited to logical expressions [19], neural networks [79, 13], graphical user interfaces [54], and state machines [73, 45]. Several coverage metrics have been introduced using data-flow to target concurrency in actor-based [81], concurrent [84, 33, 80, 72, 66], and distributed programs [67, 40]. Researchers have proposed novel criteria for dynamic web applications by targeting HTML elements [63, 87], page access and database access [14, 88], and RESTful APIs [53, 28]. Other work have targetted JavaScript specifically, focusing on its loosely typed nature [24] or DOM elements [61]. None of these techniques, however, address the asynchronous execution and its respective challenges.

7.2 Program Analysis for JavaScript

Event-dependent and asynchronous callbacks form a majority of untested code in JavaScript [35]. Prior work have used **static analysis** to model program behavior, detect anti-patterns, and provide refactorings for JavaScript code [52, 74]. Madsen et al. presented *event-based call graphs*, a program representation to model JavaScript's event-driven behavior [52]. They further introduced *promise graphs* to model the behavior of JavaScript promises, and detect promise-related anti-patterns. More recently, Turcotte et al. leveraged static analysis to identify and visualize asynchrony-related anti-patterns [83]. Arteca et al. further published resynchronizer, a refactoring tool for asynchronous JavaScript code to allow program execution speedups [22].

However, JavaScript's dynamic nature introduces many challenges and limitations to using static analysis [20]. **Dynamic analysis** has been popularly used in JavaScript to address the imprecision of static analysis in analyzing JavaScript's inherent dynamism [48, 82, 65]. Much research in this area targets understanding, debugging, and testing techniques for programs in general [39, 62, 34, 23] [78, 29, 36, 56, 49, 57], and more recently for asynchronous JavaScript in particular [18, 77, 70].

Alimadadi et al. proposed a model for understanding JavaScript event-based interactions [17]. They also extended the notion of *promise graphs* by Madsen et al. [51] and developed PromiseKeeper [18] to help find anti-patterns related to JavaScript promises. NRace [26] is also a tool with a dynamic approach, incorporating happens-before rule and multi-priority queue to detect event races in Node.js. Sun et al. introduced AGraph [77], another dynamic analysis tool that detects bugs related to asynchronous sources of JavaScript programs based on their proposed model for communications with Node.js event loop. Tools such as Sahand [15] or AwaitViz [82] are the results of efforts to visualize the behavior of asynchrony in JavaScript in order to better comprehend their behavior. The extensive research on bug detection and comprehension of asynchrony confirms our argument for the necessity of test adequacy criteria that take into account the asynchrony in JavaScript and other languages.

7.3 Visualization

Visualization has been effectively used for better comprehension and modeling of event-driven and asynchronous programs [18, 83, 17]. Tools such as Sahand [15] or AwaitViz [82] are the results of efforts to visualize the behavior of asynchrony in JavaScript in order to better comprehend their behavior. DrAsync visualizes promise lifetime in JavaScript programs to identify anti-patterns and performance bottlenecks [83]. Madsen et al. and Alimadadi et al. use graph visualization to shed light on the behavior of JavaScript promises and the interactions between them. Seifert et al. presented an interactive and integrated visualization for JavaScript's asynchronous call graphs [70]. Similar to Seifert et al., we leveraged editor

integration to facilitate the comprehension of asynchronous coverage through an interactive interface.

7.4 User Studies

User studies are prevalently used in collecting expert data [75], measuring accuracy or usefulness of tools [16, 69], and comparing multiple approaches [46]. Many prior research have conducted user studies with 10 to 14 participants [75, 46, 32]. Recruiting a similar number of participants has been common in the software engineering community, even when the user study has been the only means of evaluation [32, 69]. We further introduced JScope as an independent variable in our study, analogous to the works of Ponzanelli et al. [64] and Alimadadi et al [15].

7.5 Mutation Testing

Mutation testing is also used as an alternative approach for measuring test quality [44, 55]. It requires defining mutants (i.e. altered versions of code) and measuring how many of them are detected by the tests. While not as popular as code coverage criteria, some prior research propose mutation algorithms and testing fault detection tools for JavaScript [68]. Mirshokraie et al. have done promising research on mutation testing for JavaScript. They proposed an algorithm to select variables and branches for mutation and a function ranking metric [59] and further adopted mutation testing to build automatic test generation tools [58, 60]. Despite their effectiveness, mutation testing for JavaScript is typically very costly, and has yet to gain the popularity of code coverage [21].

Chapter 8

Conclusion and Future Work

8.1 Conclusion

In this dissertation, we proposed a set of coverage criteria for assessing the adequacy of tests in verifying asynchronous program behavior. These criteria target eventual completion, registration of reactions, and execution of reactions for asynchronous operations. We utilized dynamic program analysis to instrument JavaScript applications for measuring our proposed criteria. We designed and implemented a tool to automatically calculate and generate asynchronous coverage reports. By embedding our tool into a typical development environment, we enabled programmers to view async coverage results on top of the code, reducing the cognitive burden of combining the results from a separate view. The interactive visualizations provide developers with extended debugging and comprehension capabilities regarding asynchronous code. The results of our evaluation show that asynchronous coverage criteria are complementary to traditional metrics such as statement coverage. Through examining real issues from open-source programs, we showed that async criteria can help programmers detect insufficiencies of tests and related bugs in asynchronous code where traditional metrics can't. Our controlled user experiment also demonstrated that our tool helps improve developers' performance in tasks related to assessing test quality and debugging of asynchronous code.

8.2 Future Work

Asynchrony in Other Languages

Asynchronous programming is a language-agnostic paradigm, and not limited to JavaScript. However, JavaScript's inherent asynchrony, and its predominance in modern web development has made it a prominent choice for our research. Promises and `async/await` proved to be effective methods for utilizing asynchrony, so that they found their way into other programming languages. They are in python as *Awaitables*, in C++ as *Promises*, in Java and Dart as *Futures*, and in C# as *Tasks* [12, 7, 8, 5, 6]. These commonalities put for-

ward possibilities of generalizing async coverage criteria for languages that are inherently synchronous, but utilize asynchrony.

Automatic Test Generation

A good test suite helps to identify defects early in the development process and provides confidence in the software's behavior. Writing tests manually is a time-consuming and error-prone task, especially when dealing with complex software systems. As such, automatic test generation has become an active area of research in recent years. We examined that our proposed coverage criteria can help improve the quality of tests and discover potential issues in asynchronous JavaScript code. One possible future direction could be to utilize these criteria to model applications' behavior in order to automatically generate test suites. With the goal to maximize asynchronous coverage criteria, these automatically generated tests can contribute to robustness of asynchronous programs.

Bibliography

[1] Mocha, the fun, simple, flexible JavaScript test framework, 2022. [Online; accessed 1-September-2022].

[2] Node Tap, 2022. [Online; accessed 1-September-2022].

[3] Proxy - JavaScript, 2022. [Online; accessed 1-September-2022].

[4] Top programming languages | the state of the octoverse, 2022. [Online; accessed 17-February-2023].

[5] Asynchronous programming: futures, async, await, 2023. [Online; accessed 30-Aug-2022].

[6] Asynchronous programming with Async and Await (Visual Basic), 2023. [Online; accessed 30-Aug-2022].

[7] Awaitables, python documentation, 2023. [Online; accessed 16-Jan-2023].

[8] Future (Java Platform SE 8), 2023. [Online; accessed 30-Aug-2022].

[9] JScope - VSCode Extension for measuring Asynchronous Coverage Criteria, 2023. [Online; accessed 14-Feb-2023].

[10] JScope User Study Materials, 2023. [Online; accessed 14-Feb-2023].

[11] M. Ogden, Callback Hell, 2023. [Online; accessed 14-Feb-2023].

[12] Promises, Cplusplus.com, 2023. [Online; accessed 14-Feb-2023].

[13] Faouzi Adjed, Mallek Mziou-Sallami, Frédéric Pelliccia, Mehdi Rezzoug, Lucas Schott, Christophe Bohn, and Yesmina Jaafra. Coupling algebraic topology theory, formal methods and safety requirements toward a new coverage metric for artificial intelligence models. *Neural Computing and Applications*, pages 1–16, 2022.

[14] Manar H. Alalfi, James R. Cordy, and Thomas R. Dean. Automating Coverage Metrics for Dynamic Web Applications. In *2010 14th European Conference on Software Maintenance and Reengineering*, pages 51–60, 2010.

[15] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. Understanding Asynchronous Interactions in Full-Stack JavaScript. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 1169–1180, 2016.

[16] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. Inferring hierarchical motifs from execution traces. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 776–787, 2018.

[17] Saba Alimadadi, Sheldon Sequeira, Ali Mesbah, and Karthik Pattabiraman. Understanding JavaScript Event-Based Interactions. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 367–377, New York, NY, USA, 2014. Association for Computing Machinery.

[18] Saba Alimadadi, Di Zhong, Magnus Madsen, and Frank Tip. Finding Broken Promises in Asynchronous JavaScript Programs. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018.

[19] P. Ammann, J. Offutt, and Hong Huang. Coverage criteria for logical expressions. In *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.*, pages 99–107, 2003.

[20] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. A Survey of Dynamic Analysis and Test Generation for JavaScript. *ACM Comput. Surv.*, 50(5), sep 2017.

[21] J. H. Andrews, L. C. Briand, and Y. Labiche. Is Mutation an Appropriate Tool for Testing Experiments? In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, page 402–411, New York, NY, USA, 2005. Association for Computing Machinery.

[22] Ellen Arteca, Frank Tip, and Max Schäfer. Enabling Additional Parallelism in Asynchronous JavaScript Applications (Artifact). *Dagstuhl Artifacts Series*, 7(2):5:1–5:6, 2021.

[23] Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. A Framework for Automated Testing of Javascript Web Applications. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, page 571–580, New York, NY, USA, 2011. Association for Computing Machinery.

[24] Sora Bae, Joonyoung Park, and Sukyoung Ryu. Partition-Based Coverage Metrics and Type-Guided Search in Concolic Testing for JavaScript Applications. In *Proceedings of the 5th International FME Workshop on Formal Methods in Software Engineering*, FormaliSE '17, page 72–78. IEEE Press, 2017.

[25] Bruce Belson, Jason Holdsworth, Wei Xiang, and Bronson Philippa. A survey of asynchronous programming using coroutines in the internet of things and embedded systems. *ACM Trans. Embed. Comput. Syst.*, 18(3), jun 2019.

[26] Xiaoning Chang, Wensheng Dou, Jun Wei, Tao Huang, Jinhui Xie, Yuetang Deng, Jianbo Yang, and Jiaheng Yang. Race Detection for Event-Driven Node.js Applications. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 480–491, 2021.

[27] Andy Cockburn, Amy Karlson, and Benjamin B. Bederson. A review of overview+detail, zooming, and focus+context interfaces. *ACM Computing Surveys*, 41(1):2:1–2:31, 2009.

[28] Davide Corradini, Amedeo Zampieri, Michele Pasqua, and Mariano Ceccato. Restats: A Test Coverage Tool for RESTful APIs. *CoRR*, abs/2108.08209, 2021.

[29] Monika Dhok, Murali Krishna Ramanathan, and Nishant Sinha. Type-Aware Concolic Testing of JavaScript Programs. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 168–179, New York, NY, USA, 2016. Association for Computing Machinery.

[30] Danny Dig. Refactoring for asynchronous execution on mobile devices. *IEEE Software*, 32(6):52–61, 2015.

[31] ECMAScript 2021 Language Specification. `https://www.ecma-international.org/ecma-262/`, June 2021.

[32] Brian Ellis, Jeffrey Stylos, and Brad Myers. The factory pattern in api design: A usability evaluation. In *29th International Conference on Software Engineering (ICSE'07)*, pages 302–312, 2007.

[33] M. Factor, E. Farchi, Y. Lichtenstein, and Y. Malka. Testing concurrent programs: a formal evaluation of coverage criteria. In *Proceedings of the Seventh Israeli Conference on Computer Systems and Software Engineering*, pages 119–126, 1996.

[34] Amin Milani Fard and Ali Mesbah. JSNOSE: Detecting JavaScript Code Smells. In *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 116–125, 2013.

[35] Amin Milani Fard and Ali Mesbah. JavaScript: The (Un)Covered Parts. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 230–240, 2017.

[36] Amin Milani Fard, Ali Mesbah, and Eric Wohlstadter. Generating Fixtures for JavaScript Unit Testing. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ASE '15, page 190–200. IEEE Press, 2015.

[37] Keheliya Gallaba, Quinn Hanam, Ali Mesbah, and Ivan Beschastnikh. Refactoring asynchrony in javascript. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 353–363. IEEE, 2017.

[38] Satyajit Gokhale, Alexi Turcotte, and Frank Tip. Automatic migration from synchronous to asynchronous javascript apis. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–27, 2021.

[39] Liang Gong, Michael Pradel, Manu Sridharan, and Koushik Sen. DLint: Dynamically Checking Bad Coding Practices in JavaScript. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, page 94–105, New York, NY, USA, 2015. Association for Computing Machinery.

[40] Dominik Hellhake, Tobias Schmid, and Stefan Wagner. Using Data Flow-Based Coverage Criteria for Black-Box Integration Testing of Distributed Software Systems. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 420–429, 2019.

[41] Hadi Hemmati. How Effective Are Code Coverage Criteria? In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 151–156, 2015.

[42] Michael Hilton, Jonathan Bell, and Darko Marinov. *A Large-Scale Study of Test Coverage Evolution*, page 53–63. Association for Computing Machinery, New York, NY, USA, 2018.

[43] Laura Inozemtseva and Reid Holmes. Coverage is Not Strongly Correlated with Test Suite Effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 435–445, New York, NY, USA, 2014. Association for Computing Machinery.

[44] Yue Jia and Mark Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

[45] Kenneth Koster and David Kao. State coverage: A structural test adequacy criterion for behavior checking. pages 541–544, 01 2007.

[46] Thomas D. LaToza, Micky Chen, Luxi Jiang, Mengyao Zhao, and André van der Hoek. Borrowing from the crowd: A study of recombination in software design competitions. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 551–562, 2015.

[47] Paul Leger and Hiroaki Fukuda. Using continuations and aspects to tame asynchronous programming on the web. In *Companion Proceedings of the 15th International Conference on Modularity*, MODULARITY Companion 2016, page 79–82, New York, NY, USA, 2016. Association for Computing Machinery.

[48] Ding Li, James Mickens, Suman Nath, and Lenin Ravindranath. Domino: Understanding Wide-Area, Asynchronous Event Causality in Web Applications. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, page 182–188, New York, NY, USA, 2015. Association for Computing Machinery.

[49] Guodong Li, Esben Andreasen, and Indradeep Ghosh. SymJS: Automatic Symbolic Testing of JavaScript Web Applications. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 449–459, New York, NY, USA, 2014. Association for Computing Machinery.

[50] Matthew C Loring, Mark Marron, and Daan Leijen. Semantics of asynchronous javascript. In *Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages*, pages 51–62, 2017.

[51] Magnus Madsen, Ondřej Lhoták, and Frank Tip. A Model for Reasoning about JavaScript Promises. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017.

[52] Magnus Madsen, Frank Tip, and Ondřej Lhoták. Static Analysis of Event-Driven Node.Js JavaScript Applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, page 505–519, New York, NY, USA, 2015. Association for Computing Machinery.

[53] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. *Test Coverage Criteria for RESTful Web APIs*, page 15–21. Association for Computing Machinery, New York, NY, USA, 2019.

[54] Atif M. Memon, Mary Lou Soffa, and Martha E. Pollack. Coverage Criteria for GUI Testing. *SIGSOFT Softw. Eng. Notes*, 26(5):256–267, sep 2001.

[55] author. Memon, Atif. *Mutation Testing Advances: An Analysis and Survey*, volume 112 of *Advances in Computers*. Academic Press,, Cambridge, MA :, 2019.

[56] Amin Milani Fard, Mehdi Mirzaaghaei, and Ali Mesbah. Leveraging Existing Tests in Automated Test Generation for Web Applications. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, page 67–78, New York, NY, USA, 2014. Association for Computing Machinery.

[57] Shabnam Mirshokraie and Ali Mesbah. JSART: JavaScript Assertion-Based Regression Testing. In Marco Brambilla, Takehiro Tokuda, and Robert Tolksdorf, editors, *Web Engineering*, pages 238–252, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[58] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. PYTHIA: Generating test cases with oracles for JavaScript applications. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 610–615, 2013.

[59] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. Guided mutation testing for JavaScript web applications. *IEEE Transactions on Software Engineering*, 41(5):429–444, 2014.

[60] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. JSEFT: Automated Javascript Unit Test Generation. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10, 2015.

[61] Mehdi Mirzaaghaei and Ali Mesbah. DOM-Based Test Adequacy Criteria for Web Applications. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, page 71–81, New York, NY, USA, 2014. Association for Computing Machinery.

[62] Erdal Mutlu, Serdar Tasiran, and Benjamin Livshits. Detecting JavaScript Races That Matter. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 381–392, New York, NY, USA, 2015. Association for Computing Machinery.

[63] Hung Nguyen, Hung Phan, Christian Kästner, and Nguyen Tien. Exploring output-based coverage for testing PHP web applications. *Automated Software Engineering*, 26, 03 2019.

[64] Luca Ponzanelli, Simone Scalabrino, Gabriele Bavota, Andrea Mocci, Rocco Oliveto, Massimiliano Di Penta, and Michele Lanza. Supporting software developers with a holistic recommender system. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 94–105, 2017.

[65] Ohad Rau, Caleb Voss, and Vivek Sarkar. Linear Promises: Towards Safer Concurrent Programming. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming (ECOOP 2021)*, volume 194 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:27, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[66] Veselin Raychev, Martin Vechev, and Manu Sridharan. Effective Race Detection for Event-Driven Programs. *SIGPLAN Not.*, 48(10):151–166, oct 2013.

[67] Christopher Robinson-Mallett, Robert M. Hierons, and Peter Liggesmeyer. Achieving Communication Coverage in Testing. *SIGSOFT Softw. Eng. Notes*, 31(6):1–10, nov 2006.

[68] Diego Rodríguez-Baquero and Mario Linares-Vásquez. Mutode: Generic JavaScript and Node.Js Mutation Testing Tool. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, page 372–375, New York, NY, USA, 2018. Association for Computing Machinery.

[69] Nicholas Sawadsky, Gail C. Murphy, and Rahul Jiresal. Reverb: Recommending code-related web pages. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 812–821, 2013.

[70] Dominik Seifert, Michael Wan, Jane Hsu, and Benson Yeh. An Asynchronous Call Graph for JavaScript. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 29–30, 2022.

[71] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 488–498, New York, NY, USA, 2013. Association for Computing Machinery.

[72] Elena Sherman, Matthew B. Dwyer, and Sebastian Elbaum. Saturation-Based Testing of Concurrent Programs. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, page 53–62, New York, NY, USA, 2009. Association for Computing Machinery.

[73] Khashayar Etemadi Someoliayi, Sajad Jalali, Mostafa Mahdieh, and Seyed-Hassan Mirian-Hosseinabadi. Program State Coverage: A Test Coverage Metric Based on Executed Program States. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 584–588, 2019.

[74] Thodoris Sotiropoulos and Benjamin Livshits. Static Analysis for Asynchronous JavaScript Programs, 2019.

[75] Davide Spadini, Maurício Aniche, Margaret-Anne Storey, Magiel Bruntink, and Alberto Bacchelli. When testing meets code review: Why and how developers review tests. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 677–687, 2018.

[76] Haiyang Sun, Daniele Bonetta, Christian Humer, and Walter Binder. Efficient Dynamic Analysis for Node.Js. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, page 196–206, New York, NY, USA, 2018. Association for Computing Machinery.

[77] Haiyang Sun, Daniele Bonetta, Filippo Schiavio, and Walter Binder. Reasoning about the Node.Js Event Loop Using Async Graphs. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, page 61–72. IEEE Press, 2019.

[78] Haiyang Sun, Andrea Rosà, Daniele Bonetta, and Walter Binder. Automatically Assessing and Extending Code Coverage for NPM Packages. In *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*, pages 40–49, 2021.

[79] Youcheng Sun, Xiaowei Huang, Daniel Kroening, James Sharp, Matthew Hill, and Rob Ashmore. Structural Test Coverage Criteria for Deep Neural Networks. *ACM Trans. Embed. Comput. Syst.*, 18(5s), oct 2019.

[80] Juichi Takahashi, Hideharu Kojima, and Zengo Furukawa. Coverage Based Testing for Concurrent Software. In *2008 The 28th International Conference on Distributed Computing Systems Workshops*, pages 533–538, 2008.

[81] Samira Tasharofi, Michael Pradel, Yu Lin, and Ralph Johnson. Bita: Coverage-guided, automatic testing of actor programs. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 114–124, 2013.

[82] Ena Tominaga, Yoshitaka Arahori, and Katsuhiko Gondow. AwaitViz: A Visualizer of JavaScript's Async/Await Execution Order. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, SAC '19, page 2515–2524, New York, NY, USA, 2019. Association for Computing Machinery.

[83] Alexi Turcotte, Michael D. Shah, Mark W. Aldrich, and Frank Tip. DrAsync: Identifying and Visualizing Anti-Patterns in Asynchronous JavaScript. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 774–785, New York, NY, USA, 2022. Association for Computing Machinery.

[84] Cheer-Sun D. Yang, Amie L. Souter, and Lori L. Pollock. All-Du-Path Coverage for Parallel Programs. In *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '98, page 153–162, New York, NY, USA, 1998. Association for Computing Machinery.

[85] Yucheng Zhang and Ali Mesbah. Assertions are strongly correlated with test suite effectiveness. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 214–224, New York, NY, USA, 2015. Association for Computing Machinery.

[86] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software Unit Test Coverage and Adequacy. *ACM Comput. Surv.*, 29(4):366–427, dec 1997.

[87] Yunxiao Zou, Zhenyu Chen, Yunhui Zheng, Xiangyu Zhang, and Zebao Gao. Virtual DOM Coverage for Effective Testing of Dynamic Web Applications. In *Proceedings*

of the 2014 International Symposium on Software Testing and Analysis, ISSTA 2014, page 60–70, New York, NY, USA, 2014. Association for Computing Machinery.

[88] Yunxiao Zou, Chunrong Fang, Zhenyu Chen, Xiaofang Zhang, and Zhihong Zhao. A Hybrid Coverage Criterion for DynamicWeb Testing (S). In *SEKE*, 2013.

Appendix A

Supplamantary Data File: JScope Source Code

Description:

The accompanying zip file contains the source code of JScope to allow reproducibility. For any questions or contribution, please raise an issue or create a pull-request in this repository: https://github.com/MohGanji/jscope

Filename:

jscope-source-code.zip

Appendix B

Supplamantary Data File: User Study Materials

Description:

The appended zip file contains materials used in our user study. It includes pre-questionnaire, study tasks for control and experimental groups, and the pre-study tutorial. These documents are also available online at https://github.com/MohGanji/jscope-user-study.

Filename:

jscope-user-study-main.zip

Appendix C

Consent Form

Consent Form

Testing Asynchronous code in JavaScript

Principal Investigator:

Dr. Saba Alimadadi

Student Lead:

Mohammad Ganji

Co Investigators:

Dr. Frank Tip

SFU RESEARCH ETHICS

PURPOSE:
We have developed a novel technique for assessing JavaScript test quality. The purpose of this study is to evaluate the effectiveness of our tool in helping programmers. During the study, you will perform a few tasks related to understanding and debugging asynchronous JavaScript code.

STUDY PROCEDURES:
A day prior to the experiment, we will email you a copy of the consent form for you to study in your own time, and a pre-questionnaire form, indicating your experience and level of expertise in related fields.
During the period of this experiment, first you will be handed in a physical copy of the consent form to sign. We will give you a participant ID with which you will be identified, and you can hand it in if you later decide to withdraw from the experiment. Afterwards, you will be asked to perform a list of tasks and write down the results. After finishing the study, you will fill a post-questionnaire form about your opinion on the tool. At last, we will have a short interview to ask you qualitative questions about the your experience with the tool.

Consent forms will be administered in person at the time of experiment and thus signatures will be collected in person. You may also ask questions about the consent form and the procedure of the experiment. You may leave the experiment at any time that you wish. You can withdraw your volunteered participation without consequences.

LOCATION AND DEVICES
The location of the study will be at the building TASC1, room 9205. You are provided with a laptop containing the instructions, the tool, the code and the tasks.

POTENTIAL RISKS:
The risks related to this project are minimal and are no more than what you would encounter in your regular day.
The research team will abide by the latest provincial health guidelines in relation to the COVID19 pandemic.
The research team is fully vaccinated against COVID-19 and participants also need to be fully vaccinated in order to participate. You will be required to show proof of vaccination at the start of the session.

POTENTIAL BENEFITS:
The results of this study will help assess and improve our under-development tool for testing and debugging JavaScript applications. Developers can use the tool, in order to help them in testing and debugging their code.

DISSEMINATION:

The results of this study will be used to 1) enhance usability and efficiency of the tool used during the experiment, 2) publish a paper in a conference in software testing and debugging, and 3) as part of the thesis of the student lead in this study. You will also be informed about the web page on which the tool will be published at the end of the experiment.

FUTURE CONTACT:
After Completion of this experiment, the investigator team will not initiate any contact you. However, feel free to contact a co-investigator should you have any comments, questions or concerns.
You can obtain the results of this study from the student lead, Mohammad Ganji via email after the results have been processed, analyzed and documented.

CONTACT FOR COMPLAINTS:
If you have any concerns about your rights as a research participant and/or your experiences while participating in this study, please contact the SFU Office of Research Ethics at dore@sfu.ca or 778-782-6618.

CONFIDENTIALITY:
The data collected in this study is anonymized and confidentiality is maintained to best of the research team's ability.

Data will be recorded in the form of text documents and will not include any personal information about the participant. The gathered data will be put in a password protected folder in the investigators flash memory stick, and will be locked in a secure cabinet for 4 years after the date of the study.

Please note that posting to comments sections, liking or sharing on social media or other forums about this study may identify you as a participant. We therefore suggest that if this study was made available to you via a social media site or other online forums, you refrain from posting comments to protect your confidentiality

VOLUNTARY PARTICIPATION:
Your participation is voluntary. You have the right to refuse to participate in this study. If you decide to participate, you may still choose to withdraw from the study at any time during, or after the study without any negative consequences to the education, employment, or other services to which you are entitled or are presently receiving. We will give you a participant ID at the beginning of the study, which is associated to you. You will need to hand in that ID for us to be able to withdraw your results from the study. You should not feel pressured to participate because of an existing relationship with the research team.

CONSENT:

Taking part in this study is entirely up to you. You have the right to refuse to participate in this study. If you decide to take part and later change your mind, you can withdraw from the study at any time without giving a reason and without any negative impact on your grades, or employment, or any services to which you are presently entitled to receive.

- Your signature below indicates that you have received a copy of this Consent Form for your own records.
- Your signature indicates that you consent to participate in this study.
- You do not waive any of your legal rights by participating in this study.

____

Participant's Name

____

Participants Signature

____

Date