Code Coverage Criteria for Asynchronous Programs

Mohammad Ganji
Simon Fraser University
Canada
m_ganji@sfu.ca

Saba Alimadadi
Simon Fraser University
Canada
saba@sfu.ca

Frank Tip
Northeastern University
USA
f.tip@northeastern.edu

ABSTRACT

Asynchronous software often exhibits complex and error-prone behaviors that should be tested thoroughly. Code coverage has been the most popular metric to assess test suite quality. However, traditional code coverage criteria do not adequately reflect completion, interactions, and error handling of asynchronous operations.

This paper proposes novel test adequacy criteria for measuring: (i) 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 implement JScope, a tool for automatically measuring coverage according to these criteria in JavaScript applications, as an interactive plug-in for Visual Studio Code.

An evaluation of JScope on 20 JavaScript applications shows that the proposed criteria can help improve assessment of test adequacy, complementing traditional criteria. According to our investigation of 15 real GitHub issues concerned with asynchrony, the new criteria can help reveal faulty asynchronous behaviors that are untested yet 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 its effectiveness in improving programmers' ability to assess test adequacy and detect untested behavior of asynchronous code.

CCS CONCEPTS

• Software and its engineering → Software testing and debugging.

KEYWORDS

Code coverage, Dynamic analysis, Asynchronous JavaScript

ACM Reference Format:
Mohammad Ganji, Saba Alimadadi, and Frank Tip. 2023. Code Coverage Criteria for Asynchronous Programs . In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616292

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12...$15.00
https://doi.org/10.1145/3611643.3616292

1 INTRODUCTION

Asynchronous programming is extensively used for web development and is crucial for providing benefits such as non-blocking I/O, seamless and real-time user interactions, and efficient client-server communications. JavaScript is single-threaded, and asynchronous execution of potentially long-running tasks is what enables the applications to remain responsive while processing events. In recent years, JavaScript's Promises [1, Section 27.2] and `async/await` [1, Section 15.6] have rapidly become the most popular mechanisms for supporting asynchrony, supplanting the previous error-prone approach based on event-based programming and callbacks. However, understanding the flow of asynchronous execution and identifying and fixing faults remain challenging for developers [15, 47, 72, 77].

Developers typically rely on an application's tests to identify faults and verify the application's behavior. They often use code coverage criteria such as statement and branch coverage to assess the adequacy of their tests throughout the process, and to identify and address the shortcomings of existing tests in order to improve their quality [40, 81]. However, 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. 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.

This paper presents new coverage criteria for assessing the adequacy of tests in exercising the asynchronous behavior of JavaScript applications. These criteria quantify the adequacy of tests in covering eventual successful or exceptional completion of asynchronous operations, associating reactions with the outcomes of asynchronous operations, and execution of (chains of) reactions by the application's tests. These criteria target the semantics of JavaScript's promises and `async/await` features, and are meant to complement existing coverage metrics such as statement and branch coverage.

We implement our approach in a plugin for Visual Studio Code named JScope, which presents coverage results as a textual report, and through an interactive visualization. JScope automatically instruments an application's code to calculate and report coverage according to three criteria, namely *settlement coverage*, *reaction registration coverage*, and *reaction execution coverage*.

An evaluation of JScope on 20 JavaScript applications shows that the proposed 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 yet 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 and buggy behavior.

In summary, this paper makes the following contributions:

- New coverage criteria that quantify the degree to which key scenarios are exercised in asynchronous code,
- An instrumentation-based technique for measuring coverage according to these criteria,
- Implementation of the technique in an interactive VS Code extension named JScope that computes a coverage report and provides an interactive visualization [42], and
- An empirical evaluation, demonstrating the ability of the proposed criteria to identify test inadequacies in asynchronous code. We also report on a user study showing that JScope improves the effectiveness of programmers when testing and debugging asynchronous code.

2 BACKGROUND

In recent years, many programming languages have been extended with support for asynchrony. For example, Java and Dart now support Futures [5, 6], C# and Python support **async**/**await** [2, 3], and JavaScript first added promises, and then defined an **async**/**await** feature in terms of promises. These new features in JavaScript are used pervasively and pose significant new challenges for testing.

In this section, we provide an overview of promises and **async**/**await**, two features that have supplanted event-driven asynchronous programming in JavaScript. While our techniques do not apply directly to the latter, any event-driven API can be "promisified" into an equivalent promise-based one using standard library functions.

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.

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

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`:

```
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.

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. Invoking the `Promise` constructor and the `then` and **catch** methods creates a new promise p. However, if the `resolve` associated with the `Promise` constructor is invoked with an argument that evaluates to a promise p', or when a reaction that is registered by calling `then` or `catch` returns a promise p', the promise p' becomes *linked* with p. As such, if p' is resolved with a value v, then p is also resolved with v, and if p' is rejected with a value e, then so is p, and if p' remains pending, so does p. This 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 and assigns them to variables `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.

***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.

In an **async** function, **await**-expressions may be used to wait for a promise settle. If an expression e evaluates to a promise p, then an expression **await** e evaluates to v; if it is rejected with a value `err`, `err` is thrown as an exception that can be caught using `try/catch`.

```
async function f(){
  try {
    let v = await e;
    /* 1 */
  } catch(e){ /* 2 */ }
```

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

3 MOTIVATION AND CHALLENGES

This section elaborates on some challenges in identifying parts of asynchronous code that despite being covered by tests, are not tested "sufficiently" and thus may include bugs. We use real bug reports from Figures 1–2 to illustrate the challenging nature of locating bugs in asynchronous code. These challenges are intensified by developers' confidence in correctness of the code, when their tests exercise that code. While existing coverage metrics may show

```
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 1: Implementation of `RepoService.remove`.

```
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 2: Implementation of async function `visibility`.

full coverage of these code segments, these metrics are unable to examine the execution of scenarios specific to 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 assess 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 [15]. For instance, `await` expressions may be surrounded by `try/catch` for handling a failed completion of the async function. However, many applications 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.

3.1.1 Example 1. 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 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. Programmer confidence in 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. The execution of `RepoService.remove` could end before `webhook.remove` is 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).

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

3.2 Pending Asynchronous Operations

An asynchronous operation remains pending until it is "settled" successfully or through a failure, i.e., fulfilled or rejected. It is common to chain asynchronous operations to 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 further interactions with the system.

3.2.1 Example 2. Figure 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. A reaction is assigned to this operation that is invoked upon its successful completion, fulfilling the promise (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 is frozen, 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 fulfill the promise. As there is no reject reaction devised for 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 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 `rejects` the promise upon failure of `trigger_up` (line 31), which settles the promise and allows the execution to continue.

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, focusing on promise-based asynchrony. Figure 3 illustrates the life cycle of a promise: Upon creation, a promise is in the *pending* state from whence it may transition to the *settled* state when it is

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

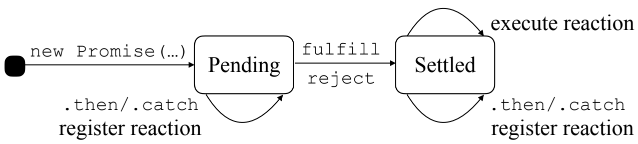

Figure 3: Illustration of the life cycle of a promise.

fulfilled or rejected. Reactions may be *registered* on a promise at any time in the pending or settled state. Such reactions will execute when the promise is settled. Our coverage criteria reflect the key steps of promise settlement, promise registration, and promise execution. It is noteworthy that none of these steps subsumes the others because: (i) settlement of a promise does not imply that reactions are registered on it, (ii) registration of a reaction of a promise does not imply that the promise will be settled (and hence that the reaction will execute), and (iii) execution of a reaction of a promise requires *both* settlement of the promise and registration of the reaction. Further, reactions may be registered on promises *after* they have settled. By proposing distinct criteria for each step, issues that result in failure to fulfill a promise and failure to register a reaction will manifest themselves through lack of coverage.

We define our criteria in terms of events in execution traces that pertain to the use of asynchronous features. We define three coverage criteria that 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 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`, `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` is invoked with an argument that is 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 (and not for linked promises).

$Reg_{fulfill}$ 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.

4.2 Coverage Criteria for Promise-Based Code

In the definitions that follow, $pid, pid', \cdots$ represent promise identifiers, $f, f', \cdots$ denote functions, and $loc, loc', \cdots$ denote source locations. Definition 1 defines a trace as a sequence of trace events (see Table 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 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 *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:*

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

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

Using Definition 2, we now define the notion of 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 (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* settlement coverage *of τ as:*

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

Our next goal is to measure 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. However, we need to consider that the rejection of a promise p may be handled by a reject reaction that is not registered directly on p itself, but at the end of a promise chain that starts with p. To capture this, we define the set of *dependent promises pid* that occur at the end of a chain of fulfill-reactions that starts at pid. In such cases, we will write $pid \rightsquigarrow pid'$, as defined below in Definition 4.

Create(*pid*, *loc*)	creation of promise *pid* at location *loc*
Fulfilled(*pid*, loc)	promise *pid* is fulfilled at location *loc*
Rejected(*pid*, 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 *pid′*
$Reg_{reject}(pid, f, loc, [pid'])$	register reject reaction f on promise *pid* at location *loc*, which may chain it to promise *pid′*
$Exec_{fulfill}(pid, f, \text{loc})$	execute fulfill reaction f on promise *pid* at location *loc*
$Exec_{reject}(pid, f, \text{loc})$	execute reject reaction f on promise *pid* at location *loc*

Table 1: Trace events for asynchronous operations.

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:*

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

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 a more convenient syntax that is highly similar to that of sequential code. 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. A *Rejected* event is emitted if an `async` function throws an exception that is not caught within its body. The code fragment following an `await` statement will be considered a fulfill reaction for the promise p returned by the `async` function, and thus a $Reg_{fulfill}$ event will be added to the trace. If the `await`-expression is in a `try/catch`, the **`catch`** statement will be the reject reaction, i.e., a Reg_{reject} event. If p is fulfilled, then an $Exec_{fulfill}$ event is emitted. Otherwise, the **`catch`** statement executes and an $Exec_{reject}$ is recorded in the trace. Assuming these trace elements, the same coverage definitions apply.

4.4 Example

Consider the following code displaying function `fun` and its tests.

```
35    function fun(inputStr) {
36        const p1 = new Promise((resolve) => {
37            resolve(JSON.parse(inputStr));
38        }).then(function f1(data) {
39            console.log(data.foo.bar)
40        }); }
41    // Tests:
42    test("T1: inputStr is valid JSON", () => {
43        fun('{"foo": {"bar": "Hello."}}'); })
44    test("T2: inputStr is not a valid JSON", () => {
45        fun('Hello.'); })
```

In order to measure `fun`'s async coverage criteria, we first obtain the following trace.

```
Create(pid_p1, L36:L38) // Start of T1
Fulfilled(pid_p1, L37:L37)
Create(pid_then, L38:L40) // Promise.then() returns a promise
Reg_fulfill(pid_p1, f1, L38:L38, pid_then)
Fulfilled(pid_then, L38:L40)
Exec_fulfill(pid_p1, f1, L38:L40)
Create(pid'_p1, L36:L38) // Start of T2
Rejected(pid'_p1, L37:L37) // Error thrown by JSON.parse() rejects p1.
Create(pid'_then, L38:L40)
Reg_fulfill(pid'_p1, f1, L38:L38, pid'_then)
```

We then identify two unique promises from the traces obtained from *T1* and *T2*. The promise created at L36:L38 achieves full (2/2) settlement coverage with a *Fulfilled* event in T1 and a *Rejected* event in *T2*. However, the promise created at L38:L40 achieves partial (1/2) settlement coverage with only one *Fulfilled* event in *T1*. Based on the observed $Reg_{fulfill}$ and Reg_{reject} events, the two promises achieve partial (1/2) and minimal (0/2) reaction registration coverage, respectively. reaction execution coverage can also be measured in a similar manner. Overall, we calculate a total of 75% settlement coverage, 25% reaction registration coverage, and 25% reaction execution coverage for function `fun`. To achieve full coverage, a reject reaction needs to be registered to both promises (e.g., adding a **`catch`** at the end of the chain). The reaction then needs to be executed through a newly-written test that rejects the promise at L38:L40.

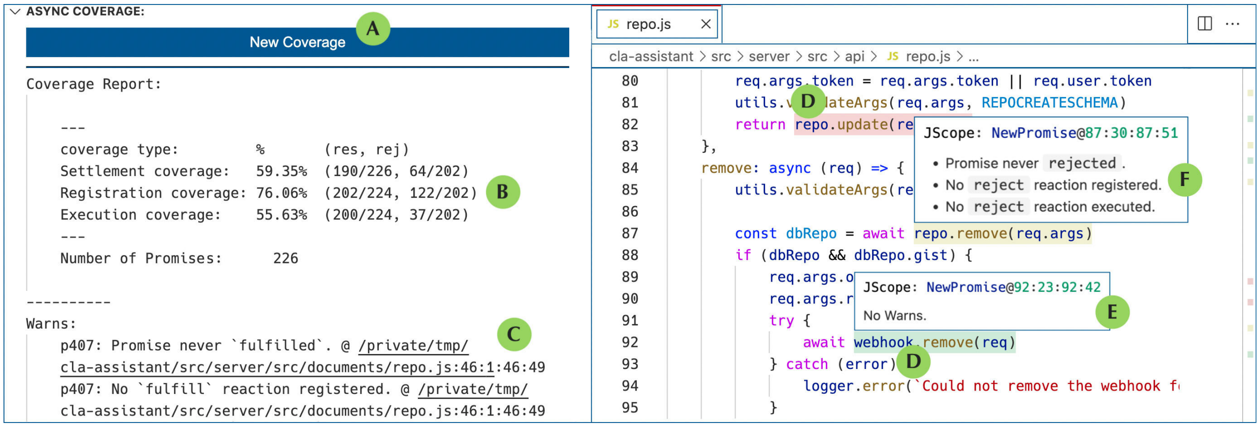

Figure 4: JScope coverage results for CLA Assistant. The open editor shows `RepoService.remove` in repo.js.

4.5 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 `e.then(···)`, where `e` is an expression that evaluates to a promise p, the promise created by the call to `then` will remain pending if p is never fulfilled causing settlement coverage to remain less than 100%, and reaction registration coverage and reaction execution coverage may remain below 100% for similar reasons. Similar scenarios arise for `async` functions.

5 APPROACH

In this section, 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, embedded within the development environment of Visual Studio Code. [4]

5.1 Instrumentation and Trace Collection

To automatically collect trace events described in Table 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.

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

5.2 Measuring Asynchronous Coverage

As promises can only be settled once, at least two tests 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 executions in a test suite will be considered as the same promise. In such cases, coverage reported by JScope should be interpreted accordingly. In particular, if full settlement coverage is reported for a promise created at location L, then this means that at least one promise created at L was fulfilled, and at least one promise created at L was rejected, meaning that both possible outcomes were observed.

We then integrate different execution paths corresponding to the same promise to locate its various settlements, registered reactions and execution of such reactions. Our analysis may miss promises in unexercised parts of code due to the incomplete nature of dynamic analysis. However, the low traditional coverage of these parts will warn the developers first. As such, async coverage is most effective when used complementary to the existing coverage criteria.

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. Similarly, to avoid unresolvable missing coverage warnings, $Reg_{fulfill}$ events are optional for `then`. 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. Our algorithm also detects promise links by locating where a promise $p1$ is fulfilled with promise $p2$, and applies all *Fulfilled* and *Rejected* events of $p2$ to $p1$ as well.

Finally, we calculate and visualize the overall async coverage by combining async coverage of all promises, and report a list of warnings for all promises' missing reactions.

Application			Objects	Traditional Coverage			Asynchronous 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	69	53	53
4. Cacache	1878	95	99	100	100	100	66	66	55
5. Github Action Merge Dependabot	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 Ref Parser	3070	256	34	88	88	78	80	92	78
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	68	45	36
13. Minipass	840	131	10	100	100	100	87	50	25
14. Grant	2756	495	29	98	97	89	58	70	56
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	51	49	43
19. Avvio	5460	180	13	94	95	91	50	56	37
20. Matched	274	30	9	96	100	78	60	100	64
AVERAGE	3385	144	39	92	91	85	64	74	57

Table 2: Summary of different coverage metrics reported by JScope and traditional coverage.

5.3 Visualizing the Asynchronous Coverage

We designed an interactive visualization integrated in VS Code, a widely used development environment, based on data gathered from a preliminary user study we conducted. Users can invoke JScope on demand (Figure 4, A) to present the results as a textual report (Figure 4, B&C) and visual cues overlayed on the code (Figure 4, D–F). JScope summarizes async coverage results in the *Coverage Overview* panel to help with overall understanding of async coverage (Figure 4, B&C). The overview includes clickable warnings, linked to the locations of their respective promises. JScope overlays relevant visual cues on the code in the editor. It highlights promises using a red-yellow-green "color spectrum" to determine their level of async coverage (Figure 4, D). As such, the promise in line 82 is marked 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 4, E&F). By leveraging the integration of focus within the context [25], we help maintain programmers' mental model of the overall program while working with individual promises.

5.4 Implementation

We used NodeProf.js [71] for instrumentation and used JavaScript Proxies to intercept the execution of built-in features for settling promises and registering their reactions [10]. We utilized programmatic APIs of Mocha [8] and Tap [9] testing frameworks for automatic execution of apps and VSCode's extension development API to integrate JScope into its editor. In our implementation of coverage criteria as per section 4, functions f that create and return a new promise object (similar to `util.promisify`) are treated specially: When a call to f is encountered, a *Create* event is generated for that call and the promise creation inside f is ignored. This custom notion of context-sensitivity [43, 79] during identifying promise-creation sites generally results in lower coverage. However, the results are more actionable as they enable detecting lack of coverage when promises are created using helper functions.

6 EVALUATION

In order for our new coverage criteria to be useful, they should be able to reveal untested asynchronous behaviors that are not detected by traditional coverage criteria. To this end, we first measure coverage according to the new criteria for 20 JavaScript applications, and study correlations with traditional coverage criteria. Next, we report on experiments that aim to determine (i) whether the new coverage criteria identify uncovered code that contains bugs, and (ii) whether using JScope can improve developers' performance when performing tasks related to assessing test adequacy and debugging.

Our evaluation targets 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 adopted a similar approach to Zhou et al. [82] and Davis et al. [27] in selecting 20 open-source JavaScript applications from GitHub. These projects used promises and/or `async/await` considerably, were accompanied by reasonable test suites, and were compatible with Graal.js [7]. They represented various sizes, domains, and architectures and the average statement coverage of the benchmark applications was 92%. We ran JScope on the subjects by automatically exercising them through their tests. We measured the results of the three asynchronous coverage metrics, and calculated statement, function, and branch coverage using Istanbul, [5] a popular JavaScript coverage

[5] https://istanbul.js.org/

	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 3: Correlation coefficients for asynchronous and traditional coverage criteria.

tool. We then examined the possible correlations of our proposed asynchronous coverage criteria with these traditional criteria.

6.1.2 Results and Discussion. The results are displayed in Table 2. The first four columns show an application's name, LOC, 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.

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 settlement, reaction registration, and reaction 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. We used the Kendall rank correlation coefficient, which does not assume a normal distribution. The results, depicted in Table 3, show no strong correlations between traditional and asynchronous coverage metrics. This indicates that traditional coverage metrics are not necessarily equipped for identifying 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.

Overall, while the high traditional coverage scores raise confidence in sufficient testing of the code, they are not equipped with identifying shortcomings of the tests in asynchronous scenarios. For instance, while 92% of the statements 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. A main application of coverage criteria is identifying code segments that may contain bugs due to insufficient coverage, which can be helpful during debugging. As such, given a set of known bugs, we investigated (1) if traditional coverage criteria raise warnings about inadequate testing of faulty asynchronous code and (2) if JScope could have helped discover these bugs.

6.2.1 Experimental Design and Procedure. We searched the repositories of the projects in Table 2 for issues that 1) involved promises and/or async/await, 2) were closed with the fixes linked to the relevant commits, and 3) had complete statement coverage in the

	Commit	App	Category	Settlement	Registration	Execution	Statement
1.	#f56491a	express-http-proxy	Unhandled Exp.	63	96	74	95
2.	#d902776	cla-assistant	Unhandled Exp.	58	75	55	94
3.	#8ff7de7	streamroller	Unhandled Exp.	60	81	67	100
4.	#8e94a60	eslint_d.js	Unhandled Exp.	70	65	65	89
5.	#6bcf8ca	checkfire	Unhandled Exp.	40	55	40	-
6.	#fff6640	postgres	Unhandled Exp.	71	83	60	91
7.	#2fc9693	haraka	Unhandled Exp.	25	33	33	-
8.	#e5615da	ioredis	Unhandled Exp.	76	69	55	95
9.	#146bb3b	install	Unhandled Exp.	50	100	62	98
10.	#0dfff52	json-schema-ref-parser	Unhandled Exp.	80	91	81	94
11.	#cbcdfc6	socketcluster-server	Unhandled Exp.	63	50	43	79
12.	#dfbafbf	clamscan	Pending Op.	58	89	62	40
13.	#48a2ddf	cla-assistant	Broken Chain	58	75	55	94
14.	#b0a86d4	avvio	Broken Chain	38	58	38	93
15.	#68342f8	libnpmteam	Unnecessary Async.	40	83	61	100

Table 4: Asynchrony-related JavaScript issues from Github.

version before the fix. We found seven bugs in six of the repositories. We expanded our search to real bugs from other projects on GitHub that met our requirements. We selected a total of 15 bugs. 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 4 displays the results. 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 async coverage numbers before the fix. The last column shows statement coverage before the fix, reported by Istanbul.js.

Overall, JScope reported insufficient coverage and relevant warnings for all bugs, addressing which could have helped detect and fix the bugs before deployment. Statement coverage, however, showed no sign of warning or insufficient testing for any of the bugs or their relevant code segments. 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 [15]. 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 [4] for higher performance and has >30k weekly downloads on the NPM registry (Table 4, row 4). It caches a single linter object to reduce overhead. Line 272 of the left code snippet in Figure 5-A shows how the async function `getCache` is invoked to asynchronously retrieve a cached ESLint linter object from a given path. The program, using **`await`**, waits until this promise fulfills. A bug was reported in this method despite the full coverage of this code segment by the tests, as depicted by the green markings by the line numbers. It 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 5-A, right snippet, lines 273–278).

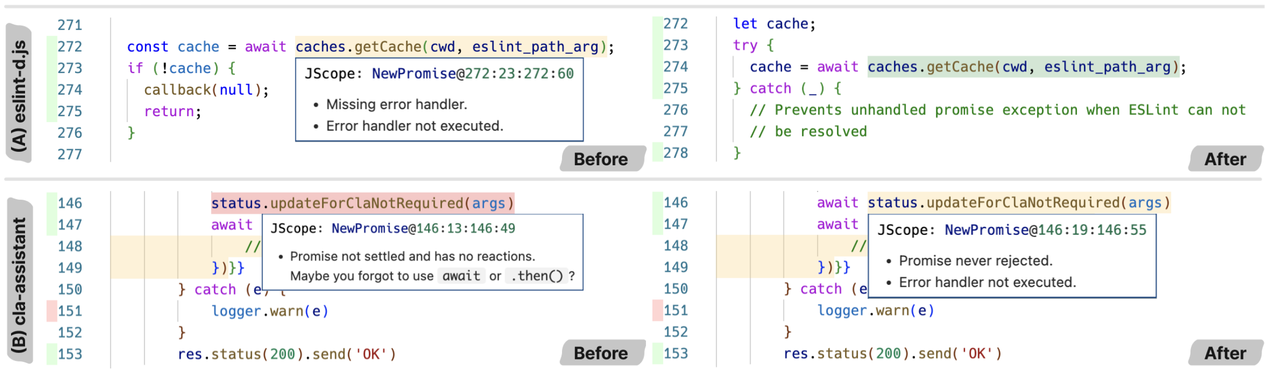

Figure 5: JScope results (highlights and warnings overlayed on code) vs. Istanbul results (markings by the line numbers).

A corresponding test was also added to the test suite that 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 5-A). Having had access to JScope during testing could have helped reveal this bug before production.

Our results in Table 4 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. 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. [6]

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 [47]. 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).

(Example B) Row 13 of Table 4 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 5-B). Upon invocation of the **async** function `updateForClaNotRequired` (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).

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

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 but were unable to reproduce it. [7]

JScope reported low async coverage for the promise on line 146 before the fix (Figure 5-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).

Pending Operations. If not explicitly settled, asynchronous operations remain pending, causing nontermination or memory leaks. Such problems 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 4, 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 broken promise chains. While generally less severe, JScope warns about their missing rejections.

Overall, async coverage criteria can effectively expose test inadequacies related to asynchrony that are not detected 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 Asynchronous Coverage 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.

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

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

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	Identifying the underlying causes of a failure
T3.B	Finding the fix to the failure

Table 5: Tasks used in the user study.

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 with 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 have two continuous *Dependent Variables* that represent the developers' performance in completing the tasks: task completion duration (seconds) and accuracy (%).

Participants. We sent out recruitment emails to graduate students' mailing lists. From the replies, we selected the ones who met our knowledge requirements of JavaScript development and testing. The majority of our participants had a medium-level expertise in JavaScript programming, and familiarity with testing. 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 version of the `body.js` file from Node Fetch, [8] a library implementing browsers' `window.fetch` in Node.js. For the debugging task, we chose a fixed bug from Docusaurus, a website building application. [9] 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 5). T1 and T2 were designed to assess effectiveness of tool in helping programmers identify well-tested and insufficiently tested functions and promises. T3 was designed to investigate the usefulness of *Tool* in helping participants identify the underlying causes of the bug (T3.A) and propose a fix (T3.B).

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. All participants signed a consent formed prior to starting the study.

Training. The participants were given refresher tutorials on main concepts of asynchronous JavaScript, coverage , and Istanbul, to ensure consistency in the knowledge required for completing the tasks.. The experimental group also received a tutorial on using JScope. Both groups were given some time to familiarize themselves with the tools and the setup of the experiment.

Task Completion. Next, the participants started performing the tasks (Table 5). 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 responses later.

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 experimental group spent an average of 12:43, 7.58, and 7:54 minutes for completing T1, T2, and T3, respectively. The control group spent 6:42, 11:58, 9:12 minutes for performing the same tasks, on average. 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 unfamiliarity with JScope, and they still achieved an average of 33% higher accuracy for T1. 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 tested portions of the code that may contain bugs. Using JScope helped the experimental group in debugging to achieve more accurate answers while spending less time locating the underlying causes 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).

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.

[8] https://github.com/node-fetch/node-fetch
[9] https://github.com/facebook/Docusaurus/issues/238

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. Our analysis for the applications in Table 2 indicates 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). The slowdown is similar to other instrumentation-based dynamic analyses for JavaScript [15, 37, 72].

6.5 Threats to Validity

There are threats pertaining to the representativeness of our participants, benchmark projects, or issues. We addressed these by randomly selecting participants who met the minimum experience requirements and projects of different sizes from different domains that met 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 detailed rubrics for grading the accuracy of the results prior to the study to address a similar bias in measuring participants' accuracy. We tried to alleviate the impact of expertise level in our study by balancing the participants' expertise levels based on their responses to our pre-questionnaire. We made JScope and our experimental data available to allow reproducibility.

7 RELATED WORK

While being the most prominent test quality assessment technique [83], **code coverage criteria** have always been under scrutiny about their effectiveness [31, 38–40]. The generic nature of traditional coverage criteria has led to the emergence of various domain-specific coverage criteria [16, 44, 51, 68, 69, 74]. Several coverage metrics have been introduced using data-flow to target concurrency in actor-based [75], concurrent [67, 80], and distributed programs [62]. Researchers have proposed novel criteria for dynamic web applications [49, 58, 84, 85], or loosely typed nature of JavaScript [22], or DOM elements [56]. None of these techniques, however, address the asynchronous execution and its respective challenges.

Event-dependent and asynchronous callbacks form a majority of untested code in JavaScript [31]. Prior work has used **static analysis** to model event-driven JavaScript [47, 48, 70]. Other work has focused on constructing promise graphs that express the relationships between promises and relevant code [47] and detecting promise anti-patterns based on promise graphs [15]. to identify performance-related anti-patterns involving promises [77] . Arteca et al. [20] present a refactoring for enabling additional concurrency by splitting and moving `await` expressions, and Gokhale et al. [36] present a refactoring for migrating applications from the use of synchronous APIs to equivalent asynchronous APIs. Moreover, **dynamic analysis** has been popularly used in JavaScript [13, 14, 45, 60, 76] to address the imprecision of static analysis in analyzing JavaScript's inherent dynamism [17]. Much research in this area targets understanding, debugging, and testing techniques for programs in general[15, 24, 30, 37, 57, 64, 72] [21, 28, 32, 46, 52, 53, 73], and more recently for asynchronous JavaScript in particular [15] [72][64]. A long line of research projects has focused on the detection and remediation of event races [11, 12, 29, 61], concurrency bugs [78], and schedule fuzzers for event-driven programs [26]. 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. **Visualization** has been effectively used for comprehension and modeling event-driven and asynchronous programs [13–15, 76, 77]. Similar to Seifert et al. [64], we leveraged editor integration to facilitate the comprehension of asynchronous coverage through an interactive interface.

Code coverage is crucial in evaluating the effectiveness of **test generation** techniques such as feedback-directed random testing [19, 59, 65], dynamic symbolic execution [23, 35, 66], and search-based and evolutionary techniques [33, 34]. Nessie [19] is a feedback-directed test generation tool for JavaScript that targets event-driven asynchrony. Event-driven asynchrony is rapidly being supplanted by promises and `async`/`await` because these features lead to a more readable and less error-prone code. However, Nessie does not provide special support for promises and `async`/`await`.

Mutation testing is also used as an alternative approach for measuring test quality [41, 50]. Despite their effectiveness, mutation testing for JavaScript is typically very costly, and has yet to gain the popularity of code coverage [18, 54, 55, 63].

8 CONCLUDING REMARKS

In this paper, we proposed a set of coverage criteria for assessing the adequacy of tests with respect to asynchronous program behavior. We designed an interactive visualization and implemented a tool to allow programmers to view async coverage results in a typical development environment. The results of our evaluation showed that async coverage metrics are complementary to traditional metrics and can help programmers detect insufficiencies of tests and related bugs in asynchronous code where traditional metrics cannot. Our user experiment also demonstrated that our tool helps improve developers' performance in tasks related to assessing test quality and debugging of asynchronous code.

The coverage criteria presented in this paper are designed for JavaScript. As was pointed out in section 2, similar features have been added to various programming languages [2, 3, 5, 6], and adapting the coverage criteria to these languages is an interesting future direction. Another avenue for future work is the development of test generation techniques that aim to improve asynchronous coverage. For example, one could imagine extending Nessie [19] to register reactions on promises returned by function calls in previously generated tests.

9 DATA AVAILABILITY

JScope and our experimental data are publicly available [42].

ACKNOWLEDGMENTS

This work was supported in part by an NSERC Discovery Grant and National Science Foundation grant CCF-1907727. We are grateful to the participants of our controlled experiments.

REFERENCES

[1] 2021. ECMAScript 2021 Language Specification. https://www.ecma-international.org/ecma-262/.

[2] 2022. Asynchronous programming with Async and Await. https://docs.microsoft.com/en-us/dotnet/visual-basic/programming-guide/concepts/async/ Accessed Aug-2022.

[3] 2022. Awaitables, python documentation. https://docs.python.org/3/library/asyncio-task.html#awaitables Accessed Jan-2023.

[4] 2022. Eslint: Pluggable JavaScript Linter. https://www.npmjs.com/package/eslint Accessed Jan-2023.

[5] 2022. Future (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html Accessed Aug-2022.

[6] 2022. Future<T> class. https://api.dart.dev/stable/2.18.7/dart-async/Future-class.html Accessed Jan-2023.

[7] 2022. Graalvm Node.js Runtime. https://istanbul.js.org/ Accessed Aug-2023.

[8] 2022. Mocha, the fun, simple, flexible JavaScript test framework. https://mochajs.org Accessed Sep-2022.

[9] 2022. Node Tap. https://node-tap.org Accessed Sep-2022.

[10] 2022. Proxy - JavaScript. https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Proxy Accessed Sep-2022.

[11] Christoffer Quist Adamsen, Anders Møller, Saba Alimadadi, and Frank Tip. 2018. Practical AJAX race detection for JavaScript web applications. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 38–48.

[12] Christoffer Quist Adamsen, Anders Møller, Rezwana Karim, Manu Sridharan, Frank Tip, and Koushik Sen. 2017. Repairing event race errors by controlling nondeterminism. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 289–299.

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

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

[15] Saba Alimadadi, Di Zhong, Magnus Madsen, and Frank Tip. 2018. Finding Broken Promises in Asynchronous JavaScript Programs. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 162 (oct 2018), 26 pages.

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

[17] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. 2017. A Survey of Dynamic Analysis and Test Generation for JavaScript. *ACM Comput. Surv.* 50, 5, Article 66 (sep 2017), 36 pages.

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

[19] Ellen Arteca, Sebastian Harner, Michael Pradel, and Frank Tip. 2022. Nessie: Automatically Testing JavaScript APIs with Asynchronous Callbacks. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1494–1505.

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

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

[22] Sora Bae, Joonyoung Park, and Sukyoung Ryu. 2017. 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* (Buenos Aires, Argentina) *(FormaliSE '17)*. IEEE Press, 72–78.

[23] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2006. EXE: automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, October 30 - November 3, 2006*, Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati (Eds.). ACM, 322–335.

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

[25] Andy Cockburn, Amy Karlson, and Benjamin B. Bederson. 2009. A review of overview+detail, zooming, and focus+context interfaces. *Comput. Surveys* 41, 1, Article 2 (2009), 31 pages.

[26] James C. Davis, Arun Thekumparampil, and Dongyoon Lee. 2017. Node.fz: Fuzzing the Server-Side Event-Driven Architecture. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, Gustavo Alonso, Ricardo Bianchini, and Marko Vukolic (Eds.). ACM, 145–160.

[27] James C. Davis, Eric R. Williamson, and Dongyoon Lee. 2018. A Sense of Time for JavaScript and Node.js: First-Class Timeouts as a Cure for Event Handler Poisoning. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 343–359.

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

[29] André Takeshi Endo and Anders Møller. 2020. NodeRacer: Event Race Detection for Node.js Applications. In *13th IEEE International Conference on Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020*. IEEE, 120–130.

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

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

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

[33] Gordon Fraser and Andrea Arcuri. 2011. Evolutionary Generation of Whole Test Suites. In *Proceedings of the 11th International Conference on Quality Software, QSIC 2011, Madrid, Spain, July 13-14, 2011*, Manuel Núñez, Robert M. Hierons, and Mercedes G. Merayo (Eds.). IEEE Computer Society, 31–40.

[34] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, Tibor Gyimóthy and Andreas Zeller (Eds.). ACM, 416–419.

[35] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, Vivek Sarkar and Mary W. Hall (Eds.). ACM, 213–223.

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

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

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

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

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

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

[42] JScope 2023. JScope. https://github.com/SEatSFU/JScope.

[43] Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. 2014. JSAI: A Static Analysis Platform for JavaScript. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 121–132.

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

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

[46] Guodong Li, Esben Andreasen, and Indradeep Ghosh. 2014. SymJS: Automatic Symbolic Testing of JavaScript Web Applications. In *Proceedings of the 22nd ACM*

SIGSOFT International Symposium on Foundations of Software Engineering (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 449–459.

[47] Magnus Madsen, Ondřej Lhoták, and Frank Tip. 2017. A Model for Reasoning about JavaScript Promises. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 86 (oct 2017), 24 pages.

[48] Magnus Madsen, Frank Tip, and Ondřej Lhoták. 2015. 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* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 505–519.

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

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

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

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

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

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

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

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

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

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

[59] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*. IEEE Computer Society, 75–84.

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

[61] Veselin Raychev, Martin T. Vechev, and Manu Sridharan. 2013. Effective race detection for event-driven programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes (Eds.). ACM, 151–166.

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

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

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

[65] Marija Selakovic, Michael Pradel, Rezwana Karim, and Frank Tip. 2018. Test generation for higher-order functions in dynamic languages. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 161:1–161:27.

[66] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, Michel Wermelinger and Harald C. Gall (Eds.). ACM, 263–272.

[67] Elena Sherman, Matthew B. Dwyer, and Sebastian Elbaum. 2009. 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* (Amsterdam, The Netherlands) *(ESEC/FSE '09)*. Association for Computing Machinery, New York, NY, USA, 53–62.

[68] S. Sinha and M.J. Harrold. 1999. Criteria for testing exception-handling constructs in Java programs. In *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*. 265–274.

[69] Khashayar Etemadi Someoliayi, Sajad Jalali, Mostafa Mahdieh, and Seyed-Hassan Mirian-Hosseinabadi. 2019. 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)*. 584–588.

[70] Thodoris Sotiropoulos and Benjamin Livshits. 2019. Static Analysis for Asynchronous JavaScript Programs. In *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom (LIPIcs, Vol. 134)*, Alastair F. Donaldson (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 8:1–8:30.

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

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

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

[74] Youcheng Sun, Xiaowei Huang, Daniel Kroening, James Sharp, Matthew Hill, and Rob Ashmore. 2019. Structural Test Coverage Criteria for Deep Neural Networks. *ACM Trans. Embed. Comput. Syst.* 18, 5s, Article 94 (oct 2019), 23 pages.

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

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

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

[78] Jie Wang, Wensheng Dou, Yu Gao, Chushu Gao, Feng Qin, Kang Yin, and Jun Wei. 2017. A comprehensive study on real world concurrency bugs in Node.js. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.). IEEE Computer Society, 520–531.

[79] Shiyi Wei and Barbara G. Ryder. 2015. Adaptive Context-sensitive Analysis for JavaScript. In *29th European Conference on Object-Oriented Programming (ECOOP 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 37)*, John Tang Boyland (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 712–734.

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

[81] Yucheng Zhang and Ali Mesbah. 2015. Assertions Are Strongly Correlated with Test Suite Effectiveness. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 214–224.

[82] Jingyao Zhou, Lei Xu, Gongzheng Lu, Weifeng Zhang, and Xiangyu Zhang. 2023. NodeRT: Detecting Races in Node.Js Applications Practically. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1332–1344.

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

[84] Yunxiao Zou, Zhenyu Chen, Yunhui Zheng, Xiangyu Zhang, and Zebao Gao. 2014. Virtual DOM Coverage for Effective Testing of Dynamic Web Applications. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 60–70.

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