TÉCNICO LISBOA

Practical Detection of JavaScript Concurrency Bugs using Callback Graphs

Bernardo Vasconcelos Freitas de Almeida Furet

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. João Fernando Peixoto Ferreira

Examination Committee

Chairperson: Prof. Alberto Manuel Rodrigues da Silva
Supervisor: Prof. João Fernando Peixoto Ferreira
Member of the Committee: Prof. António Manuel Ferreira Rito da Silva

November 2020

Acknowledgments

I would like to dedicate this Thesis to my girlfriend Camila Barreto and my mother Teresa Freitas, for all the support and motivation throughout the years that led me to this point and also for never stop believing in me. Thank you for everything.

Also a special thank you to my friends Margarida Costa and Paulo Alves, for all the conversations that made us three go through this process together, helping each other.

Thank you too to my brother Henrique Furet, my grandmother Maria José Freitas, my girlfriend's parents Alexandra Barreto and Aníbal Manuel, my father Nuno Furet and my friends Seomara Félix and António Charana for their concern and support.

I would also like to acknowledge my dissertation supervisor Prof. João Ferreira for all the insight, knowledge and support provided that made this Thesis possible. Thank you, professor, for your kindness and friendship.

Last but not least, to all other professors that believed in me, gave me opportunities to develop my abilities (they know who they are) and made me grow and improve as an IT and Computer Engineer student.

A huge, sincere thank you to each and every one of you,

Bernardo Furet

Abstract

JavaScript is becoming an increasingly popular programming language [1, 2] that works in both client-side, within the browser, and server-side, through Node.js. One of its most important and widely used features is, despite being single threaded, the capability to schedule operations, thus emulating an asynchronous behaviour. These operations are non-blocking, meaning that other code can be ran while the program is waiting for responses, event triggers or just for a timer to run out. This asynchronous nature gives flexibility to developers, but it can also lead to concurrency bugs, since the order of execution might be, sometimes, non-deterministic.

The purpose of this thesis is to explore the recently introduced Callback Graph model [3] to automatically detect concurrency bugs in JavaScript. We focus on two specific cases of concurrency issues, the cases of Broken Promise and Unexpected Execution Order bugs, and we propose the design of a solution that detects those issues automatically. Our implementation is built on top of Async-TAJS [4], a static analysis tool for JavaScript code that implements the Callback Graph model. We evaluate our solution on a benchmark of asynchronous JavaScript code. Our results show that the proposed solution can effectively detect the two cases of bugs considered. As an additional contribution, we created a new dataset of 74 code examples of asynchronous JavaScript code that developers can use to test their analysis tools.

Keywords

JavaScript, Asynchronous, Data race, Static analysis, Concurrency

Resumo

JavaScript é uma linguagem que, ultimamente, tem ganho imensa popularidade [1, 2], operando quer em client-side, através dos browsers, quer nos servidores, a partir de Node.js. Uma das suas funcionalidades mais usadas e importantes é, apesar de funcionar principalmente numa única thread, a sua assincronia, gerindo e adiando operações que estão pendentes. Estas operações tornam-se não bloqueantes, permitindo a execução de outro código, enquanto o programa espera pela sua conclusão, por eventos ou por timers. Esta natureza assíncrona proporciona flexibilidade aos programadores, mas, por outro lado, pode levar a bugs de concorrência, dado que a ordem de execução das operações pode ser, nalguns casos, indeterminada.

Esta tese visa explorar o modelo de Callback Graph [3], recentemente introduzido, para detectar problemas de assincronia em código JavaScript. Focamo-nos em duas categorias específicas de bugs de concorrência, Broken Promise e Unexpected Execution Order, propondo uma solução para os detectar automaticamente. A nossa implementação usa a ferramenta Async-TAJS [4], que faz análise estática de código JavaScript e implementa o modelo de Callback Graph. Para avaliarmos a nossa solução, usamos uma benchmark de código JavaScript assíncrono. Os nossos resultados mostram que a solução proposta consegue detectar os dois casos particulares considerados. Como contribuição adicional, criámos um novo conjunto de 74 ficheiros de código JavaScript assíncrono que os programadores podem usar para testar as suas próprias soluções.

Palavras Chave

JavaScript, Assincronia, Data race, Análise estática, Concorrência

Contents

1 Introduction 2
1.1 Objectives . . . 4
1.2 Contributions . . . 5
1.3 Thesis Outline . . . 5

2 Motivating Examples 6
2.1 Broken Promise Bugs . . . 7
2.2 Unexpected Execution Order Bugs . . . 8
2.3 Summary . . . 12

3 Background and Related Work 13
3.1 The Event Loop . . . 15
3.2 Static vs Dynamic Analysis . . . 16
3.3 Graph Models for JavaScript Analysis . . . 18
3.3.1 Callback Graph . . . 18
3.3.2 Promise Graph & PromiseKeeper . . . 21
3.3.3 Async Graph & AsyncG . . . 25
3.3.4 Call graphs . . . 31

4 Approach 35
4.1 Broken Promise Bugs . . . 37
4.2 Unexpected Execution Order Bugs . . . 38

5 Implementation 42
5.1 TAJS – Type Analyzer for JavaScript . . . 43
5.2 Broken Promise Bugs . . . 43
5.3 Unexpected Execution Order Bugs . . . 47

6 Evaluation 55
6.1 Datasets of JavaScript Asynchronous Bugs . . . 57
6.2 Broken Promise Bugs . . . 59

6.3 Unexpected Execution Order Bugs 61

7 Conclusion **66**

7.1 System Limitations 67

7.2 Future Work 70

List of Figures

1.1 Comparison between asynchronous calls without using Promises (on the left) and with using Promises (on the right). 3

1.2 Example of a subtle data race issue. 4

2.1 The example on the left shows a data race issue: both reactions are registered on the original promise. Thus, regardless the order in which they resolve, the data they return will never depend flow from one to the other. On the right, the example is corrected: the reaction that increments is registered first and, then, the reaction that checks for the value is registered on the promise returned by the reaction that increments the value. Thus keeping the order of execution: first the increment, then the check. 8

2.2 The same code is represented in both sides, in terms of execution order. On the left, the developer may mistakenly think that `o.print` will be defined when its call arrives. On the right, the code was simply organized to map directly to the execution order. `setTimeout`, an asynchronous function, will defer the execution of its callback to after the synchronous code is complete. It is made more obvious on the right side that this execution will not work as expected. 9

2.3 Example of the Unexpected Execution Order bug. `validateString` will be called after `storeString` without assuring the latter has completed its task. 10

2.4 The snippet on the left fixes the problem by making use of the `storeString` signature that allows it to receive a callback as a second argument, to be called once `storeString` completes. By calling `validateString` inside that callback, it is ensured `validateString` will only be executed when we are expecting: after `storeString` completes. The snippet on the right fixes the issue by using the Promise returned by `storeString`, registering a new callback onto it. Calling `validateString` on that callback only, ensures it will be executed after `validateString` completes. 11

3.1 Checking code in order to find any possible values of v at the time of line 2 can be useful to exclude one of the branches, cutting down unnecessary verifications. Likewise, some optimization can be made in order to check that a, at line 9, will only have the value 1, thus entering the then branch of the if and there is no need to waste time checking the else branch. This strategy is similar to what some compilers do [5]. 17
3.2 Callback Graph model graphical representation of the code snippet displayed in Fig. 1.1. . 20
3.3 Example of a Promise Graph. Taken and adapted from [6]. 22
3.4 Example of a Promise Graph as it is outputted by the PromiseKeeper. Taken from [7]. . . 25
3.5 The execution order of each callback is different from the registration order. Taken from [8]. 26
3.6 Example of an Async Graph for buggy code. Taken from [8]. 29

4.1 Example of the Unexpected Execution Order bug, similar to the one in Fig.2.3, but with the both the storing and the validation of the string being asynchronous, leading to non-deterministic results. 39

5.1 Callback Graph model applied to the examples provided on Fig. 2.1, respectively. 44
5.2 A Broken Promise bug that will always result in an uncaught exception is shown on the left with the respective graph on the right. 45
5.3 Broken Promise verification for the left snippet presented in Fig. 2.1. 46
5.4 Callback Graph model for the code snippet displayed in Fig. 2.3. 49
5.5 Callback Graph model for the code snippet displayed in Fig. 4.1. 50
5.6 Enhanced model to reason about Unexpected Execution Order issues for the code snippet displayed in Fig. 2.3. 51
5.7 Enhanced model to reason about Unexpected Execution Order issues for the corrected version of code snippet displayed in Fig. 2.3. 53

6.1 Enhanced model to reason about Unexpected Execution Order issues for the code snippet displayed in Fig. 4.1. 62

7.1 The same code from Fig. 4.1, but with explicit callbacks instead of default reactions, on the left, with the respective graph on the right. 69
7.2 Example of manually tagging the reaction registration that forks the Promise to avoid being tagged as a Broken Promise bug. 71

List of Tables

6.1 Table illustrating which C benchmarks use which asynchronous APIs. 58

6.2 Table illustrating which I benchmarks use which asynchronous APIs. 59

6.3 Table illustrating which O benchmarks use which asynchronous APIs. 59

1

Introduction

Contents

1.1 Objectives 4
1.2 Contributions 5
1.3 Thesis Outline 5

JavaScript is currently one of the most popular programming languages [1, 2], being used for both the front-end of web applications, especially via frameworks, and for the back-end, through Node.js, allowing complete JavaScript application servers. At the root of this popularity and flexibility is JavaScript's support for asynchronous programs, which are typically written in an event-driven style that heavily relies on callbacks that are invoked when an asynchronously executed operation has completed.

Given its asynchronous nature and the event-driven paradigm JavaScript offers, the order of execution might be, sometimes, non-deterministic and unexpected. This allows for responsive applications, but it also introduces complexity and non-expected behaviour. For instance, if a program makes an asynchronous call that involves communication with a remote server, while the program waits for the server to respond back, other code can continue its execution. Despite preventing the application from stopping while waiting for external responses, it also contributes to certain elements to be deferred instead of being executed immediately, possibly introducing race conditions if there are dependencies between data that comes from the said external sources and code that kept running.

In order to facilitate the construction of asynchronous JavaScript code, the concept of Promise [9] was introduced, with the ECMAScript 2015 [10, 11]. A Promise provides an encapsulation for the result of an asynchronous operation. These operations were already part of JavaScript, but Promises made it easier to deal with and handle this kind of operations, due to the layer of abstraction they provide. It makes the code more readable and organized, reducing the number of bugs, since it provides a way to write asynchronous code that executes in a top-down manner, through method chaining, as opposed to the more traditional way (see Fig. 1.1), where people would fall into the well-known callback hell [12].

Promises were well-received within the JavaScript community and are widely used. In fact, some libraries and frameworks already had their own implementations of this concept (e.g., jQuery's Deferred [13–15] and Bluebird [16]). The popularity of the Promise concept led to its standardization and, consequently, its addition to the JavaScript specification. Today, about 75% of JavaScript frameworks use Promises [17].

```
1  try {
2    doTask1( p1, r1 => {
3      doTask2( r1, r2 => {
4        doTask3( r2, r3 => {
5          // etc..
6        } );
7      } );
8    } );
9  } catch ( e ) {
10   // Handle error.
11 }
```

```
1  doTask1( p1 )
2    .then( doTask2 )
3    .then( doTask3 )
4    .then( r3 => /*etc..*/ )
5    .catch( e => /*Handle error.*/ )
6  ;
```

Figure 1.1: Comparison between asynchronous calls without using Promises (on the left) and with using Promises (on the right).

```
1  client.connect( PORT );
2
3  client.send( message );
```

Figure 1.2: Example of a subtle data race issue.

Promises make the programmer's job easier and enable code that is more elegant, but asynchronous programming still poses some challenges. As pointed out by Sotiropoulos and Livshits [3], recent studies showed that concurrency bugs found in JavaScript programs are sometimes caused by asynchrony. Consider the code presented in Fig. 1.2. This example shows a common case where the user wants to send a message to a client. First, the user needs to connect with the client and then send the message. The problem is that connecting to the client (an external entity) is usually an asynchronous operation. So when the call to `client.send` is made, the user might not yet be connected to the client. To complicate the matter even more, since sending a message also requires contacting the remote entity, `client.send` is, most likely, asynchronous as well. So, the order in which each operation will actually be executed is not deterministic. When testing this kind of programs, the tests might not accuse any bug, because the operations conveniently happened in an order that did not raise any errors.

The main reason why these problems occur is the complex mechanism that supports asynchronous code in JavaScript. This mechanism, responsible for the reception and handling of events and the scheduling and execution of asynchronous operations (Promises, timers, async I/O, etc.), is the event loop [18–20]. As the name says, it consists of a loop, constantly executing several different types of operations, by phases. Each phase is responsible for handling a specific type of operations. These are called the macrotasks. Between each macrotask, another kind of operations, the so-called microtasks, can be executed. This contributes to a complex structure underneath JavaScript that can lead to data race bugs, or, more broadly, concurrency bugs.

1.1 Objectives

Since it is easy to introduce subtle errors when writing asynchronous code in JavaScript, it is important to develop automatic methods and tools that detect those errors. The goal of this work is to explore the recently introduced Callback Graph model [3] to automatically detect concurrency bugs in JavaScript.

We focus on two specific cases of concurrency issues, the cases of Broken Promise and Unexpected Execution Order bugs, and we propose the design of a solution that detects those issues automatically, so that they can be reported to the developer. Our solution is built on top of Async-TAJS [4], a static analysis tool for JavaScript code that implements the Callback Graph model.

1.2 Contributions

We expect to be able to contribute with new knowledge that might be useful for the software engineering community. In particular, the sub-community focused on program analysis and bug finding.

Specifically, we contribute with:

- Three datasets: One, more general, to exercise asynchronous behaviour from multiple APIs. The other two are focused on each of the motivating examples. These datasets contain small programs with asynchronous code that vary from simple to edge cases and even more complex patterns.
- Methods based on the Callback Graph to automatically detect Broken Promise issues.
- New model based on the Callback Graph to automatically detect Unexpected Execution Order issues.
- Extension to the Async-TAJS tool that reports Broken Promise issues to the user.

1.3 Thesis Outline

This document will start by presenting two cases of concurrency bugs that pose as motivating examples for the development of the tool.

After that, a thorough description about concepts that are involved with these bugs and JavaScript are presented, followed by a description of the model we use to solve these issues, along several other works dedicated to concurrency issues and JavaScript analysis.

The fourth and fifth chapters will dive into the approaches taken to tackle the concurrency bugs that serve as motivating examples, as well as the implementation of those approaches, respectively.

The evaluation of the results gathered by the tool are analyzed in the following chapter.

Finally, the conclusion will address the global vision of the work and its results, as well as describing the obstacles faced and what can be done as future work.

2

Motivating Examples

Contents

2.1 Broken Promise Bugs 7
2.2 Unexpected Execution Order Bugs 8
2.3 Summary 12

There are several common concurrency bugs, all related to the asynchronous nature of JavaScript. We will be focusing on two of them: (a) Broken Promise bugs and (b) Unexpected Execution Order bugs.

The former has to do with the flow of the data through Promise chains. The latter has to do more with shared data between different execution contexts created by the scheduling of operations for different, unintended execution timings. The case of the Broken Promise can also create a problem of Unexpected Execution Order, for very specific cases, if there is data being accessed by multiple callbacks that were not registered in the correct order.

In this chapter, we describe these two types of bugs.

2.1 Broken Promise Bugs

Broken Promise bugs are also known as Broken Promise Chain bugs. This type of bug occurs when the Promise reactions are not registered in the same chain, but are instead accidentally registered to the same root Promise. This creates one new chain per new registered reaction, rather than keeping the same flow.

Promises do not have a particularly different API, so programmers, unaware of these details, can easily register reactions continuously on the same root Promise, without realizing that each new registered reaction will return a new Promise, instead of changing the original one. Promises' values are immutable. Each Promise will encapsulate only one value, after being fulfilled or rejected, and that value will not change. Each Promise does not store a queue of the registered reactions, since each reaction will create a new Promise that will encapsulate the result of the registered callback, after its execution. It is especially easy to fall into this issue when there are larger volumes of code, when Promises are being generated through auxiliary procedures (e.g., dynamically generated through auxiliary functions). In such cases, the developer might be registering new reactions on already forked Promises if said procedures are not well done or if the developer is not tracking (like storing in a variable) the most recently generated Promise.

Debugging this issue is also complicated, because JavaScript just keeps executing with whatever value was passed, which sometimes can trigger other bugs later in the execution, pointing to a more general error that happened exclusively due to this bug, making it harder to understand the real issue came from this situation. Even worse, the execution could simply display values that could appear correct (no error generated), masking the issue underneath as if it were correct.

An example of this bug is illustrated in Fig. 2.1. On the left, it shows a data race bug. The Promise chain is being accidentally forked. Two chains are being created from the root Promise. This will produce two new Promises, each receiving the value encapsulated by the root Promise. The two new Promises will not be able to modify that value for the other one to read. After the callbacks execute, receiving that

```javascript
const logValue = v => console.log(
    'Expecting 2:', v );

const increment = v => v + 1;

const p = new Promise( resolve => {
  resolve( 1 );
} );

p.then( increment );

p.then( logValue );
```

```javascript
const logValue = v => console.log(
    'Expecting 2:', v );

const increment = v => v + 1;

const p1 = new Promise( resolve => {
  resolve( 1 );
} );

const p2 = p1.then( increment );

const p3 = p2.then( logValue );
```

Figure 2.1: The example on the left shows a data race issue: both reactions are registered on the original promise. Thus, regardless the order in which they resolve, the data they return will never depend flow from one to the other. On the right, the example is corrected: the reaction that increments is registered first and, then, the reaction that checks for the value is registered on the promise returned by the reaction that increments the value. Thus keeping the order of execution: first the increment, then the check.

value, one of the new Promises will encapsulate the value 2 (`value + 1`), while the other will simply log the value, which will be the one used to resolve the initial Promise. Instead of the expected `2`, it will log the value `1`. On the right side of the figure, the correct code (i.e., as intended) is shown. Notice how the first `then` chains to the root Promise, just like on the buggy example, creating a new Promise `p2`. The next chaining is done on the newly created Promise, providing only one Promise chain in the end, ensuring dependency between the values (`p1`; `p2`; `p3`).

This is a simple case, where each callback only uses the value passed to it and where everything occurs right away, within the same microtask phase. More complicated cases, where the callbacks will resolve after an unknown amount of time, or cases where a variable, global to both callbacks, is being written and read by both callbacks, may require that the Promise chains are well defined, along with other code that keeps running in the meantime, but can also work if this bug is occurring. This is dangerous, because the behaviour might be non-deterministic (due to the unknown and possibly random time the callbacks take to resolve and the shared objects).

2.2 Unexpected Execution Order Bugs

Unexpected Execution Order bugs happen when some procedures are executed at an unintended timing, different from what the programmer was expecting. Specifically, it has to do with misplacing method calls, in relation to the availability of the data that these calls are intended to manipulate (i.e., there is data dependent on the order of these calls). This happens due to some functions having asynchronous behaviour, postponing the operations to a later time, without the user being aware of it, or simply scheduling to a time that is not exactly the one the developer wanted. It can vary from very simple cases, where asynchronous code is placed before or in the middle of synchronous code, with the developer expecting

```
const o = {};

setTimeout( () => {
  o.print = console.log;
}, 1000 );

o.print( 'test' );
```

```
const o = {};

o.print( 'test' );

setTimeout( () => {
  o.print = console.log;
}, 1000 );
```

Figure 2.2: The same code is represented in both sides, in terms of execution order. On the left, the developer may mistakenly think that `o.print` will be defined when its call arrives. On the right, the code was simply organized to map directly to the execution order. `setTimeout`, an asynchronous function, will defer the execution of its callback to after the synchronous code is complete. It is made more obvious on the right side that this execution will not work as expected.

the execution to occur as written; to very complex cases, where the dependencies are not clear nor obvious or where dependencies are not known at first to be required. It can also vary between execution order from the same API, to execution order between multiple asynchronous APIs and even between these and synchronous code, as mentioned above.

A very simple example of the Unexpected Execution Order bug can be seen in Fig. 2.2. On the left side, the code can seem correct, at a first glance. The intention is to wait 1 second before defining the `o.print` method. The problem is that the definition is inside an asynchronous function. So when we reach line 7 and we call `o.print`, there will be an error, because it is not defined at that point. Its definition is postponed to after the synchronous code has finished executing. Rearranging the code to the be on par with the order of execution, it will result in what is shown on the right side of the figure. In this case, it is clear that it will not work, because we define `o` as an empty object and immediately call a non-existent method.

It is not always possible to organize the code as shown on the right side (for example, due to locally scoped variables that demand the asynchronous calls to be placed at that place only and not after the totality of the synchronous). Besides, for the developer to organize it that way, they have to already know some of the functions have an asynchronous signature, resulting in the developer hardly writing code with this bug. But, for complex code, there may be no other way for the developer to organize their code. Especially if the code is composed by multiple asynchronous calls, the developer may think that the callbacks written first (for asynchronous APIs) will execute always first, which may not be true and is dependent on the API and its scheduling. Asynchronous APIs, in general, are not particularly different from synchronous APIs, so less experienced programmers may not be aware of how the scheduling is being done or if it is being done at all.

Common ways to write asynchronous code dependent on other asynchronous operations is to either:

1. Enclose the code with Promises, using the Promise chains to define an explicit execution flow. In this case, the Promises usage and the execution flow is transparent to the developer, meaning it is the developer controlling what will happen and when will happen.

2. Use functions that receive a callback to be called once the asynchronous task has completed. In this case, the developer needs to be aware of what the function receiving the callback does and how it does it. Mainly, timings for the asynchronous execution and how the flow is propagated. Unlike the previous case, the execution flow is given by the higher order functions and is not obvious to the developer; the specification is on the documentation, instead, and/or on the APIs source code. To complicate matters, not all functions that receive a callback perform asynchronous operations.

The first case is less error prone, because the developer is the one controlling the flow. Errors can still happen, especially if dealing with large volumes of code and a complex code base. The second case is more complicated, because it is not transparent to the developer what may be happening and how timings are defined.

A more interesting example can be seen in Fig. 2.3. This example aims to capture a more realistic pattern. It makes use of the asynchronous signature of a higher order function receiving a callback that will be called once the asynchronous task is complete.

```
1  function doThrow( msg ) {
2    throw new Error( msg );
3  }
4
5  function Manager() {}
6
7  Manager.prototype.storeString = function( str, onCreate ) {
8    return Promise.resolve()
9      .then( () => ( this.str = str ) )
10     .then( onCreate )
11   ;
12 };
13
14 Manager.prototype.validateString = function( str ) {
15   return str === this.str
16     ? console.log( 'Success!' )
17     : doThrow( 'Different!' )
18   ;
19 };
20
21 var manager = new Manager();
22
23 var str = 'str';
24
25 manager.storeString( str );
26
27 manager.validateString( str );
```

Figure 2.3: Example of the Unexpected Execution Order bug. `validateString` will be called after `storeString` without assuring the latter has completed its task.

The intention here is to have an object of type `Manager` that stores a string and that receives a string and validates it by checking if it is stored. If it is stored, it logs "Success!"; if not, it throws an exception. The storing of the string is an asynchronous operation, emulating what would happen if the storing of the string was done on a remote server, for instance. On a more general note, this example emulates manipulating data between remote environments, such as setting up a remote connection or database and make use of it while manipulating data on/through it. Here, the user is storing `str`. Then, it is validating the same `str`. So, intuitively, it should be a success. However, given the asynchronous nature of the `storeString` function and the synchronous nature of the `validateString` function, the former will be postponed and `validateString` will be executed first. This will lead to an exception being thrown, because, at the time `validateString` makes the check, `storeString` has not yet executed and has not yet stored the string.

In this case, the developer only knows about the asynchronous behaviour of `storeString`, and that it returns a Promise where the storing is done, after analyzing the code and looking inside the functions themselves. In larger code bases, often including third-party libraries, the programmers may not be aware of this asynchrony and may write their code as shown in the example, thinking `validateString` will always execute after `storeString`. With the advent of `async/await`, it is made more explicit when a function is asynchronous. But the lack of `async` on a function does not guarantee the function is not asynchronous or performs asynchronous operations. Thus, this bug can be introduced.

In any case, let's assume, from now on, it is known that `storeString` is asynchronous and has an asynchronous signature, either because it is explicitly in the documentation it schedules the operation or because the developer checked that the function receives a callback or even because the developer is aware that it returns a Promise (in newer versions of JavaScript this could be explicitly noted by defining the function with `async`). With that in mind, in order to fix this bug, we must ensure the validation of the string is done only after the string is stored (i.e., we want to ensure `validateString` occurs after

```
1    var str = 'str';
2
3  - manager.storeString( str );
4  + manager.storeString( str, stored => {
5  +   manager.validateString( str );
6  + } );
7
8  - manager.validateString( str );
```

```
1    var str = 'str';
2
3    manager.storeString( str )
4  +   .then( stored => {
5  +     manager.validateString( str );
6  +   } )
7  + ;
8
9  - manager.validateString( str );
```

Figure 2.4: The snippet on the left fixes the problem by making use of the `storeString` signature that allows it to receive a callback as a second argument, to be called once `storeString` completes. By calling `validateString` inside that callback, it is ensured `validateString` will only be executed when we are expecting: after `storeString` completes. The snippet on the right fixes the issue by using the Promise returned by `storeString`, registering a new callback onto it. Calling `validateString` on that callback only, ensures it will be executed after `validateString` completes.

`storeString`). To achieve such behaviour, the call to `validateString` will have to be moved to a callback passed as the second argument to `storeString`, to be called after the former completes. Alternatively, since `storeString` even returns the Promise, we could register a callback on the returned Promise. Fig. 2.4 illustrates both the solutions. By applying any of these solutions, it is ensured `validateString` will only be called after `storeString` completes.

2.3 Summary

The two types of bugs described in this chapter can easily be introduced in JavaScript code and lead to errors difficult to debug. This motivates the development of automated methods capable of detecting these types of concurrency bugs.

3

Background and Related Work

Contents

3.1 The Event Loop 15
3.2 Static vs Dynamic Analysis 16
3.3 Graph Models for JavaScript Analysis 18

In this chapter we provide an overview of some JavaScript features and we present tools and models that can be used to automatically detect concurrency issues.

3.1 The Event Loop

When it comes to the JavaScript asynchrony, there is an underlying structure that is crucial: the event loop [18–20].

As mentioned in the first Chapter, the event loop consists of a loop, constantly executing several different types of operations, by phases. Each phase is responsible for handling specific types of operations, recurring to thread pools, for Node.js, and Web APIs, for the browser, to manage those tasks. Once a task is ready, the respective scheduled callback will be added back to the JavaScript runtime environment, waiting for its turn to be executed, which will be whenever the call stack is empty.

The tasks handled by the event loop phases are the macrotasks. If a macrotask schedules another macrotask of the same phase, the respective callback will only be executed on the next event loop iteration. In a simplified form, these are the phases:

- Timers: Will look for expired timers (`setTimeout` and `setInterval`) and process their callbacks.
- Poll: Will add to the macrotask queue callbacks registered from incoming events, including triggered by completed async I/O tasks. This phase is blocking, as it waits for any I/O events, if there is nothing more it can do. In this case, if synchronous code appears (e.g., coming from a triggered event), it will be added to the macrotask queue and then added to the call stack.
- Check: In Node.js only, checks for callbacks set by `setImmediate`, adds them to the macrotask queue and, consequentially, to the call stack, to be executed.
- Close: Adds to the macrotask queue callbacks set by any close event (sockets, files).

Between each phase, the loop checks for any microtask operations. There are two types of microtasks, each with a respective queue. First, in Node.js only, it will execute callbacks set through `process.nextTick`. Then, in the order they were registered, callbacks registered on Promises and callbacks set through `queueMicrotask` will be executed. Unlike with the macrotasks, if these callbacks add more callbacks to their respective microtask queue, those newly added callbacks will also still be executed during this microtask cycle, meaning it is possible to starve (block) the event loop with recursive calls of microtask callbacks. Before advancing to the next macrotask, both of the microtask queues must be empty. This means that it is also possible to starve the event loop by adding callbacks back and forth between the two microtask queues.

All of this complexity can lead to concurrency issues, if the developers are not aware of these idiosyncrasies laying beneath JavaScript. Since each phase deals with a specific type of operation, the

execution order of asynchronous calls might not be as expected, compromising dependencies between calls.

This motivates investigation for an automated analysis to detect bugs coming from asynchronous operations.

3.2 Static vs Dynamic Analysis

To analyze JavaScript code, both approaches of static analysis and dynamic analysis have been used by researchers [21, 22]. A tool that performs static analysis runs directly over the code (the text) and there is no need to actually run the code in order for the tool to achieve some result. With dynamic analysis, the JavaScript code needs to be executed in order for the analyzer to reach any result. Both have advantages and disadvantages one over the other.

With static analysis, since the code does not need to be executed, any timing overhead coming from the code execution will not happen. Therefore, there is a quicker access to the analysis result, in general. Imagine code that takes a long time to execute or that performs longstanding computations. Taking a dynamic analysis approach, a user would only have access to the analysis result after all of the code has been executed, which might not be convenient, in some cases, because the developer might realize some of the bugs when the code is executing, defeating the point, in some cases. Besides, another positive aspect of static analysis is, since the code does not run, it will not manipulate any data that would later require a possible cleanup. Finally, a major advantage is that a developer gets an analysis on their code while they are designing it. This is a crucial advantage, because there is no need for trial-and-error approaches. The code is being designed and developed; the developer runs the tool; the tool outputs its analysis; the developer is able to check what and where possible bugs exist and immediately correct them. Thus, when the code firstly executes, is, ideally, already debugged.

However, with static analysis, a big batch of code may require a longer time to analyze. Since the code will not execute, all possible paths must be analyzed (e.g., all if-then-else paths). This grows exponentially, because it cannot check only one path, at least by following a naïve approach (see Fig. 3.1). To avoid going through all paths, some techniques have been developed (e.g., keeping context and state).

But keeping the context, while analyzing, can be, itself, a complicated task. A variable inside one function may come from another part of the code. If that variable contains a callback function, the analyzer needs to know which one it is, in order to perform the analysis. This can lead to a large search on the code, delaying the conclusion of the analyzer's work. Especially when JavaScript, a dynamically typed language, is being considered. Often, there is a trade-off between performance and soundness and completeness. Thus, the trade-off could lead to incorrect results, due to the lack of information, since the tool might not check the whole context, in order to be faster or less memory intensive.

```javascript
function f( v ) {
  if ( v > 1 ) {
    someComplicatedLogic( someValue );
  } else {
    anotherComplicatedLogic( anotherValue );
  }

  const a = 1;
  if ( a === 1 ) {
    // do something.
  } else {
    // do some other thing.
  }
}
```

Figure 3.1: Checking code in order to find any possible values of `v` at the time of line 2 can be useful to exclude one of the branches, cutting down unnecessary verifications. Likewise, some optimization can be made in order to check that `a`, at line 9, will only have the value `1`, thus entering the `then` branch of the `if` and there is no need to waste time checking the `else` branch. This strategy is similar to what some compilers do [5].

As seen in Fig. 3.1, depending on the value of `v`, a branch will be chosen. A static analysis must check both paths, because it has no way of guessing the value of the variable at a given time, at least if it does not check the whole execution context, looking for `v`. There are, notwithstanding, some ways to optimize the analysis, in terms of choosing only a subset of paths:

1. If the condition comes as an argument of a function: by performing a traceback to the function calls, in order to understand where the argument comes from. This, of course, would be recursive, per call, thus, expensive in time. Some depth limitations could be imposed, to limit the traceback, but that might not reach the definition of the value (e.g., if it comes as external input), resulting in wasting time without reaching a conclusion and ending up going through all paths nonetheless. Otherwise, some paths could be discarded, achieving better performance.

2. If the condition is defined on the current scope. That is, the variable used to choose the path being defined just somewhere above the conditional branch. In this case, there is the issue of side effects. But since JavaScript has immutable datatypes, this can be inferred with security, for some cases.

Often, tools that perform static analysis offer some parameters that can be tuned to improve the time or accuracy of the analysis. Fortunately, the tool and model we will be using, Async-TAJS and Callback Graph, already offer a set of tuning parameters to allow choosing how sensitive the analysis should be when assembling the graph structure, providing some adaptation to the types of programs to be evaluated. These parameters allow to adjust sensitivity to context.

Regardless of these tuning parameters, and precisely due to performing static analysis, some limitations of it are that this type of analysis cannot deal accurately with variable-timing operations (e.g., timers

and async I/O, in JavaScript), because it is not possible to predict how long it will take to complete some tasks, like a remote request or a write into a file, while waiting to trigger an already exhausted timer, and the trigger of events coming from user interactive interfaces (for instance as seen on the web), in order to ascertain correctly if a data race will occur (e.g., which file will be fully written first; could a button be pressed before some initialization code is performed?).

Several models and tools exist, with different configurations and built for different purposes, that can help tracking and correct these issues.

3.3 Graph Models for JavaScript Analysis

In this work, we have looked into several models and tools that perform JavaScript code analysis. We focus on static analysis based on graph models. We use the Callback Graph model, but we also describe the Promise Graph model [6], used by PromiseKeeper [23], and the Async Graph [8] model, implemented through AsyncG [24]. Finally, we present a comparative study between different analyzers for JavaScript code [25]. We start with the model we used to analyze our concurrency bugs serving as motivating examples.

3.3.1 Callback Graph

Precisely due to the event-driven, asynchronous nature of JavaScript, along with its increasing popularity, JavaScript developers have been facing some difficulties dealing with asynchronous code. This posed as motivation for the creation of the Callback Graph, introduced by Sotiropoulos and Livshits [3]. Our study will focus on automatically detection of concurrency issues using this model.

This scheme is one of the first static analysis tool supporting ES6 Promises and represents the call flow of asynchronous operations, producing an oriented, acyclic graph. It deals with almost all of the asynchronous primitives described on the ECMAScript specification, up to the 7th edition. The model can be used in tandem with a set of parameters, to allow fine tuning, in order to be adapted to more types of programs. To generate the graph, the analysis makes use of a context-sensitivity strategy.

Sotiropoulos and Livshits make four contributions with this model:

- A λ_q calculus, modeling asynchronous operations, like timers, Promises and asynchronous I/O. It is a variation of the calculi presented by Loring et al. and Madson et al. [6, 26], which in turn is an extension to the λ_{JS} calculus, developed by Guha et al. 2010 [27].

- A static analysis, to handle asynchronous JavaScript operations, making use of the aforementioned calculus, designed to be sound.

- The Callback Graph, which, as mentioned before, represents the execution order between asynchronous functions. Especially when paired with the proposed callback-sensitive analysis, that makes use of the graph to retrieve information about temporal relations, to correctly propagate the data flow. On top of that, the analysis uses a new context-sensitivity flavor, QR-sensitivity, to distinguish asynchronous callbacks.

- Evaluation of the performance and precision of the analysis.

Even though the proposed λ_q calculus is able to cover asynchronous behaviour up to the ECMAScript 7th Edition, it does not model the `Promise.all` method and it has some limitations when it comes to more recent editions, such as handling the `async` and `await` keywords or the asynchronous generators and iterators. All of the asynchronous constructors and methods are then modeled like in the ECMAScript specification, through this calculus.

One of the main advantages of this model is that it preserves the execution order of callbacks with the event loop behaviour in mind. When it comes to Promises, this is modeled directly by the calculus presented above. There are two cases:

- Settling a Promise with containing callbacks: When the Promise is fulfilled or rejected (i.e., settled) and has callbacks registered, those callbacks are added to a list, preserving the order of execution. Those are the callbacks that will be triggered upon settling the Promise.

- Register a callback on an already settled Promise: This case happens when a Promise already settled, but then a reaction is registered. In this case, it is known beforehand if the Promise fulfilled or rejected, so only the relevant reaction is appended to the scheduling list. An example: for cases like `p.then( f1, f2 )`, if `p` rejected, only `f2` will be scheduled.

For timers and async I/O, when performing a static analysis, as stated before, it is not possible to infer how operations that depend on external factors will behave. In this particular case, it is not possible to guess exactly at which state a program is when a timer expires or an async I/O operation completes. The authors assume the order of these callbacks is unspecified. Regardless, nested callbacks will still be represented with their correct execution order.

The Callback Graph itself represents the callbacks as nodes, where each node has associated a context — the context of the function in the program. A path from one node $\mathtt{n}_1$ to another node $\mathtt{n}_2$ means that the callback represented by $\mathtt{n}_2$ will be called after $\mathtt{n}_1$ is called. This property is transitive. Unconnected nodes mean the execution order of the callbacks they represent is non-deterministic or unspecified.

Fig. 3.2 shows a simple example of the Callback Graph model, applied to the code snippet presented in Fig. 1.1. It is important to remark that this model only captures the asynchronous callbacks (i.e., the

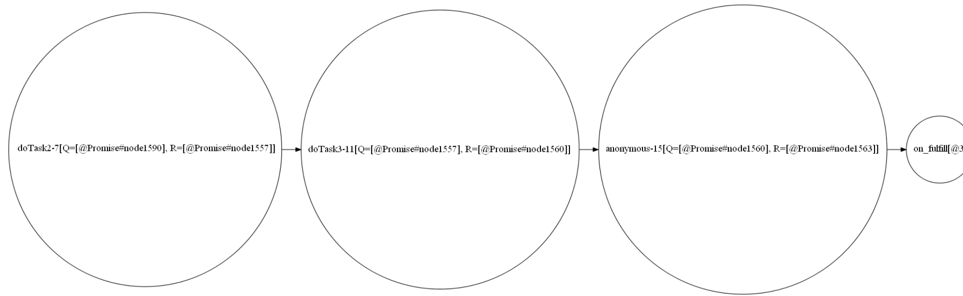


Figure 3.2: Callback Graph model graphical representation of the code snippet displayed in Fig. 1.1.

registration of reactions). Hence why `doTask1` does not appear on the generated graph, assuming `doTask1` just creates and returns a Promise.

As it was stated before, the model comes with a set of parameters, to adjust its sensitivity.

One of the key features offered by this model is that it keeps the context between function calls. This allows to keep track of a happens-before relation, to enrich context information. The authors call this type of analysis callback-sensitive, as it propagates the state from a callback to another, instead of to the caller. In other words, the edges of the graph, between nodes, propagate the state from a node exit point to the next node entry point.

Since the authors concluded this feature does not work with contexts in order to improve precision of the analysis, the event loop is still represented by a single program point (i.e., one corresponding context only). To keep the context sensitivity, though, the state is propagated each time to the event loop, to be joined with the initial state, thus keeping it updated for each new call of the same callback and then propagating it back to the callbacks. In conclusion, there is still some imprecision with this approach.

To increase the precision of the Callback Graph and, consequentially, the precision of the analysis, the callbacks can be distinguished depending on their invocation context. This concept is called context-sensitivity. It is still not very useful on its own, because it does not help differentiating between asynchronous callbacks, considering context-sensitive analysis helps differentiating calls based on the receiver. The receiver of asynchronous callbacks is often the same for different callbacks and calls (for example, the global object).

To overcome this issue, the authors introduce QR-sensitivity. This concept helps distinguishing callback calls by dividing them by their call context and by the object fulfilled by their return value. This increases precision of the analysis, since it allows to clearly distinguish the same function, when called multiple times, depending on the queue object they are fulfilling (one callback call fulfills one Promise, for example).

For the implementation, an extension to TAJS [28–31] was used, denominated Async-TAJS. This is a static analyzer for JavaScript, designed to be sound. It performs a flow-sensitive and context-sensitive analysis. It outputs a set containing all reachable states, starting from any initial state, along with a call

graph. It is capable of detecting some type-related errors, such as generated implicit type-conversion and call of non-callable variables.

The evaluation is done on a set of hand-written code and a set of real-world modules, collected from GitHub.

The model provides different parameterizations that were used to perform the evaluation:

1. Neither callback-sensitive nor QR-sensitive: NC-No.
2. Not callback-sensitive, but QR-sensitive: NC-QR.
3. Callback-sensitive, but not QR-sensitive: C-No.
4. Both callback-sensitive and QR-sensitive: C-QR

The evaluation measure of each analysis depends on three factors:

1. The number of analyzed callbacks.
2. The precision of the computed Callback Graph, measured as being the ratio between the number of callback pairs with a determined execution order over the total of callback pairs.
3. The number of errors reported, as described in [32]. Less errors means it is more precise.

Results show this model can analyze medium-sized JavaScript programs. On average, the precision was around 79%. The tuning of the parameters serves to manage a trade-off between precision and performance, making the analysis more flexible, since it can be adjusted for specific types of programs. The usage of the C-QR parametrization achieves a precision of 88%, on average, while also reducing the number of errors.

In this thesis, we will not only be using this model to detect concurrency bugs, we will also contribute by extending the model in order to improve its capability of detecting and reporting on specific concurrency issues (see Chapter 2 for motivating examples).

3.3.2 Promise Graph & PromiseKeeper

Promise graphs were introduced by Madsen et al. [6]. The object of study is the standardized concept of Promise. The goal is to analyze code containing Promises and find bugs related to them. For this, the paper contributes with a classification of common Promise-based errors, frequently reported in Q&A platforms like StackOverflow.

Given the ECMAScript 2015 standard informally specifies the behaviour and semantics of Promises, in order for the Promise Graph to be accurate and well defined, the authors defined a λ_p calculus, by extending the λ_{JS} calculus proposed by Guha et al. [27], to formally explain the semantics of Promises,

Figure 3.3: Example of a Promise Graph. Taken and adapted from [6].

although it just covers a subset of Promise-based codes. This is a small-step reduction semantics, covering the behaviour of Promise objects and methods.

Errors in code containing Promises are frequent. Therefore, the concept of Promise Graph is introduced. As it was mentioned in the introduction, Promises were important to develop asynchronous code, due to abstraction they provide, making it easier to structure multiple asynchronous calls and handle errors thrown by such operations. However, the semantics of Promises, even though informally defined, are quite complex. Also, Promises are implemented on top of callbacks, which means there is not a particular new syntax for them, meaning there are no checks at static time for Promise errors. Thus, the frequent bugs related to Promises appear.

This motivated the creation of the Promise Graph, whose main goal is to, after performing static analysis on the code, provide a graph based structure that captures the control flow and the data flow of the Promises in a program, with the goal of demonstrating the flow of values through Promise reactions.

Consequently, the Promise Graph is composed of:

- Value node: One per each value allocation site.
- Promise node: One per each Promise allocation site.
- Function node: One for each function.
- Resolve or Reject edge: Labeled with its type. Associates a value node to a Promise node, indicating that the value will be used to resolve or reject the Promise.
- Registration edge: Connects a Promise node to a function node. Represents the registration of the function on the Promise.
- Link edge: From a Promise node to another Promise node, represents dependency.
- Return edge: A function node is connected to a value node through this edge, indicating the function will return that allocation value.

Fig. 3.3 displays an example of Promise Graph, taken and adapted from [6]. With this layout, the Promise Graph helps identifying some types of bugs, but, for some cases, extra information is needed, coming from syntactic checks or with the help of other call graphs. Nonetheless, the Promise Graph is essential to spot these common bugs, proposed and classified by the article:

- Dead Promise: A created Promise that is missing the call to either `resolve` or `reject`, despite having chained reactions. The Promise will never be fulfilled nor rejected. This is visible in the graph, whenever there is a Promise that does not contain any `resolve` nor `reject` edges.

- Missing Resolve Reaction or Reject Reaction: Happens when a Promise either resolves or rejects with a non-undefined value, but there are not any reactions registered. However, Promise callbacks differ slightly from event handlers, so even if the Promise has fulfilled or rejected, a reaction can still be registered in the future.

- Missing Exceptional Reject Reaction: The Promise is rejected implicitly, by throwing an error. But there is not any `reject` reaction registered to catch the error. Such bug can be visualized in the graph, when there is a Promise with a `reject` edge, but not with a `reject` registration edge.

- Missing Return: A registered reaction will accidentally return `undefined`, due to the lack of a `return` statement, and there are other registered reactions on the Promise chain. This may cause the loss of the value being passed through the chain.

- Multiple Resolve or Reject: Case of having, in the creation of a Promise, multiple calls to `resolve` or `reject`. Only the first call will affect the behaviour of the chain. The Promise Graph will help tracking these bugs, if multiple `resolve` or `reject` edges go to the same Promise. However, not all of these cases scenarios will be because of the bug, so the user needs to use eternal information (such as happens-before relations) to be sure.

- Unnecessary Promise: Extra Promise wrapper between two Promises, created inside a reaction. This is usually unnecessary, because the value returned is automatically returned as a Promise value. This can be detected through the Promise Graph, in cases where the unnecessary Promise is not fulfilled nor rejected and in cases where the callback registered just wraps the value with either `resolve` or `reject`.

- Broken Promise Chain: Occurs when the reactions are not registered in chain, but are accidentally registered to the same root Promise. This will create one chain per new registered reaction, rather than keeping the same flow. The Promise Graph can be inspected to help tracking these cases, but some of them could be legitimate behaviours, so the values need to be backtracked, in order to find a possible fork of the chain. This case is illustrated in Fig. 2.1.

This model was applied to 600 code snippets from StackOverflow, figuring out the type of errors and their reasons. It was concluded that it would benefit developers if they had access to tools like this one, that allows reasoning about complex structures such as Promises.

Given such an approach was quite a novelty and had relevant results, two of the three authors, along two other authors, decided to extend this concept, taking on its limitations [7].

One of the limitations was the lack of an automated technique to build the graphs. For that, they present the PromiseKeeper [23]. This tool performs dynamic analysis to construct a Promise Graph. This paper also addressed other limitation the Promise Graph concept had, such as correctly describing the behaviour of the synchronization methods `Promise.all` and `Promise.race`, handling the propagation of exceptions and handling default reactions.

The graph generated by the PromiseKeeper is different from the original Promise Graph presented by Madsen et al.. The new model performs dynamic analysis on the code, rather than static analysis. This creates major differences. One case example Alimadadi et al. [7] present is that an expression that creates a Promise is reused, a static analysis graph will have only one node, abstracting all possible uses of that expression, whereas the graph generated by performing dynamic analysis will actually refer to each use/execution of the expression and also, to reflect what happens at execution time, each Promise can only be fulfilled or rejected with one value, so each Promise node can only have, at most, one incoming edge (representing the one value used to resolve the Promise).

The newly generated graph, being an extension of the one proposed by Madsen et al., presents the same components, but, in some cases, with some slight changes:

- Synchronization node: New node, representing a synchronization operation (for instance, `Promise.all` or `Promise.race`). Serves to keep track of how the state on Promises created by the aforementioned methods depends on the input of each of the Promises converging there and the order in which each Promise is fulfilled or rejected.
- Return or Throw edge: Referred as "Return edge" previously. Now includes `throw` statements and can be labeled to indicate an implicit return.
- Two types of synchronization edges: 1) From a Promise node to a Synchronization node; 2) From a Synchronization node to a Promise node. The former represents the convergence of an input Promise to the synchronization node (i.e., a Promise being passed to `Promise.all` or `Promise.race`), labeled with the state. The latter represents the resulting Promise of the methods application, labeled with the state of the resulting Promise.

To handle default reactions, the graph will generate the default reactions and place the nodes and the edges accordingly.

In addition to the bugs classified by Madsen et al., Alimadadi et al. classify some of those common bugs as anti-patterns and add the following:

- Unreachable reactions: In case there are unsettled Promises (i.e., always in pending state) passed to the `Promise.all` method, any reaction it has registered will never be called, because it will keep waiting for all Promises to resolve. The only exception is if a Promise rejects, in which case `Promise.all` will reject with that value. A similar situation happens with the method `Promise.race`,

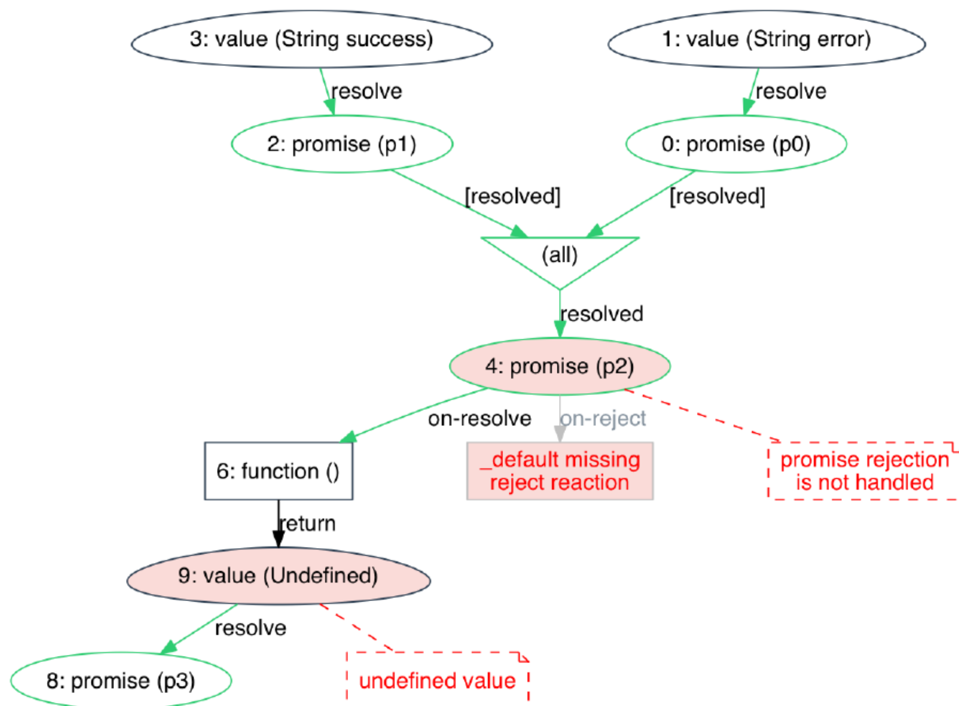


Figure 3.4: Example of a Promise Graph as it is outputted by the PromiseKeeper. Taken from [7].

if all Promises passed down to it never resolve. For the more recent `Promise.allSettled` case, it will wait indefinitely, if at least one Promise is pending indefinitely.

The resulting Promise Graph, output by the PromiseKeeper, seen in Fig. 3.4, is styled with particular shapes and colors for each type of node, facilitating the analysis. It also tries to infer automatically, as much as possible, the common anti-patterns, presenting tags pointing to such situations.

3.3.3 Async Graph & AsyncG

Another graph-based model is the Async Graph [8]. It was created to address the Node.js environment, especially the scheduling of operations using the Node.js event loop.

To support this model, the authors built AsyncG [24]: a tool that generates the Async Graph, by tracking all asynchronous calls. The tool automatically identifies bugs related to event scheduling and other asynchronous operations, at runtime. Thus, performs a dynamic analysis of the code. It was the first tool to detect bugs by misuse of several types of asynchronous APIs in Node.js applications.

On the grounds of what was stated above, the authors make four contributions:

- Introduction of the Async Graph. A graph-based structure that allows to reason about the execution of event-driven Node.js programs.
- Implementation of the AsyncG. A tool to build the Async Graph, during a program execution.

```
let foo;
Promise.resolve( {} ).then( v => {
  foo = v;
} );
setTimeout( () => {
  foo.bar = function() { /* ... */ }
}, 0 );
process.nextTick( () => {
  foo.bar();
} );
```

Figure 3.5: The execution order of each callback is different from the registration order. Taken from [8].

- AsyncG analyzes the Async Graph and reports any potential bugs and code smells.
- AsyncG is able to detect common bugs in Node.js, related with asynchronous operations, and also newly-found bug patterns.

Along with all of the possible ways to defer operations in JavaScript mentioned in the introduction, Node.js provides a default API called `EventEmmiter` [33]. This allows to create objects that emit events and attach listeners (callbacks) that will execute whenever an event is emitted, thus deferring the execution of the callbacks to when an event is fired. This kind of events is used in Node.js for remote I/O operations. For example, when connecting to a Node.js server, when the server is ready, triggers an event; then, any listener registered from an external application will be executed at that time. Due to the common practices of JavaScript, allowing chaining between methods that receive callbacks, a long chain of code may be created, plugging multiple even handlers together. The execution order of those callbacks might not be the order in which they appear written and that can lead to concurrency bugs (i.e., the developer expects an incorrect order of execution).

A simple example can be seen in Fig. 3.5 (retrieved from [8]). Here, as it was seen in the introduction, the microtasks will execute first and then the macrotasks, even though the registration order is for a microtask (Promise), then a macrotask (setTimeout) and finally another microtask (`process.nextTick`). Between the two microtasks, `process.nextTick`, even though registered in last, has precedence over the Promise. The execution order will first call the `process.nextTick` callback, attempting to call method `bar` on object `foo`. But `foo` is simply `undefined` at this time. This is a simple example, but, for complex code, it may be complicated to find the source of the error and why a callback is not being fired when the developer is expecting.

Helping finding these bugs poses as a motivation for the development of the AsyncG. The tool should assist the developers in such a way that allows them to understand the order of the callbacks, since that is the cause for most data races. The authors talk about four main challenges in order to develop AsyncG in a way that will cover the mentioned functionalities:

- Formalization: New abstraction to represent asynchronous calls.

- Visualization: Representation of the execution of asynchronous callbacks, to illustrate the event loop scheduling of the operations. The formalization used of the event loop is presented in [26].

- Implementation: AsyncG must track all sources of asynchronous execution (including external ones) and relate them to event queue. But it must not cause any side effects on the application, since it will run alongside it. It provides some overhead comparable to that of a debugger.

- Automatic Bug Detection: The tool should identify bugs that cannot be easily found with existing tools, especially bugs that require knowledge about the event loop mechanism.

From those challenges, the authors describe the Async Graph, to fulfill the first and second points, and AsyncG, to address the third and fourth points.

For the former, the authors created a time-oriented graph, describing the asynchronous flow of a Node.js application. Each tick of the event loop is represented by a vertical line. Each node belongs to a tick, representing when the call happens, and has associated a source code location, to indicate from where the callback (the event) is triggered. Each tick can have various types of nodes:

- $\Box$ — Callback Registration (`CR`) node: Registration of one or more callbacks to be executed. It may be executed right away (a Promise constructor: its callback always executes immediately) or deferred (microtsasks creating microtasks or timers, etc.).

- $\bigcirc$ — Callback Execution (`CE`) node: Execution of a callback.

- $\star$ — Callback Trigger (`CT`) node: An event emission or a Promise action (`resolve` or `reject`), triggering the execution of a previously registered callback.

- $\triangle$ — Object Binding (`OB`) node: Creation of a Promise or emitter object.

Each pair of nodes α, β is connected through either:

- a direct edge with the semantics α *causes the execution of* β: $\alpha \rightarrow \beta$

- a dashed edge, that may be labeled: $\alpha \leftarrow -- \beta$

For the direct edge, there are the following cases:

- `CR` $\rightarrow$ `CE`: A callback execution can be triggered by its registration. As mentioned above, this happens on the callback passed to the Promise constructor.

- `CT` $\rightarrow$ `CE`: A callback trigger will cause a callback execution. This happens with an event emission, for instance.

- `CE` $\rightarrow$ `x`: Where `x` represents all callback nodes that will be executed because of this callback execution.

For the dashed connections, the following cases may occur:

- CR $\leftarrow - -$ CE: Represents the binding between the callback registration and the callback execution.
- OB $\overset{\text{relation}}{\leftarrow - -}$ x: This edge is labeled and represents a relation between a node x with a Promise or an event emitter. The label expresses the type of the relation (i.e., the name of the event, the name of the Promise or *link*, to indicate the join of a returned Promise from a `then` callback to the Promise chain).

As for the AsyncG tool implementation, the authors based it on NodeProf [22]. NodeProf instruments the internal Abstract Syntax Tree representation in a way that avoids adding any overhead, unlike other source-code instrumentation frameworks, that wrap the code, creating new frames on the call stack. NodeProf also supports the instrumentation of Node.js internal libraries and supports some of the latest features, such as `async/await`.

To build the Async Graph, AsyncG intercepts all function calls, using NodeProf, and uses the method `functionEnter( func, receiver, args )` before executing the intercepted function, where `func` is the intercepted function, `receiver` is the receiver object of the function and `args` are the arguments of the function. When exiting the function call, AsyncG will call `functionExit( func, retVal )`, where `retVal` is the returned value of the intercepted callback.

To identify event loop ticks, AsyncG maintains a shadow stack. A new tick will only start when the shadow stack is empty.

To handle callback registration, it is important to check if the higher order function receiving the callback is from an asynchronous API. This check is done by an auxiliary method. Then, it checks which argument refers to the callback function, the event loop tick that will execute the callback and if the callback is to be fired once only or multiple times (`setImmediate` vs. event callback). Finally, check the type of the object the function call is bound to. Different ways to determine the previous checks were implemented, in order to fetch the information, due to the multiple APIs.

After all the checks done, a CR node is created and placed on the current tick in the graph and will be inserted in a list $L^{func}_{pending}$, later to be used to map the respective CE node.

To finalize the assembling of the graph, AsyncG will map the callbacks' executions to the respective registrations. A context validator will determine if the current context is favorable (i.e., type of the current tick, current shadow stack and the type of object bound to this function call) for any pending callback in the aforementioned list $L^{func}_{pending}$. If there is a valid CR node, it means the current function call was registered by that CR node. Since a callback is being executed, the tool also needs to create a CE node. This new CE node will connect to the CR node through a dashed edge, labeled with the the type of relation. A direct edge is used to link the CR node to a CT node, which may have fired the callback being currently executed. If the callback is only to be executed once, it is removed from $L^{func}_{pending}$.

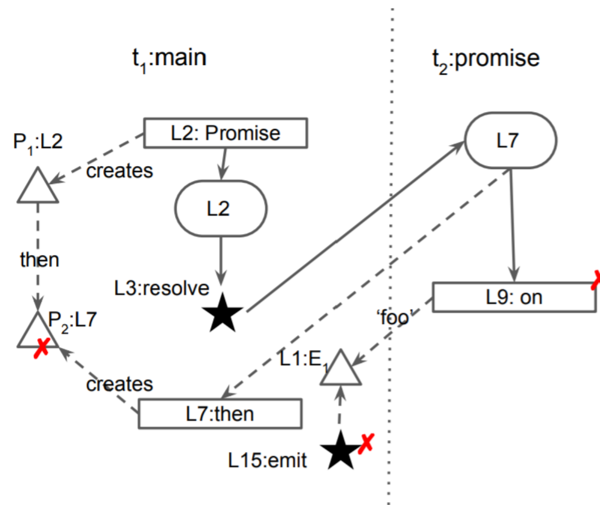


Figure 3.6: Example of an Async Graph for buggy code. Taken from [8].

To visualize the generated graph, AsyncG can use DOT [34] or D3.js [35], a JavaScript library, to display the graph in the browser. An example is provided in Fig. 3.6.

An important feature offered by AsyncG is the automatic bug detection to aid developers with debugging. Some bugs require a broader knowledge about the application, in order to be identified, but some specific types can be inferred automatically:

- Scheduling bugs: Incorrect use of APIs that schedule operations.
  - Recursive Microtasks: As previously stated, when a microtask is scheduled, it will always be executed as quickly as possible. When a microtask is scheduled from a microtask, it will always execute in the same cycle. Thus, recursively scheduling microtasks, will create an endless loop of microtask execution.
  - Mixing similar APIs: Happens when the developer accidentally uses a method, expecting the result of a similar method (e.g., `process.nextTick`, `setImmediate` and `setTimeout` with a delay of 0). AsyncG will warn when the uses of these methods in the same tick lead to wrong event execution order.
  - Unexpected Timeout Execution Order: This happens when the developer writes functions that may be scheduled in a different order. The example given by the authors is `setTimeout( foo, 101 ); setTimeout( bar, 100 )`.
- Emitter bugs: Related to emitters.
  - Dead Listeners: Occurs when a listener is registered for an event that will never be emitted.
  - Dead Emits: Occurs when an event is fired, but there are not any listeners for it.

- Invalid Listener Removal: The developer tries to remove a listener for an event, passing a function that was never registered as a listener of that event.
- Duplicate Listeners: The same function is registered more than once for the same event.
- Add Listener with Listener: As the name states, this happens when a listener is registered for a specific event through a listener of that event.

• Promise bugs: Incorrect use of Promises, including `async`/`await`. Some of these were covered by the above model/tool, given it is Promise-oriented.

  - Dead Promise: Promises that are never resolved nor rejected.
  - Missing Reaction: A Promise is resolved or rejected, but there is not any reaction registered (i.e., no calls to `then`, `catch`, etc., including `await`).
  - Missing Exceptional Reject Reaction: A call to `catch` is missing at the end of the Promise chain.
  - Missing Return: A registered callback will implicitly return `undefined` due to the lack of a `return` statement, and there are other registered reactions on the Promise chain. This may cause the loss of the value being passed through the chain.
  - Multiple Resolve or Reject: Multiple calls to `resolve` or `reject` inside the same function will not change the result of the first call.

Even though these cases are often a sign of buggy implementations, some may actually be intended. Imagine making multiple calls to a remote server, but those calls need to be made in order, without caring about a specific value returned. In this case, a Promise chain can be built, to guarantee the order of the asynchronous requests, but the callbacks will not return any value. Hence, no bug.

Besides the above cases, there are still other scenarios where there is a bug, but AsyncG cannot detect it alone. They can be detected by analyzing the Async Graph:

• Expecting Callbacks to Run Synchronously: This happens when the developer writes code, after writing an asynchronous function call, and expects the code of the asynchronous callback to run before the code written after it. In other words, there are dependencies between the code, but the order might not be maintained due to the scheduling.

• Broken Promise Chain / Unnecessary Promise: Can be noticed by analyzing the graph and back-tracking the Promise chain.

As it can be seen, these model and tool cover almost all asynchronous API in the Node.js environment. By performing a dynamic analysis, over a static analysis, and with NodeProf to instrument the Abstract Syntax Tree, the overhead is minimal, along with the fact it just provides the path the program traveled.

3.3.4 Call graphs

To finalize, since the work presented in this thesis makes use of and analyzes code using a particular case of call graph, in this subsection we present several static code analyzers that produce a call graph. We summarize how they compare with each other, based on a study made by [25]. Each of these tools has their own unique features, that may be useful to explore, in order to integrate with the model we will be using.

The generated call graphs are, generically, composed by (a) nodes, representing the functions of a program, identified by an ID, a label and the source code location; and (b) edges, that represent a call between the two nodes (functions). Each pair of nodes is connected through, at most, one edge (omitting multiplicity of calls between functions, due to some tools' limitations). These graphs are used to detect quality and security issues in JavaScript programs. More precisely, they can detect functions that are never called, for instance. By examining the graph, some other factors can be asserted, like checking if the number of arguments passed to functions are correct. This type of graph is also useful, along with control-flow graphs, in mutation testing [36], automated refactoring [37] and defect prediction [38].

Performing static analysis has some advantages over dynamic analysis, as mentioned before, but there are, in turn, some disadvantages: static checking cannot perform an analysis on methods whose behaviour is simply runtime dependent. For instance, `eval` or Reflection [39]. On the other hand, static checking can be more efficient, since it does not require the code to be executed.

Therefore, improvement of static call graph construction is encouraged. For that reason, five tools are compared:

1. IBM WALA [40]: A framework for static and dynamic program analysis for Java, having a JavaScript front-end. Only the feature of call graph assembling performing static analysis is used. WALA produces several edges between the same pair of nodes, if there are multiple calls between those functions, but since the authors have decided to simplify, they changed the serializer to merge them all into one. To indicate function calls at the global scope, if any, a top-level node was created.

2. Google Closure Compiler [41]: A JavaScript compiler, optimized. This compiler builds a call graph for its own purpose only, so it had to be modified in order to output it. It adds an artificial root node, to designate calls at the global scope.

3. ACG (Approximate Call Graph) [42]: This tool can operate in two modes: pessimistic and optimistic. These modes have to do with how interprocedural flows are handled. In this study, the pessimistic one was used. The implementation had to be changed to add a root node, representing the global scope, as well as merging multiple edges for the same node pair into one.

4. npm callgraph [43]: An npm module that parses JavaScript and creates the program's call graph.

5. TAJS (Type Analyzer for JavaScript) [28–31]: A Java tool for dataflow analysis to build call graphs and infer type information. This is the tool that serves as the base for the Async-TAJS tool that is used in this work.

By testing all and each of these tools, both quantitatively, to count the amount of detected calls, and qualitatively, by validating the found call edges for each graph per tool and comparing them with each other's graph, the general conclusion is that none is perfect: some edges are missing on some graphs produced by some tool, but appear in others, while some other edges are missed on others tool, but appear, again, in others. Bottom line, together, they can complement each other. The amount of resources used per tool is also tracked, to assess usability and performance on real-world cases.

For the particular case of TAJS, since it is the foundation for Async-TAJS, the one we use, it did not report any edge that was not found by at least one other tool. However, it was one of the two, along ACG, to be able to find an edge representing a complex control flow. Both it and WALA were also the only two to report edges coming from higher-order functions.

With all these comparisons, the authors point the main contributions of their study as:

- Evaluation of capabilities and performances of five widely adopted static JavaScript call graph generation tools.
- Quantitative and qualitative comparison of each tool results on 26 benchmark programs and 6 real-world Node.js modules.
- Manually validated dataset of call edges found by these tools on the 26 benchmark programs.

To perform a balanced comparison between each tool, some arrangements had to be done. Some tools had to be patched, as it was already covered above. Besides, in order to extract and dump the call graphs that were, otherwise, only in memory, some more modifications were made. After the extraction, all dumps were converted into DOT format and then into a JSON representation, in order to be compared. The only exception was Closure, which was dumped directly as JSON.

For the comparison, three test input groups were defined:

1. Real world single file examples: Consists of programs with varying complexity of code, contained in one single JavaScript file. The size of the programs is such that it is possible to analyze them manually, to then compare with the tools. The benchmark used was the SunSpider benchmark, form the WebKit browser engine [44].
2. Real world multi-file Node.js examples: Programs composed by modules, to be ran in Node.js. The authors collected the modules from GutHub. Each module contains several files, to test exports and dependencies. The selection criteria included that the modules had a test suite with at least

75% code coverage. Furthermore, each selected module had to be used by at least other 100 modules.

3. Generated large examples: Aimed to measure performance by stress testing the tools. This collection consists of large examples. The authors did not find any open source datasets that would contain only the language feature understood by the tools in question, so they opted for generating themselves JavaScript programs that conform to the ECMAScript 2015 specification (highest standard all tools conform to). Two categories were defined:

   (a) Simple: contains only code with simple logic and statements, like variable declarations, object creation and property access, loops. No complex control flows. Three files were generated for this category.

   (b) Complex: In addition to the above, contains function calls, with different number of parameters, including callbacks, function expressions and logging statements. Two files were generated for this category.

For comparing the resulting graphs of each tool, two criteria were considered, as previously mentioned:

- Quantitative: Comparing the number of nodes and the number of edges.
- Qualitative: An automated script was created, based on [45], to detect common edges the various tools found. On the JSON files, each node and edge is assigned metadata, indicating which tool found which node and edge. This identification is done by tracking the file path, line and column information of the function on the source code, given JavaScript functions are often anonymous. Some tools had to be adapted in order to output all of these details.

The results for smaller programs of the SunSpider dataset are very similar between each tool, but that is probably because there is also not much room for disagreement. Larger problems already have some discrepancy between each call graph tool. For this dataset, two of the authors performer manual evaluation.

For the Node.js modules, npm callgraph and WALA were unable to deal with module requiring. TAJS, even though it supports requiring modules, it has some limitations, so it could not detect calls between different files. Thus, only Closure Compiler and ACG were used here. The results are quite different for each, bar two cases (one about nodes and the other about edges). But this is expected for complex programs.

In terms of performance, Closure, ACG and TAJS were the best. As for the remaining two, WALA was 30% faster than npm callgraph, although it consumes more memory.

For medium-sized programs, ACG was the best, followed by TAJS. For the large set, Closure used the least memory, despite TAJS being faster. With these results, the authors suggest ACG and Closure keep a simple structure inside, performing better for smaller sets, but taking longer to analyze for larger sets, with several edges.

In conclusion:

- Recursive calls are not handled by all tools. Closure Compiler presented better handling in this case.
- Inner functions are not handled by every tool, which may result in some false positives.
- WALA and TAJS are the only two tools that detect higher order function calls.
- ACG and TAJS can detect more complex control flows and non-trivial call edges.
- Closure relies on name-matching, causing false or missing edges.
- WALA is able to analyze some dynamic calls from strings, but only to some extent.
- npm callgraph incorrectly assigns edges from anonymous functions at the global scope.
- Closure has better performance for very large inputs, with high recall, at the expense of precision.
- ACG consumes the least memory, while running faster, for small to medium programs.
- WALA and npm callgraph are not satisfactory for analyzing programs with million lines of code.

4

Approach

Contents

4.1 Broken Promise Bugs 37
4.2 Unexpected Execution Order Bugs 38

This Chapter describes how each concurrency bug described in Chapter 2 is approached, in order to be detected by an automated process.

4.1 Broken Promise Bugs

A Broken Promise bug, simply put, happens when a reaction is registered on a Promise that already had reactions registered. Or, from another perspective, is when the same root Promise is used to generate more than one Promise chain. An approach, even though apparently naïve, to ensure these issues are caught by the static analysis is to follow these steps:

1. Each time a new Promise is created, assign a unique identifier (UID) to that Promise. This UID will be used to define and identify that Promise and only that Promise. A Promise is always created by the Promise constructor (`new Promise`) or by any method from the Promise API, including the ones in the Promise prototype (`then`, `catch` and `finally`) [9]. This UID also identifies the root of a potential Promise chain. Let's call this value `R`.

2. If the Promise was created by registering a new reaction onto an already existing Promise, then assign another value to the newly created Promise: Let's call it `Q`. This value will be the same as the `R` of the Promise where the reaction was registered to. An unique value can be used to represent the absence of value, for when a Promise is being created at the top level, instead through the registration of a reaction.

3. Store all Promises in the form of `[R,Q]` (or any other form that includes both `Q` and `R`). These two values are enough to identify a specific Promise and where it comes from. `Q` identifies where it comes from; `R` identifies the Promise itself.

4. After the static analysis is complete (and all the Promises were tagged with `Q` and `R`), we now have complete information about how the Promises relate to each other, in terms of chains. So the next step is to find which Promises have the same `Q` value. If there are multiple Promises with the same `Q`, that means all those Promises were created by reactions registered in the same Promise, whose `R` value is that `Q`. This means the Promise with that value `R` was forked, giving birth to all those Promises with the same value `Q`. The only exception is if Promises with the same `Q` also have the same `R`. This happens if different callbacks generate the same Promise (i.e., different callbacks resolve the same Promise).

More technically, `Q` stands for queue object: the object where the callback is being registered to. In this case, the Promise where the reaction is being registered to. `R` stands for dependent queue object: the object being generated from the registration. In this particular case, the new Promise being created by the reaction.

With this in practice, we can identify forked Promises. From those forked Promises, all that remains is to filter which forked Promises are Broken Promise bugs.

Forking a Promise is not by itself an issue. It only becomes a real issue in some cases if, for example, there is data depending on it. This may either come from the flow of the Promise chain, via the arguments and the return value of the callbacks, or from data declared in an outer scope that requires the callbacks to be executed at specific timings. Or even the data dependency may happen in an external server, whose order of the events received depends on the Promise chain, but that is impossible to know just by the JavaScript code. Besides, given JavaScript is a dynamically typed language, it makes it virtually impossible for most cases to automatically understand when the data passed through the chain is not being used as intended by the developer. More so, a forked Promise could also be intentional. There is not a simple way to infer if a forked Promise is intentional or just came from lack of attention. In the end, it depends on the intentions of the developer.

In either case, some communities might consider registering multiple reactions on the same Promise a bad practice or an anti-pattern [46, 47]. In fact, the result of forking a Promise can entirely be replaced by creating a new Promise with the desired value and develop a Promise chain from there, making forking Promises apparently futile (nonetheless, in some edge case scenarios, perhaps it may be useful to fork a Promise).

So we decided to consider all forked Promises as potential Broken Promise bugs.

4.2 Unexpected Execution Order Bugs

The problem that derives from the Unexpected Execution Order bug is data access and manipulation (read and write) shared by the asynchronous operations whose order will not be the expected one, causing undesirable results.

Therefore, we are interested in finding cases of different execution timings that use the shared data in an unintended way. For instance, going back to the example shown in Fig. 2.2, we are interested in knowing when any data is manipulated. More specifically, we are interested in knowing when `o.print` is read and when it is written. `o.print` is read (and only read) synchronously and written (and only written) asynchronously.

Fig. 2.3 contains a more interesting example. Lines 25 and 27 is where the Unexpected Execution Order bug occurs, because both those calls access and manipulate the value stored in `manager.str` not only at different execution timings, but because the order of reads and writes is not the desirable one. The call to `storeString`, appearing first, asynchronously writes `manager.str`. The call to `validateString`, appearing last, synchronously reads `manager.str`. We have an asynchronous write and a synchronous read on the same data. The read will happen first and the write will happen last. The

value written through `storeString` will never be read. And the value that will be read will be something potentially uninitialized and unexpected.

In both these figures there is a deterministic result. It will always throw an exception. But, for the code in Fig. 2.3, the validation itself could also be asynchronous (e.g., validating through a remote server), which could lead to non-deterministic results.

```
1  function Manager() {}
2
3  Manager.prototype.storeString = function( str, onCreate ) {
4    return new Promise( resolve => {
5      setTimeout( () => {
6        resolve( this.str = str );
7      }, Math.random() * 1000 );
8    } ).then( onCreate );
9  };
10
11 Manager.prototype.validateString = function( str ) {
12   return new Promise( ( resolve, reject ) => {
13     setTimeout( () => {
14       if ( str === this.str ) {
15         resolve( 'Success!' )
16       } else {
17         reject( 'Different!' );
18       }
19     }, Math.random() * 1000 );
20   } )
21     .then( console.log )
22     .catch( console.warn )
23   ;
24 };
25
26 var manager = new Manager();
27
28 var str = 'str';
29
30 manager.storeString( str );
31
32 manager.validateString( str );
```

Figure 4.1: Example of the Unexpected Execution Order bug, similar to the one in Fig.2.3, but with the both the storing and the validation of the string being asynchronous, leading to non-deterministic results.

Fig. 4.1 illustrates exactly that. Here, both `storeString` and `validateString` defer their execution by an unknown time. There is no guarantee on which will complete first. Sometimes, the validated string will be stored first, other times it will not and the program will warn `Different!` on the console. So the best approach is to alert the developer not only when there are data writes executing first and data reads executing, but to alert the user when there are data writes and data reads that can **potentially** be executed in the any order.

It is necessary to track reads and writes not only when they appear, but also when they may execute in an interchangeable order. Therefore, an approach capable of detecting this type of problem would be:

1. To split all the different execution timings into nodes. Each node thus represents a different execution timing. An execution timing is either the synchronous code that executes when a program is started (i.e., the code added to the call stack) or each different execution added to the event loop tasks and queues. This includes both microtasks and macrotasks. For Promises, it is important to keep track of the chains, to know which reaction will happen only on the next level. So, like on the Broken Promise bug, this approach also records each Promise with its `Q` and `R` values.

2. For each node, record which data is read and written.

3. Finally, organize the nodes by the execution order, according to the event loop, whenever possible, with the synchronous code always as the starting point. For nodes whose execution timing cannot be guaranteed (for instance, timers or operations dependent on external services), assume they can happen at any possible time, according to the event loop. For example, if the code contains a Promise with one reaction and two `setTimeout` with a random delay after it, the Promise reaction will always execute first, because it is a microtask, but the `setTimeout` can happen in any order. Our approach will thus create all the possible paths, generating two paths: one where one `setTimeout` happens before the other and vice-versa, but the Promise reaction will always execute before them, in either case.

To extract results from this approach and detect Unexpected Execution Order bugs, the next step is to check if data is written on a node n_2 that comes after (not necessarily immediately) a node n_1 that reads the same data.

Regarding the order the code is executed for Promises and Promise reactions, each reaction registration will schedule to the next context level. A level is generated by queuing the reactions' callbacks according to registration order of the first reaction registration for each respective Promise chain, for the same context. As an example, if three Promises are created and each has two reactions registered through `then`, where those reactions are composed solely by synchronous code, each first `then` callback for each Promise will execute by the order of their (the first) registration and only then, when all first `then` callbacks are done, it goes to the next level and the second callback for each Promise will execute, in order of the first reaction registration for that Promise chain, for the same context. It follows the order of the Promises generated by the first reaction registered resolves, because that is when the first scheduling of an operation and its completion occurs (i.e., its callback was added to the queue). Any other microtask, created inside one of those callbacks will be scheduled into the next level of reaction execution, incrementally for each inner registered reaction on their chain, following a LIFO [48] approach, placed right before the next respective reaction for that current outer Promise chain, assuming they are resolved

immediately. That is, when the next callback for that outer Promise would execute, first it will exhaust the respective inner registered reaction, for each inner Promise chain, in order of the first reaction registered completion for each inner Promise, for that level. Basically, each inner Promise reaction will have its level defined as the current level (e.g., the level of the outer Promise if this inner Promise resolves immediately), plus the level of the reaction where they are created in. For forked Promises, the reactions for the same level of the forked Promise will be queued following the FIFO [49] methodology. This applies in the same manner in case there are multiple reactions in each chain of the forked Promise, being queued by their respective level. For inner reactions that are not resolved immediately (e.g., waiting for another operation), the same will apply but only counting from the moment when they resolve.

As for `process.nextTick`, it goes into its own queue. This queue is verified and emptied before the other microtasks' queue. If this queue is filled while the microtasks queue is being emptied, it will be traveled again. And, again, after it is emptied, the event loop will check again the microtasks queue. The event loop will not go to the next phase until both queues are empty.

To summarize, this approach separates all the execution timings and allows for comparison on how the data is accessed and manipulated between them.

5

Implementation

Contents

5.1 TAJS – Type Analyzer for JavaScript 43
5.2 Broken Promise Bugs 43
5.3 Unexpected Execution Order Bugs 47

The previous Chapter describes two approaches to detect the asynchrony bugs considered in this work. In this Chapter, we discuss our implementation of the approaches described. The source code of our implementation is publicly available as a GitHub repository.[1]

As mentioned before, we use the Async-TAJS [4] tool to perform static analysis in the JavaScript code. This tool is itself an extension to the TAJS [28–31] tool. It was extended with the model of the Callback Graph, introduced by Sotiropoulos and Livshits [3]. Our goal is to explore the Callback Graph to detect concurrency bugs. This Chapter also describes how the implementation uses this model to perform the detection.

5.1 TAJS – Type Analyzer for JavaScript

The TAJS is a tool written in Java designed to be sound [28, 50]. It performs a flow-sensitive and context-sensitive analysis. It outputs a set containing all reachable states, starting from any initial state, along with a call graph. It is capable of detecting some type-related errors, such as generated implicit type-conversion and call of non-callable variables. With the added extension, it is also able to generate the Callback Graph for the input program.

Our intention is to extend this tool even further, taking off from the Callback Graph modifications, making it possible for the tool to detect the concurrency issues presented in the above sections while the code is being developed and alert the user with useful information, so they can understand there are problems in the code, which specific types of issues the user is facing and where those issues are located.

The Async-TAJS is able to already generate a file displaying the Callback Graph. The next step is to parse the Callback Graph (its internal structure), to collect information about a possible data race bug.

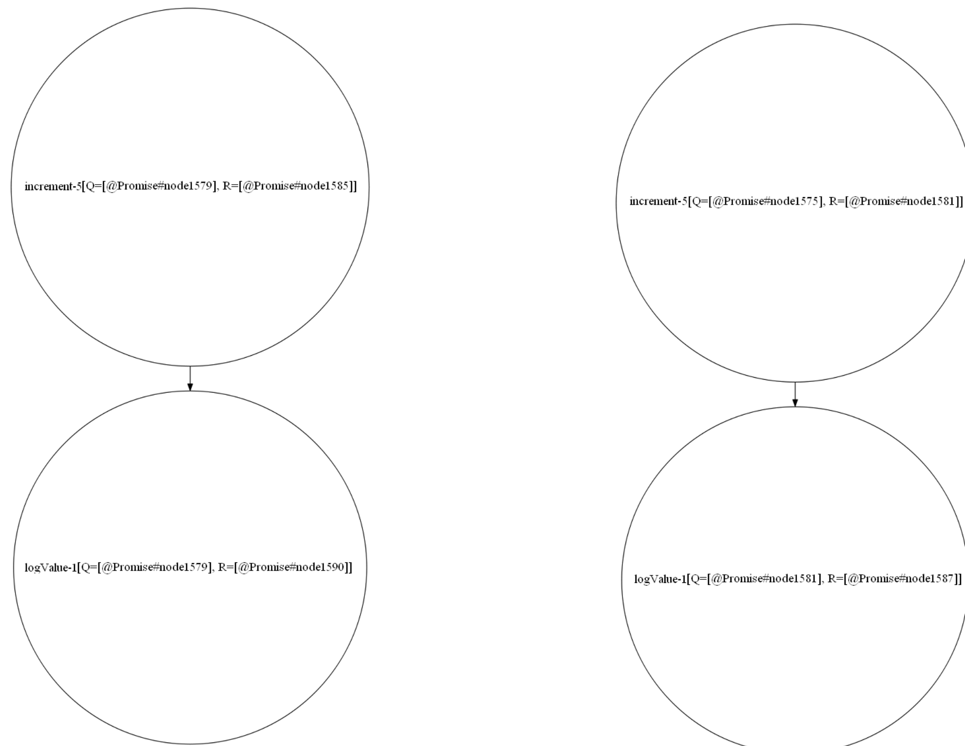
Each data race issue has its owns quirks; thus, its own way to parse the Callback Graph, looking for the necessary information, useful for each particular issue.

For our analysis, we used the tuning parameters for the QR-sensitivity analysis, considering this tuning is the only one that allows to distinguish between different calls of the same asynchronous callback. Therefore, allows for a precise distinction of the callback calls. We implemented our solutions with that parametrization in mind.

5.2 Broken Promise Bugs

We took on the approach detailed in Section 4.1 of the previous Chapter, intending to alert the developer about all cases of a forked Promise.

[1] Source code of implementation: `https://github.com/BernardoFuret/async-tajs`.



(a) Callback Graph of a code snippet with a Broken Promise issue.

(b) Callback Graph of the fixed code.

Figure 5.1: Callback Graph model applied to the examples provided on Fig. 2.1, respectively.

Consider the code snippets presented in Fig. 2.1. The Callback Graph model collects the asynchronous callbacks registered in the Promises, through `then` and `catch`. It then generates nodes from these methods, with associated `Q` and `R` values, introduced in Section 4.1. These are concepts developed by the Callback Graph model and already implemented in the Async-TAJS.

Fig. 5.1 displays the graphs generated by running Async-TAJS on the example snippets.

At first, both graphs seem similar. Function `increment` comes first and there is an edge from it to the function `logValue`. Do not forget that the graph structure itself presents happens-before relations between the asynchronous callbacks. So this is correct. Both functions `increment` and `logValue` are composed only of synchronous code; `increment` is registered first and `logValue` is registered second. So `increment` will always be executed before `logValue`. The problem illustrated on the code snippet on the left, from Fig. 2.1, is that, even though `increment` is registered before `logValue`, their flow is not the same. The Promise where both reactions are registered is the same. The root Promise is forked into two Promises. So even though the execution order for each callback is correct, the way the data flows through them is not.

Upon a closer look to the graphs generated by the tool, inside the nodes, there is information about the `Q` and the `R` for each node. Even though the graphs are similar, the key difference is on the `Q` and

the `R` values. The second node of the graph on the left, corresponding to the code snippet with the Broken Promise bug, has a value for `Q` different than the value for the `R` of the first node. So the callback represented by the second node did not register on the Promise generated by the first callback (the `Q` of the second node is not the `R` of the first node). With this information it is possible to conclude that the data flow will not be propagated from `increment` to `logValue`. Furthermore, and now this is the important part to identify the Broken Promise bug, both nodes have the same `Q`, meaning they both registered on the same Promise object.

There is a single case that requires special attention: the case where both an `onFulfill` and an `onReject` reactions are registered to the same Promise. This is a completely legitimate case if, and only if, both reactions generate the same Promise, handling both success and reject of a single point in the Promise chain. In practice, this happens when a call to `then` receives two callbacks. This generates two nodes for the same Promise (same `R`), each handled by a different callback. If this case was not taken into consideration, it would result in false positives, because both reactions will be registered in the same Promise (same `Q`). It is simple to filter, because both reactions will register on the same Promise and generate the same dependent queue object (same `R`). The tool just needs to ignore cases with the same `Q` and the same `R`. During execution, at that point, depending on if there is an exception, the program will follow either one or the other reaction, but they both generate the same Promise.

```
1 const p1 = new Promise( ( res, rej ) => {
2   rej( 1 );
3 } );
4
5 p1.then( console.log );
6
7 p1.catch( console.warn );
```

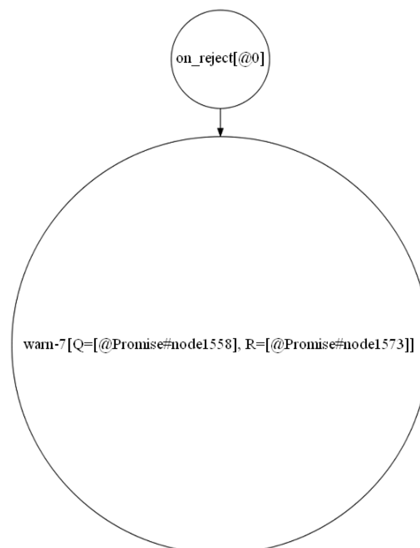


Figure 5.2: A Broken Promise bug that will always result in an uncaught exception is shown on the left with the respective graph on the right.

While all this information may be displayed in the graph, it is not trivial for the developer to generate the graph and then carefully look into all the generated information, making sure nothing is overlooked. Besides, for large programs, with heavy use of Promises and other asynchronous APIs, the graph will take larger proportions and it will be hard to analyze solely by eye. Furthermore, there are several nodes

that will not present their `Q` and `R` values, because they are native functions. This is a design decision taken by the Async-TAJS developers.

Fig. 5.2 illustrates yet another interesting example of a Broken Promise where both `onFulfill` and `onReject` reactions register on the same Promise. The user may be lead to assume the `catch` will capture the exception thrown by `reject`, but this is actually a case of Broken Promise, considering both `then` and `catch` register on the same root Promise, but generate different Promises (each method call generates its own unique Promise). This differs from the previous case of two callbacks registered by `then`, because two distinct Promises are generated from the callbacks (one Promise per callback). It is also important to recall that `catch` is just syntactic sugar for a call to `then` without a first argument and with the argument to `catch` being the second argument to `then`. What is happening here is that when the Promise is rejected, it will propagate the value as an exception, that has two paths to follow, since the Promise has been forked:

1. It will go into the omitted (thus default) `onReject` reaction registered through the `then` on line 5, which will simply throw the argument received. The newly generated Promise (from the `then` call) does not have any more reactions registered, so the exception will be thrown into the top level.

2. It will go into the `console.warn` reaction and be contained.

The graph displayed on the figure does not show the totality of the information and just shows that the default `onReject` reaction will occur before the `console.warn`, which simply does not make clear what the problem is. Notwithstanding, the necessary information is available on the internal structure of the graph.

To facilitate the debugging process, we have implemented functionality that receives the internal graph structure and looks for the data about the `Q` and `R` values of each node to apply the approach described in Section 4.1. As a result, the tool outputs a file pointing to each Promise that was forked (to the creation of that forked Promise) and to where it was forked (all the multiple reactions that register to it), with location by line and column numbers. Fig. 5.3 exemplifies the output for the left snippet on Fig. 2.1.

```
Possible Broken Promise between positions: 9:1 and 11:1!
Forked from position: 5:11.
```

Figure 5.3: Broken Promise verification for the left snippet presented in Fig. 2.1.

The deliberate choice of stating "**Possible** Broken Promise" is due to the fact the forked Promise may be intended. In that case, it is not a bug; thus, not a Broken Promise bug. With this information, the developer has all the data they need to check what is happening at those locations and fix the issue.

Implementation details For Async-TAJS to be able to detect cases of this bug, it has to analyze the internal structure of the resulting Callback Graph for the specific input program. All the necessary information for an automatic detection is present on the Callback Graph, either directly or indirectly. Since this tool is written in Java, we created a new class that receives the internal structure of the Callback Graph, after the static analysis on the program has completed. At that point, the Callback Graph object contains the nodes representing each asynchronous callback with the respective information about queue objects (`Q` and `R` attributes, previously mentioned). Then, there is a method specific to handle the analysis for the specific bug of Broken Promise. This method groups the nodes by their `Q` and maps them to their respective `R`. So now we have a map between each queue object (the object taking the registration) and their dependent queue objects (the objects generated by registering on that queue object). For the dependent queue objects, we are not interested in duplicate values (same `R`), because, as mentioned above, the only way for more than one node to have the same `R` and the same `Q` is if these nodes represent different callbacks for the same Promise. We are interested in finding different Promises (different `R`) being generated from the same Promise (same `Q`). Now, we just have to check which `Q` map to more than one `R`. For each `R`, a new Promise is forking the same root Promise. We iterate the map, filtering `Q` entries with less than two `R`, then sort by the Promise identified by `Q`, by location in the file (line and column number) and then, for each `R` we also sort in the same manner. To facilitate managing and sorting file locations, an auxiliary class was created, receiving the TAJS object containing the location information and implementing the `Comparable` interface. Finally, we create the output, alerting for the possible Broken Promise cases, sorted by Promise fork and where that fork occurs, also sorted by file location, as can be seen in Fig. 5.3. To put this into practice, we also needed to change a core class to access the Callback Graph object and create an instance of our `CallbackGraphAnalysis` class. We also added an option to allow the user to enable/disable the analysis to the Callback Graph. All of this code was documented with JavaDoc.

5.3 Unexpected Execution Order Bugs

Following the approach described in Section 4.2, the tool will be looking for writes on data that will happen before reads on the same data, for different execution timing contexts.

Some execution timings cannot be modeled accurately, since they depend on external factors. In general, microtasks can be modeled correctly comparatively to other microtasks (including `process.nextTick`), as long as they do not depend on external factors to resolve. Macrotasks can be modeled as occurring after the microtasks, in case the macrotasks are independent (e.g., `setTimeout` without being wrapped in a Promise chain) and in case the Promises do not depend on external factors. For instance, if a `setTimeout` appears before a Promise, on the same context, and the `setTimeout` writes on data that is

read on the Promise, it is possible there is an Unexpected Execution Order bug here, especially if that same data is never read again.

However, between macrotasks it may not be possible to know when each will trigger in relation to another. For example, if two `setTimeout` appear one after the other and the first one has a delay greater than the last one, they will not necessarily be executed in the reverse order. For instance, if there are two `setTimeout` one after the other, with delays of `1` and `0`, respectively, the first one, even though it has a delay greater than the second one, will (most likely) execute first. This can even be dependent on the system implementation. For greater values of the delays, the order of execution seems to follow the delay values, but there is nothing guaranteeing it will always have that behaviour.

The Callback Graph model is on par with this behaviour. That and the way some asynchronous tasks operate (time-dependent, disk access, etc.) is why it models the execution order of timers and async I/O as unspecified. There is no way to guess when some external operation will complete during a static analysis. The best is for the developer to make sure the execution order is guaranteed by their code (e.g., wrapping these operations with Promises). With this implementation we intend to alert the developer for cases where the order is not guaranteed to be consistent (e.g., a timer and disk write whose callbacks manipulate the same data, without any specific chaining to guarantee order).

As an automatic solution to track this bug was being developed, it was made clear that the Callback Graph model was not enough to help track this bug. The model does not cover synchronous code, but we need to keep track of what happens on the synchronous code, to later compare it with what happens asynchronously, checking the possibility of writes happening only in an asynchronous environment, while reads happen synchronously. Fig. 2.3 illustrates precisely this: `validateString` reads synchronously while `storeString` writes asynchronously. However, the Callback Graph also does not record all the operations that happen in the code. It just gathers asynchronous callback calls and stores them in an abstract-like representation that does not keep reads and writes of data happening in them.

Fig. 5.4 displays the Callback Graph for the code snippet in Fig. 2.3. The first node of the graph corresponds to the anonymous callback `() => ( this.str = str )`, in line 9, that will store the string. The second node represents the callback that will be executed in the next line, executed after the previous callback, as they both are part of the same Promise chain. This second node represents the default `onFulfill` reaction. Even though the graphical representation of the graph does not display it, the internal structure of the graph stores information about this callback (line number, etc.). This default reaction is used because `onCreate` is `undefined` (`storeString` does not receive a second argument).

The graph only shows that there is a callback in line 9 (the one with the assignment, that stores the string) that always happens before a default `onFulfill` reaction. Peeking into the internal structure, it is also known that both callbacks are part of the same Promise chain. To the developer, this does not mean much, in terms of tracking Unexpected Execution Order issues. The only use this information could have

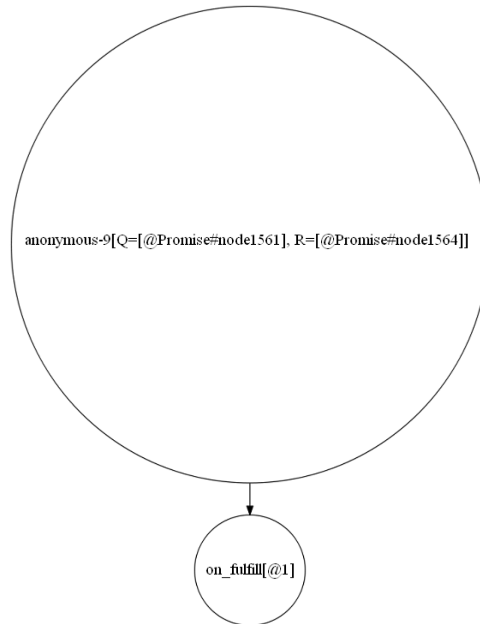


Figure 5.4: Callback Graph model for the code snippet displayed in Fig. 2.3.

is if the developer knows beforehand that the default reaction should be a specific callback. Meaning `onCreate` should be a valid callback. But for the developer to know that already, they would have first actually passed the argument with the callback. More so, since `storeString` returns a Promise, the user can chain to the returned Promise, resulting in not passing a callback, but, instead, chaining it. This would still generate an `onFulfill` node on the graph, because the second parameter to `storeString` would not be passed in the end. Default reactions are not synonym that something is missing that should not be missing.

Considering again the code from Fig. 4.1, its corresponding graph is presented in Fig. 5.5. This graph presents more issues, since there are several clear happens-before relationships that the graph fails to display.

The two first nodes represent each of the `setTimeout`. These operations may happen in any order between each other, since, regardless of their delay value being random, the Callback Graph models the timers API order as undefined, but it is guaranteed one of these will be the first asynchronous operation to execute, because all other asynchronous operations depend on them, via the Promises wrapping the `setTimeout`. The problem is, according to the graph, each `setTimeout` will occur always before any of the reactions registered for any Promise, by presenting a directed edge from both of them to the `on_fulfil[@0]` node. This node represents the default `onCreate` reaction that will be used (since no argument is passed). This is not true, because each `setTimeout` is wrapped in a Promise that has a chain depending on it. So once a `setTimeout` triggers, since Promises are microtasks, the dependent Promise callbacks for the triggered `setTimeout` chain will always resolve before another (the

Figure 5.5: Callback Graph model for the code snippet displayed in Fig. 4.1.

other) `setTimeout` executes. This means the graph can be misleading, since it is supposed to represent happens-before relations. The flow with a starting point on a Promise chain triggered by a `setTimeout` will always complete before the other `setTimeout` triggers. In practice, the two `setTimeout` will not execute before any other microtask executes; only one will. The graph is not very explicit in displaying this as it may come from a design limitation, due to modeling the timings for `setTimeout` as unspecified.

Furthermore, the default `onReject` reaction, represented by the `on_reject[@0]` node, automatically registered by the `then` in line 21, will never happen before the default `onFulfil` reaction, represented by the `on_fulfil[@3]` node, automatically registered by the `catch` in line 22, because if `onReject` happens, it throws, orienting the flow to the `console.warn`. So the default `onFulfil` reaction, represented by the `on_fulfil[@3]` node, will never happen in that case (there should not be an arrow in the graph).

Likewise, if the flow goes into `console.log` (which does not throw), the `console.warn` cannot happen. So the `console.warn` will never happen if `console.log` happens. Either one of the other executes (assuming `console.log` does not throw). These last cases only have to do with Promise flow and are not dependent on the modeling of `setTimeout` timings.

Regardless of all this, what the developer needs is not a graph to analyze. It is more useful for the developer to receive explicit information about the problem. For an automated tool to know the above, it would need to know the same the developer knows beforehand, to distinguish when a default reaction should actually be a specific callback, for instance. This is unreliable, not to say impossible for most cases. Plus, the major issue is the data dependency between two different execution contexts: one synchronous and the other asynchronous. The graph also does not show (nor its internal structure contains) any information about data reads and writes. Concluding, the Callback Graph model is not enough to identify Unexpected Execution Order issues.

Extending the Callback Graph Given the above, we propose an enhanced version of the Callback Graph that captures each different context. It is a combination of a flowgraph and the Callback Graph that provides more useful elements to the analysis. It follows the approach described in Section 4.2, representing execution timings as nodes. Any timings that cannot be precisely defined at the static analysis level will generate all possible paths for where they could happen. Besides, each node will store the data it reads and writes. With all this information, it is now possible to reason about the timings of reads and writes and determine if there are reads before writes and/or no reads after writes.

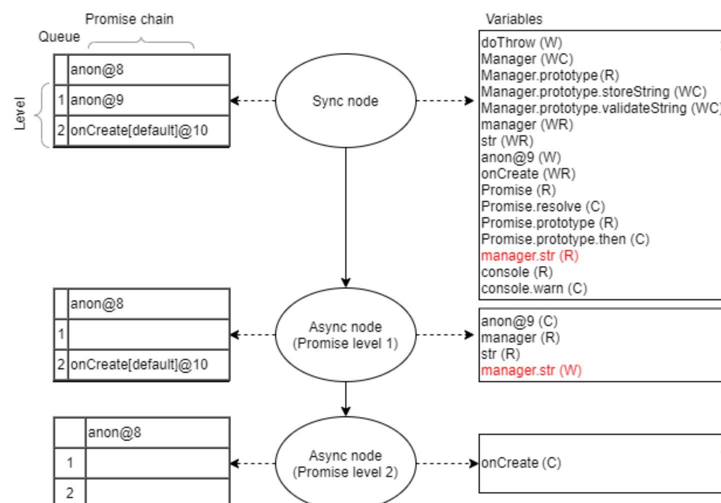


Figure 5.6: Enhanced model to reason about Unexpected Execution Order issues for the code snippet displayed in Fig. 2.3.

On Fig. 5.6 we present a prototype of what should be the result of the analysis for the code presented in Fig. 2.3. This structure provides all the necessary information for an automatic analysis to detect data

racing issues triggered by Unexpected Execution Order bugs.

As it can be seen, this model has a node for each distinct execution timing. The first node is the one that corresponds to the synchronous code. Then, the microtasks; in this case, the Promises callbacks registered in `storeString`. On the right side, there is a board for each node with the variables accessed during that execution context. It does not matter the order here within each context, since what we are interested in is in the order between different contexts. The letters after each variable name mean:

- `R`: Read;
- `W`: Write;
- `C`: Call (this is also a read, for all purposes);

Internally, each of these variables should have a unique ID, to avoid name clashes and to also keep the same reference on different variable names.

On the left side of each node there is the respective Promise queue at the node's exit point. This queue has the shape of a tri-dimensional matrix. The rows are for the level of the chain and the columns are for each Promise chain, in order of appearance. Each cell is itself a queue, to support the scheduling strategy described in Section 4.2, for Promises created during the execution of asynchronous callbacks and for forked Promises.

Analyzing this model, we can see that in the first variables' board there is a read happening on `manager.str`, but not a write. And on the second variables' board there is a write on the same `manager.str`. There is not a read on the second board nor on the third. So this is a potential Unexpected Execution Order issue. The third board only shows `onCreate`, which was `undefined`, thus the default identity reaction (i.e., `v => v`) is used instead.

There are other variables that were read before being written, like `Manager.prototype` or even `Promise`, but these are values created (written) natively by the JavaScript engine.

Correcting the code, by applying the suggestion on the left from Fig. 2.4, and analyzing again, we now have the result presented in Fig. 5.7.

We can see that now there is not a single write on any address happening on any board that comes after a board where a read happened on that same address. For the particular case of `manager.str`, it is written on the second board and only read on the third board, guaranteeing the correct order of the operations.

This model collects all the necessary information to reason about how data is accessed and manipulated during the different execution timings, allowing it to detect Unexpected Execution Order bugs.

Implementation details This feature remains to be implemented. We started developing the implementation of this model, but due to time constraints and to the immense volume of code composing

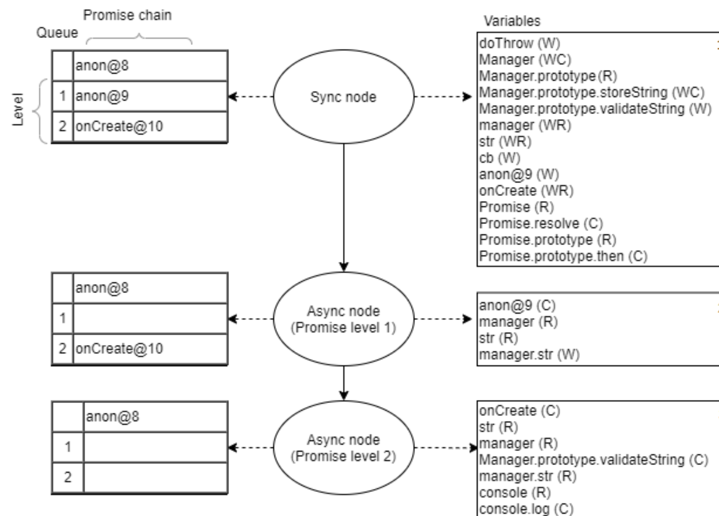


Figure 5.7: Enhanced model to reason about Unexpected Execution Order issues for the corrected version of code snippet displayed in Fig. 2.3.

Async-TAJS, it was not possible to conclude it. The lack of documentation for the tool also made it hard to understand the code at first. Before developing this new model, we explored the internal structure of the Callback Graph, which took time, to no avail. We tried to understand if, from its internal structure, it was possible to access the data reads and writes. But all this information was being abstracted while the Callback Graph was being built. This was our original idea: to extend the Callback Graph. So then we tried to use the Flowgraph structure, to access the necessary information, since this structure does record the reads and writes of data. However, this structure is also one of the main features of TAJS, so the code that generates it is scattered through hundreds of Java files and thousands of lines of code, without detailed documentation to help, requiring large amounts of time to read the code and understanding where the relevant actions are happening. Besides, the Flowgraph does not take particular attention for asynchronous operations, in a similar way the Callback Graph does. So, in the end, we tried to combine both structures, by populating the Callback Graph with some of the information gathered by the Flowgraph. This was proven to be difficult, due to how the two models are processed internally and how they each store the relevant information. In the end, the best would be to have a dedicated model: the one we described in this Section. It is a mixture of both the Callback Graph, because it handles the different asynchronous execution timings, and the Flowgraph, that gathers and records the information necessary to understand which data is being read and written. Due to time constraints, this remains to be concluded. In practical terms, our idea was to add one extra method to the `CallbackGraphAnalysis` to handle this case, just like there is one method to deal with the Broken Promise bug. But due to how auxiliary methods needed to be created, it is cumbersome to have all this in one single class. So then our idea was to have the `CallbackGraphAnalysis` to implement a strategy design pattern, for example, and extracting each bug-handling method (and their auxiliary methods) to dedicated classes (one class

per concurrency bug). Possibly into its own package to completely abstract our additions.

6

Evaluation

Contents

6.1 Datasets of JavaScript Asynchronous Bugs 57
6.2 Broken Promise Bugs 59
6.3 Unexpected Execution Order Bugs 61

In this chapter, we describe the datasets used to evaluate our work and we present the evaluation results.

6.1 Datasets of JavaScript Asynchronous Bugs

Studying specifically the asynchronous behaviour of JavaScript is a fairly recent research topic, so we were unable to find dedicated JavaScript datasets for asynchronous code. Thus, we designed our own set of micro-benchmarks with asynchronous code ranging from very simple, common cases, to more complex edge cases. These datasets are publicly available as a GitHub repository.[1]

Micro benchmarks One of the goals with this set was to exercise the current state of Async-TAJS for both simple cases and corner cases, to trigger any possible bugs and limitations the tool might have. Then, for each specific motivating example, smaller sets of benchmarks were designed to test those cases specifically. Again, each set ranging from simple cases, to complex edge cases. The two specific sets for each of the notable cases cover all possible (or the most possible) cases that could be encountered in practice.

All datasets contain code with bugs and code without bugs, to aid in a better finding of false positives. The files are organized following a simple pattern. They are named starting with a letter, to identify the purpose of the code they contain and how they are to be used:

- `C`: Is for code that is correct, regarding the programmer's intentions. This code will execute in the way the developer wants it to.
- `I`: Represents code that contains asynchrony bugs. The developer wanted the program to do something, but an unintended bug leads to unexpected results.
- `O`: Stands for other examples that do not necessarily add anything new in terms of concepts, but contain code organized more like it appears on real-world programs, or simply display edgier patterns.

To execute the Async-TAJS with the necessary parameters and to manage the results generation we programmed a small script. This script does some pre parsing, on the files to be analyzed, calls Async-TAJS with the desired configurations, setting a QR-sensitivity analysis, and then gathers the results in an organized way.

The first phase was to apply Async-TAJS to the general set of benchmarks, containing 74 programs (32 of type `C`, 28 of type `I` and 14 of type `O`). These benchmarks were composed of:

[1] GitHub repository for the datasets we designed: https://github.com/BernardoFuret/Thesis-dissertation-Datasets.

- Promise reaction registration (`then`, `catch`, `finally`). The tool analyses these cases correctly, except `finally` which has a buggy implementation
- Promise synchronization methods. Only `Promise.race` is supported, but may generate incorrect results for the case where there could either be a fulfillment or a rejection first on the Promises passed to the method. The other synchronization methods are not implemented so are simply not marked as asynchronous and ignored.
- Usage of `queueMicrotask`. Not implemented by the tool, thus ignored.
- Usage of `process.nextTick`. Same as above.
- Usage of `setTimeout` and `setInterval`. `setTimeout` works correctly. `setInterval` has a buggy implementation.
- Usage of `setImmediate`. Not implemented, so it simply is not marked as asynchronous and is ignored.
- Events and async I/O. The tool has issues with importing some libraries required (for Node.js).

In addition to all those, there are some programs that combine the multiple APIs and test more specific behaviours, like dynamic registration of reactions, loops or calling asynchronous APIs inside already scheduled operations.

Tables 6.1, 6.2 and 6.3 detail which asynchronous code is exercised by each `C`, `I` and `O` benchmarks, respectively.

Our primary target environment is Node.js, although most of the test programs work in the browser as well. The browser does not support operations like `process.nextTick`, however. On the other hand, it offers other asynchronous APIs that we did not exercise, like `window.requestAnimationFrame`.

General datasets We intended to also test the result of our work with real-world examples of code, from real programs and libraries. Unfortunately, the TAJS itself has some limitations. When it encounters

C Benchmarks

	C00	C01	C02	C03	C04	C05	C06	C07	C08	C10	C11	C13	C14	C15	C16	C17	C19	C20	C21	C22	C23	C24	C25	C26	C27	C28	C29	C30	C31	C32	C33	C41
Promise (then;catch)	✓	✓	✓		✓	✓	✓	✓	✓						✓	✓	✓														✓	✓
Promise synchronization					✓	✓	✓	✓	✓																							
Promise finally										✓																						
queueMicrotask																								✓	✓							✓
process.nextTick																										✓	✓					✓
setTimeout											✓			✓		✓		✓	✓												✓	
setInterval																				✓	✓											
setImmediate												✓		✓								✓	✓									
Async I/O													✓	✓														✓	✓			
Events				✓																										✓		

Table 6.1: Table illustrating which C benchmarks use which asynchronous APIs.

I Benchmarks

	I01	I02	I03	I04	I05	I06	I07	I08	I09	I10	I11	I13	I14	I15	I16	I17	I18	I19	I20	I21	I22	I30	I31	I32	I40	I41	I42	I45
Promise (then;catch)	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓										✓	✓	✓	✓	✓	✓	✓
Promise synchronization																						✓	✓	✓				
queueMicrotask										✓						✓												
process.nextTick											✓						✓											
setTimeout													✓													✓		
setInterval														✓														
setImmediate															✓												✓	
Async I/O																		✓										
Events																			✓	✓	✓							

Table 6.2: Table illustrating which I benchmarks use which asynchronous APIs.

O Benchmarks

	O000	O001	O006	O007	O008	O010	O011	O012	O013	O014	O016	O017	O020	O021
Promise (then;catch)	✓	✓	✓	✓	✓			✓	✓	✓	✓	✓	✓	✓
Promise synchronization				✓	✓									
async/await						✓								
queueMicrotask								✓						
process.nextTick								✓						
setTimeout				✓	✓	✓	✓	✓		✓		✓		✓
setInterval														
setImmediate								✓						
Async I/O								✓						

Table 6.3: Table illustrating which O benchmarks use which asynchronous APIs.

a JavaScript program that requires built-in Node.js modules, it simply cannot locate the modules [25]. These built-in modules come with the Node.js installation precisely because they contain necessary code to perform common operations. So it is more often than not for Node.js programs to import at least one of those modules, making it difficult to properly evaluate our work. With code destined for the browser, the tool faces issues from HTML code that loads third party libraries via the `<script>` tag from external repositories.

Nonetheless, we used our own programs and the benchmarks assembled by the developers of Async-TAJS to evaluate our tool. Our tool was able to process correctly all 36 cases from the Async-TAJS dataset.

We also decided to analyze the 6 libraries used to evaluate Async-TAJS by its authors, but only to evaluate the Broken Promise bugs finding implementation. The libraries used were controlled-promise [51], fetch [52], honoka [53], axios [54], Pixiv API Client [55] and Glob [56].

For the motivating examples, each was posteriorly analyzed and evaluated separately, through more dedicated datasets that we created, respectively for each type of bug, with the above limitations in mind.

6.2 Broken Promise Bugs

To evaluate the implementation of Broken Promise bugs finding, we assembled a set of 13 small programs: 6 of those contained code that is correctly designed, when it comes to the programmer's inten-

tions; the other 7 contain Broken Promise bugs, resulting in undesired results due to the Broken Promise issues they contain.

The expectation is for the tool to not report any Broken Promise bug in `C` files and report all of the Broken Promise bugs in the `I` files.

The results:

- for `C` files: Of the 6 files, 2 accused possible Broken Promise issues. This happens because there are forked Promises in those 2 files. Our implementation follows the approach of reporting all forked Promises as possible Broken Promise issues (see Section 4.1). In these 2 files, the programmer wanted to fork the Promise, but it is impossible for an automatic tool to know it unless it has the same knowledge about the program the developer does.

- for `I` files: The analysis, on all of the 7 files, reported Broken Promise bugs and all of the Broken Promise bugs that were present in each program were found and reported correctly.

The results show that there are only 2 false positives, accompanied by 0 false negatives. The 2 false positives are impossible to automatically detect as intended behaviour, since they depend solely on the intention of the programmer. In any case case, as previously mentioned, these do not constitute a problem, because forking Promises can be considered a bad practice or an anti-pattern by some communities [46, 47].

As for the Async-TAJS benchmarks, there was 1 report of a Broken Promise bug, which was a true positive, and there were 0 false negatives.

Regarding the real-world modules used by the Async-TAJS authors, 3 of them (fetch, axios and Glob) were not possible to evaluate due to Async-TAJS limitations when requesting global Node.js modules. Other 2 (honoka and Pixiv API Client) of the remaining ones presented issues that made TAJS halt the analysis earlier than it should. In these cases, no Broken Promises were detected, even if introduced in code written using these libraries, due to the sudden halt of the TAJS analysis. Finally, the remaining library (controlled-promise) was analyzed correctly all the way through. If we create small programs that uses this library and contain Broken Promise bugs without making use of the library's Promise-using APIs, it will find the Broken Promise bug. However, the Async-TAJS was reporting variables from the library that should hold Promises as `undefined`, preventing the correct report of the broken Promise bugs associated with these variables, in case the Broken Promise bugs were introduced by making use of the library's APIs (that use these variables).

6.3 Unexpected Execution Order Bugs

To evaluate the detection of this asynchrony bug, we have assembled a set of 10 main micro programs and 4 specific micro programs. These just combine asynchronous code with synchronous code and different asynchronous APIs. It uses primarily Promises and `setTimeout` for the asynchronous APIs, considering these are the ones fully and properly supported by TAJS.

Similarly to the previously described bug, this dataset is comprised of 5 files of type `C` and 5 files of type `I`. In addition, there were created 4 files of type `O`, representing very complicated execution timings coming from the scheduling or complicated patterns. These aim to test how the model adapts to complicated and/or complex code designs (Promises scheduling from asynchronous callbacks, with multiple reactions, etc.). Finally, beyond those 14 micro programs, at least 1 other was created, being a variant of one of the files comprising the 14 files dataset. This was useful to unveil some limitations of the implementation of the Callback Graph in Async-TAJS.

The implementation of this functionality has yet to be concluded, so it is not possible to evaluate the tool itself.

Nevertheless, the model was designed to detect all cases of this bug, considering the way we defined this problem in Section 4.2.

Detailed example To illustrate how this proposed model detects Unexpected Execution Order bugs, we manually analyzed the benchmark whose code is presented in Fig. 4.1. This is a complex example, with variable, nondeterministic results, due to the random delay passed to each `setTimeout`, simulating real-world applications connecting to external sources.

Fig. 6.1 shows a visual representation of that analysis. This visual model attempts to demonstrate how the information is gathered and how its data is related. To avoid clogging the image with irrelevant information to the problem, some variable names were omitted from the variables boards (e.g., `Manager.prototype`, `Promise.prototype`, `Math`).

This program peculiarity is that each `setTimeout` receives a random delay value. So either one can trigger before the other. In any case, the synchronous code will always execute first. On the image, we see the flow starts with the synchronous node, as it always should. After the synchronous code is done, we see which Promises and which other tasks were scheduled. Remember, the boards pointed by the dotted arrows from the node represent the exit state of the node. Albeit not made explicitly in the image, to avoid getting cumbersome, the pending Promises will only settle after a respective `setTimeout` resolves (each Promise wraps one specific `setTimeout`). So they each will only settle after each `setTimeout` is ready. In other words, nothing can happen until one `setTimeout` triggers. The Promises that are pending are not placed in any particular order on the queue. In fact, to be strictly correct, pending Promises should be awaiting in another place; they should only be pushed to

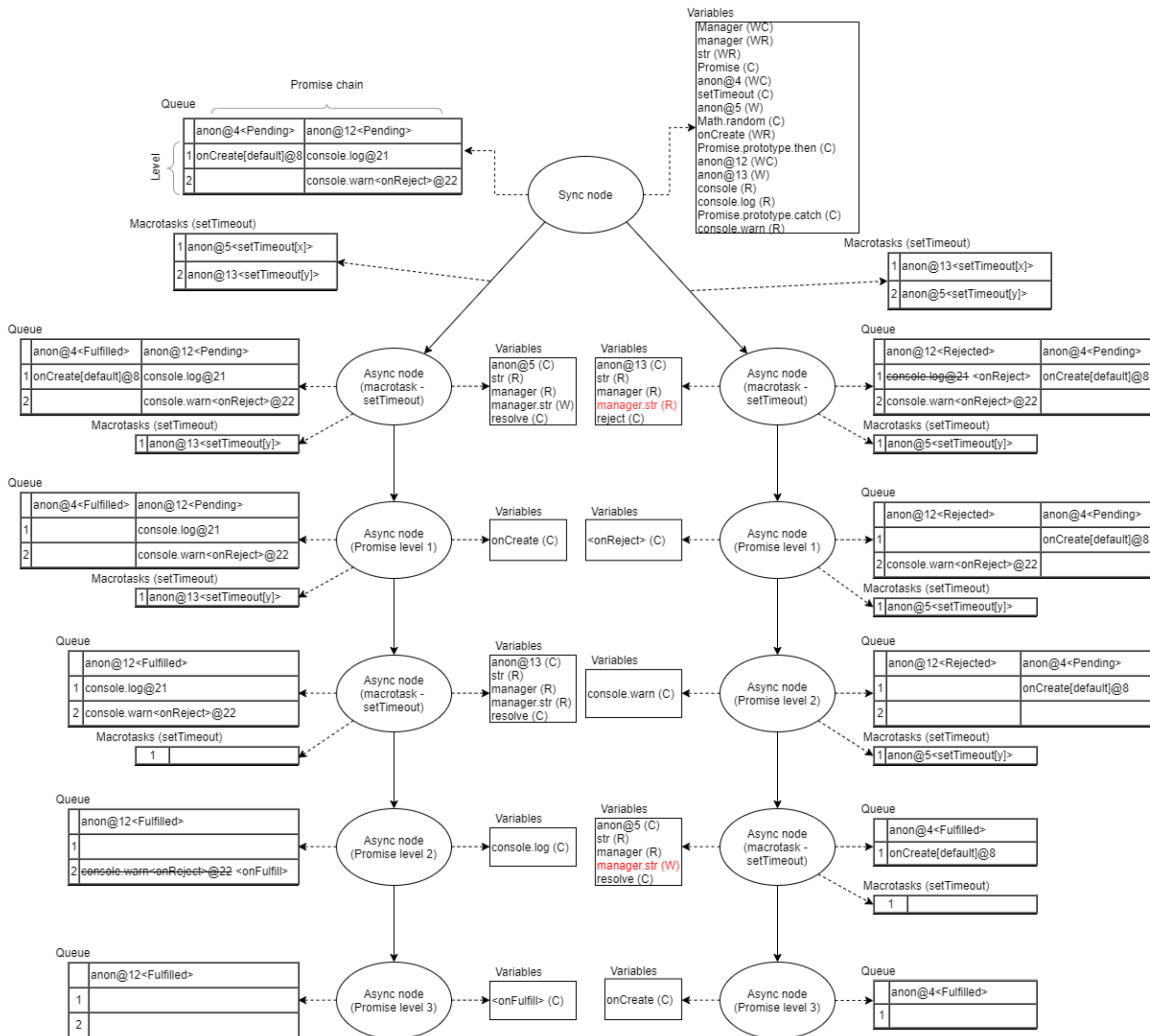


Figure 6.1: Enhanced model to reason about Unexpected Execution Order issues for the code snippet displayed in Fig. 4.1.

the microtasks queue once they are ready. This microtasks queue is referred simply as ”Queue”, in the image. The await for the `setTimeout` triggering is behaviour handled externally by the responsible APIs for these operations. Thus, the model in the image only represents, in the macrotasks queue, the callbacks for events that are ready. Through a static analysis, it is impossible to foresee which of the two `setTimeout` will trigger first, so we have to account both cases.

The outer edges of the synchronous node have dotted arrows indicating the state of the macrotasks queue. Here, the dotted arrows come from the outer edges (instead of from the node itself), because each edge represents one possible path. The `x` and `y` on the macrotasks queue represent the `setTimeout` delays. Assuming `x` is less than `y`, on the left outer edge of the synchronous node, we have the case where the callback scheduled by the `setTimeout` in line 5 will execute first. On the right outer

edge, we have the case where the callback scheduled by the `setTimeout` in line 13 will execute first.

Focusing first on the path from the left outer edge, we now have an asynchronous node, representing the calling of the callback scheduled by the `setTimeout` in line 5. Looking at the queues, we now see that one of the Promises is fulfilled. This is the Promise created in line 4 that was wrapping the `setTimeout` in line 5. The macrotasks queue was also updated, to remove the callback that was just executed. At this point, we have one Promise fulfilled, that has one reaction registered (seen in the queue as `onCreate[default]@8`), which corresponds to the default reaction, since the parameter `onCreate` was not used; and one `setTimeout` callback ready (because it is in the macrotasks queue). And, of course, there still is one pending Promise. Microtasks always take precedence over macrotasks. So the Promise reaction is called. After that, we have only the ready `setTimeout` callback and the pending Promise. Therefore, the `setTimeout` callback is executed, represented in the second asynchronous node labeled with `macrotask - setTimeout`.

Looking at the queues, we now see the macrotasks queue is empty and that the Promise that was pending is now fulfilled. It had two reactions registered, at different execution timings (levels), due to the chained methods `then( console.log ).catch( console.warn )` from lines 21 and 22 This is visible in the microtasks queue: the first level has the `console.log` scheduled from line 21 and the second level has `console.warn`.

All that is left is to resolve the two Promise reactions, represented by the two following asynchronous nodes.

To resolve the first reaction, `console.log` is called. Since the Promise returned by the call of this reaction fulfilled (i.e., did not reject; `console.log` did not throw), the reaction registered through `catch` (`console.warn`) will not be reached and the default `onFulfill` identity reaction (`v => v`) is used instead (do not forget that `catch( cb )` is syntactic sugar for `then( undefined, cb )`).

There is nothing left to do in the queues, so this path is complete. Analyzing it (from the synchronous node to the last asynchronous node), in particular, checking the variables boards, we verify that there is not a single read (`R`) or a call (`C`) on a variable defined by the user (that is, ignoring the likes of `console` and other native data) before it has been written to (`W`). This is correct. In practice, following the code, what happened was the `setTimeout` that stores the string (coming from `storeString`) triggered before the `setTimeout` that verifies if the string was stored (coming from `verifyString`). The string was stored and only then validated, resulting in success.

But the analysis of the program is not complete yet. This was just one of the two possible paths. Now we will be focusing on the path resulting from the right outer edge of the synchronous node.

The first asynchronous node that appears is relative to the call of the callback scheduled by the `setTimeout` in line 13. This callback will make some assertions, visible in the code on lines 14 and 16, with the `if`/`else` statements. `this.str`, at this point is `undefined`, so the Promise will be rejected.

`this.str` appears in the variables board for this node as `manager.str`, being read. The Promise that rejected had two reactions registered: one through the first parameter of `then` and the other through `catch`. Since the Promise rejected, the reaction registered through `then` will not be reached and the default `onReject` reaction is used instead (this reaction simply throws the argument received). This can be seen in the image, by looking at the queue pointed by this first asynchronous node. The `onFulfill` reaction (which was `console.log`) is struckthrough. The macrotasks queue now contains only one ready callback. Like for the other path, microtasks take precedence over macrotasks, so the settled (ready) rejected Promise will have its ready reactions executed first. This means the default `onReject` is called, simply throwing the argument. It is represented by the next node: the asynchronous node referring to the first level of Promise reactions in this path. The state at the exit of this node shows that there still is one Promise callback ready. So it is executed before any macrotasks' callbacks.

Represented by the next asynchronous node, referring to the second level of Promise reactions, `console.warn` is called, since it was the reaction scheduled in case there was a rejection (e.g., an exception being thrown) immediately prior to it. Exiting this node, only a pending Promise exists, along with the ready macrotask callback.

Since the only ready callback is the one scheduled by the `setTimeout` in line 5, it is time to call it. This is where `this.str` will be written for the first time. This callback also resolves the Promise wrapping the `setTimeout` that scheduled the callback. Finally, the last Promise is not pending anymore and its only reaction, the default `onFulfill` reaction (since the parameter `onCreate` was not used), is called.

To conclude the analysis, this path needs to be checked for the existence of data being read before being written. There is one case, highlighted in red: `manager.str` is being read when the callback from the `setTimeout` is called, but it is only written when the callback scheduled by the `setTimeout` in line 5 called, three execution timings later, without ever being read again, indicating a potential Unexpected Execution Order bug.

The image shown is only a visual representation to help explaining how this model can capture and, from the captured information, infer over it in order to find Unexpected Execution Order bugs. The image simplifies several concepts, like the microtasks queue, especially on how Promises are handled while pending and how their reactions are stored internally. Having all that information displayed correctly in the image not only would take much more space, it would also contribute to a more complex image, making it more complicated to understand where the relevant information is, just to exemplify how it can be evaluated.

This model we propose will always be able to identify cases of data dependencies like this one (reads before writes) that are indeed incorrectly programmed. Furthermore, this model, with this information, can also track other concurrency bugs. For example, if it reaches a state (the last state) where there are only pending Promises with nothing else left to do. This is a Dead Promise bug, as described by [8]. This

model also gathers enough information that can be tuned to be able to catch more specific cases. For example, if there are multiple writes in different execution timings without ever being read and multiple reads on execution timings before the writes, these could also be Unexpected Execution Order issues. In the end, all the information necessary to reason about execution timings, data and how they relate, leading to Unexpected Execution Order bugs, is gathered by this approach and this model.

7

Conclusion

Contents

7.1 System Limitations 67
7.2 Future Work 70

JavaScript provides a feature that allows concurrency in a single-threaded main environment, managed by the event loop, allowing to schedule operations while they are not ready, keeping other code running. With this powerful behaviour comes complexity, making JavaScript code prone to concurrency bugs. Therefore, techniques to reduce the amount of these bugs programmers negligently introduce in their code must be developed.

In this work, we developed a solution capable of detecting asynchrony bugs in JavaScript code. We used the Callback Graph model [3], implemented in Async-TAJS [4], to analyze and reason about asynchronous JavaScript code. Furthermore, we extended the implementation to detect two specific cases of JavaScript concurrency bugs: Broken Promise bugs and Unexpected Order bugs. Broken Promise bugs are now detected correctly. As for Unexpected Execution Order bugs, we show that the Callback Graph model is not enough to correctly detect these issues. Thus, we present a new model, capable of gathering all the necessary information to infer if this bug is present in JavaScript code.

A further contribution of our work is a new dataset of JavaScript code containing different types of asynchrony bugs, along with two smaller, more focused datasets. These datasets can be used by other tool developers to evaluate their tools.

Despite encountering several problems and limitations during the development of this project, our results show that our proposed solution is capable of detecting the asynchrony bugs considered in our study.

7.1 System Limitations

During different phases of our research, we encountered several problems and limitations with both Async-TAJS and the Callback Graph model.

Async-TAJS limitations One limitation that was already mentioned on some of the previous Sections is the fact that Async-TAJS itself has issues requiring some other JavaScript files needed, which posed some difficulties on the evaluation of larger, real-world programs containing dependencies from global packages/modules.

Other limitations come from the fact that Async-TAJS does not support JavaScript syntax above the ES5 [57] specification (for example, it does not support arrow functions, spread/rest operator, trailing commas). So transpiling code was necessary. We built a script to automate the whole process (transpiling and analysis). The main problem with transpiling is if there is code with dependencies, which requires all the necessary code to be transpiled (not only the code on the target file for analysis). This would require our script to analyze the target file in order to find dependencies and then track the location of the dependencies transpile them and recursively apply this approach. Ideally, this should be done

by Async-TAJS itself, as it is the one parsing the code. To overcome this complication, we decided to bundle the file we would be analyzing along with its dependencies, as if it was a standalone project, and transpile the bundling. We use Webpack [58] for the bundling with Babel [59] to perform the code transformation back to, at most, ES5. Although this now allows Async-TAJS to perform the analysis, it can hurt debugging, because the file being analyzed now contains all the code involved, often resulting in dozens of thousands of lines of code (considering minification is already disabled). As a consequence, now bugs instead of being on specific files with few lines of code, they will be all on the same file, containing much more lines of code, making it hard to spot the right place where it needs to be corrected, on the original file.

Even after bundling and transpiling the code, we faced more problems from the Async-TAJS analysis as it is unable to process code that contains `switch` statements with `default`-case not being at the last position, for example, despite being perfectly accepted by JavaScript.

Other technical issues were found, unrelated to the JavaScript specification in use. For example, if a reaction is being registered through `Function.prototype.bind` [60], the Callback Graph will not assume this call as returning a bound function (e.g, `p.then( console.warn.bind( console, 'Warning:' ) );`), hurting the evaluation. For programs containing loops registering an already defined callback to a Promise chain, the Async-TAJS seems to remain pending indefinitely. This does not happen if the callback is registered dynamically (created inside the loop).

Callback Graph issues Aside from general limitation of Async-TAJS, already mentioned, and the Callback Graph limitations and implementation issues in detecting the Unexpected Execution Order concurrency bug mentioned in Section 5.3, the Callback Graph also exhibits incongruent behaviour.

Considering again the code from Fig. 4.1 and its generated graph presented in Fig. 5.5. If we replace each of the default reactions for explicit callbacks with the exact same behaviour, the analysis produces a different graph. These changes are shown in Fig. 7.1. Note that we did not change any of the Promise methods (i.e., the `catch` was not changed to `then` with two arguments).

The graph is slightly different than the original, even though the only differences it should present would be the labels of the nodes (to show now the new function names and the `Q` and `R` values). And if we introduce synchronous code, which will not change the way the asynchronous callbacks are executed, since there is no scheduling nor new contexts, the graph could be vastly different, depending on the changes. It could even go to the point of displaying the happens-before relations more correctly according to the program flow, regarding the Promises chain, as explained in section 5.3. Likewise, if we convert the `catch` to a `then` whose first argument is an explicit callback with the same behaviour as the default `onFulfill` callback, the graph generated will be yet a different graph. But all of these programs are equivalent in terms of happens-before relations (i.e., the code added will not change any

```
function Manager() {}

Manager.prototype.storeString = function(
  str, onCreate ) {
  return new Promise( resolve => {
    setTimeout( () => {
      resolve( this.str = str );
    }, Math.random() * 1000 );
  } ).then( onCreate || function A( v ) {
    return v;
  } );
};

Manager.prototype.validateString =
  function( str ) {
  return new Promise( ( resolve, reject )
    => {
    setTimeout( () => {
      if ( str === this.str ) {
        resolve( 'Success!' );
      } else {
        reject( 'Different!' );
      }
    }, Math.random() * 1000 );
  } ).then(
    function S( v ) {
      console.log( 'S', v );
    },
    function R( v ) {
      throw v;
    }
  ).catch( function E( v ) {
    console.warn( 'E', v );
  } );
};

var manager = new Manager();

var str = 'str';

manager.storeString( str );

manager.validateString( str );
```

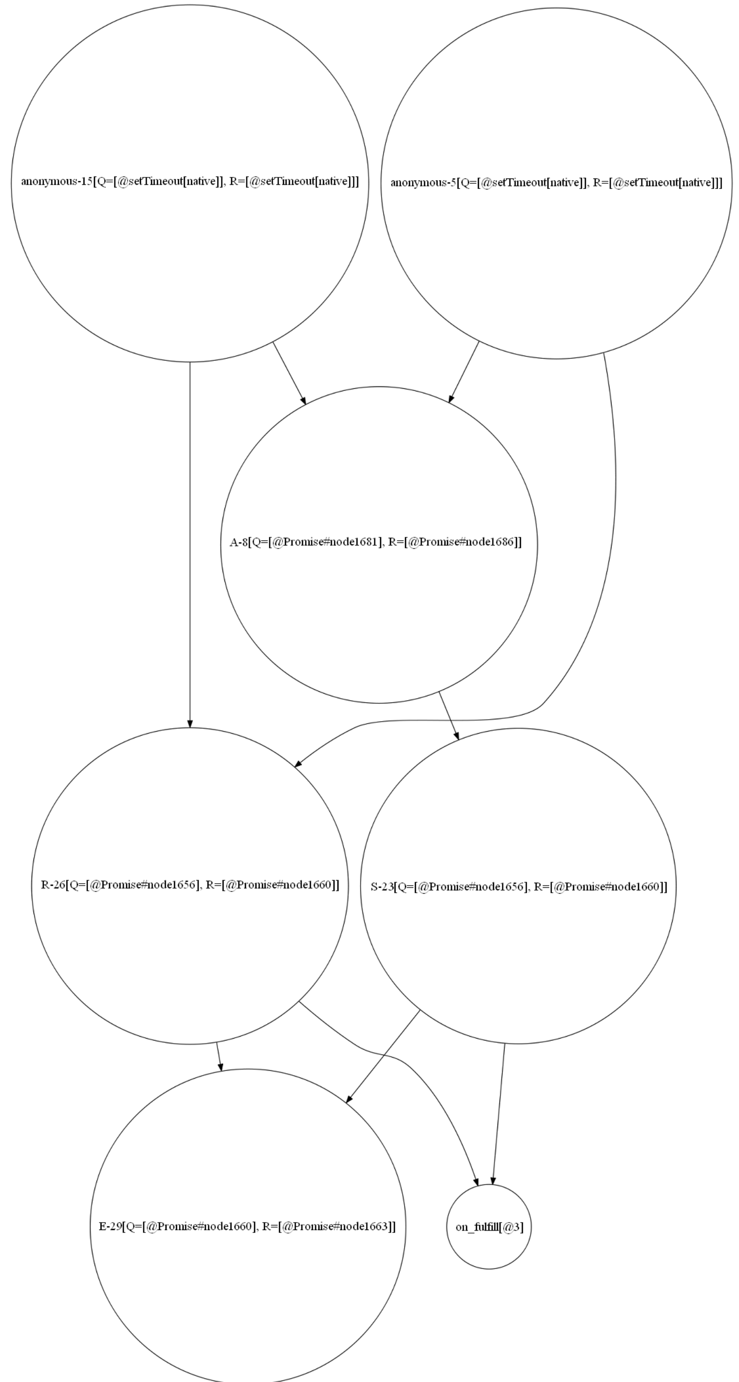


Figure 7.1: The same code from Fig. 4.1, but with explicit callbacks instead of default reactions, on the left, with the respective graph on the right.

of the scheduling). The `setTimeout` nodes are still shown in the same way, though, but that has to do solely with the way the Callback Graph models the timers. The part that was not correct in the graph for the original example (the part about the Promise chains) is the only part that is prone to changes when callbacks are introduced/removed and their code is changed (again, only considering non-asynchronous APIs inside the callbacks).

This volatile implementation of the Callback Graph could pose some difficulties, causing inaccuracy in the analysis.

7.2 Future Work

Besides some aspects of our work that can be improved, we also want to leave some ideas and suggestions to further improve our work.

Detection of Broken Promise Bugs The detection of Broken Promise bugs can potentially be improved. Our approach is tracking all forked Promises, because forked Promises can be either intentional or not and can be considered a bad practice. So we suggest two approaches to try to refine the accuracy of the results:

- Gather information about the context of execution. From it, somehow infer if the fork is intentional or not.
- Similarly to JavaDoc [61] and ESLint [62], use a special syntax on JavaScript comments, to indicate that the reaction being registered on the same root Promise, on the following line, is intentional. This would tell the static analyzer to not treat the tagged reaction registration as a Broken Promise occurrence.

The first approach could process the context and calculate a confidence factor. It is not a simple task to automate this, considering that the design choice comes from the intention of the programmer. For simple cases, it may detect erroneous occurrences, flagging it as a Broken Promise bug. But there are cases where the issues will be manifested possibly only on remote servers, for instance, whose requests came from several places in the JavaScript code, without apparent relation between them, complicating automation.

The second approach, illustrated in Fig. 7.2, may generate more accurate results, because the developer tags the forks along with the writing of the code. If a forked Promise is not tagged to be ignored, then it either is (a) an unintentional forked Promise and, therefore, a Broken Promise bug; or (b) the developer does want to fork the Promise, but simply forgot to tag it as such.

```javascript
let c = v => console.log( v );

const p = Promise.resolve( 1 );

p.then( v => {
  c( v );
} );

/***ignore: promise-fork */
p.then( v => {
  c( v + 1 );
} );
```

Figure 7.2: Example of manually tagging the reaction registration that forks the Promise to avoid being tagged as a Broken Promise bug.

In this last case, then the user shall receive a warning of a Broken Promise, as they usually would. Regardless, the user attention to that forked Promise is caught and the user can either correct it, by correcting the Broken Promise issue, or simply tag it to be ignored, because the fork was intentional.

Disadvantages of this approach is that it requires a manual work by the developer; it relies on the user. For a program that contains lots of forked Promises, this could be tiresome. It is extra work to the programmer.

Detection of Unexpected Order Bugs As it has been mentioned, the detection of Unexpected Execution Order issues has yet to be fully implemented. Also, this model, as we described it, gathers enough information to reason about other concurrency bugs, such as Dead Promise. It can be further explored.

Tool support The Async-TAJS itself, and the implementation of the Callback Graph, have some issues, as discussed in the previous Section. The TAJS itself has fixed some problems and has improved, but those changes are not present yet at Async-TAJS and are not easily portable.

Lastly, not all asynchronous APIs are modeled by the Callback Graph or supported by Async-TAJS (e.g., `Promise.allSettled`, `setImmediate`). JavaScript is a language that is in constant evolution, with new APIs being developed, like `Promise.any`, so both the model and the tool need to be updated as time goes by.

Bibliography

[1] Piotr Sroczkowski, "100 most popular languages on GitHub in 2019," https://brainhub.eu/blog/most-popular-languages-on-github/, 2019, [Online; accessed 6-November-2019].

[2] GitHub, "GitHub Octoverse — Top languages," https://octoverse.github.com/#top-languages, 2019, [Online; accessed 6-November-2019].

[3] T. Sotiropoulos and B. Livshits, "Static analysis for asynchronous javascript programs," *arXiv preprint arXiv:1901.03575*, 2019.

[4] theosotr, "Async TAJS," https://github.com/theosotr/async-tajs, 2019, [Online; accessed 20-September-2020].

[5] "GCC Docs — 3.10 Options That Control Optimization," https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html, [Online; accessed 21-November-2019].

[6] M. Madsen, O. Lhoták, and F. Tip, "A model for reasoning about javascript promises," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, p. 86, 2017.

[7] S. Alimadadi, D. Zhong, M. Madsen, and F. Tip, "Finding broken promises in asynchronous javascript programs," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, p. 162, 2018.

[8] H. Sun, D. Bonetta, F. Schiavio, and W. 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*. IEEE Press, 2019, pp. 61–72.

[9] "ECMAScript® 2015 Language Specification — Promise Objects," https://www.ecma-international.org/ecma-262/6.0/#sec-promise-objects, June 2015, [Online; accessed 7-November-2019].

[10] "ECMAScript® 2015 Language Specification," https://www.ecma-international.org/ecma-262/6.0/, June 2015, [Online; accessed 3-October-2020].

[11] Benjamin Diuguid, “Asynchronous Adventures in JavaScript: Promises,” https://medium.com/dailyjs/asynchronous-adventures-in-javascript-promises-1e0da27a3b4, February 2016, [Online; accessed 3-October-2020].

[12] “Callback hell,” http://callbackhell.com/, [Online; accessed 6-November-2019].

[13] “jQuery GitHub,” https://github.com/jquery/jquery, [Online; accessed 9-November-2019].

[14] “jQuery,” https://jquery.com/, [Online; accessed 3-October-2020].

[15] “jQuery API Documentation — Deferred Object,” https://api.jquery.com/category/deferred-object/, [Online; accessed 3-October-2020].

[16] “Bluebird GitHub,” https://github.com/petkaantonov/bluebird, [Online; accessed 9-November-2019].

[17] K. Gallaba, A. Mesbah, and I. Beschastnikh, “Don’t call us, we’ll call you: Characterizing callbacks in javascript,” in *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2015, pp. 1–10.

[18] “The JavaScript Event Loop,” https://flaviocopes.com/javascript-event-loop/, April 2018, [Online; accessed 13-November-2019].

[19] MDN contributors, “Concurrency model and the event loop,” https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop, September 2019, [Online; accessed 13-November-2019].

[20] “The Node.js Event Loop, Timers, and process.nextTick(),” https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/, [Online; accessed 13-November-2019].

[21] M. Madsen, F. Tip, and O. Lhoták, “Static analysis of event-driven node.js javascript applications,” *SIGPLAN Not.*, vol. 50, no. 10, p. 505–519, Oct. 2015. [Online]. Available: https://doi.org/10.1145/2858965.2814272

[22] H. Sun, D. Bonetta, C. Humer, and W. Binder, “Efficient dynamic analysis for node. js,” in *Proceedings of the 27th International Conference on Compiler Construction*. ACM, 2018, pp. 196–206.

[23] Northeastern University Programming Research Lab, “PromiseKeeper,” https://github.com/nuprl/PromiseKeeper, 2018, [Online; accessed 28-November-2019].

[24] Haiyang-Sun, “AsyncG,” https://github.com/Haiyang-Sun/AsyncG, 2018, [Online; accessed 5-December-2019].

[25] G. Antal, P. Hegedus, Z. Tóth, R. Ferenc, and T. Gyimóthy, “Static javascript call graphs: A comparative study,” in *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2018, pp. 177–186.

[26] M. C. Loring, M. Marron, and D. Leijen, “Semantics of asynchronous javascript,” in *ACM SIGPLAN Notices*, vol. 52, no. 11. ACM, 2017, pp. 51–62.

[27] A. Guha, C. Saftoiu, and S. Krishnamurthi, “The essence of javascript,” in *European conference on Object-oriented programming*. Springer, 2010, pp. 126–150.

[28] Aarhus University, “TAJS: Type Analyzer for JavaScript,” https://www.brics.dk/TAJS/, [Online; accessed 23-October-2020].

[29] S. H. Jensen, A. Møller, and P. Thiemann, “Type analysis for javascript,” in *International Static Analysis Symposium*. Springer, 2009, pp. 238–255.

[30] ——, “Interprocedural analysis with lazy propagation,” in *International Static Analysis Symposium*. Springer, 2010, pp. 320–339.

[31] S. H. Jensen, M. Madsen, and A. Møller, “Modeling the html dom and browser api in static analysis of javascript web applications,” in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 59–69.

[32] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf, “Jsai: a static analysis platform for javascript,” in *Proceedings of the 22nd ACM SIGSOFT international symposium on Foundations of Software Engineering*. ACM, 2014, pp. 121–132.

[33] “Events — Node.js Documentation,” https://nodejs.org/api/events.html, [Online; accessed 4-December-2019].

[34] “The DOT Language,” https://www.graphviz.org/doc/info/lang.html, [Online; accessed 5-December-2019].

[35] “D3.js, Data-Driven Documents,” https://d3js.org/, [Online; accessed 5-December-2019].

[36] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, “Efficient javascript mutation testing,” in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 2013, pp. 74–83.

[37] A. Feldthaus, T. Millstein, A. Møller, M. Schäfer, and F. Tip, “Tool-supported refactoring for javascript,” in *ACM SIGPLAN Notices*, vol. 46, no. 10. ACM, 2011, pp. 119–138.

[38] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos, “Graph-based analysis and prediction for software evolution,” in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 419–429.

[39] "ECMAScript® 2015 Language Specification — The Reflect Object," https://www.ecma-international.org/ecma-262/6.0/#sec-reflect-object, June 2015, [Online; accessed 8-December-2019].

[40] S. Fink and J. Dolby, "Wala–the tj watson libraries for analysis," 2012.

[41] M. Bolin, *Closure: The Definitive Guide: Google Tools to Add Power to Your JavaScript.* " O'Reilly Media, Inc.", 2010.

[42] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, "Efficient construction of approximate call graphs for javascript ide services," in *2013 35th International Conference on Software Engineering (ICSE).* IEEE, 2013, pp. 752–761.

[43] Gunar C. Gessner, "npm callgraph," https://www.npmjs.com/package/callgraph, 2018, [Online; accessed 13-November-2019].

[44] "SunSpider 1.0.2 JavaScript Benchmark," https://webkit.org/perf/sunspider/sunspider.htm, [Online; accessed 10-December-2019].

[45] O. Lhoták *et al.*, "Comparing call graphs," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering.* ACM, 2007, pp. 37–42.

[46] "Promises chaining," https://javascript.info/promise-chaining, May 2020, [Online; accessed 6-October-2020].

[47] Bobby Brennan, "ES6 Promises: Patterns and Anti-Patterns — Calling .then() multiple times," https://medium.com/datafire-io/es6-promises-patterns-and-anti-patterns-bbb21a5d0918#1de1, September 2017, [Online; accessed 6-October-2020].

[48] Wikipedia contributors, "Stack (abstract data type) — Wikipedia, the free encyclopedia," https://en.wikipedia.org/wiki/Stack_(abstract_data_type), 2020, [Online; accessed 30-September-2020].

[49] ——, "Fifo (computing and electronics) — Wikipedia, the free encyclopedia," https://en.wikipedia.org/wiki/FIFO_(computing_and_electronics), 2020, [Online; accessed 30-September-2020].

[50] Aarhus University, "TAJS," https://github.com/cs-au-dk/TAJS, 2020, [Online; accessed 23-October-2020].

[51] Vitaliy Potapov, "controlled-promise," https://github.com/vitalets/controlled-promise, July 2018, [Online; accessed 26-October-2020].

[52] GitHub, "window.fetch polyfill," https://github.com/github/fetch, October 2020, [Online; accessed 26-October-2020].

[53] kokororin, “honoka,” https://github.com/kokororin/honoka, April 2020, [Online; accessed 26-October-2020].

[54] axios, “axios,” https://github.com/axios/axios, August 2020, [Online; accessed 26-October-2020].

[55] alphasp, “Pixiv API Client,” https://github.com/alphasp/pixiv-api-client, October 2019, [Online; accessed 26-October-2020].

[56] isaacs, “Glob,” https://github.com/isaacs/node-glob, November 2019, [Online; accessed 26-October-2020].

[57] “ECMAScript® Language Specification,” https://www.ecma-international.org/ecma-262/5.1/, June 2011, [Online; accessed 17-October-2020].

[58] “Webpack,” https://webpack.js.org/, [Online; accessed 17-October-2020].

[59] “babel,” https://babeljs.io/, [Online; accessed 17-October-2020].

[60] MDN contributors, “Function.prototype.bind(),” https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/bind, August 2020, [Online; accessed 18-October-2020].

[61] “JavaDoc,” https://www.oracle.com/java/technologies/javase/javadoc-tool.html, [Online; accessed 21-September-2020].

[62] “ESLint Configuration,” https://eslint.org/docs/user-guide/configuring, [Online; accessed 21-September-2020].