Practical Detection of JavaScript Concurrency Bugs using Callback Graphs

BERNARDO FURET, Instituto Superior Técnico, Universidade de Lisboa

JavaScript is becoming an increasingly popular programming language [12, 21] 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 [22] 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 [24], 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.

CCS Concepts: • **Software and its engineering** → *Software maintenance tools.*

Additional Key Words and Phrases: JavaScript, Asynchronous, Data race, Static analysis, Concurrency

ACM Reference Format:
Bernardo Furet. 2020. Practical Detection of JavaScript Concurrency Bugs using Callback Graphs. 1, 1 (November 2020), 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

JavaScript is currently one of the most popular programming languages [12, 21], 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. While this prevents the application from stopping while waiting for external responses, it also contributes to certain elements to be deferred and not being executed right away, possibly introducing race

Author's address: Bernardo Furet, bernardo.furet@tecnico.ulisboa.pt, Instituto Superior Técnico, Universidade de Lisboa, Lisboa.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2020 Association for Computing Machinery.
XXXX-XXXX/2020/11-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
client.connect( PORT );
client.send( message );
```

Fig. 1. Example of a subtle data race issue.

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* [5] was introduced, with the ECMAScript 2015 [4, 9]. 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. Today, about 75% of JavaScript frameworks use Promises [11].

However asynchronous programming still poses some challenges. For example, as pointed out by Sotiropoulos and Livshits [22], recent studies showed that concurrency bugs found in JavaScript programs are sometimes caused by asynchrony. The code in Fig. 1 shows a common example 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, responsible for the reception and handling of events and the scheduling and execution of asynchronous operations (Promises, timers, etc.): the *event loop* [2, 6, 19]. 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.

The goal of this work is to explore the recently introduced Callback Graph model [22] to automatically detect concurrency bugs in JavaScript, through static analysis using Async-TAJS, a JavaScript static analysis tool. We focus on two specific cases of concurrency issues, the cases of *Broken Promise* and *Unexpected Execution Order* bugs. The contributions of this work are: (a) methods based on the Callback Graph to automatically detect Broken Promise issues, (b) new model based on the Callback Graph to automatically detect Unexpected Execution Order issues, (c) extension to the Async-TAJS tool that reports Broken Promise issues to the user and (d) three datasets to exercise asynchronous JavaScript behaviour.

2 MOTIVATING EXAMPLES

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.

Broken Promise bugs. Also known as *Broken Promise Chain* bugs, occur when 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

```
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 );
```

```
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 );
```

Fig. 2. The example on the left shows a data race issue: both reactions are registered on the original promise, forking it. On the right, the example is corrected: the reactions are registered in order on the resulting Promise, keeping the order of execution: first the increment, then the check.

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.

An example of this bug is illustrated in Fig. 2. On the left, it shows a data race bug. The Promise chain is being accidentally forked. Two chains are being created on the root Promise. This will produce two new Promises, each with 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. One of the new Promises will encapsulate the value 2 (`value + 1`). 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 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`).

Unexpected Execution Order bugs. This type of bugs happens 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.

The example presented in Fig. 3 aims to capture a realistic pattern. 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 website, 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 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 `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

```
function doThrow( msg ) { throw new Error( msg ); }
function Manager() {}
Manager.prototype.storeString = function( str, onCreate ) {
  return Promise.resolve()
    .then( () => ( this.str = str ) )
    .then( onCreate )
  ;
};
Manager.prototype.validateString = function( str ) {
  return str === this.str
    ? console.log( 'Success!' )
    : doThrow( 'Different!' )
  ;
};
var manager = new Manager();
var str = 'str';
manager.storeString( str );
manager.validateString( str );
```

Fig. 3. Example of the Unexpected Execution Order bug. `validateString` will be called after `storeString` without ensuring the latter has completed its task.

a callback on the returned Promise. By applying any of these solutions, it is ensured `validateString` will only be called after `storeString` completes.

3 BACKGROUND & RELATED WORK

There are several models and tools that perform JavaScript code analysis in an attempt to aid the developers writing better JavaScript code. We use the Callback Graph model, but we also describe the Promise Graph model [18], used by PromiseKeeper [20], and the Async Graph [23] model, implemented through AsyncG [14]. Finally, we present a comparative study between different analyzers for JavaScript code [8].

3.1 Callback Graph

Introduced by Sotiropoulos and Livshits [22], it is implemented through Async-TAJS [24], a JavaScript static analysis tool. Our study focuses 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.

The Callback Graph represents the asynchronous callbacks as nodes, where each node has associated a context. A path from one node n_1 to another node n_2 means that n_2 will be called after n_1 is called. This property is transitive. Unconnected nodes mean the execution order of the callbacks they represent is non-deterministic or unspecified (e.g., `setTimeout`).

Fig. 4 shows a simple example of the Callback Graph model. With this model, the authors introduce *QR-sensitivity*. This concept helps distinguishing callback calls by dividing them by their call context and by the

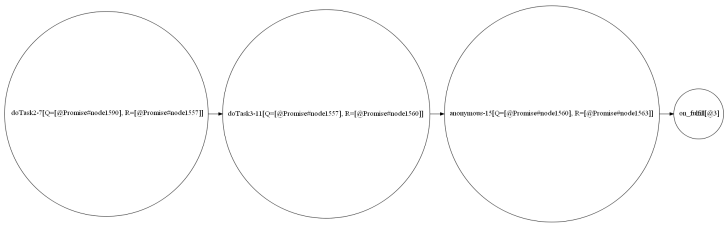

Fig. 4. Callback Graph model example.

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).

3.2 Promise Graph & PromiseKeeper

Promise graphs were introduced by Madsen et al. [18]. The object of study is the standardized concept of Promise. The goal is to analyze code containing Promises and find bugs related to them. The authors defined a λ_p calculus, by extending the λ_{JS} calculus proposed by Guha et al. [13], to formally explain the semantics of Promises.

The authors also present the PromiseKeeper [20], a tool that performs dynamic analysis to construct an enhanced Promise Graph. The resulting Promise Graph 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 Async Graph & AsyncG

Another graph-based model is the Async Graph [23]. 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 [14]: 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.

3.4 Call graphs

Since the work presented in this thesis makes use of and analyzes code using a particular case of a call graph, in this subsection we present several static code analyzers that produce a call graph, based on a study made by [8]. Each of these tools has their own unique features that may be useful to explore. Included in this study is the TAJS [15–17]; the tool serving as the base for Async-TAJS. This study compares the advantages and disadvantages of each model, as well as of static and dynamic analysis.

4 APPROACH

Each concurrency bug is approached in a different manner.

4.1 Broken Promise

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`) [5]. 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.
(3) Store all Promises in the form of `[R,Q]`. These two values are enough too 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, 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.

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. Forking a Promise not always becomes an issue. So it 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. 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 [7, 10]. 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 (nonetheless, in some edge case scenarios, perhaps it may be useful to fork a Promise). So we decided to consider all forked Promises to be reported as possible Broken Promises.

4.2 Unexpected Execution Order

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, at runtime. For instance, going back to the example shown in Fig. 3, we are interested in knowing when `storeString` and `validateString` will execute, because they write and read `manager.str`, respectively, at different execution timings. The call to `storeString` asynchronously writes `manager.str`. The call to `validateString` 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 this case there is a deterministic result. It will always throw an exception. But if the validation itself was asynchronous (e.g., validating through a remote server), it could lead to non-deterministic results.

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) Split all the different execution timings into nodes. Each node 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.
(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, 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 create all the possible paths, generating two paths: one where one `setTimeout` happens before the other and vice-versa, but always after the Promise reaction.

To extract results from this approach and detect Unexpected Execution Order bugs, 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. This approach separates all the execution timings and allows for comparison on how the data is accessed and manipulated between them.

5 IMPLEMENTATION

As mentioned before, we use the Async-TAJS [24] tool to perform static analysis in the JavaScript code. This tool extends TAJS [15–17], by implementing the Callback Graph model [22].

5.1 TAJS – Type Analyzer for JavaScript

Performs a flow-sensitive and context-sensitive static analysis. It is capable of detecting some type-related errors, such as generated implicit type-conversion and call of non-callable variables.

Our intention is to extend this tool even further, making it possible for the tool to detect the concurrency issues presented in the Section 2 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.

For our analysis, we performed QR-sensitive 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.

5.2 Broken Promise

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

Consider the code snippets presented in Fig. 2. The Callback Graph model collects the asynchronous callbacks registered in the Promises and generates nodes from these methods, with associated `Q` and `R` values. Fig. 5 displays the graphs generated by running Async-TAJS on the example snippets.

The graphs present, inside the nodes, information about the `Q` and the `R` for each node. The second node of the graph on the left, corresponding to the snippet with the Broken Promise bug, has a value for `Q` different than the value for `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

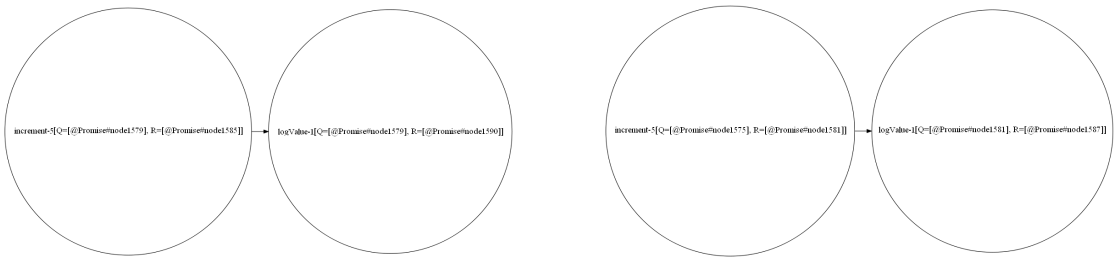

Fig. 5. Callback Graph model applied to the examples provided on Fig. 2, respectively.

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, and different `R`, meaning they generate different Promises (i.e., they are not different callbacks to handle the same Promise, which happens when `then` receives two arguments).

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. With this information, the developer has all the data they need to check what is happening at those locations and fix the issue.

5.3 Unexpected Execution Order

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.

For execution timings whose specific timings cannot be known through a static analysis, assume they can happen in all possible orders. 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. 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.

Extending the Callback Graph. We propose an enhanced version of the Callback Graph, that captures each different context. 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. Besides, each node will store the variables 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.

On Fig. 6 we present a prototype of what should be the result of the analysis for the code presented in Fig. 3. As it can be seen, this model has a node for each distinct execution timing context. Connected from each node there is a board indicating the variables accessed during that execution context, tagged with a letter: `R` for read; `W` for write and `C` for call (this is also a read).

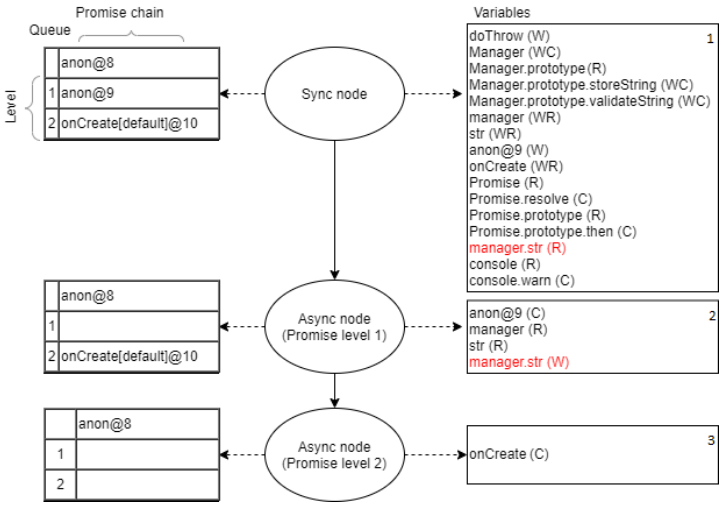

Fig. 6. Enhanced model to reason about Unexpected Execution Order issues for the code snippet displayed in Fig. 3.

Analyzing this model, we can see that `manager.str` is read during an execution timing before it is written (emphasized in red). After being written, it is never read again. There are other variables that were read before being written, like `Manager.prototype`, but these are values created (written) natively by the JavaScript engine.

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.

6 EVALUATION

6.1 Datasets of JavaScript Asynchronous Bugs

We assembled a set of 74 small programs composed of almost all asynchronous API supported by the language to date, considering our work is oriented more for the Node.js environment. Furthermore, we assembled two other smaller datasets: the first with 13 files to test our implementation that detects Broken Promise issues; the second with a total of 14 programs to test our implementation that detects Unexpected Execution Order bugs. In addition, we also used the benchamrks designed by the authors of Async-TAJS and the 6 real world modules the same authors used to evaluate our solutions.

6.2 Results

We first ran Async-TAJS over our set of general benchamrks and detected several limitations of the tool. Some asynchronous APIs were not recognized or were modeled incorrectly by the tool. TAJS itself also has issues that prevented a proper evaluation for some programs.

Broken Promise bugs. Considering the dataset designed to exercise the detection of this type of bugs, the results were 2 false positives and 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. 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.

Unexpected Execution Order bugs. Due to time constraints, this solution is not fully implemented yet. However, as explained in Section 5.3, the model gathers all the info necessary to reason about execution timings, data and how they relate, in order to detect Unexpected Execution Order bugs.

7 CONCLUSION

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 [22], implemented in Async-TAJS [24], 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. The former are now detected correctly. As for the latter, 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

Async-TAJS limitations. The tool has issues requiring other JavaScript files needed, which posed some difficulties on the evaluation of larger, real-world programs containing dependencies from global packages/modules. It also does not support JavaScript syntax above the ES5 [3] specification. More so, it is unable to process code containing **`switch`** statements with **`default`**-case not being at the last position.

Callback Graph issues. If Promise default reactions are replaced with callbacks that would result in the exact same program flow, the Callback Graph may generate different graphs, even though the happens-before relations and the flow is the same, causing inaccuracy in the analysis.

7.2 Future Work

Detection of Broken Promise Bugs. Our approach tracks all forked Promises. Considering some forked Promises might be intentional, we leave a suggestion of a solution that allows the programmer to tag the Promises that are forked as an intentional fork (for example, in the same way ESLint [1] rules can be disabled), to reduce false positives.

Tool support. Not all asynchronous APIs are modeled by the Callback Graph or supported by Async-TAJS (e.g., `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 can be updated as time goes by.

REFERENCES

[1] [n.d.]. ESLint Configuration. https://eslint.org/docs/user-guide/configuring. [Online; accessed 21-September-2020].

[2] [n.d.]. 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].

[3] 2011. ECMAScript® Language Specification. https://www.ecma-international.org/ecma-262/5.1/. [Online; accessed 17-October-2020].

[4] 2015. ECMAScript® 2015 Language Specification. https://www.ecma-international.org/ecma-262/6.0/. [Online; accessed 3-October-2020].

[5] 2015. ECMAScript® 2015 Language Specification | Promise Objects. https://www.ecma-international.org/ecma-262/6.0/#sec-promise-objects. [Online; accessed 7-November-2019].

[6] 2018. The JavaScript Event Loop. https://flaviocopes.com/javascript-event-loop/. [Online; accessed 13-November-2019].

[7] 2020. Promises chaining. https://javascript.info/promise-chaining. [Online; accessed 6-October-2020].

[8] Gábor Antal, Péter Hegedus, Zoltán Tóth, Rudolf Ferenc, and Tibor Gyimóthy. 2018. Static JavaScript Call Graphs: A Comparative Study. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 177–186.

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

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

[11] Keheliya Gallaba, Ali Mesbah, and Ivan Beschastnikh. 2015. Don't call us, we'll call you: Characterizing callbacks in JavaScript. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 1–10.

[12] GitHub. 2019. GitHub Octoverse | Top languages. https://octoverse.github.com/#top-languages. [Online; accessed 6-November-2019].

[13] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2010. The essence of JavaScript. In *European conference on Object-oriented programming*. Springer, 126–150.

[14] Haiyang-Sun. 2018. AsyncG. https://github.com/Haiyang-Sun/AsyncG. [Online; accessed 5-December-2019].

[15] Simon Holm Jensen, Magnus Madsen, and Anders Møller. 2011. 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, 59–69.

[16] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type analysis for JavaScript. In *International Static Analysis Symposium*. Springer, 238–255.

[17] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2010. Interprocedural analysis with lazy propagation. In *International Static Analysis Symposium*. Springer, 320–339.

[18] Magnus Madsen, Ondřej Lhoták, and Frank Tip. 2017. A model for reasoning about JavaScript promises. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 86.

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

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

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

[22] Thodoris Sotiropoulos and Benjamin Livshits. 2019. Static Analysis for Asynchronous JavaScript Programs. *arXiv preprint arXiv:1901.03575* (2019).

[23] Haiyang Sun, Daniele Bonetta, Filippo Schiavio, and Walter Binder. 2019. Reasoning about the Node.js event loop using async graphs. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Press, 61–72.

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