An Asynchronous Call Graph for JavaScript

Dominik Seifert
National Taiwan University
Taiwan, Taipei
d01922031@ntu.edu.tw

Michael Wan
National Taiwan University
Taiwan, Taipei
b07201003@ntu.edu.tw

Jane Hsu
National Taiwan University
Taiwan, Taipei
yjhsu@csie.ntu.edu.tw

Benson Yeh
National Taiwan University
Taiwan, Taipei
pcyeh@ntu.edu.tw

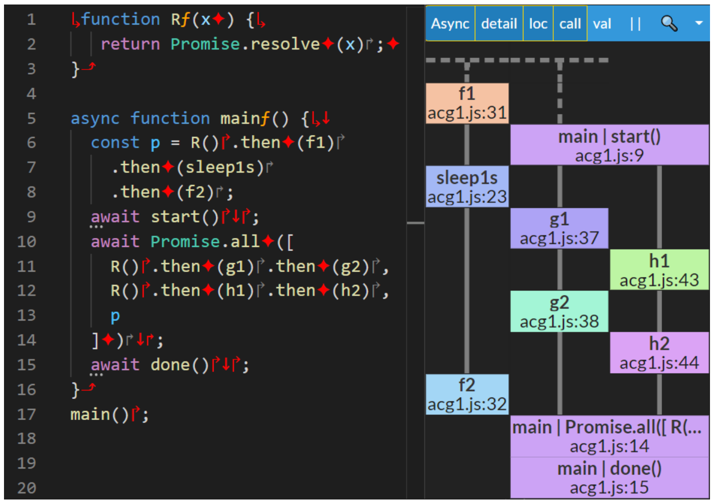
ABSTRACT

Asynchronous JavaScript has become omnipresent, yet is inherently difficult to reason about. While many recent debugging tools are trying to address this issue with (semi-)automatic methods, interactive analysis tools are few and far between. To this date, developers are required to build mental models of complex concurrent control flows with little to no tool support. Thus, asynchrony is making life hard for novices and catches even seasoned developers off-guard, especially when dealing with unfamiliar code. That is why we propose the Asynchronous Call Graph. It is the first approach to capture and visualize concurrent control flow between call graph roots. It is also the first concurrency analysis tool for JavaScript that is fully interactive and integrated with an omniscient debugger in a popular IDE. First tests show that the ACG works successfully on real-world codebases. This approach has the potential to set a new standard for how developers can analyze asynchrony.

CCS CONCEPTS

• **Software and its engineering → Concurrent programming structures**.

ACM Reference Format:
Dominik Seifert, Michael Wan, Jane Hsu, and Benson Yeh. 2022. An Asynchronous Call Graph for JavaScript. In *44nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3510457.3513059



```
1   function R(x) {
2     return Promise.resolve(x);
3   }
4
5   async function main() {
6     const p = R().then(f1)
7       .then(sleep1s)
8       .then(f2);
9     await start();
10    await Promise.all([
11      R().then(g1).then(g2),
12      R().then(h1).then(h2),
13      p
14    ]);
15    await done();
16  }
17  main();
18
19
20
```

Figure 1: Screenshot of a sample application in the Dbux VSCode Extension with the ACG on the right.

1 INTRODUCTION AND BACKGROUND

While a **dynamic call graph** establishes the caller-callee relationship between executed files and functions[3], the novel Asynchronous Call Graph (ACG) captures control flow where there is no direct caller: at the roots. Unlike **asynchronous event graphs**[4][2][1], the ACG maps out complete asynchronous control flow. To that end, it uncovers CHAIN and FORK relationships between all Call Graph Roots (CGRs). The ACG is also the first unified approach to deal with all three types of JS asynchronous events.

Background. Given a function, file or script tag `f`, we denote any of its executions as **context** f. For any interrupted function's asynchronous continuation[1] a (virtual) **context** is inserted. Each context starts when it is pushed onto and ends when popped from the stack. JS's non-preemptive property ensures that each context runs to completion without interruption, unless it encounters `await` or `yield` expressions. Any context directly invoked by the JS engine's event loop[1] is a **Call Graph Root (CGR)**.

We denote the entry point CGR as R_0 and the CGR of any event or context `x` as `cgr(x)`. If `p` is a promise, `cgr(p)` denotes its first CGR[1]. CGRs are scheduled by **Asynchronous Events (AEs)**. Each AE `e` has at least two properties: `sched(e)`, the event that scheduled it, and `to(e)`, the CGR that executes upon its activation. `e` and `to(e)` have a 1:1 relationship. `to(e)` is also called `e`'s CGR and `e` is `to(e)`'s AE. There are three types of AEs: (1) **AWAIT** is scheduled by an `await` expression inside an async function's context f. `to(e)` is the virtual context representing f's asynchronous continuation, starting right after the `await` expression. (2) **THEN** is scheduled by `q = p.THEN(f[, g])` for some promise `p`. We use `THEN` to represent `then`, `catch` or `finally`. We refer to the context of its handler function f or g as *thenCb*. In this case, `to(e)` is *thenCb*. (3) Asynchronous callbacks **CB** are scheduled by a call to an uninstrumented function `f` which takes at least one argument `cb` of type `function`, e.g. `f(cb)`. In this final case, `to(e)` is the executed callback context *cb*.

2 ASYNCHRONOUS CALL GRAPH

```
async function f() {
  FA
  await 0;
  FB
}
```

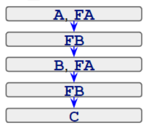
(a) f

```
A
await f();
B
await f();
C
```

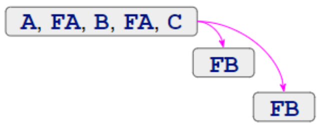
(b) Ex. 2: serial

```
A
f();
B
f();
C
```

(c) Ex. 3: concurrent



(d) ACG for Ex. 2 has CHAINs



(e) ACG for Ex. 3 has FORKs

Figure 2: Ex. 2 and 3 illustrate how ACGs use CHAINs and FORKs to make serial and concurrent control flows explicit. CGRs are labeled with lists of all their executed events.

The ACG is a DAG. Its nodes are CGRs. At the core of ACG construction is the `ADD_EDGE` algorithm: given a CGR `to(e)` of some AE `e`, add all edges ϵ_i := `from(e)`^{i} ⟶ `to(e)`. The difficulty lies in finding the CGR(s) `from(e)` that logically "executed before" `to(e)`, since such relationship has not been previously defined.

CHAIN vs. FORK. In order to better capture concurrency, we categorize all edges into CHAINs or FORKs. CHAINs are to convey a clear serial flow of events, while FORKs imply a lack thereof. We, respectively, define **virtual thread** (short: thread) and **chain-subgraph** as maximal path and maximal subgraph consisting only of CHAINs. Edge construction first tries to find and insert CHAINs. If not found, a FORK from `cgr(sched(e))` is inserted instead. To that end, we partition all AEs into two groups with different rulesets:

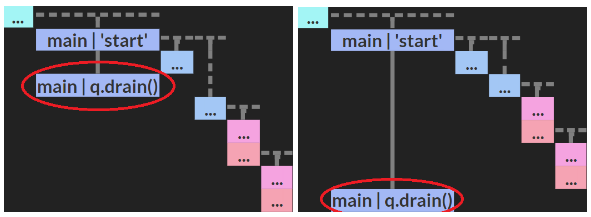
(i) The first group consists of all AWAIT- and THEN-type of AEs, as well as promisified CBs[1]. These are CHAINed according to "promise and async function chain and nesting semantics"[1]. This type of CHAIN propagates errors, meaning that a single error handler can be placed to catch all errors of the same thread. These AEs can induce FORKs if `to(e)` is the first CGR of a promise chain[1] and all its nesting[1] promise chains.

(ii) All remaining AEs are non-promisified CBs. They are CHAINed iff any of the following two heuristics apply: **(h1)** CHAIN all CGRs of the same event handler and **(h2)** CHAIN recursive callbacks. This type of CHAIN does not propagate errors. In absence of heuristics, language semantics provide insufficient information to determine control flow reliably. One remaining challenge is to find a general solution to categorizing edges between non-recursive nested callbacks into CHAIN or FORK.

Ex. 1: `q = p.then(f).then(g)`. Let `G(x)` be a chain-subgraph starting at `cgr(x)`, and `G(cgr(p))` not empty. Promise chain semantics imply: $\{R_0 \rightarrow$ `G(cgr(p))` $\rightarrow$ `G(`f`)` $\rightarrow$ `G(`g`)`$\}$. Furthermore, the last two edges are ensured to be CHAINs. However, the designation of the first edge depends on whether `q` is chained-to[1] its root R_0. Fig. 2 illustrates the distinction with two more examples.

[1]This term is further explained in our Technical Report[6], §1

3 RESULTS



(a) async-js(1) – with bug. (b) async-js(2) – without bug.

Figure 3: async-js code, before and after fixing the bug. The circle indicates the `drain` event CGR.

We implemented the ACG as part of Dbux. It is open source and documentation is available online[5]. In order to explore the ACG's real-world utility, we used it on eleven popular Node.js and frontend projects, incl. webpack, sequelize and Editor.md.

In the async library, we investigated an open order-violation bug[2]. The bug's ACG is shown in Fig. 3a. Its `main` function finishes almost immediately, because the non-empty task queue's `drain` event activates too early. We could find the culprit within minutes, after following the obvious out-of-order `drain` event and its scheduler. When fixed (Fig. 3b), the `drain` event is triggered after all tasks completed. We shared the solution with the authors[2].

Lastly, we found a new bug in sequelize when investigating how it deals with two concurrent `findOrCreate` queries. The bug is visible as a console error message: "cannot start a transaction within a transaction". Using the ACG, we were able to achieve the following: (i) Verify our assumption that the error is caused by the second `findOrCreate` query, indicating that concurrent transactions are not supported. (ii) Quickly identify and track down the cause of an atomicity violation bug. (iii) Compare control flow patterns between two variations of the test case. Our accepted bug report[3] summarizes the bugs. More details can be found in our TR[6], §2.

ACKNOWLEDGMENTS

This work has received some financial support from Life is Learning Co., Ltd. (Taipei, Taiwan). Lastly, we thank Kuan-Yu Chen and Jyun-Wei Chen for their contributions.

REFERENCES

[1] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. 2016. Understanding asynchronous interactions in full-stack JavaScript. In *Proceedings of the 38th International Conference on Software Engineering*. ACM. https://doi.org/10.1145/2884781.2884864
[2] Saba Alimadadi, Di Zhong, Magnus Madsen, and Frank Tip. 2018. Finding broken promises in asynchronous JavaScript programs. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (Oct 2018), 1–26. https://doi.org/10.1145/3276532
[3] Susan L Graham, Peter B Kessler, and Marshall K McKusick. 1982. Gprof: A call graph execution profiler. *ACM Sigplan Notices* 17, 6 (1982), 120–126.
[4] Magnus Madsen, Ondřej Lhoták, and Frank Tip. 2017. A model for reasoning about JavaScript promises. , 24 pages. https://doi.org/10.1145/3133910
[5] Dominik Seifert and Michael Wan. 2021. *Dbux Documentation: Asynchronous Call Graph*. Retrieved 2/2022 from https://domiii.github.io/dbux/acg
[6] Dominik Seifert, Michael Wan, Jane Hsu, and Benson Yeh. 2022. *Technical Report: An Asynchronous Call Graph for JavaScript*. Retrieved 2/2022 from https://github.com/Domiii/dbux/blob/master/pub/icse-seip-2022/tr.pdf

[2]https://github.com/caolan/async/issues/1729
[3]https://github.com/sequelize/sequelize/issues/13554