# Technical Report: An Asynchronous Call Graph for JavaScript

## Dominik Seifert

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

## Jane Hsu

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

## ABSTRACT

This Technical Report serves as a supplementary document to the "An Asynchronous Call Graph for JavaScript" article. It provides extra background information and showcases results.

Important note to the reviewer: we discovered two more serious faux pas that snuck into the final submission draft of the paper. We are in the process of writing a correction.

One error (at L78) pertains to the amount of participants of our workshop (9 out of 10; not: 10 out of 11).

Another error pertains to Table 1: the "total threads" are not defined unambiguously, and do not include most orphaned forks (forks without a parent) that arise from (i) files being loaded and (ii) edges missed by the dynamic callback patcher. We want to re-design the table to unambiguously address that issue.

#### **CCS CONCEPTS**

 $\bullet$  Software and its engineering  $\to$  Concurrent programming structures.

#### **ACM Reference Format:**

## Michael Wan

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

#### Benson Yeh

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

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.

© 2021 Association for Computing Machinery. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...\$15.00 https://doi.org/10.1145/nnnnnnnnnnnnnnnn

Seifert, et al.

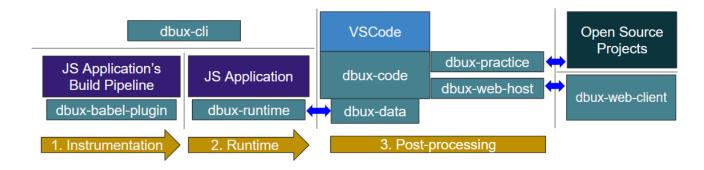


Figure 1: Dbux Architecture.

#### 1 A BRIEF INTRODUCTION TO DBUX

#### 1.1 Architecture

Dbux [3] has four applications and several supplementary modules, as depicted in Fig. 1. Several shared modules, such as the dbux-com mon modules, are not shown.

Dbux's three stages *instrument*, *runtime* and *post-processing* are implemented in four collaborating applications:

- dbux-babel-plugin instruments the target application and injects the dbux-runtime. It requires to be run with Babel [1].
- dbux-runtime records the target application's execution trace and streams it to a server in real time.
- dbux-code is a one-click-installable extension to VSCode, available on the VSCode Marketplace, complete with extensive documentation. It also has prepared several real-world projects, bugs and experiments to try it out on. Upon activation, it starts a server to wait for the execution data produced by dbux-runtime. When received, it post-processes it with the help of the dbux-data module before presenting it to the user. Data is processed and presented as soon as it is received, meaning that applications can be debugged while they are still running.
- dbux-cli is to Dbux, what "nyc" is to the coverage reporter Istanbul <sup>1</sup>, that is: a convenient command line tool that makes it easier for developers to execute a JS application with Dbux enabled, without having to prepare a build pipeline. Instead, it uses a modified version of @babel/register <sup>2</sup> to inject dbux-babel-pluqin on the fly.

#### 1.2 Call Graph Assembly

NOTE: In the following, we refer to the "dynamic call graph" just as "call graph". Dbux does not have a static call graph.

We model the call graph as follows:

(1) We refer to a call graph node, that is the recorded execution of a file or function, as an "executionContext", or **context** for short. Given a function f, we denote the context that represents the i'th execution of some function f as f<sub>i</sub>.

- (3) Any function execution  $f_i$  is considered a Call Graph Root (CGR), if it has no parent caller. This implies that the function was either directly invoked by the JavaScript engine's event queue, or the first invoked function was not recorded.
- (4) Traditionally the above rule set was sufficient to build a JavaScript dynamic call graph. However, ES2017 [2] introduced async functions, which need special attention due to their property of interruptibility: we refer to the  $i^{th}$  execution of some async function h as  $h_i$ . The context  $h_i$  is considered a real context. When executed, a virtual con**text**  $h_i^1$  is added as a child to  $h_i$ . Furthermore, any await p; expression tells the scheduler to **interrupt** the current control flow for one tick of the asynchronous queue, or, if p is a promise, until that promise has been settled. Once await has concluded, that is, after the promise has settled, execution of  $h_i$  continues. At this point, our instrumentation adds a new virtual context  $h_i^k$ . That context represents the asynchronous continuation of the interrupted real context  $h_i$  at a later point in time. We assume that each virtual context  $h_i^k$ , i > 1 is also a CGR.

<sup>(2)</sup> Edges represent the caller - callee relationship. For uninterruptible functions: if during it's i'th execution, f calls some function g, then, for some j,  $g_j$  is a child -or **callee**- of  $f_i$ , and  $f_i$  is a parent -or **caller**- of  $g_j$ .

In general, all events, contexts and roots are ordered by time of occurrence. Dbux's synchronous call graph implementation renders all CGRs linearly in that order.

<sup>1</sup>https://istanbul.js.org/

<sup>&</sup>lt;sup>2</sup>https://babeljs.io/docs/en/babel-register

## 1.3 Developer Survey

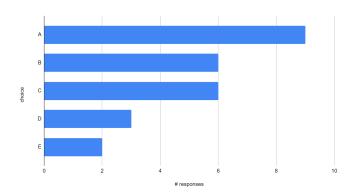


Figure 2: Survey Results: What type of programming problems are the most difficult to deal with? (A) Asynchronous behavior (setTimeout; setInterval; Process.next; promise; async/await etc.) (B) Third-party APIs (e.g. Node API, Browser API, other people's libraries, modules etc.) (C) Programming logic (D) Syntax (E) Events.

During a workshop in summer 2020 that introduced Dbux to 20 TAs of a local JavaScript Bootcamp provider, we asked the participants what type of bugs they found most difficult to deal with. A total of 10 participants filled out our survey. The top choice for the (multiple choice) "programming problems" was "asynchronous behavior" with 9 votes, while the second place only received 6.

,, Seifert, et al.

#### 2 ASYNCHRONOUS SEMANTICS PRIMER

Fig. 3 illustrates the three types of Asynchronous Events (AE). In all three cases, the resulting Asynchronous Call Graph (ACG) should feature three nodes, connected by two CHAINs.

Below are several illustrations of asynchronous programs and their expected conceptual ACG.

```
let p = P().
  then(f1);

p.then(f2).
  then(f3);

p.then(f4).
  then(f5);
```

Figure 4: Promises (CHAIN vs. FORK)

```
let p = P().
  then (f0);
p. then (g).
                      f0
  then (f3);
                            g
                                  f4
                       G
p.then(f4).
  then (f5);
                                  f5
                      f1
function g() {
                      f3
  G
  return P().
    then (f1);
}
```

Figure 5: Nested Promises (CHAIN vs. FORK)

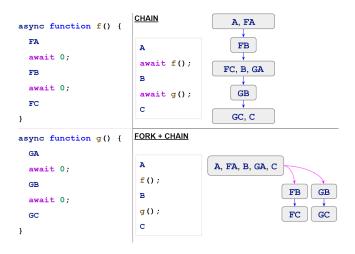


Figure 6: AWAIT (CHAIN vs. FORK)

#### 2.1 Promise Creation Semantics

To better motivate PromiseLinks, consider the following. In JavaScript, promises can be created in four ways. Somewhat counter-intuitively, (i), (ii) and (iii) do *not* cause an asynchronous event on their own. However, all of them can nest promises:

```
async function send(fpath) {
                                       function send(fpath) {
                                                                         function send(fpath, cb) {
 const file = await openFile(fpath);
                                        return openFile(fpath).
                                                                          openFile(fpath, function (file) {
                                          then (function (file) {
const cont = await readFile(file);
                                            return readFile(file);
                                                                            readFile(file, function (cont) {
                                          then (function (cont) {
await sendFile(cont);
                                            return sendFile(cont);
                                                                              sendFile(cont, function () {
                                          }).
                                                                                cb && cb();
                                          then(function() {
                                            console.log('File sent!');
console.log('File sent!');
                                                                                console.log('File sent!');
                                          });
                                                                              });
                                       }
                                                                            });
                                                                          });
```

Figure 3: Three types of AEs implementing a series of three operations: openFile → readFile → sendFile

The (i) Promise constructor takes an executor function which in turn is provided two parameters: the resolve and reject functions which are to be called to fulfill the promise. The executor function is called synchronously from the constructor. The Promise constructor is commonly used to wrap asynchronous callbacks into promises. This process is commonly referred to as "promisification".

(ii) Promise.resolve(x) and Promise.reject(x) are equivalent to using the (i) Promise constructor and synchronously calling resolve or reject respectively. Promise.all and Promise.race can further be used to nest multiple promises into one.

When (iii) an async function is called, the runtime environment creates a new promise. Its call expression value is set to that promise. Async functions execute synchronously until the first await is encountered. This means that if an async function concluded without explicitly invoking an await expression or any of the three other types of events, it does not trigger an asynchronous event. Promises can further be nested by returning them from an async function.

(iv) promise chaining (then, catch, finally).

. Seifert, et al.

#### 3 CONCURRENT DATA FLOW: RESULTS

These are the results from the "Concurrent Data Flow" extension on the three producer-consumer problems:

CrossThreadDataDependencies
producing = 0 producer\_consumer\_base.js:39
lastProducingItem = 0 producer\_consumer\_base....
buffer producer\_consumer\_base.js:47
key seedrandom.js:183
producingBuffer producer\_consumer\_base.js:113
consumerQueue = [] producer\_consumer\_async.j...
nItems = 0 producer\_consumer\_base.js:35
consuming = 0 producer\_consumer\_base.js:38
consumingBuffer producer\_consumer\_base.js:69
producerQueue = [] producer\_consumer\_async.js:...

Figure 7: Async Function Implementation

CrossThreadDataDependencies
producing = 0 producer\_consumer\_base.js:39
lastProducingItem = 0 producer\_consumer\_base...
buffer producer\_consumer\_base.js:47
key seedrandom.js:183
producingBuffer producer\_consumer\_base.js:113
consumerQueue = [] producer\_consumer\_promis...
nItems = 0 producer\_consumer\_base.js:35
consumingBuffer producer\_consumer\_base.js:69
consuming = 0 producer\_consumer\_base.js:38
producerQueue = [] producer\_consumer\_promise...

Figure 8: Promise Implementation

CrossThreadDataDependencies producing = 0 producer\_consumer\_base.js:39 lastProducingItem = 0 producer\_consumer\_base.... buffer producer\_consumer\_base.js:47 key seedrandom.js:183 producingBuffer producer\_consumer\_base.js:113 nItems = 0 producer\_consumer\_base.js:35 consuming = 0 producer\_consumer\_base.js:38 consumingBuffer producer\_consumer\_base.js:69 Figure 9: Callback Implementation

```
TRACE DETAILS
                                                                                 Async Stack
 err query.js:72
                                                ret[key] = _include.model;
        -||潭|
 Value: {name.message.st...} (Error)
                                                if (_include.include) {
                                   60 \
                                                                                 ቹ ■ findOrCreate-parallel.js @ findOrCreate-parallel.js:1
                                                   .merge(ret, this._collectModel = main @findOrCreate-parallel.js:11
  message: SOLITE ERROR: cannot start a tr..
                                                                                 ☐ Model.findOrCreate @ model.js:2270
  stack: Error: SQLITE ERROR: cannot start a.

    Seguelize.transaction @ seguelize.is:1073

  cause: null
  sql: BEGIN DEFERRED TRANSACTION;
                                                                                 ☐ Transaction.prepareEnvironment @ transaction.js:107
 Executions: 2x
                                                                                 ☐ Transaction.begin @ transaction.js:156
                                            return ret:
 Async (root=16313, from=16312)
                                                                                 ☐ QueryInterface.startTransaction @ query-interface.js:1191
 Debug id: 529861
                                                                                 : Sequelize.query @ sequelize.js:511
 Research
                                                                                  70 ~
                                            if (err) {
                                                                                 err.sql = this.sql+;
                                                                                 :■ ■ Query.run @ query.js:219
                                              throw this.formatError (err) ; 6
                                                                                 :≣ ≡ [cb] Promise @ query.js:225
                                                                                 let result = this.instance+;
                                                                                  :≡ ■ Query.executeSql @ query.js:227
                                                                                 ; anonymous) @ sqlite3 | sqlite3.js:7
                                                                                      [cb] normalizeMethod @ sqlite3 | sqlite3.js:86
                                            if (this.isInsertQuery*(results, meta
                                              this.handleInsertQuery (results, me; ≡ Query.afterExecute @ query.js:233
                                              if (!this.instance + +) {
                                                                                  — : Query._handleQueryResponse  @ query.js:69
                                                                                   :■ ■ Query.formatError @ query.js:349
DATA FLOW
```

Figure 10: The sequelize ACS when the first thrown error is selected in the code.

## 4 PROJECT RESULTS

## 4.1 Project ACS

Fig. 10 shows the asynchronous call stack of the sequelize bug. The stack prominently features the sequelize API call that caused it: findOrCreateCall. For contrast: err.stack is empty. formatEr ror returns a new Error object with its stack only containing the functions within its current CGR, up to Query.afterExecute.

#### 4.2 Project ACGs

In the following, we list the raw ACG results of the nine projects that we had to omit from the article for brevity (enhanced for contrast).

TODO: Editor.md screenshot is missing because the application file importer cannot deal with a special property of the Editor.md log (it created contexts that were never sent to the server; possibly due to erroneous <script> loads). The data file is Ok, but the importer needs a bit more resilience. Aiming to fix within 1-2 weeks.

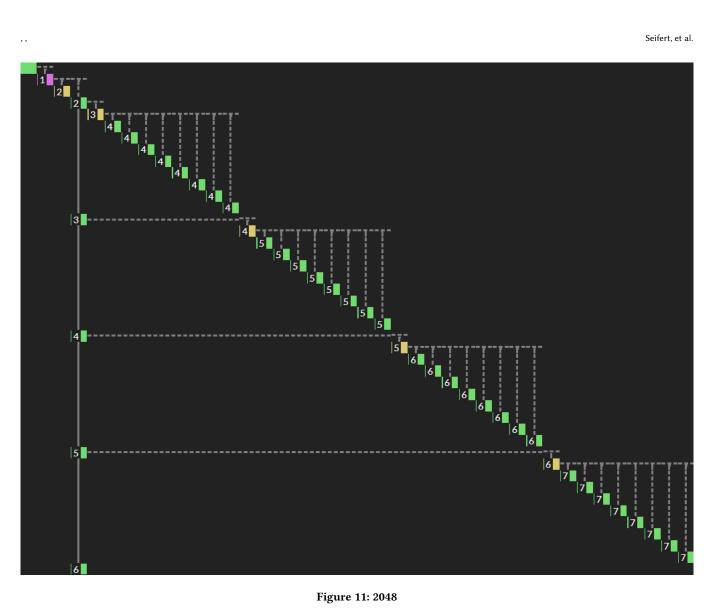


Figure 11: 2048

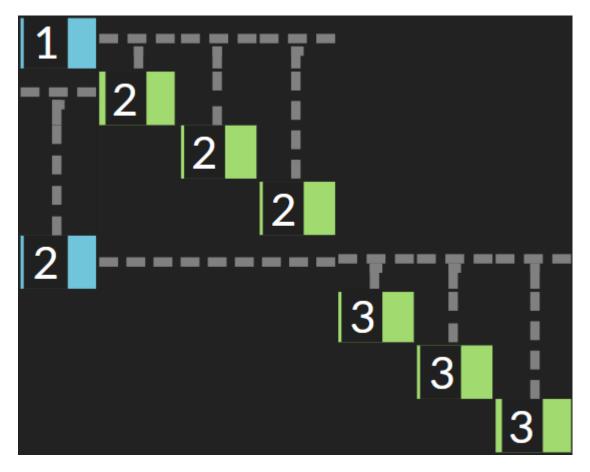


Figure 12: Bluebird

,, Seifert, et al.

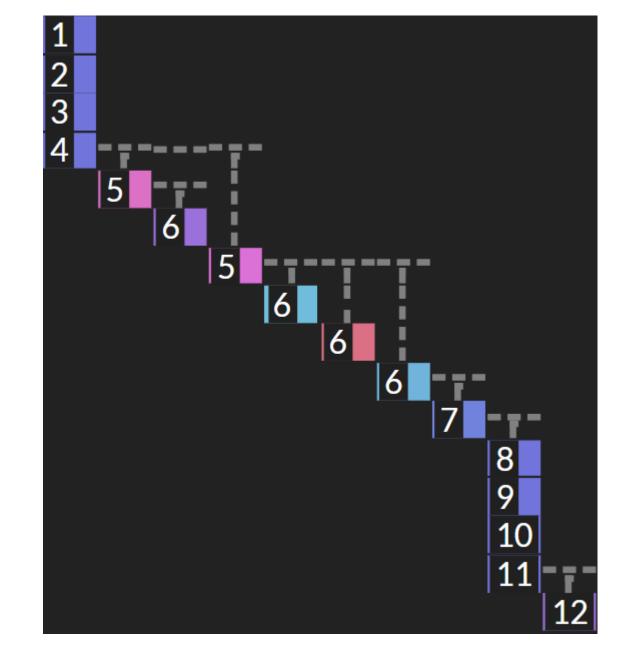


Figure 13: Express



Figure 14: Hexo

Seifert, et al.

1277	1	1335
1278		1336
1279	2	1337
1280	3	1338
1281		1339
1282		1340
1283	3	1341
1284	2	1342
1285	3	1343
1286	3	1344
1287	2	1345
1288	3	1346
1289		1347
1290	3	1348
1291	3	1349
1292	2	1350
1293	3	1351
1294		1352
1295		1353
1296		1354 1355
1297	2	1356
1298 1299	3	1357
1300	3	1358
1301	3	1359
1302	3	1360
1303	3	1361
1304	2	1362
1305	3	1363
1306		1364
1307	3	1365
1308	3	1366
1309		1367
1310	2	1368
1311		1369
1312	2	1370
1313		1371
1314		1372
1315	2	1373
1316	2	1374
1317	3	1375
1318		1376
1319	3	1377
1320	3	1378
1321	3	1379
1322	3	1380
1323	3 3	1381
1324	3	1382
1325 1326	3	1383 1384
1327	3	1385
1328	3	1386
1329	3	1387
1330	<u> </u>	1388
1331	32	1389
1332	11	1390
		1391
1334	Figure 15: node-fetch	1392

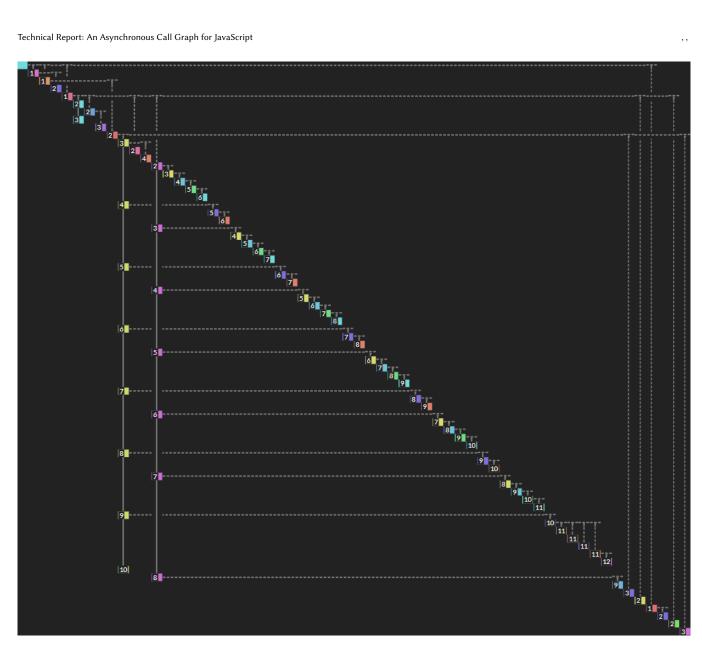


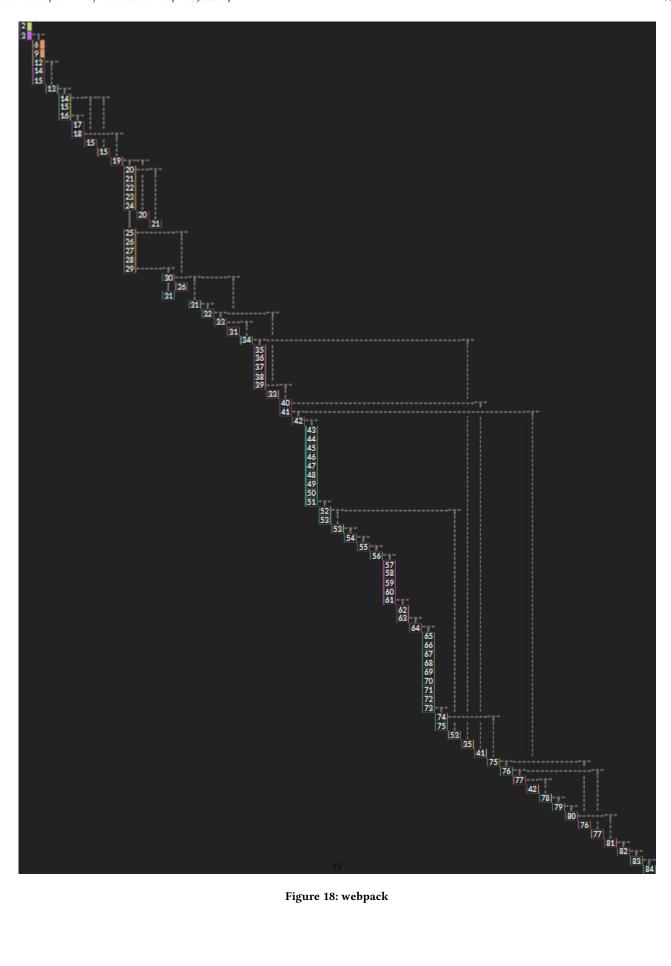
Figure 16: socket.io

,,





Figure 17: todomvc



,, Seifert, et al.

## **REFERENCES**

[1]	(accessed in 1	2/2020) 1	Rahel ht	tns://babel	is io/docs	en/index htm	nl
1	l (accesseu III I	.4/4040). 1	<i>οαυει</i> . πι	tps.//babei	13.10/ uocs	CII/IIIUCX.IIIII	ш

[2] Ecma International. 2021. ECMAScript® 2017 Language Specification. Retrieved 10/2021 from https://262.ecma-international.org/8.0/

[3] Dominik Seifert and Michael Wan. 2019. Dbux. Retrieved 10/2021 from https://github.com/Domiii/dbux