Automatic Migration from Synchronous to Asynchronous JavaScript APIs

SATYAJIT GOKHALE, Northeastern University, USA
ALEXI TURCOTTE, Northeastern University, USA
FRANK TIP, Northeastern University, USA

The JavaScript ecosystem provides equivalent synchronous and asynchronous Application Programming Interfaces (APIs) for many commonly used I/O operations. Synchronous APIs involve straightforward sequential control flow that makes them easy to use and understand, but their "blocking" behavior may result in poor responsiveness or performance. Asynchronous APIs impose a higher syntactic burden that relies on callbacks, promises, and higher-order functions. On the other hand, their nonblocking behavior enables applications to scale better and remain responsive while I/O requests are being processed. While it is generally understood that asynchronous APIs have better performance characteristics, many applications still rely on synchronous APIs. In this paper, we present a refactoring technique for assisting programmers with the migration from synchronous to asynchronous APIs. The technique relies on static analysis to determine where calls to synchronous API functions can be replaced with their asynchronous counterparts, relying on JavaScript's `async`/`await` feature to minimize disruption to the source code. Since the static analysis is potentially unsound, the proposed refactorings are presented as suggestions that must be reviewed and confirmed by the programmer. The technique was implemented in a tool named *Desynchronizer*. In an empirical evaluation on 12 subject applications containing 316 synchronous API calls, *Desynchronizer* identified 256 of these as candidates for refactoring. Of these candidates, 244 were transformed successfully, and only 12 resulted in behavioral changes. Further inspection of these cases revealed that the majority of these issues can be attributed to unsoundness in the call graph.

CCS Concepts: • **Software and its engineering → Software evolution**; **Maintaining software**; **Software maintenance tools**; • **Theory of computation → Program analysis**.

Additional Key Words and Phrases: Refactoring, JavaScript, Static Analysis, Asynchronous Programming

ACM Reference Format:
Satyajit Gokhale, Alexi Turcotte, and Frank Tip. 2021. Automatic Migration from Synchronous to Asynchronous JavaScript APIs. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 160 (October 2021), 27 pages. https://doi.org/10.1145/3485537

1 INTRODUCTION

The JavaScript ecosystem provides equivalent synchronous and asynchronous Application Programming Interfaces (APIs) for many commonly used I/O operations. When faced with a choice, programmers often favor the synchronous APIs because they enable solutions with simple sequential control flow and avoid the complexities of asynchronous programming. However, this greater ease of use comes at a price, as synchronous solutions prevent applications from making

Authors' addresses: Satyajit Gokhale, gokhale.sa@northeastern.edu, Northeastern University, Boston, MA, USA; Alexi Turcotte, turcotte.al@northeastern.edu, Northeastern University, Boston, MA, USA; Frank Tip, f.tip@northeastern.edu, Northeastern University, Boston, MA, USA.

2475-1421/2021/10-ART160
https://doi.org/10.1145/3485537

progress while the I/O operation is pending, given that JavaScript does not support concurrency at the language level. As a result, the use of synchronous I/O in client-side JavaScript applications may cause user-interfaces to become non-responsive, while on the server-side the use of such APIs may negatively affect both responsiveness and throughput. This paper explores the development of automated tool support for assisting programmers with the migration from synchronous APIs to equivalent asynchronous APIs.

As an example, consider the function `fs.readFileSync` in the `fs` (file system) package that is part of the Node.js distribution [OpenJS Foundation 2021]. This function, when given the name of a file, will return a string value representing the file's contents. The operation is *synchronous* in the sense that, when this function is invoked, program execution will block until the read operation has completed. At that time, the function returns a string containing the file contents or throws an exception if an I/O error occurred. The use of blocking synchronous APIs such as `fs.readFileSync` prevents any other event handler from executing, and is considered undesirable because it reduces the application's responsiveness.

To avoid blocking, programmers can use *asynchronous* APIs to perform I/O operations. Two types of asynchronous APIs are commonly used:

- In *event-based asynchronous APIs*, a callback function must be provided that will be invoked upon completion of the I/O operation. For example, the function `fs.readFile` takes a callback as an argument that is invoked when the read operation completes. The callback is invoked asynchronously with two arguments: an error object (or null if no error occurred), and a string containing the file contents if no error occurred.
- In *promise-based asynchronous APIs*, a promise [ECM 2020, Section 25.6] is returned that is settled upon completion of the I/O operation. For example, the function `fs.promises.readFile` returns a promise that is eventually fulfilled with a string value representing the file contents, or rejected with an error message if an I/O error occurred.

These two approaches are functionally equivalent, but promise-based APIs are rapidly gaining in popularity due to the lower syntactic burden and improved facilities for error handling, especially when used in combination with JavaScript's recent **`async`**/**`await`** feature [ECM 2020, Section 25.7].

Despite the advantages of asynchronous APIs in terms of improved responsiveness and scalability, many JavaScript applications continue to rely on synchronous APIs, presumably because of their greater ease of use. This paper presents a semi-automatic refactoring to help users migrate from synchronous APIs to promise-based asynchronous APIs, in combination with the **`async`**/**`await`** feature. The refactoring replaces synchronous calls with asynchronous calls, and automatically determines when functions need to become **`async`** so that asynchronous calls within their body can be awaited using **`await`**. When functions become **`async`**, functions that invoke them may have to become **`async`** as well, which is determined by inspecting the program's call graph. The suggested refactorings may fail to preserve behavior for several reasons. Most significantly, the call graph may be unsound—nodes and edges may be missing because the analysis is unable to reason precisely about JavaScript's dynamic features. Unsoundness may also arise because, after the refactoring, interleavings will be possible where event handlers are scheduled at times when an introduced asynchronous call is being awaited, and we do not attempt to determine how this may impact program behavior. For these reasons, the proposed transformations are presented as *suggestions* that the programmer needs to inspect carefully and confirm, by running tests and inspecting the code.

We implemented the refactoring in a tool named *Desynchronizer*, which supports 51 different synchronous APIs. In an empirical evaluation, we applied *Desynchronizer* to 12 subject applications containing 316 synchronous API calls. *Desynchronizer* identified 256 of these calls as candidates for refactoring. Of these candidates, 244 were transformed successfully, and only 12 resulted in

behavioral changes. Further inspection of these problems revealed that the majority of them can be attributed to unsoundness in the call graph.

In summary, our work makes the following contributions.

(1) We present a technique to generate refactoring suggestions to help users migrate their use of synchronous APIs to asynchronous equivalents.
(2) We implemented this approach in *Desynchronizer*.
(3) We present an empirical study where we apply *Desynchronizer* to 12 applications. In these applications, *Desynchronizer* successfully transformed 244 of the 256 synchronous API calls that were identified as candidates for refactoring. The majority of the 12 cases in which *Desynchronizer* did not preserve behavior can be attributed to unsoundness in the static call graphs upon which *Desynchronizer* relies.

2 BACKGROUND

Nearly all software relies on functions in I/O libraries to interact with file systems, databases, and servers. In JavaScript, library I/O functions can be synchronous or asynchronous. In *synchronous I/O library functions*, execution of the application's source code is suspended until the I/O operation has completed. At this time, the results of the I/O operation are returned as the function's return value. On the other hand, a call to an *asynchronous I/O library function* returns immediately. At a later time, when the I/O operation completes, the application receives notification that the results of the I/O operation have become available. Two mechanisms for providing such notifications are in widespread use:

- In *event-based asynchronous APIs*, a callback function that was passed to the I/O library function is invoked with the result of the I/O operation.
- In *promise-based asynchronous APIs*, the promise [ECM 2020, Section 25.6] that was returned by the I/O library function is resolved or rejected with the value that represents the results of the I/O operation.

Below, we briefly discuss synchronous and asynchronous I/O libraries using examples, and discuss the tradeoffs of the two mechanisms.

2.1 Synchronous I/O

Figure 1(a) shows a function `compress` that uses the `gzipSync` function provided by the built-in `zlib` package of Node.js to compress its argument `input`.

On line 12, `compress` is invoked on the string value `"example"`, causing `gzipSync` to be invoked on line 5 to compress this value. Execution of the program will be suspended until the compression operation has completed, at which time a buffer containing the compressed string will be assigned to variable `output`, which is then printed. If an error occurs, the handler on line 8 will execute.

As can be seen in the figure, the use of synchronous APIs such as `gzipSync` leads to code with straightforward control flow that is easy to understand. However, use of such *blocking I/O prevents any event handler from executing*, which, e.g., leads to applications becoming unresponsive while I/O is taking place, which can be problematic.

2.2 Event-Driven Asynchronous I/O

Figure 1(b) shows a variation on the example of (a) in which an equivalent event-based asynchronous API is used. Here, the function `gzip` is invoked on line 17, to start the compression operation. The second argument is a callback function that is *invoked asynchronously when the I/O operation has completed*. This callback takes two arguments, `err` and output. If the operation was successful, `err`

```
1  let zlib = require('zlib');
2
3  function compress(input){
4    try {
5      let output = zlib.gzipSync(input);
6      console.log(output);
7    } catch (err){
8      console.log("Error:␣" + err);
9    }
10 }
11
12 compress("example")
13
```

(a)

```
14 let zlib = require('zlib');
15
16 function compress(input){
17   zlib.gzip(input, (err, output) => {
18     if (err){
19       console.log("Error:␣" +
20                   err);
21     } else {
22       console.log(output);
23     }
24   })
25 }
26
27 compress("example")
28
```

(b)

```
29 let util = require('util');
30 let gzip = util.promisify(
31   require('zlib').gzip
32 );
33
34 async function compress(input){
35   try {
36     let output = await gzip(input)
37     console.log(output);
38   } catch (err){
39     console.log("Error:␣" + err);
40   }
41 }
42
43 compress("example")
44
```

(c)

Fig. 1. Example usage of `gzip` using synchronous and asynchrous APIs: (a) uses a synchronous API, (b) uses an event-driven asynchronous API, (c) uses a promise-based asynchronous API.

has the value `null` and output contains the result of compressing the input value, which is then printed on line 22. If the operation fails, an error message is printed on line 39.

The use of an asynchronous API such as `gzip` has the advantage that it is *non-blocking*, thus preventing the responsiveness issues triggered by the use of synchronous APIs. However, the use of event-based asynchronous APIs leads to convoluted and error-prone control flow [Madsen et al. 2015] sometimes referred to as "callback hell".

An alternative asynchronous method of performing I/O uses promises and **`async`**/**`await`**, and is depicted in Figure 1(c)—before describing that code, however, we will review JavaScript's promises and **`async`**/**`await`** features.

2.3 Brief Review of Promises and Async/Await

In recent years, the JavaScript community has rapidly adopted promises as a mechanism for asynchronous programming that is more convenient and less error-prone than event-based programming. Here, we present a very brief summary of promises. For complete details, the reader is referred to the ECMAScript specification [ECM 2020, Section 25.6], and a formal account of the semantics of promises can be found in [Madsen et al. 2017].

A *promise* represents the value computed by an asynchronous computation, and is one of three states: *pending*, *fulfilled*, or *rejected*. Upon creation, a promise is in the pending state until it is settled (i.e., fulfilled or rejected).

Creating promises. A promise is created by invoking the `Promise` constructor, e.g.:

```
let p = new Promise((resolve, reject) => {...})
```

Here, `resolve` and `reject` are functions that must be invoked to resolve or reject a promise, respectively. For example, the following code

```
let p1 = new Promise((resolve, reject) => resolve(17))
```

creates a promise that is fulfilled immediately with the value 17.

Registering reactions on promises. Programmers can register *reactions* on promises, i.e., functions that are invoked asynchronously when a promise is fulfilled or rejected. This is accomplished using the `then` method, e.g., extending the previous example with

```
p1.then( (v) =>  v+1 )
```

registers a reaction on `p1` that is invoked with its argument `v` bound to the value that `p1` was resolved with (i.e., 17). This reaction will be executed asynchronously and return `v+1`, i.e., 18.

Creating promise chains. The `then` method returns a promise that is resolved with its reaction's return value, enabling the construction of *promise chains*. In the following example:

```
let p = new Promise((resolve, reject) => resolve(17))
p.then( (v) => v + 1 )
 .then( (v) => console.log(v) )
```

a promise is created that is fulfilled with the value 17. The reaction registered on this promise returns 18, causing the first invocation of `then` to return a promise that is resolved with that value. On the last line, a reaction is registered on the latter promise, causing `18` to be printed.

Promises also support a method `catch` to facilitate error handling, which is typically used at the end of a promise chain. For example, in the following example

```
let p = new Promise( ... )
p.then( ... )
 .then( ... )
 .catch( (err) => console.log("An error occurred") )
```

the reaction passed to `catch` will be executed if any of the previous promises in the chain are rejected (e.g., if an uncaught exception occurs).

Promise linking. A special case arises when a reaction returns a value that is a promise p. In such cases, the promise p' returned by `then` or `catch` becomes *linked* with p. Concretely, if p is resolved with a value v, then p' is resolved with v as well, if p is rejected with value e, then p' is rejected with e as well, and if p remains pending, so does p'. For example, the following code fragment:

```
let p = Promise.resolve(17)
let q = new Promise((resolve, reject) =>
  setTimeout( () => resolve(18) , 1000)
)
p.then( () => q ) // the promise returned by p.then becomes linked with q
 .then( console.log ) // prints 18 after 1 second
```

creates promises `p` and `q`. When `p` is fulfilled, its reaction is executed and returns `q`, so `q` and the promise returned by `p.then()` become linked. After 1 second, `q` resolves to 18, so the promise returned by `p.then()` resolves to 18 as well, causing the second reaction to execute, which prints this value.

Synchronization using `Promise.all`. The `Promise.all` function provides a mechanism for waiting for a set of promises to be fulfilled in no particular order. It takes as an argument an array of promises $p_1, \cdots, p_n$ and returns a promise p'. If each p_i is fulfilled with a value v_i, then p' is resolved with an array $[v_1, \cdots, v_n]$. If any p_i is rejected, then p' is immediately rejected with that value, regardless of the disposition of the promises p_j $(j \neq i)$.

Interoperability with `async`/`await`. The `async`/`await` feature provides syntactic sugar on top of promises. A function declared as `async` returns a promise that is fulfilled with the function's return value. Inside an `async` function, `await`-expressions may be used to wait for a promise to be settled. If an awaited promise p is fulfilled with value v, then an expression `await` p evaluates to v; if it is rejected with a value `err`, `err` is thrown as an exception that can be caught using standard `try`/`catch` error handling.

2.4 Promise-Based Asynchronous I/O

Figure 1(c) shows another version of the example, in which a promise-based version of the asynchronous `gzip` API is used. Here, the `promisify` function from the standard package `util` is used to convert the callback-based `gzip` API into an equivalent promise-based API. *Promisification* is a technique for converting an event-based API into an equivalent promise-based API, and involves the creation of a promise that is fulfilled or rejected when the callback is invoked, depending on whether an error occurred.

The use of a promise-based API enables us to turn `compress` into an `async` function, in which the `await`-expression on line 36 will suspend execution of `compress` until the promise returned by `gzip` is resolved or rejected. Standard `try/catch` syntax can be used for error handling, leading to code that is very similar to that in Figure 1(a). Crucially, the use of the asynchronous `gzip` function does *not* block program execution, allowing other event handlers to execute until the promise returned by `gzip` has been settled.

In summary, promise-based asynchronous APIs share the convenient syntax and support for error handling using `try`/`catch` with synchronous APIs, while providing the responsiveness benefits of event-based asynchronous APIs.

2.5 Performance Benefits of Asynchronous APIs

Asynchronous APIs (both event-based and promise-based) may enable multiple I/O requests to be processed concurrently. To illustrate this point, consider Figure 2, which shows the relevant code fragments of two versions of a server application[1] that performs a compression operation to elements of an array of files. Compression is widely used by server-side web applications, in order to reduce the size (and hence the time needed to transfer) of the responses it sends.

Figure 2(a) shows a version that uses the synchronous API discussed in Section 2.1. Here, the `forEach`-loop on line 65 is used to invoke `gzipSync` on each file in an input array `files`, storing the result in a newly created array `compressedFiles`.

Figure 2(b) shows an alternative solution that uses the promise-based asynchronous API discussed in Section 2.4. In this version, lines 75–76 create an array of promises that will eventually resolve to the gzipped files. After creating all of these promises, the synchronization operation `Promise.all` is used to wait for all of these promises to be fulfilled.

These two versions of the applications have very different performance characteristics. In version (a), the gzip operations occur in strictly sequential order, with each gzip operation starting after the previous one has finished.

[1]The full source code of this application is available as part of supplemental materials.

```
61  const gzipSync = require('zlib').gzipSync;
62
63  function compressFiles(files) {
64    const compressedFiles = [];
65    files.forEach(file => {
66       compressedFiles.push(gzipSync(file));
67    });
68    return compressedFiles;
69  }
```

(a)

```
70  let util = require('util');
71  let gzip =
72    util.promisify(require('zlib').gzip);
73
74  async function compressFiles(files) {
75    const cFileProms =
76      files.map(f => gzipPromise(f));
77
78    return await Promise.all(cFileProms);
79  }
```

(b)

Fig. 2. Excerpt of a server application for compressing a list of files: (a) synchronous version using `gzipSync` (b) promise-based asynchronous version using `gzip`

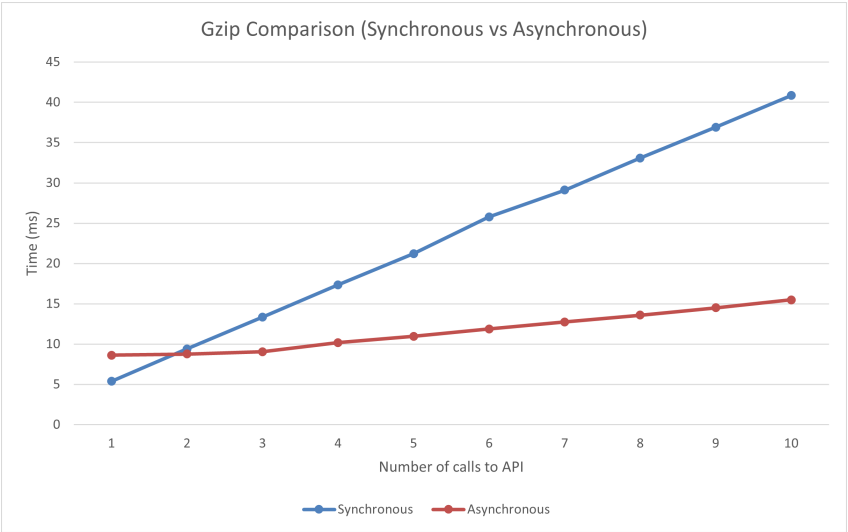

Fig. 3. Gzip performance comparison (sync vs async).

In version (b), all `gzip` operations are started at once, without waiting for the previous operation to terminate. Although JavaScript does not have concurrency at the language level, it relies on I/O libraries that take advantage of the fact that modern operating systems are capable of handling multiple I/O requests concurrently.

While this suggests that asynchronous solutions might scale better than sequential ones, we are not aware of published research in which this difference in performance is confirmed experimentally. Therefore, we constructed a set of synthetic benchmarks in which the performance of synchronous APIs such as the one in Figure 2(a) is compared against that of a corresponding asynchronous API such as the one in Figure 2(b). Figure 3 below visualizes the results on such an experiment for the `gzipSync` and promisified `gzip` APIs[2]. The figure shows that, if there is only one call to the `gzipSync`/`gzip` API, the synchronous solution requires 5 msec and the asynchronous solution 8 msec. However, as the number of API calls increases, the asynchronous solution quickly outperforms the

[2]The results for similar experiments involving 10 APIs can be found in the supplemental materials.

```
80  const express = require('express');
81  const fs = require('fs');
82  const zlib = require('zlib');
83
84  const processRequest = (basePath) => {
85    const variants = fs.readdirSync(basePath);
86    return variants.map(variant =>
87      JSON.parse(fs.readFileSync(basePath + variant, 'utf8'))
88    );
89  }
90
91  const app = express();
92  app.get('/product/details', (request, response) => {
93      const data = processRequest(__dirname + '/variants/');
94      response.send(zlib.gzipSync(JSON.stringify(data)));
95  });
96
97  var port = process.env.PORT || 5000;
98  app.listen(port);
```

Fig. 4. Example server application that uses synchronous APIs.

synchronous one (e.g., 40 msec for the synchronous solution and only 15 msec for the asynchronous solution if there are 10 API calls).

To obtain such performance benefits, programmers should refactor their code to migrate from synchronous to asynchronous APIs. The next section presents such a refactoring.

3 MIGRATING FROM SYNCHRONOUS TO ASYNCHRONOUS APIS

This section explores the challenges associated with refactoring applications so that uses of synchronous APIs are replaced with their promise-based asynchronous counterparts. Here, we will illustrate the issues and review the code transformations involved in the refactoring using a motivating example that represents a small server application[3]. Section 4 will present our approach for automating this refactoring.

Figure 4 shows a simple server application that was built using the `express` framework [Exp 2021] on the Node.js platform [OpenJS Foundation 2021]. The application responds to requests for product details by invoking the `processRequest` function on line 93. This function obtains a list of all files containing descriptions of different variants of the product by calling `readdirSync` on line 85. Then, it relies on the `Array.map` function to read each file by invoking `readFileSync` and calls `JSON.parse` to parse it and create its JSON representation on line 87. The resulting array is converted to a string and then compressed by invoking `gzipSync` on line 94 before being returned to the user.

Note that the use of three synchronous APIs (`readdirSync` on line 85, `readFileSync` on line 87, and `gzipSync` on line 94) will render the server unresponsive while it waits, e.g., for a file to be read.

To improve scalability, the application can be refactored using our *Desynchronizer* tool to use the asynchronous counterparts for these APIs resulting in the code that is shown in Figure 5[4]. The key steps in this refactoring include: replacing calls to synchronous API functions with **`await`**-expressions that refer to the corresponding asynchronous API function, and changing functions containing migrated API functions into **`async`** functions, as **`await`**-expressions can only be used inside **`async`** functions [ECM 2020, Section 25.7]. In addition, the callers of functions that have become **`async`**

[3]This application is available as part of supplemental materials.

[4]The code shown here has been modified slightly for readability, by changing variable names and eliminating common subexpressions. The original version produced by our tool is included as part of supplemental materials.

```
101  const express = require('express');
102  const fs = require('fs');
103  const zlib = require('zlib');
104  const util = require('util');
105  const readdir = util.promisify(fs.readdir);
106  const readFile = util.promisify(fs.readFile);
107  const gzip = util.promisify(zlib.gzip);
108
109  const processRequest = async (basePath) => {
110    const variants = await readdir(basePath);
111    return Promise.all(variants.map(async (variant) =>
112      JSON.parse(await readFile(basePath + variant, 'utf8'))
113    ));
114  };
115
116  const app = express();
117  app.get('/product/details', async (request, response) => {
118    const data = await processRequest(__dirname + '/variants/');
119    response.send(await gzip(JSON.stringify(data)));
120  });
121
122  var port = process.env.PORT || 5000;
123  app.listen(port);
```

Fig. 5. Refactored version of the server application that uses asynchronous APIs.

may have to become **async** functions as well, depending on if and how they use the return value of the invoked function; for example, if a function needs to **await** a call to an asynchronous function, it will need to become **async** as well. In cases where API functions are invoked inside loops or in invocations of callbacks passed to `Array.forEach`, we perform a simple analysis to determine whether the loop iterations or callback invocations are independent (i.e., if they operate on different state in each iteration or invocation). If so, we rely on `Promise.all` to enable each iteration or invocation to proceed concurrently. If not, the loop order is preserved and the promises computed in each iteration are awaited in succession.

Concretely, in the case of our motivating example, the source code changes can be summarized as follows:

- The calls to `readdirSync`, `readFileSync`, and `gzipSync` are replaced with **await**ed calls to promisified versions of the library functions `readdir` (line 110), `readFile` (line 112), and `gzip` (line 119, respectively,
- `processRequest` has become an **async** function (lines 109–114), to enable the use of **await**-expressions inside its body.
- Similarly, the callback functions passed to `variants.map` (lines 111–113) and `app.get` (lines 117–120) are made **async** as well, to enable the use of **await** in their bodies.
- Finally, the synchronization function `Promise.all` is used on line 111 to wait until each element of the array of promises returned by the call to `variants.map` has settled.

We constructed a set of synthetic benchmarks to measure the performance of the synchronous implementation shown in Figure 4 and compared the results against the asynchronous implementation shown in Figure 5. The code in Figure 4 loops over user requests one-by-one, and blocks on a file access needed to process the request, whereas the code in Figure 5 can processes requests *concurrently* using promises and **async**/**await**.

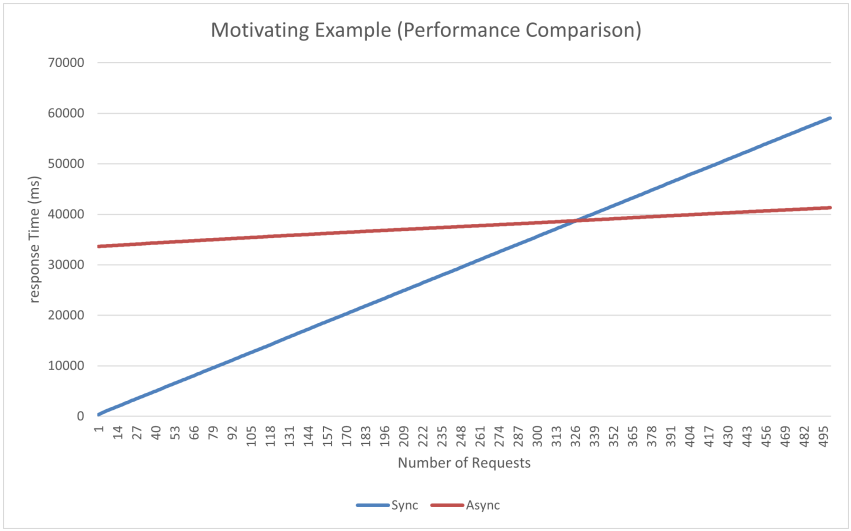

Fig. 6. Performance of server example, comparing response time for synchronous and asynchronous implementations.

We observed that the synchronous implementation of the server performs better for a low number of clients, whereas the asynchronous implementation performs much better when dealing with a high number of concurrent clients. The performance breakdown can be seen in Figure 6.

The next section will present our approach for inferring these transformations.

4 APPROACH

We present a three-step approach for automatically determining how to migrate from synchronous APIs to their asynchronous counterparts, consisting of the following steps:

Identify Transformation Candidates. The first step of our approach is to identify synchronous API calls that we aim to transform, and determine the extent of the required transformations. Once a synchronous call is identified and made asynchronous, it must be enclosed in an `await` expression to preserve the original semantics of the program: If the original, synchronous call returned a value of type *T*, the now asynchronous call will return a value of *Promise<T>*, and awaiting this call will ensure that the expected value becomes available. In JavaScript, `await` expressions can only appear in `async` functions, so this transformation *propagates* up from callee to caller until the top-level of the program or an already asynchronous function is reached. The result of this step is a *transformation set* of functions that will need to become `async`, and will be discussed in more detail in Section 4.1.

Discard Unsupported Transformations. Each transformation set is then checked to ensure that all functions can be transformed, since the transformation to support asynchrony is an all-or-nothing proposition. There are several reasons why such a set may need to be discarded: for example, transformations that would require making a constructor into an `async` function must be abandoned, as constructors cannot be `async` in JavaScript. Section 4.2 reviews the preconditions that must hold in order to allow migration to asynchronous APIs.

Transform Source Code. The last step is to analyze the transformation sets and apply the necessary transformations to the code. These range from simple transformations, such as turning calls to synchronous APIs into `await`-expressions, to more complex transformations,

such as changing a map of a (now asynchronous) function into an array of promises that is awaited simultaneously using `Promise.all`. Further details are presented in Section 4.3.

4.1 Identify Transformation Candidates

The first step in the approach is to identify transformation candidates and build a transformation set for each synchronous API call. Algorithm 1 shows how transformation sets are constructed.

Algorithm 1: BuildTransformationSet

```
   Data: n_sync: a synchronous API call
   Data: CG: the call graph of the program
1  let T := [];
2  let P := [n_sync];
3  while P not empty do
4      let p := pop (P);
5      if p not visited then
6          if p is a reaction or p is argument to promise constructor then
7              continue;
8          let callers := callers of p in CG;
9          for c in callers do
10             if c test runner then
11                 remove c from callers;
12         T := T ∪ callers;
13         P := P ∪ callers;
14         mark p as visited;
15 return T;
```

On line 1, the algorithm initializes a list T, which will contain all functions requiring transformation, and line 2 initializes P, the list of functions that still need processing. While there are still functions to process (line 3), a function p is popped from P (line 4). If it has not been visited (line 5), the algorithm identifies all callers of p in the call graph (line 8), and adds them to the list T of functions that would need to be transformed (line 12), and also to the list of functions P that still need to be processed (line 13), as we need to recursively visit their callers. When no additional functions can be found, T is returned.

There are a few situations where the traversal of the call graph can be ended early. The first case we will discuss is depicted on lines 9-11 of the algorithm. If a function c is part of a test harness (like Jest [Jes 2021] or Mocha [Moc 2021]), there is no need to transform it or propagate the transformation through it, as they are already equipped to handle asynchrony.

The next case, depicted on lines 6-7, is more subtle. If a function that is flagged for transformation is either an argument to the promise constructor, or is a reaction registered through `then` or **`catch`**, then the transformation need not be propagated further. Recall that when a function is made **`async`**, its semantics are altered such that it will now return a promise. In the first case, the promise constructor does not do anything with the return value of the function passed to it, and so no further transformations are necessary. In the second case, reactions registered with `then` and **`catch`** *already* return promises, and so the promise returned by the newly asynchronous reaction will be *linked* with the promise that is already returned, which will preserve the original semantics.

To illustrate what a transformation set would look like, consider the following snippet:

```
124  function inner(listOfFiles) {
125    listOfFiles.map(readFileSync);
126  }
127  function outer(listOfFiles) {
128    inner(listOfFiles);
129  }
130  outer(["a.txt", "b.txt"]);
```

We draw the reader's attention to the synchronous API call inside of the call to `Array.map` in the body of function `inner`. Since `map` invokes the callback function that is passed to it as an argument[5], our algorithm would add `map` to the transformation set for the call to `readFileSync`. Since `map` is called by `inner`, which is called by `outer`, which is called at the top level of the program, the following transformation set will be computed:

{`fs.readFileSync`, `Array.map`, `inner`, `outer`, program root}.

4.2 Discard Unsupported Transformations

Once transformation sets have been constructed for each synchronous API call, it is necessary to check if the associated transformations are feasible. In general, the transformation of a synchronous API call must be abandoned if any function in its transformation set is called in a context which is flagged as not transformable by Algorithm 2:

Algorithm 2: IsSetViable

```
   Data: T: a transformation set
1  for function f in T do
2      if f is a constructor or
3      f returns a promise or
4      f unhandled extern
5      f does not support IIFE then
6          return false;
7  return true;
```

The cases where transformation sets are abandoned can be found on lines 2-5, and consist of:

constructors. Constructors cannot receive the `async` modifier. This precludes the transformation of synchronous API calls in constructors or in functions called by constructors because `await`-expressions can only be used in the body of `async` functions.

functions already returning promises. Transformations involving functions that already return promises are also rejected. If these functions were to be made `async`, they would return a promise linked to another promise, and while this is not an issue in and of itself given the behavior of linked promises, the insertion of `await` expressions at call sites to such functions is fraught. To illustrate, consider the code in Figure 7, where the transformation would alter the value stored in `r` (it would no longer be a promise). Determining the return type of a function is difficult in JavaScript due to the dynamism of the language. In the case of this example, our approach infers that the function returns a promise from the fact that reactions are registered on it using `then`, and therefore abandons the transformation.

[5]To determine this, we rely on a call graph builder that is equipped with models for external functions such as `map` and `forEach`.

```
let n = "foo.txt";
let q;
function foo() {
  q = fs.readFileSync(n);
  // ...
  return some_promise;
}

let r = foo(); // is a promise
// ...
r.then(() => { ... });
```

(a) Untransformed code.

```
let n = "foo.txt";
let q;
async function foo() {
  q = await fs.promises.readFile(n);
  // ...
  return some_promise;
}

let r = await foo(); // not a promise
// ...
r.then(() => { ... });
```

(b) Transformed code.

Fig. 7. A rejected transformation involving a function that already returned a promise.

unsupported external functions. The technique supports the external functions `Array.map` and `Array.forEach`, as will be discussed in further detail in Section 4.3. If calls to synchronous APIs occur in (functions invoked by) callbacks passed to other external functions, the transformation is abandoned.

asynchronous calls at the top level. JavaScript only allows the use of `await` inside the body of `async` functions. That said, if a transformation would require an `await`-expression at the top-level of the program, our approach will attempt to introduce an immediately-invoked (asynchronous) function expression (IIFE). Unfortunately, this is not always possible: ES6 introduced *static* `import` and `export` statements to JavaScript. `import` statements can be safely hoisted outside of the IIFE (as one can only import string literals), but `export` statements cannot be, as they can refer to values computed at run time that will be out-of-scope if removed from the IIFE.

4.3 Transform Source Code

After eliminating the transformation candidates that do not satisfy the conditions presented in Section 4.2, the final step is transformation of the source code. Figure 8 presents these transformations as a set of rewrite rules. Throughout these rules, we make reference to T_f, the set of functions that are to be made asynchronous, obtained by taking the union of all transformation sets that have not been discarded. Note that these rules asbtract away from minor details in JavaScript syntax, e.g., in rule Async-Function, `fun f(A) {B}` refers to *any* JavaScript function, including arrow functions.

The first two rules, Async-Function and Async-Call, are concerned with making functions asynchronous. These transformations simply add the `async` modifier to functions, and turn function calls into `await`-expressions if the function is in T_f. All call sites identified by our call graph as calling a newly asynchronous function are `await`-ed in this way.

Next, ForEach-ForOf defines translation of `Array.forEach` into an `async`-ready `for ... of` loop. If the callback `f` is being made asynchronous, the body B of `f` is found, and the name e of the single argument of `f` is extracted (note: the callback passed to `Array.forEach` must be a single-argument function). Then, a `for ... of` loop is constructed as follows: the loop iterator variables are $[i, e]$ (i for the array index, which is unused, and e, the callback argument name extracted previously), and the loop iterates over the entries of the array. Note that if the callback passed to `Array.forEach` contains a return, we do not perform this transformation; typically, these callbacks do not contain returns as `Array.forEach` does not return anything. The body of the loop is simply the body of the callback, and requires no additional transformation.

$$\frac{\texttt{f} \in T_f}{\texttt{fun f(A) \{B\}} \;\longrightarrow\; \texttt{async fun f(A) \{B\}}}\ \textsc{(Async-Function)} \qquad \frac{\texttt{f} \in T_f}{\texttt{f(args)} \;\longrightarrow\; \texttt{await f(args)}}\ \textsc{(Async-Call)}$$

$$\frac{\texttt{f} \in T_f \qquad B \text{ body of f} \qquad \text{no returns in } B \qquad e = \text{the single argument of f}}{\texttt{arr.forEach(f)} \;\longrightarrow\; \texttt{for([i, } e\texttt{] of arr.entries()) } \{B\}}\ \textsc{(forEach-forOf)}$$

$$\frac{\texttt{f} \in T_f \qquad \texttt{f equivalent to (v) => e} \qquad \texttt{f}_\texttt{A} = \textbf{transform}(\texttt{f})}{\texttt{arr.forEach(f)} \;\longrightarrow\; \texttt{await Promise.all(arr.map(f}_\texttt{A}\texttt{))}}\ \textsc{(forEach-map)}$$

$$\frac{\exists\, \texttt{f} \in B \mid \texttt{f} \in T_f \qquad B' = \textbf{transform}(B) \qquad \texttt{f}_\texttt{A} = \texttt{async fun f}_\texttt{A}\texttt{() } \{B'\}}{\texttt{get p() } \{B\} \;\longrightarrow\; \texttt{get p() } \{\texttt{return f}_\texttt{A}\texttt{()}\}}\ \textsc{(getter)}$$

$$\frac{\texttt{f} \in T_f \qquad \texttt{f}_\texttt{A} = \textbf{transform}(\texttt{f})}{\texttt{arr.map(f)} \;\longrightarrow\; \texttt{await Promise.all(arr.map(f}_\texttt{A}\texttt{))}}\ \textsc{(Promise-All)}$$

$$\frac{P \in T_f \qquad \mathit{imports} = \textbf{hoistImports}(P) \qquad P' = \textbf{removeImports}(P)}{P \;\longrightarrow\; \mathit{imports};\ \texttt{(async () => } \{P'\}\texttt{)()}}\ \textsc{(Top-Level-IIFE)}$$

Fig. 8. Rewrite rules describing the sync-to-async transformation.

Another rule, ForEach-map, depicts a special case of a `forEach` transformation, when the callback passed to `forEach` is an arrow function expression with a single expression as its body. This situation is more elegantly handled by a transformation to `Array.map`. There is nothing in principle preventing a transformation to a `for ... of` loop like above, this is simply more in line with observed language paradigms (programmers often use `Array.forEach` and `Array.map` with arrow functions of this style).

Rule Promise-All depicts a case where `Array.map` is translated into an equivalent construct that properly handles asynchronous function calls. If the callback `f` is being made asynchronous, the required transformation (depicted here with $\texttt{f}_\texttt{A} = \textbf{transform}(\texttt{f})$) is applied, and the new call to $\texttt{arr.map}(\texttt{f}_\texttt{A})$ is wrapped in an awaited `Promise.all`. This has the effect of waiting until the (now asynchronous) applications of $\texttt{f}_\texttt{A}$ for each of `arr`'s elements completes, semantics similar to that

of the original map. The reasoning behind this transformation is that mapping an asynchronous function on an array would result in an array of promises, and awaiting a call to `Promise.all` will wait for all of the promises to settle.

Getter defines the translation of a special case of function definitions. Here, if the body B of some getter `p` contains a function to transform, the body is transformed into B' and wrapped in a new asynchronous function f_A, and the body of the getter is changed to immediately return a call to that function. Getters in JavaScript are forbidden from being made asynchronous, and so the body of the getter is translated into an anonymous async function which is immediately called and returned. This causes the getter to return a promise, and so calls to the getter can be awaited.

Finally, Top-Level-IIFE depicts the transformation for introducing immediately invoked asynchronous function expressions to the top level of the program. The transformation relies on an external function, **hoistImports**, that simply identifies and collects all of the `import` statements from the program body, P. P' is the transformed program body with the `import` statements removed. The transformed body is surrounded by an asynchronous IIFE, which in JavaScript looks like `(async () => {...})()`, which can accommodate `await` expressions in P'.

The motivating example in Figures 4 and 5 shows code examples of transformation rules Async-Function, Async-Call, and ForEach-ForOf.

4.4 Call Graph Construction

Our technique relies on call graphs to determine functions that transitively invoke methods containing synchronous calls and that may need to be changed into `async` functions. In principle, any call graph construction algorithm can be used for this purpose, allowing for a range of tradeoffs in terms of cost and precision. In practice, the high level of dynamicity in the JavaScript language makes it extremely challenging to compute call graphs soundly while at the same time computing results that are sufficiently precise and scalable to be useful. Very few practical implementations of call graph construction algorithms for JavaScript exist, and we are not aware of any that can easily be applied to the Node.js applications that we consider.

In this project, we aim to use call graphs in an interactive tool that generates refactoring suggestions that need to be reviewed and confirmed by the programmer. In this setting, some unsoundness can be tolerated, but the analysis must complete in a reasonable amount of time. We opted to design and implement a simple underapproximate algorithm, which is based on the approximate call graph construction algorithm by Feldthaus et al. [2013]. The algorithm is a constraint-based analysis that simultaneously constructs a data flow graph and a call graph. The data flow graph shows how functions and classes flow from their declaration to expressions that may hold a reference to them. The call graph shows, for each call-expression, what functions may be invoked from that call site. Similar to Feldthaus et al., only the flow of functions and classes is tracked (i.e., the flow of primitive values and objects is ignored). However, our algorithm extends the algorithm by Feldthaus et al. in several signficant ways: it accommodates all features in ECMAScript 2020 [ECM 2020], such as classes, the ECMAScript Module System (ESM) and many other features that did not exist in JavaScript at the time of the work by Feldthaus et al.

The algorithm is specified as a set of constraint generation rules and implemented using GitHub's CodeQL language [Avgustinov et al. 2016]. While the exact details of the algorithm are not pertinent to the scope of our project, the reader may find a comprehensive presentation in Appendix A.

4.5 Implementation

The technique described in this section was implemented in a tool named *Desynchronizer*. The tool is implemented in JavaScript (running on NodeJS v14.15.4), and uses the Babel [BabelJS 2021] parser to build the program AST. The static call graph builder is implemented in version of CodeQL

Table 1. Evaluation Subject Applications

Project	LOC	Files	Functions	Sync API Calls
deepforge	34,541	314	7,169	44
meteor-desktop	13,017	120	1,172	228
apps	2,349	26	179	46
switchBoard	14,476	268	426	70
flatsheet	28,701	32	3,129	20
bonescript	12,832	56	410	139
FiltersCompiler	3,115	25	174	49
a11y-metrics	1,589	28	56	14
ember-watson	2,445	142	227	97
adapt_authoring	45,354	665	16,573	31
TurboScript	34,264	23	2,307	51
useragent	8,295	12	95	6

v2.2.4 wherein we modified a library, as by default CodeQL ignores externally defined functions such as the synchronous APIs targeted by our approach, and has little support for externs such as `Array.map` and `Array.forEach`. We modified CodeQL's definitions for external library functions to reflect how they invoke callback functions. The transformations themselves are carried out by replacing subtrees in the AST and using Babel to regenerate the code afterwards. Our code and experiments are available as an artifact [6].

5 EVALUATION

In the previous two sections, we have described the approach behind *Desynchronizer*, as well as the implementation of the tool itself. This section reports on an evaluation of *Desynchronizer*, in which we aim to answer the following research questions:

- **RQ1 (Applicability)** How many transformations are proposed by *Desynchronizer*?
- **RQ2 (Soundness)** How often can the unsoundness of proposed transformations be detected by running application test suites?
- **RQ3 (Performance)** What is the impact of refactoring on the performance of applications?
- **RQ4 (Processing Time)** What is the performance of *Desynchronizer*?

We answer these through a series of experiments conducted on 12 subject applications from `npm`.

5.1 Subject Applications

Table 1 lists some general information about the projects selected for our evaluation. The first row of the table reads: the `deepforge` project has 34,541 lines of code across 314 files, contains 7,169 functions, and 44 synchronous API calls.

To find these projects, we ran a QL query over some 50k projects that highlighted projects which had calls to synchronous API functions[7] supported by our tool. We designed our query to look for calls to these functions in the main project sources (excluding test code, examples, config files, etc.). To determine which of these projects to use in our evaluation, we cloned and built the projects from Github and kept those which had tests that ran and passed (note: very few JavaScript projects

[6] Link to artifact: https://doi.org/10.5281/zenodo.5502210

[7] `readFileSync`, `writeFileSync`, `readdirSync`, `accessSync`, `appendFileSync`, `chmodSync`, `fchmodSync`, `lchmodSync`, `chownSync`, `fchownSync`, `lchownSync`, `mkdirSync`, `mkdtempSync`, `statSync`, `lstatSync`, `fstatSync`, `linkSync`, `symlinkSync`, `readlinkSync`, `realpathSync`, `unlinkSync`, `rmdirSync`, `renameSync`, `openSync`, `closeSync`, `existsSync`, `copyFileSync`, `truncateSync`, `ftruncateSync`, `utimesSync`, `futimesSync`, `fsyncSync`, `writeSync`, `readSync`, `fdatasyncSync`, `gzipSync`, `gunzipSync`, `brotliCompressSync`, `brotliDecompressSync`, `deflateSync`, `inflateSync`, `deflateRawSync`, `inflateRawSync`, `unzipSync`, `execSync`, `spawnSync`, `execFileSync`, `pbkdf2Sync`, `generateKeyPairSync`, `randomFillSync`, `scryptSync`

have test suites, fewer still have test suites where all tests pass). Concretely, the QL query identified 14k of the original 50k projects containing > 5 synchronous API calls, of which several hundred built and had running tests—from these, we selected projects at random.

5.2 Experimental Design

We ran the call graph builder on each of the projects, choosing the application tests as the entry point for our analysis. In a few cases (`switchBoard` and `flatsheet`), the test code invoked the application code dynamically, which prevented the call graph builder from recognizing application code as being reachable. In these cases, we selected all `.js` files as entry points.

We then ran *Desynchronizer* on each project, applied every transformation that *Desynchronizer* suggested, and reran the application test suite on the transformed application. *Desynchronizer* was configured to log when it abandoned transformations; this, together with the number of synchronous API calls from Table 1, enables us to answer **RQ1**. To answer **RQ2**, instances where *Desynchronizer* introduced failing tests were recorded. We timed the test execution time pre- and post-transformation to help answer **RQ3**. Finally, the entire process was timed, from the beginning of call graph construction to the end of the transformation, to answer **RQ4**.

Experiment Infrastructure. All of our experiments were performed on a server running CentOS Linux release 7.8.2003 (Core), with 2x 32-core processors, running at 2.35GHz. The server is equipped with 128GB of RAM. Timing tests were run on a quiet machine, with no other processes running.

5.3 Experimental Results

Table 2 summarizes the experimental results. The first row of the table reads: it took 222 seconds to build the call graph for the `deepforge` project, and 0.9 seconds to apply the transformation; the call graph is 1,020kB; in total, 14 synchronous API calls supported by *Desynchronizer* were detected in the call graph, 13 were transformed, and 5 additional transformations were applied. The run time of the `deepforge` test suite was 7.6 seconds before and after applying the transformations.

Table 2. Evaluation Results

	Transformation Stats			Sync Calls			Related Fns	Test Run Time	
Project	Build (s)	CG Size (kB)	Trans (s)	Detected	Trans	Trans(%)	Trans.	before	after
`deepforge`	222	1,020	0.9	14	13	93	5	7.6	7.6
`meteor-desktop`	380	1,400	25.1	22	18	82	38	11.6	11.6
`apps`	75	52	0.4	13	8	62	13	10.8	10.5
`switchBoard`	3208	5,400	371.3	65	56	86	28	2.2	2.2
`flatsheet`	236	257	0.5	16	16	100	8	0.5	0.5
`bonescript`	83	160	0.3	11	11	100	6	1.3	1.3
`FiltersCompiler`	114	1,004	15.9	49	22	45	1	19.5	19.4
`a11y-metrics`	73	99	0.2	5	3	60	2	1.1	1.1
`ember-watson`	89	397	0.8	96	96	100	52	1.6	1.6
`adapt_authoring`	1120	1,900	61.4	19	8	42	5	5.8	5.7
`TurboScript`	141	296	0.1	4	3	75	0	50.6	50.8
`useragent`	486	551	0.2	2	2	100	0	0.4	0.4
Total/Average	518.9	1044.6	39.7	316	256	81	158	9.4	9.3

RQ1: How many transformations are proposed by *Desynchronizer*?

Consider Table 2, particularly the columns under **Sync Calls**, which represent the total number of synchronous API calls identified, and the total number that were successfully transformed by *Desynchronizer*. In all, *Desynchronizer* suggested transformations for 256 of 316 synchronous API

calls. To make sense of the landscape of abandoned transformations, we broke down the reasons for abandonment by type. In total, 60 transformations were abandoned: 4 were due to a constructor definition being in the transformation set, 5 were due to functions already returning promises, and the remaining 51 were due to calls to unsupported external functions in the transformation set (calls to `readFile`, `writeFile`, `http.server.timeout`, `Array.reduce` and `Array.filter`, and test related functions such as `describe.each` and `describe.skip`).

Answer to RQ1: *Desynchronizer* identifies the majority of synchronous API calls (256 of 316) as candidates for refactoring.

Table 3. Coverage Results

	Test Coverage				Sync Function coverage					
Project	Statement	Branch	Function	Line	Trans covered	Trans Total	Trans %	Total covered	Total	Total %
deepforge	49.52	73.41	24.60	49.52	13	13	100.0	14	14	100.0
meteor-desktop	58.21	50.56	62.41	58.30	05	08	27.77	6	22	27.27
apps	50.02	42.86	20.83	50.02	06	08	75.00	10	13	76.92
switchBoard	65.15	75.74	52.48	65.15	48	56	85.71	51	65	78.46
flatsheet	00.60	00.40	00.90	00.64	00	16	00.00	00	16	00.00
bonescript	88.85	71.24	66.67	88.85	11	11	100.0	11	11	100.0
FiltersCompiler	90.99	78.44	94.63	90.83	22	22	100.0	47	49	95.91
a11y-metrics	56.06	51.47	28.21	55.90	00	03	00.00	00	05	00.00
ember-watson	93.76	96.08	90.23	93.76	95	96	98.95	95	96	98.95
adapt_authoring	51.82	66.55	55.67	51.82	08	08	100.0	18	19	94.73
TurboScript	49.86	34.93	50.81	50.40	00	03	00.00	01	04	25.00
useragent	92.32	68.87	66.67	92.32	01	02	50.00	01	02	50.00
Total/Average	62.26	59.21	51.17	62.29	209	256	81.64	254	316	80.37

RQ2: How often can the unsoundness of proposed transformations be detected by running application test suites?

Desynchronizer relies on an unsound static call graph analysis, and the refactorings produced by the tool may not preserve program behavior. To determine how often incorrect refactoring suggestions are offered to the user, we applied every suggested transformation to the code, and ran the application's test suite after transformation to observe if test failures occurred.

We found that in 12 cases, *Desynchronizer* suggested transformations that resulted in behavioral changes. In 9 of these cases, we determined that the problem could be attributed directly to unsoundness in the call graph. In 3 cases, the problem related to the test infrastructure (in particular, two cases because `async`/`await` cannot be used in conjunction with the `done` function in the test framework, and one case where a mock function needed updating). While this is promising, we must emphasize that, in general, running an application's tests may not reveal all such incorrect transformations. To provide a fuller picture, we have included the coverage of the application test suites in Table 3, which quantify how much of an application's code is exercised by the test suite—e.g., 80% statement coverage means that 80% of an application's statements are exercised by the tests. On average, we found 62.26% statement coverage, 59.21% branch coverage, 51.17% function coverage, and 62.29% line coverage. Further, we found that 254 (80.37%) of the identified 316 synchronous API calls were covered. Overall, we think this is reasonable coverage.

Answer to RQ2: Only 12 out of 256 transformations resulted in behavioral changes detectable by running an application's tests.

RQ3: What is the impact of refactoring on the performance of applications?

Translation to asynchronous APIs may result in improved application performance by enabling the application to take advantage of the underlying operating system to process multiple I/O requests concurrently. As is suggested in Figure 3, asynchronous APIs outperform their synchronous counterparts if many simultaneous requests are processed, though a slowdown may occur if there is no concurrency. To measure the effect of our transformation on the performance of applications, we collected the run times of the test suites before and after transformation. Table 2 summarizes these results in the **Test Run Time** columns labeled 'before' and 'after'. In each case, the original and transformed code had similar execution times. While the code transformations do not improve test suite run times in a meaningful way, we note that none of these tests exercised the application *at scale* by processing many concurrent requests.

Answer to RQ3: The transformations suggested by *Desynchronizer* do not negatively impact the run time of an application's tests.

RQ4: What is the performance of *Desynchronizer*?

To assess the usability of *Desynchronizer*, we measured how long it took to run it on the applications we selected for our evaluation. The results can again be found in Table 2, under the **Transformation Stats** columns (see the 'Build' and 'Refactoring' subcolumns). The build time includes the time it takes to build the QL database, which is required by QL to execute its queries.

We see that *Desynchronizer* generates the call graph in < 2 minutes for over half of the projects, while in one case the call graph is generated in over 50 minutes. The performance varies highly from one application to another, due to the nature of our call graph building algorithm. `adapt_authoring` is a reasonably large project with 665 files and over 16,000 functions to be analyzed. `switchBoard` on the other hand had poor performance due to the choice of entry point resulting in an exceedingly large number of relevant files.

The time to actually build transformation chains and refactor a project is low on all projects save for `switchBoard`. Longer refactoring times correlate with the larger number of related functions that need to be transformed.

Answer to RQ4: The performance of *Desynchronizer* is mostly determined by the cost of call graph construction, which is reasonable in most cases.

6 THREATS TO VALIDITY

The main threat to validity is the fact that the transformations proposed by our technique may not preserve behavior. Loss of soundness may occur for various reasons, chiefly (i) the call graphs that our technique relies on may be unsound, and (ii) replacing synchronous calls with asynchronous calls may enable arbitrary event handlers to be interleaved when the de-synchronized call is awaited. We view such unsoundness as inevitable, given the highly dynamic nature of JavaScript. We have therefore expressly opted to focus on the development of a technique that is practical, and our evaluation reports that refactorings that are not behavior-preserving are suggested only in a small fraction of all cases.

Beyond concerns about soundness, it is possible that the set of subject applications is not representative of JavaScript projects at large. However, the projects were chosen at random from a pool of thousands of JavaScript projects that had synchronous API calls. This pool was pruned to

include only projects with a test suite in which every test passed. We believe that we have mitigated the risk of bias in our selection process through our random project selection.

To start call graph generation, we manually determined entry points for each application, and in nearly all cases this consisted of the application's tests. We assume that the tests are well written and cover the majority of the application functionality. A sub-optimal test suite could result in the failure of *Desynchronizer* to identify some synchronous API or potential semantics changes caused by the refactoring. This has the potential of negatively impacting our evaluation results, though as seen in Table 2, *Desynchronizer* transforms the majority of synchronous functions, and in Table 3 we see that most of our subject applications have good test coverage.

7 RELATED WORK

The work most closely related to ours can be subdivided into two broad categories. Some work [Gallaba et al. 2017] [Okur et al. 2014a] [Lin et al. 2015] [Lin et al. 2014] is concerned with translating one asynchronous/concurrent programming paradigm to another, while others [Dig et al. 2009a] [Wloka et al. 2009] [Khatchadourian et al. 2019] [Dig et al. 2009b] [Okur et al. 2014a] focus on introducing asynchrony or concurrency in synchronous code. To our knowledge, our work is the first technique in the latter category that is concerned with refactoring synchronous APIs into asynchronous equivalents, and that is concerned with JavaScript.

Gallaba et al. [2017] explored how to refactor older, callback-based JavaScript code to use promises. Where their work is concerned with converting one style of asynchronous programming to another, ours is concerned with converting synchronous code to asynchronous code.

Most previous research on refactoring of asynchronous or concurrent code is concerned with languages such as C#, Java, and Android. Okur et al. [2014b] present *Asyncifier*, a tool that converts callback-based code to use the .NET `async`/`await` constructs. Lin et al. [2014] present Asynchronizer, a tool for extracting long running computations in Android applications using the `AsyncTask` asynchronous programming construct, using a static points-to analysis to establish the safety of transformations. Lin et al. [2015] also present a tool for correcting misuse of old asynchronous programming idioms, and modernizing the code. These approaches focus on converting older asynchronous code to newer asynchronous code, whereas *Desynchronizer* introduces asynchrony where none was present.

Other work on C# includes work by Okur et al. [2014a] on two refactoring tools, *Taskifier* and *Simplifier*, for introducing Task abstractions and transforming them into higher-level design patterns. This is another example of a migration from one asynchronous paradigm to another (from low level `Thread` abstractions to higher level asynchronous design patterns).

Dig et al. [2009b] present *Relooper*, a refactoring tool for converting sequential loops into parallel loops in Java programs. Java introduced a `ParallelArray`, capable of applying a function to each element in parallel, and their tool performs a variety of program analyses to establish the safety of the transformation—both our and their tools present transformations as suggestions to mitigate potential unsoundness.

When Java introduced parallel streams, Khatchadourian et al. [2019] proposed a refactoring tool for migrating between sequential and parallel streams, in a similar spirit to how *Desynchronizer* proposed refactorings from synchronous to asynchronous APIs. Wloka et al. [2009] is concerned with a whole-program transformation to make programs *reentrant*, which crucially allows them to be safely deployed in parallel. While this is a sync-to-async transformation tool like *Desynchronizer*, the use case is quite different, as we introduce asynchrony on a smaller scale.

Dig et al. [2009a] introduce a refactoring tool called *Concurrencer* for introducing data structures from the `java.util.concurrent` library. *Concurrencer* can parallelize arbitrary code, though users

need to explicitly specify shared data for *Concurrencer* to work. In contrast, *Desynchronizer* only refactors synchronous API calls, but generates refactoring suggestions automatically.

As Java's support for concurrent programming matured, tools were needed to help programmers choose the right constructs: Schäfer et al. [2011] present *Relocker*, an automated tool that assists programmers with refactoring synchronized blocks into `ReentrantLocks` and `ReadWriteLocks`, to facilitate the performance tradeoffs associated with different types of locks. Other work by Schäfer et al. [2010] introduces synchronization dependencies that refactoring engines must respect to preserve the correctness of commonly used refactorings in the presence of concurrency.

More abstractly, Song et al. [2018] propose a workflow refactoring to change the layout of regions of code to maximize concurrency and block-structuredness of applications. Zhang et al. [2015] also explore this abstract space, detailing tooling to separate concurrency from the application design. This would allow programmers to decouple parallelism from core application functionality.

8 CONCLUSION

The JavaScript libraries provide synchronous and asynchronous APIs for many commonly used I/O operations. While the asynchronous mechanisms are preferable because they enable better responsiveness and performance in applications, programmers often opt for the synchronous APIs because of their greater ease of use. We have presented a technique that analyzes an application that uses synchronous APIs and automatically infers how it can be refactored to use asynchronous APIs instead. The technique relies on a static call graph analysis that is unsound, so the inferred refactorings are presented as suggestions that should be reviewed by the programmer.

We implemented the refactoring in a tool called *Desynchronizer*, which we evaluated on 12 subject applications containing 316 synchronous API calls. *Desynchronizer* identified 256 of these as candidates for refactoring. Of these candidates, 244 were transformed successfully, and only 12 resulted in behavioral changes. Further inspection of these cases revealed that the majority of these issues can be attributed to unsoundness in the call graph.

ACKNOWLEDGMENTS

This research was supported in part by National Science Foundation grants CCF-1715153 and CCF-190772. The second author was also partially supported by NSERC.

A APPENDIX: CALL GRAPH CONSTRUCTION ALGORITHM

Our call graph construction algorithm performs a data-flow analysis to determine, for each expression e in the program, a set of classes and functions that e may refer to. The algorithm has the following characteristics:

- the algorithm only tracks the data flow of classes and functions (i.e., the flow of primitive values and objects is not tracked)
- the algorithm does not keep track of aliasing, and fields in objects are represented by their name only
- the algorithm operates on an Abstract Syntax Tree $\mathcal{A}$ and constructs a data flow graph $\mathcal{D}$ and a call graph C simultaneously
- The algorithm is specified declaratively, using rules that state when edges in the graph are constructed.

A.1 Nodes

Figure 9 lists the types of nodes that occur in the data flow graph $\mathcal{D} = (\mathcal{V}, \mathcal{E})$ constructed by the algorithm. In the sequel, $a \in \mathcal{A}$ represents an AST Node, π is a location in an AST, i is a number,

$\mathcal{V}$	::=	$\mathbf{Exp}(\pi)$	value of expression at π
	|	$\mathbf{Var}(\pi)$	variable declared at π
	|	$\mathbf{Prop}(n)$	prop with name n
	|	$\mathbf{Fun}(\pi)$	function declaration/expression at π
	|	$\mathbf{Class}(\pi)$	class declaration at π
	|	$\mathbf{Callee}(\pi)$	Callee of call at π
	|	$\mathbf{Arg}(\pi, i)$	i^{th} argument of call at π
	|	$\mathbf{Param}(\pi, i)$	i^{th} parameter of function at π
	|	$\mathbf{Ret}(\pi)$	return value of function at π
	|	$\mathbf{Res}(\pi)$	result of call at π
	|	$\mathbf{Import}(n, v)$	Import of name v into module n
	|	$\mathbf{Export}(n, v)$	Export of name v from module n

Fig. 9. Nodes in the data flow graph $\mathcal{D}$.

$$\mathcal{V}(a^{\pi}) = \begin{cases} \{\ \mathbf{Var}(\pi')\ \} & \text{if } a \equiv v \text{ and } \lambda(\pi, v) = \pi' \\ \{\ \mathbf{Param}(\pi, i)\ \} & \text{if } a \equiv i^{\text{th}} \text{ parameter of function defined at } \pi \\ \{\ \mathbf{Prop}(p)\ \} \cup \{\ \mathbf{Import}(n, p) \,|\, e \equiv v, \mathit{NameSpaceVar}(v, n)\ \} & \text{if } a \equiv e.p \\ \{\ \mathbf{Callee}(\pi)\ \} & \text{if } a \equiv f(), \texttt{new } f() \\ \{\ \mathbf{Fun}(\pi)\ \} & \text{if } a \equiv \texttt{function}\,(\cdots)\{\cdots\} \\ \{\ \mathbf{Exp}(\pi)\ \} & \text{otherwise} \end{cases}$$

Fig. 10. Mapping from AST nodes to nodes in the data flow graph.

$$\frac{e = e'^{\pi}, \mathit{File}(\pi) \in R}{\mathcal{V}(e') \rightarrow \mathcal{V}(e), \mathcal{V}(e') \rightarrow \mathbf{Exp}(\pi)} \quad \text{(Assign)} \qquad \frac{e\ ?\ e' : e''^{\pi}, \mathit{File}(\pi) \in R}{\mathcal{V}(e') \rightarrow \mathbf{Exp}(\pi), \mathcal{V}(e'') \rightarrow \mathbf{Exp}(\pi)} \quad \text{(ConditionalExpr)}$$

$$\frac{e\ \&\&\ e'^{\pi}, \mathit{File}(\pi) \in R}{\mathcal{V}(e') \rightarrow \mathbf{Exp}(\pi)} \quad \text{(LogicalAnd)} \qquad \frac{e\ ||\ e'^{\pi}, \mathit{File}(\pi) \in R}{\mathcal{V}(e) \rightarrow \mathbf{Exp}(\pi), \mathcal{V}(e') \rightarrow \mathbf{Exp}(\pi)} \quad \text{(LogicalOr)}$$

$$\frac{e \equiv (e')^{\pi}, \mathit{File}(\pi) \in R}{\mathcal{V}(e') \rightarrow \mathcal{V}(e)} \quad \text{(ParenthesizedExpr)} \qquad \frac{\{\overline{p:e}\}^{\pi}, \mathit{File}(\pi) \in R}{\mathcal{V}(e_i) \rightarrow \mathbf{Prop}(p_i)} \quad \text{(ObjLiteral)}$$

$$\frac{\texttt{function}\ f(\cdots)\{\cdots\}^{\pi}, \mathit{File}(\pi) \in R}{\mathbf{Fun}(\pi) \rightarrow \mathbf{Exp}(\pi), \text{if it has a name: } \mathbf{Fun}(\pi) \rightarrow \mathbf{Var}(\pi)} \quad \text{(FunctionExpr)}$$

$$\frac{\texttt{function}\ f(\cdots)\{\cdots\}^{\pi}, \mathit{File}(\pi) \in R}{\mathbf{Fun}(\pi) \rightarrow \mathbf{Var}(\pi)} \quad \text{(FunctionDecl)}$$

$$\frac{f(\overline{e})^{\pi} \text{ or } \texttt{new}\ f(\overline{e})^{\pi} \text{ or } r.f(\overline{e})^{\pi}, \mathit{File}(\pi) \in R}{\mathcal{V}(f) \rightarrow \mathbf{Callee}(\pi), \mathcal{V}(e_i) \rightarrow \mathbf{Arg}(\pi, i), \mathbf{Res}(\pi) \rightarrow \mathbf{Exp}(\pi)} \quad \text{(FunctionCall)}$$

$$\frac{r.f(\overline{e})^{\pi}, \mathit{File}(\pi) \in R}{\mathcal{V}(r) \rightarrow \mathbf{Arg}(\pi, 0)} \quad \text{(FunctionCall-Receiver)} \qquad \frac{\texttt{return}\ e^{\pi}, \mathit{File}(\pi) \in R}{\mathcal{V}(e) \rightarrow \mathbf{Ret}(\phi(\pi))} \quad \text{(Return)}$$

$$\frac{\mathbf{Fun}(\pi) \rightarrow^{*} \mathbf{Callee}(\pi'), 0 \leq i \leq \mathit{NrArgs}(\pi')}{\mathbf{Arg}(\pi', i) \rightarrow \mathbf{Param}(\pi, i)} \quad \text{(ArgToParam)} \qquad \frac{\mathbf{Fun}(\pi) \rightarrow^{*} \mathbf{Callee}(\pi')}{\mathbf{Ret}(\pi) \rightarrow \mathbf{Res}(\pi')} \quad \text{(RetToRes)}$$

Fig. 11. Rules for constructing edges that reflect intraprocedural data flow.

v is a name (i.e., a string value), and n is a module (identified by its file name). We will use a^{π} to indicate that AST node a occurs at location π in $\mathcal{A}$.

A.2 Intraprocedural Data Flow

Figure 11 shows rules for constructing edges that capture intraprocedural control flow. The rules rely on an auxiliary function $\mathcal{V}$ (shown in Figure 10) that maps nodes in the Abstract Syntax Tree $\mathcal{A}$ to one or more nodes in the data flow graph $\mathcal{D}$. Here, $\lambda(\pi, v) = \pi'$ denotes the fact that the declaration of a variable v that is referenced at location π occurs at location π'. In other words,

variables have a unique representative whose location is associated with its declaration. In all but one case, $\mathcal{V}$ maps an AST node to a single node in $\mathcal{D}$. The sole exception consists of property-access expressions of the form $e.p$. Each such expression is mapped to a node $\mathbf{Prop}(p)$. However, if the expression e is a variable v that is bound in file n using a namespace import of the form `import * as` v `from` n', the node is additionally mapped to a node $\mathbf{Import}(n, v)$. We will further explain this case in Section A.4.

In the data-flow rules, $\mathit{File}(\pi)$ represents the source file in which location π occurs. Furthermore, e, e' represent expressions, f represents a name of a function (excluding the `require` function), p represents the name of a property, and R represents the set of reachable functions. We will use subscripts and overbar notation to represent sequences, e.g., $\{\overline{p:e}\}$ represents an object literal that initializes property p_0 with expression e_0, property p_1 with expression e_1, and so forth. We will use $\rightarrow^*$ to reflect the transitive closure of $\rightarrow$, and for a function call at location π, $\mathit{NrArgs}(\pi)$ denotes the number of actual parameters passed in the call. Finally, for a `return`-expression at location π, $\psi(\pi)$ will denote the location of the surrounding function.

As an example, consider the rule (Assign):

$$\frac{e = e'^{\pi},\ \mathit{File}(\pi) \in R}{\mathcal{V}(e') \rightarrow \mathcal{V}(e),\ \mathcal{V}(e') \rightarrow \mathbf{Exp}(\pi)}$$

This rule states that, if an assignment $e = e'$ occurs at location π in a reachable file, then edges $\mathcal{V}(e') \rightarrow \mathcal{V}(e)$ and $\mathcal{V}(e') \rightarrow \mathbf{Exp}(\pi)$ are constructed[8].

As another example, consider the rule (FunctionCall):

$$\frac{f(\overline{e})^{\pi} \text{ or } \mathsf{new}\ f(\overline{e})^{\pi} \text{ or } r.f(\overline{e})^{\pi},\ \mathit{File}(\pi) \in R}{\mathcal{V}(f) \rightarrow \mathbf{Callee}(\pi),\ \mathcal{V}(e_i) \rightarrow \mathbf{Arg}(\pi, i),\ \mathbf{Res}(\pi) \rightarrow \mathbf{Exp}(\pi)}$$

This rule states that, if a function call $f(\overline{e})$ or $\mathsf{new}\ f(\overline{e})$ or $r.f(\overline{e})$ occurs at a location π that occurs in a reachable file, then an edge $\mathcal{V}(f) \rightarrow \mathbf{Callee}(\pi)$ is constructed, reflecting the fact that any function that may flow to data flow node representing f will flow to $\mathbf{Callee}(\pi)$. Furthermore, the rule constructs edges $\mathcal{V}(e_i) \rightarrow \mathbf{Arg}(\pi, i)$, connecting the data flow node constructed for each argument e_i to $\mathbf{Arg}(\pi, i)$. Finally, the rule constructs an edge $\mathbf{Res}(\pi) \rightarrow \mathbf{Exp}(\pi)$, reflecting data flow from the node $\mathbf{Res}(\pi)$ that represents the value returned by the function to $\mathbf{Exp}(\pi)$, the node that represents the entire call-expression.

As a final example, we consider the rule (ArgToParam):

$$\frac{\mathbf{Fun}(\pi) \rightarrow^* \mathbf{Callee}(\pi'),\ 0 \leq i \leq \mathit{NrArgs}(\pi')}{\mathbf{Arg}(\pi', i) \rightarrow \mathbf{Param}(\pi, i)}$$

This rule states that, if transitive data flow exists from a node $\mathbf{Fun}(\pi)$ to a node $\mathbf{Callee}(\pi')$, then edges $\mathbf{Arg}(\pi', i) \rightarrow \mathbf{Param}(\pi, i)$ are constructed to reflect the flow from actual parameters to corresponding formal parameters.

A.3 Rules for Exports

Figure 12 shows the rules for the various cases of exporting features from ECMAScript Modules. For example, Rule (NamedExport-Var) has the following form:

$$\frac{\begin{array}{c}\texttt{export let } v = \cdots^{\pi} \textit{ or} \\ \texttt{export const } v = \cdots^{\pi} \textit{ or} \\ \texttt{export var } v = \cdots^{\pi},\ n = \mathit{File}(\pi) \in R\end{array}}{\mathbf{Var}(\pi) \rightarrow \mathbf{Export}(n, v)}$$

and handles the cases where a named export is declared by having the declaration of a variable v be preceded by the keyword `export`. In such cases, an edge $\mathbf{Var}(\pi) \rightarrow \mathbf{Export}(n, v)$ is constructed. The rules (NamedExport-Fun) and (NamedExport-Class), are similar and handle cases where

[8]Here and in other rules, we slightly abuse notation by pretending that $\mathcal{V}$ maps an AST node to a single value. More precisely, the rule constructs edges $d' \rightarrow d$ and $d' \rightarrow \mathbf{Exp}(\pi)$ where $d \in \mathcal{V}(e)$ and $d' \in \mathcal{V}(e')$.

$$\frac{\texttt{export let } v = \cdots^{\pi} \textit{ or } \texttt{export const } v = \cdots^{\pi} \textit{ or } \texttt{export var } v = \cdots^{\pi}, n = \mathit{File}(\pi) \in R}{\mathbf{Var}(\pi) \rightarrow \mathbf{Export}(n, v)} \text{(NamedExport-Var)}$$

$$\frac{\texttt{export function } f(\cdots)^{\pi}, \quad n = \mathit{File}(\pi) \in R}{\mathbf{Var}(\pi) \rightarrow \mathbf{Export}(n, f)} \text{(NamedExport-Fun)}$$

$$\frac{\texttt{export class } C \texttt{ \{ } \cdots \texttt{ \}}^{\pi}, n = \mathit{File}(\pi) \in R}{\mathbf{Var}(C) \rightarrow \mathbf{Export}(n, C)} \text{(NamedExport-Class)}$$

$$\frac{\texttt{export \{ } f \texttt{ \}}^{\pi}, n = \mathit{File}(\pi) \in R}{\mathbf{Var}(\pi) \rightarrow \mathbf{Export}(n, f)} \text{(NamedExport-Spec)} \quad \frac{\texttt{export \{ } v \texttt{ as } v' \texttt{ \}}^{\pi}, \quad v' \neq \texttt{default}, n = \mathit{File}(\pi) \in R}{\mathbf{Var}(\pi) \rightarrow \mathbf{Export}(n, v')} \text{(NamedExport-Spec-Rename)}$$

$$\frac{\texttt{export default } e^{\pi}, \quad n = \mathit{File}(\pi) \in R}{V(e) \rightarrow \mathbf{Export}(n, \texttt{default})} \text{(DefaultExport-Exp)} \quad \frac{\texttt{export default function } f^{\pi}(\cdots), \quad n = \mathit{File}(\pi) \in R}{\mathbf{Var}(\pi) \rightarrow \mathbf{Export}(n, \texttt{default})} \text{(DefaultExport-Fun)}$$

$$\frac{\texttt{export default class } C \texttt{ \{ } \cdots \texttt{\}}^{\pi}, \quad n = \mathit{File}(\pi) \in R}{\mathbf{Var}(\pi) \rightarrow \mathbf{Export}(n, \texttt{default})} \text{(DefaultExport-Class)} \quad \frac{\texttt{export \{ } v^{\pi} \texttt{ as default \}}\ \pi', \quad n = \mathit{File}(\pi) \in R}{V(\pi) \rightarrow \mathbf{Export}(n, \texttt{default})} \text{(DefaultExport-Spec)}$$

$$\frac{\texttt{export \{ } v \texttt{ as } v' \texttt{ \} from } n^{\pi} \quad n' = \mathit{File}(\pi) \in R}{\mathbf{Export}(n, v) \rightarrow \mathbf{Export}(n', v')} \text{(SelectiveReExport)} \quad \frac{\texttt{export \{ default as } v \texttt{ \} from } n^{\pi} \quad n' = \mathit{File}(\pi) \in R}{\mathbf{Export}(n, \texttt{default}) \rightarrow \mathbf{Export}(n', v)} \text{(ReExportDefault)}$$

$$\frac{\texttt{export * from } n^{\pi}, \mathbf{Export}(n, f) \text{ exists} \quad n' = \mathit{File}(\pi) \in R}{\mathbf{Export}(n, f) \rightarrow \mathbf{Export}(n', f)} \text{(BulkReExport)}$$

Fig. 12. Rules for constructing edges for exports.

classes and functions are exported by name, and (NamedExport-Spec) handles the cases where named exports are specified in an *export specifier*.

The next set of rules is concerned with *default exports*. For example, Rule (DefaultExport-Fun) takes the following form:

$$\frac{\texttt{export default function } f^{\pi}(\cdots), \quad n = \mathit{File}(\pi) \in R}{\mathbf{Var}(\pi) \rightarrow \mathbf{Export}(n, \texttt{default})}$$

and states that for an export of the form `export default function(){ ... }` at location π, an edge $\mathbf{Var}(\pi) \rightarrow \mathbf{Export}(n, \texttt{default})$ is created. Similarly, Rules (DefaultExport-Exp) and (DefaultExport-Class) handle the case where an expression or a class is designated as the default, respectively. Rule (DefaultExport-Spec) handles the case where a default export is specified as part of an *export specifier*.

The last three rules are concerned with situation where a module re-exports features that it imports from other modules. These rules are similar to those discussed above.

A.4 Rules for Imports

Figure 13 shows the rules for the various cases of importing features into ECMAScript Modules.

For example, Rule (NamedImport) handles the case where a specific named export is imported. The rule states that if a module contains a code fragment `import { v } from n`, where n is the name of a module a where π is the location of v, then an edge $\mathbf{Import}(\pi, v) \rightarrow \mathbf{Var}(\pi)$ is constructed. Rule (NamedImport-Rename) handles the similar case where the variable is renamed upon import. Rule (ImportDefault) handles situations where a module's default export is imported. The rule states that, if module n's default export is imported and bound to variable v, then an edge $\mathbf{Import}(n, \texttt{default}) \rightarrow \mathbf{Var}(v)$ is constructed. Rule (BulkImport) handles the case where the imported module's named exports are bound to the properties of a "namespace object". Here,

$$\frac{\texttt{import \{ } v^{\pi} \texttt{ \} from } n^{\pi'},\ File(\pi') \in R}{\mathbf{Export}(n, v) \rightarrow \mathbf{Import}(\pi, v), \mathbf{Import}(\pi, v) \rightarrow \mathbf{Var}(\pi), n \in R} \quad \text{(NamedImport)}$$

$$\frac{\texttt{import \{ } v \texttt{ as } w^{\pi} \texttt{ \} from } n^{\pi'},\ File(\pi') \in R}{\mathbf{Export}(n, v) \rightarrow \mathbf{Import}(\pi, w), \mathbf{Import}(\pi, w) \rightarrow \mathbf{Var}(\pi), n \in R} \quad \text{(NamedImport-Rename)}$$

$$\frac{\texttt{import * as } v \texttt{ from } n^{\pi}, File(\pi) \in R \quad \text{module } n \text{ has export with name } w,}{\mathbf{Export}(n, w) \rightarrow \mathbf{Import}(\pi, w)\ n \in R} \quad \text{(BulkImport)}$$

$$\frac{\texttt{import } v \texttt{ from } n^{\pi}, File(\pi) \in R}{\mathbf{Import}(n, \texttt{default}) \rightarrow \mathbf{Var}(v), n \in R} \quad \text{(ImportDefault)}$$

Fig. 13. Rules for constructing edges for imports.

$$\frac{\texttt{class } C \texttt{ \{ ... \}}^{\pi}, File(\pi) \in R}{\mathbf{Class}(\pi) \rightarrow \mathbf{Var}(\pi)} \quad \text{(ClassDecl)}$$

$$\frac{\texttt{new } e^{\pi}, \texttt{class } C \texttt{ \{ ... constructor (...)\{ ... \}}^{\pi'} \texttt{ ... \}}, \mathbf{Class}(C) \rightarrow^{*} \mathbf{Callee}(\pi), File(\pi) \in R}{\mathbf{Fun}(\pi') \rightarrow \mathbf{Callee}(\pi)} \quad \text{(CtorCall)}$$

$$\frac{\texttt{class } B \texttt{ \{ ... constructor (...)\{ ... \}}^{\pi'} \texttt{ ... \}},\ \texttt{class } C \texttt{ \{ ... constructor (...)\{ super (...)}^{\pi} \texttt{ \} ... \}}, File(\pi) \in R}{\mathbf{Fun}(\pi') \rightarrow \mathbf{Callee}(\pi)} \quad \text{(SuperCall)}$$

Fig. 14. Rules for constructing edges for classes.

we rely on the fact that a reference to a variable v will be mapped to an *Import* node by the $\mathcal{V}$ function if it was bound in a namespace import.

Note that each of the rules in Figure 13 adds the imported module's file to the set of reachable files, R.

A.5 Rules for Classes

Figure 14 shows the rules that show the generation of edges for features related to classes.

Rule (ClassDecl) handles class declarations:

$$\frac{\texttt{class } C \texttt{ \{ } \cdots \texttt{ \}}^{\pi}, File(\pi) \in R}{\mathbf{Class}(\pi) \rightarrow \mathbf{Var}(\pi)}$$

and states that, if a class declaration occurs at a location π, then an edge $\mathbf{Class}(\pi) \rightarrow \mathbf{Var}(\pi)$ is constructed.

Rule (CtorCall) handles constructor calls:

$$\frac{\texttt{new } e^{\pi}, \texttt{class C \{ } \cdots \texttt{ constructor } (\cdots)\texttt{\{ } \cdots \texttt{ \}}^{\pi'} \cdots \texttt{ \}},\ \mathbf{Class}(C) \rightarrow^{*} \mathbf{Callee}(\pi), File(\pi) \in R}{\mathbf{Fun}(\pi') \rightarrow \mathbf{Callee}(\pi)}$$

The rule states that if a constructor call `new` e occurs at a location π, and there exists transitive data flow $\mathbf{Class}(C) \rightarrow^{*} \mathbf{Callee}(\pi)$ and class C has a constructor at location π', then an edge $\mathbf{Fun}(\pi') \rightarrow \mathbf{Callee}(\pi)$ is constructed. Rule (SuperCall) handles super-calls that occur in a constructor.

A.6 Implementation

We have implemented this analysis in QL [Avgustinov et al. 2016]. Our implementation follows the presented rules closely and handles a number of other features, including arrow functions, getters/setters, function calls using `call`, `apply`, and `bind`, and imports using the CommonJS `require` function. Our analysis also incorporates support for a range of methods in JavaScript's standard libraries, for which we rely on QL's existing externs definitions. An open-source release of the analysis can be found as part of the artifact associated wit this paper, which was submitted for artifact evaluation.

REFERENCES

2020. ECMAScript 2020 Language Specification. https://www.ecma-international.org/ecma-262/11.0/.

2021. Express Web Framework for JavaScript. http://expressjs.com/.

2021. Jest: A delightful JavaScript Testing Framework. https://jestjs.io/.

2021. Mocha - the fun, simple, flexible JavaScript test framework. https://mochajs.org/.

Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented Queries on Relational Data. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*. 2:1–2:25. https://doi.org/10.4230/LIPIcs.ECOOP.2016.2

BabelJS. 2021. Babel Parser Documentation. https://babeljs.io/docs/en/babel-parser. Accessed 2021-03-20.

Danny Dig, John Marrero, and Michael D. Ernst. 2009a. Refactoring sequential Java code for concurrency via concurrent libraries. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. 397–407. https://doi.org/10.1109/ICSE.2009.5070539

Danny Dig, Mihai Tarce, Cosmin Radoi, Marius Minea, and Ralph E. Johnson. 2009b. Relooper: refactoring for loop parallelism in Java. In *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*. 793–794. https://doi.org/10.1145/1639950.1640018

Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2013. Efficient construction of approximate call graphs for JavaScript IDE services. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. 752–761. https://doi.org/10.1109/ICSE.2013.6606621

Keheliya Gallaba, Quinn Hanam, Ali Mesbah, and Ivan Beschastnikh. 2017. Refactoring Asynchrony in JavaScript. In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*. IEEE Computer Society, 353–363. https://doi.org/10.1109/ICSME.2017.83

Raffi Khatchadourian, Yiming Tang, Mehdi Bagherzadeh, and Syed Ahmed. 2019. Safe automated refactoring for intelligent parallelization of Java 8 streams. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 619–630. https://doi.org/10.1109/ICSE.2019.00072

Yu Lin, Semih Okur, and Danny Dig. 2015. Study and Refactoring of Android Asynchronous Programming (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. 224–235. https://doi.org/10.1109/ASE.2015.50

Yu Lin, Cosmin Radoi, and Danny Dig. 2014. Retrofitting concurrency for Android applications through refactoring. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*. 341–352. https://doi.org/10.1145/2635868.2635903

Magnus Madsen, Ondrej Lhoták, and Frank Tip. 2017. A model for reasoning about JavaScript promises. *PACMPL* 1, OOPSLA (2017), 86:1–86:24. https://doi.org/10.1145/3133910

Magnus Madsen, Frank Tip, and Ondrej Lhoták. 2015. Static analysis of event-driven Node.js JavaScript applications. In *Proc. of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. 505–519. https://doi.org/10.1145/2814270.2814272

Semih Okur, Cansu Erdogan, and Danny Dig. 2014a. Converting Parallel Code from Low-Level Abstractions to Higher-Level Abstractions. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*. 515–540. https://doi.org/10.1007/978-3-662-44202-9_21

Semih Okur, David L. Hartveld, Danny Dig, and Arie van Deursen. 2014b. A study and toolkit for asynchronous programming in C#. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. 1117–1127. https://doi.org/10.1145/2568225.2568309

OpenJS Foundation. 2021. Node.js. https://nodejs.org/en/.

Max Schäfer, Julian Dolby, Manu Sridharan, Emina Torlak, and Frank Tip. 2010. Correct Refactoring of Concurrent Java Code. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010.*

Proceedings. 225–249. https://doi.org/10.1007/978-3-642-14107-2_11

Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2011. Refactoring Java programs for flexible locking. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic (Eds.). ACM, 71–80. https://doi.org/10.1145/1985793.1985804

W. Song, H. Jacobsen, S. C. Cheung, H. Liu, and X. Ma. 2018. Workflow Refactoring for Maximizing Concurrency and Block-Structuredness. *IEEE Transactions on Services Computing* (2018), 1–1.

Jan Wloka, Manu Sridharan, and Frank Tip. 2009. Refactoring for reentrancy. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*. 173–182. https://doi.org/10.1145/1595696.1595723

Yang Zhang, Dongwen Zhang, Weixing Ji, and Yizhuo Wang. 2015. Refactoring for Separation of Concurrent Concerns. In *Algorithms and Architectures for Parallel Processing*, Guojun Wang, Albert Zomaya, Gregorio Martinez, and Kenli Li (Eds.). Springer International Publishing, Cham, 105–118.