Refactoring Asynchrony in JavaScript

Keheliya Gallaba*, Quinn Hanam†, Ali Mesbah†, Ivan Beschastnikh†
*McGill University, Canada
†University of British Columbia, Canada
keheliya.gallaba@mail.mcgill.ca, [qhanam, amesbah]@ece.ubc.ca, bestchai@cs.ubc.ca

***Abstract*—JavaScript is a widely used programming language that makes extensive use of asynchronous computation, particularly in the form of *asynchronous callbacks*. These callbacks are used to handle tasks, from GUI events to network messages, in a non-blocking fashion.**

Asynchronous callbacks present developers with two challenges. First, JavaScript's try/catch error-handling mechanism is not sufficient for proper error handling in asynchronous contexts. In response, the JavaScript community has come to rely on the *error-first protocol*, an informal programming idiom that is not enforced or checked by the runtime. Second, JavaScript callbacks are frequently nested, making them difficult to handle (also known as callback hell). Fortunately, a recent language extension called *promises* provides an alternative to asynchronous callbacks. The adoption of promises, however, has been slow as refactoring existing code to use promises is a complex task.

We present a set of program analysis techniques to detect instances of asynchronous callbacks and to refactor such callbacks, including callbacks with the error-first protocol, into promises. We implement our techniques in a tool called PROMISESLAND. We perform a manual analysis of four JavaScript applications to evaluate the tool's precision and recall, which are, on average, 100% and 83%, respectively. We evaluate PROMISESLAND on 21 large JavaScript applications, and find that PROMISESLAND (1) correctly refactors callbacks to promises, (2) outperforms a recent related refactoring technique, and (3) runs in under three seconds on all of our evaluation targets.

I. INTRODUCTION

Modern web applications make extensive use of JavaScript, which is one of the most widely used programming languages in the world[1]. Unlike many modern languages, programs written in JavaScript are single-threaded. This means that JavaScript programs cannot create additional threads to execute long running tasks such as IO operations. To overcome this limitation, developers use JavaScript APIs that accept a callback parameter [19]. Such API functions begin a task, such as an HTTP request, in another process initiated by the JavaScript engine and immediately return control to the callee. When the task completes, the JavaScript engine executes the callback function (asynchronously) to notify the program and to pass the program any data generated by the task. Because the API function returns control immediately, the task does not block the rest of the program. However, this model of invoking a callback asynchronously complicates control-flow and degrades program comprehension [8], [43]. More specifically, JavaScript lacks proper mechanisms to manage error handling in asynchronous callbacks and to maintain nested asynchronous callbacks.

In JavaScript, an error that occurs during the execution of an asynchronous task cannot be handled with the traditional try/catch mechanism because the asynchronous task is run outside the existing call stack. For example, in Listing 1 an exception generated during setTimeout will **not** be caught inside the catch block. Therefore, an error generated by an asynchronous function, such as setTimeout, can only be handled by passing it as a parameter to the callback function.

```
1  try {
2    setTimeout(function() {
3                  throw new Error('Uh oh!');
4                }, 2000);
5  } catch (e) {
6    console.log('Caught the error: ' + e.message);
7  }
```

Listing 1. A JavaScript snippet illustrating that try/catch statements are ineffective for handling errors in asynchronous callbacks.

The JavaScript community has come up with a convention for error propagation in asynchronous contexts called the *error-first protocol*. In this idiom-based protocol, the first parameter of the callback is reserved for communicating errors and the other parameters are used for passing data. Listing 2 illustrates this protocol: an error generated by the asynchronous function fs.readFile is passed to the anonymous function callback as the argument err and the callback includes appropriate error-handling code. A key limitation of the error-first protocol is that it is a convention and developers are not obligated to use it. As a result, developers manually check the function arguments to see whether a function follows the protocol, making it an error-prone practice.

```
1  fs.readFile('/foo.txt', function(err, result) {
2    if (err) {
3      console.log('Unknown Error');
4      return;
5    }
6    console.log(result);
7  });
```

Listing 2. A JavaScript snippet illustrating the error-first protocol.

Another challenge developers face when using asynchronous callbacks is callback nesting. A callback function that contains calls to other asynchronous functions creates what is known as "callback hell" [35]. This problem is exacerbated with anonymous functions, which are declared

[1]According to the 2017 survey of 64K developers by Stack Overflow, for the fifth year in a row, JavaScript is the most commonly used programming language [40].

within other functions and obfuscate the control flow within their parent function. A recent empirical study has found that asynchronous nested callbacks are prevalent in practice [19].

Promises is a new feature of ECMAScript6 [1] designed to help with the error handling and nesting problems associated with asynchronous callbacks. Promises explicitly register handlers for executions that are successful and executions that produce errors. This removes the need for the error-first convention. We carried out an exploratory study in which we explored the extent to which developers refactor callbacks to promises and how they perform such refactorings (detailed in Section IV). We found no mention or use of refactoring tools: it seems that today developers manually refactor asynchronous callbacks into promises.

In this paper, we propose a set of static analysis techniques to support automated refactoring of asynchrony in JavaScript by: (1) discovering instances of asynchronous callbacks, and (2) transforming these instances into promises.

We implemented these techniques in an open source tool called PROMISESLAND [6] and evaluated it by following three research questions:

RQ1: Can PROMISESLAND accurately identify instances of asynchronous callbacks to be converted?

RQ2: Can PROMISESLAND correctly refactor asynchronous callbacks to promises?

RQ3: Is PROMISESLAND efficient?

For the evaluation, we used 21 open source JavaScript projects containing a total of 108,615 lines of code. PROMISESLAND ran in under three seconds on all of the projects we evaluated. We reason about the correctness of PROMISESLAND given a sound and complete points-to graph and assume correct use of the error-first protocol. While no static method for building a points-to graph for JavaScript code is both sound and complete, we provide evidence that available approximations are sufficiently accurate for our analysis in most cases.

In an evaluation of recall, PROMISESLAND automatically discovers 39 of 47 (89%) asynchronous callbacks as candidates for automated refactoring. In an evaluation of precision on 188 asynchronous callbacks from 56 subjects, test suites for these projects – which execute the 188 asynchronous callbacks at least once – pass before and after PROMISESLAND performs the refactoring. This suggests that the refactoring was performed correctly in all cases.

II. RELATED WORK

Refactoring support for asynchronous programming in Android apps was proposed by Dig et al. [14], [28]. However, they target the Java programming language.

JavaScript is a challenging language for software engineering and recent research advances have made the use of static analysis on JavaScript more practical [9], [17], [26], [27], [29], [33], [34], [39]. Other techniques mitigate the analysis challenges by using a dynamic or hybrid approach [7], [21], [31], [44]. Others have considered to improve the core language through abstraction layers [42].

We performed an empirical study [19] on the use of callbacks in 138 JavaScript programs. The findings of that study motivate the need for tool support in managing callbacks. Three relevant findings are (1) more than half of all callbacks are asynchronous (both in client-side and server-side code), (2) on average every 5th function adheres to the error-first protocol (i.e., developers are inconsistent in their use of this idiom), and (3) 27% of the studied programs used promises (indicating that tool support can increase the adoption of promises).

The closest prior work is by Brodu et al. [12] who propose a compiler for converting nested callbacks into a sequence of Dues, which is a simpler version of promises. There are several drawbacks to this approach: (1) the source code does not change, so it does not eliminate the issues with understandability, (2) Dues do not support the critical notions of rejection and resolution in promises and can therefore only re-write the error-first protocol in a simplified notation, (3) their approach requires developers to manually specify asynchronous callbacks that are suitable candidates to be converted. In contrast PROMISESLAND is completely automated. Specifically, when evaluating PROMISESLAND on projects evaluated in [12], we found that our technique is able to refactor 235% more asynchronous callbacks than the tool proposed in [12].

The third-party library Bluebird [10], provides functions to wrap callbacks as promises. This library does not automatically refactor code; developers must locate callback functions and correctly perform the refactoring. PROMISESLAND automatically locates and refactors callbacks to promises.

A number of other JavaScript refactoring tools have been previously proposed. For example, Meawad et al. [30] proposed a tool to refactor `eval` statements into safer code. Feldthaus et al. [15], [16] developed a technique for semi-automatic refactoring with a focus on renaming. And, several projects consider ways to detect and refactor legacy JavaScript code to use classes, which are part of the ES6 standard [36], [37]. This existing work does not focus on detecting or refactoring of asynchronous callbacks.

III. BACKGROUND

In JavaScript, an *asynchronous function* is used to schedule a long-running task, so that the execution of the program can continue in a non-blocking manner. Asynchronous functions initiate or schedule a task to be completed in the future, but return the control back to the caller before the task completes. Callers of asynchronous functions must often specify a continuation function that (1) acts as the point where control is returned to the program after the asynchronous task completes, and (2) accepts data or errors generated by the task [12].

A. Drawbacks of callbacks

A *callback* is a parameter that is used as a function. JavaScript developers frequently use callbacks to specify continuations for asynchronous task [12]. However, using callbacks as continuations has three drawbacks.

Callback nesting. When asynchronous functions must be completed sequentially, callbacks must be nested. Because

each callback adds a new function definition and indentation level, this decreases the understandability of the program and is known as "callback hell" [35]. Listing 3 illustrates an example with three levels of callback nesting.

```
getUser('jackson', function(error, user) {
  if (error) {
    handleError(error);
  } else {
    getNewTweets(user, function(error, tweets) {
      if (error) {
        handleError(error);
      } else {
        updateTimeline(tweets, function(error) {
          if (error) handleError(error);
        });
      }
    });
  }
});
```

Listing 3. A sequence of asynchronous operations.

Callback nesting in JavaScript is pervasive and is widely considered a barrier to software comprehension. A study on callbacks in JavaScript found that most callbacks are nested to two levels and that nesting can be as deep as 8 levels [19].

Error handling. Because asynchronous tasks are executed outside the call stack, errors generated during the execution of asynchronous tasks must be propagated to callbacks as arguments. Consider Listing 3: the two callbacks accept an error parameter, which they must check to determine if an error was raised in the asynchronous function. This style is known as the *error-first protocol* and produces code which is less understandable because (1) it is inconsistent - there is no standard way for checking errors and (2) it obfuscates the control flow by adding extra branches to the callback function.

Synchronization. Callbacks do not have built-in synchronization and nothing prevents a callback from being invoked multiple times. When such multiple-invocation behavior is undesirable, it may lead to bugs. Listing 4 illustrates an example in which the callback cb is invoked twice when foo evaluates to true. Client code that provides a cb to this code must use expensive defensive programming techniques to catch and enforce a proper number of cb invocations.

```
handler(cb, foo) {
  if(foo) cb(foo);
  cb(foo);
}
```

Listing 4. Callbacks are vulnerable to synchronization bugs.

B. A promising solution

A *promise* is a design pattern that handles asynchronous events and solves many of the callback-related problems described previously. While promises have been used for some time in JavaScript with third party libraries, the next ECMA specification (version 6) [2] of the language has promises built in. With the promises design pattern, instead of accepting a callback as the continuation function, an asynchronous function returns a Promise instance. This instance represents a value that will be available sometime in the future, for example, after a deferred task has completed.

```
getUser('jackson')
  .then(getNewTweets, handleError)
  .then(updateTimeline, handleError);
```

Listing 5. A sequence of async operations composed with promises.

Promises solve many of the problems associated with callbacks. Consider Listing 5, which is semantically equivalent to Listing 3, but uses promises. Callback nesting is eliminated by chaining promises – in Listing 3 the promise for the second function call getNewTweets is chained to the promise returned by getUser. Error handling is now explicit. The success handler (the first parameter of .then) and an error handler (the second parameter of .then) are specified separately without additional control flow – In Listing 3 getNewTweets and updateTimeline are success handlers, while handleError is the error handler for both promises. Basic synchronization is now handled automatically because promises guarantee that the error and success handlers only execute once.

IV. Refactoring to Promises in Practice

We carried out an exploratory study to better understand the extent and manner in which developers refactor callbacks into promises. Our exploratory study consists of three parts: (1) a manual inspection of issues on GitHub related to promises, (2) a manual inspection of pull requests on GitHub related to promises, and (3) an automated mining of commits that refactored asynchronous callbacks to promises.

A. Exploring issues on refactoring

The first part of our study explored posts in GitHub's issue tracking system. GitHub is one of the most popular collaborative software-development platforms among JavaScript developers [18] and provides the largest publicly available dataset including developer discussions and development history. We used GitHub's search feature to find issues related to refactoring of asynchronous callbacks to promises.

We used the query "promise callback language:JavaScript stars:>30 comments:>5 type:issue" to search for GitHub issue discussions that were non-trivial (containing at least 5 comments), associated with projects that were popular (starred by at least 30 users) and contained the terms promise and callback. This search resulted in 4,342 issues. We considered the first 11 issues (on the first results page) and manually inspected the discussions associated with each issue. We found that in the majority of issues (8 out of 11), the final consensus was to refactor the code to use promises instead of using asynchronous callbacks. Many discussion participants agreed that using promises would be beneficial to the project: *"I'm very pleased with the amount of additional safety and expressiveness I've gained by using promises. I'm not dismissing callbacks per se, but personally I find it much simpler to reason*

about code using promises than code using callbacks & utility libraries like async."[2]

However, the main reasons for reluctance to migrate to promises were (1) promises may cause significant changes to existing APIs, and (2) the prohibitively high development effort associated with the change.

We then narrowed down the search by including the term refactor.[3] This resulted in 351 issues, of which we inspected the first 80. Many of these issues indicated strong demand to refactor code to use promises. For example, one participant stated: *"So this is something that is purely for devs but I think it is about time to do this. i.e. git-task is a great candidate to take full advantage of promises and it would have made implementation of #602 much easier."* [4]

Many of these requests came from users of JavaScript libraries who wanted promises as part of the library API: *"Are there any plans for promise support, alongside the callbacks and streams? Proper promise support in any-db and any-db-transaction would be really nice :)"*[5], and *"Add promise API option?"*. [6] Some of the users encouraged the move to promises by sharing their own experiences of using promises: *"We've recently converted pretty large internal codebases from async.js to promises and the code became smaller, more declarative, and cleaner at the same time."* [7]

Our study of GitHub issues indicates a strong desire from developers to have this refactoring performed on the systems they use and maintain. Developers also noted that development time is a factor in avoiding migration, which motivates the need for a tool to simplify this process by both finding refactoring candidates and by automating the transformation.

B. Exploring refactoring pull requests

The second part of our study explored pull requests on GitHub to determine whether developers acted on suggestions for refactoring asynchronous callbacks to promises. We did this by searching GitHub for pull requests associated with refactoring asynchronous callbacks to promises and manually inspecting the results. Our search used the following query string: "Refactor promises language:Javascript stars:>20 type:pr". The search resulted in 451 pull requests, of which we inspected the first 80. We observed that most of these pull requests were submitted as improvements to the project and involved replacing callbacks with promises. These were either native promises supported by the runtime or ones provided by third-party libraries like *Bluebird, Q*, or *RSVP*. We found that developers generally prefer promises, and also refactor asynchronous callbacks into promises in practice. A more detailed listing of the discussions we explored in our study, along with listings of relevant quotes, can be found in our online study site [4].

[2] https://github.com/share/ShareJS/issues/268

[3] Complete query: "Refactor promises language:JavaScript stars:>20 type:issue". We lowered the number of stars to capture more projects.

[4] https://github.com/FredrikNoren/ungit/issues/603

[5] https://github.com/grncdr/node-any-db/issues/66

[6] https://github.com/addyosmani/psi/issues/56

[7] https://github.com/meetfinch/decking/issues/18

C. Mining commits for refactorings

In the third part of our study, we used BugAid [23], a commit mining tool, to search for examples of asynchronous callback to promise refactoring in practice. For each commit in a project's history, BugAid inspects changes at the AST level. We implemented an AST pattern-matching algorithm on top of BugAid that searches for calls to promise constructors (i.e., new Promise(...)) that are inserted into existing functions. This pattern is conservative but mainly corresponds to asynchronous callbacks being replaced with promises.

We mined all the 134 subject systems studied by Hanam et al. [23]. These were selected based on curated lists of popular Node.js applications and the top NPM modules by stars and number of packages that depend on them. We discovered 39 valid instances of asynchronous callback to promise refactorings across nine projects. This further indicates that developers are interested in performing this refactoring in practice. We manually inspected each of these instances and found that five of them conform to the standard refactoring pattern matching recommendations in developer blog posts [13].

D. Findings and goal

Our exploratory study demonstrates that developers see many advantages in migrating to promises. However, because of the complex control flow associated with asynchronous callbacks, manually refactoring callbacks to promises can be difficult. These results point to a need for tooling which can automatically perform these refactorings.

Our goal is to develop a technique that can automatically refactor asynchronous callbacks to promises. The approach will (1) automatically detect candidate functions for refactoring and (2) automatically refactor these functions, and their callsites, to use promises.

V. Approach

In this section, we describe our techniques for identifying refactoring candidates and transforming them to use promises.

A. Assumptions and preliminaries

It is important to note that when describing the correctness of our approach, we make two assumptions. The first assumption is that we have sound and complete points-to analysis information. In theory this is not possible, but in practice, approximate solutions are often good enough [17], [29], [39]. The second assumption is that an error parameter name matches the regular expression $e|err|error$ iff it is part of the error-first protocol. We base this assumption on our empirical experience with many large JavaScript codebases. We demonstrate that our method is correct under these assumptions, and in our empirical evaluation (Section VII) we show that these assumptions are reasonable in most cases.

Throughout, we use $async$ to denote an asynchronous, built-in, JavaScript API function, such as process.nextTick or fs.readFile; we use *cb* to denote a callback, or a function that is passed as an argument to other functions. For example, Listing 6 gives an abstract example of function f that uses an

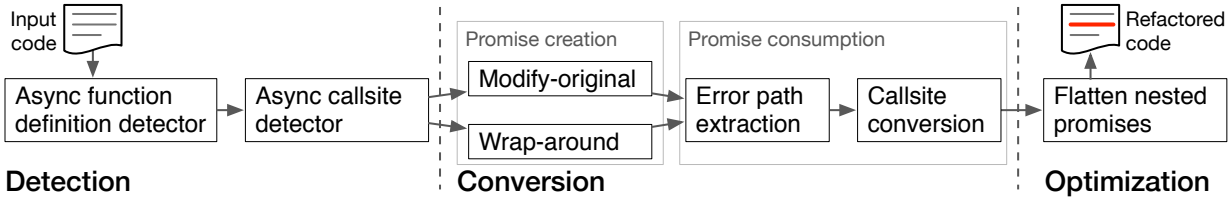

Fig. 1. Overview of our approach.

```
1  function f(cb_f) {
2    async(function cb_async(error, data) {
3              if(error) cb_f(error, null);
4              else cb_f(null, data);
5    });
6  }

8  function cb_f(error, data) {
9    if(error) {
10     // Handle error
11   }
12   else {
13     // Handle data
14   }
15 }
16 f(cb_f);
```

Listing 6. Abstract functions and callsites in the original program P.

asynchronous callback cb_f; the last line in the listing contains a callsite to f.

Given a program P, we transform it into P' by transforming sub-elements of P. Figure 1 illustrates this process. In Section V-B we describe a process to automatically detect instances of functions f in P that can be refactored using our method. In Section V-C we describe a process to derive a new asynchronous function f' that returns a promise from f. In Section V-D we describe a process to derive $succ$ and err, the success and error handlers for the promise from cb_f. In Section V-E we review the process to derive the new call site of f' from the original call site of f. Section V-F describes an optimization to flatten nested promises.

B. Identifying asynchrony

Our automated refactoring first detects refactoring candidates by searching for functions that use asynchronous callbacks. A function that uses an asynchronous callback is characterized by the use of one of its parameters as the callback argument of a known asynchronous function. The parameter can be passed directly to the asynchronous function, or indirectly by calling it inside the closure of another function. For example, consider Listing 6. Our technique infers that f is a refactoring candidate because cb_f is indirectly used as the callback argument of $async$, since it is called inside the anonymous function cb_{async}.

More formally, we consider a function f to be a refactoring candidate if $async$ is directly invoked inside f's body and if one of the following conditions is true:

(1) $cb_f = cb_{async}$, or
(2) cb_f is invoked inside the closure of cb_{async}

We look for instances of $async$ using a whitelist of calls that we know to be asynchronous. This whitelist includes a variety of asynchronous APIs, including DOM events, network calls, timers, and I/O. The complete list is available online [6].

C. Transforming the asynchronous function

In this subsection, we specify our transformations for deriving f' from f. We propose two strategies for transforming identified instances of asynchronous callbacks into promises, namely *modify-original* and *wrap-around*.

Strategy 1: Modify-original

In our exploratory study (Section IV), we observed the relative frequency of different kinds of promise refactorings performed by developers in practice. The modify-original strategy is based on the most frequent pattern that we observed.

Preconditions. Candidate instances for the modify-original refactoring to promises need to meet the following preconditions. The rationale behind these preconditions is also provided next to each item:

(1) cb_f **is invoked inside the scope of** cb_{async}
As described earlier, cb_f can be passed to the asynchronous function either directly as a callback or indirectly (inside another anonymous function called cb_{async}). We currently support the indirect way of passing (cb_f inside cb_{async}). This is because we use the function body of cb_{async} to detect the success path and error path. Thus the precondition ensures that $cb_f \neq cb_{async}$.

(2) cb_f **is not used in** cb_{async} **except as described by the previous precondition**
After the transformation, cb_f will only be called after the Promise is fulfilled or rejected. So if cb_f is invoked in other places in the function, the transformation to Promises will cause inconsistencies. Therefore to ensure that f does not use cb_f for anything other than as a callback function this precondition is needed.

(3) f **always returns** undefined
This condition checks that the function does not return anything. We found that in most cases, when an asynchronous function returns a value, the return value is used as an identifier for synchronization. This precondition eliminates cases where such an alternate synchronization method is used. Custom synchronization strategies require detailed knowledge of their implementation to produce a valid refactoring and are therefore not handled by either of our transformation strategies.

```
- function addTranslations(translations, call){
    translations = JSON.parse(translations);
    fs.readdir(dirname + '/../client/src/
        translations/',
            function (err, pofiles) {
      if (err) {
-       return callback(err);
      }
      var vars = [[]];
      pofiles.forEach(function (file) {
        var loc = file.slice(0, -3);
        if ((file.slice(-3) === '.po') && (loc !==
            'template')) {
          vars.push({tag: loc, language:
              translations[[loc]]});
        }
      });
-     return callback(vars);
    });
  }
- addTranslations(trans, jobComplete);
```

Listing 7. An example of an asynchronous callback before refactoring to promises – from KiwiIRC #581.

```
+ function addTranslations(translations){
+   return new Promise(function (resolve, reject){
      translations = JSON.parse(translations);
      fs.readdir(dirname + '/../client/src/
          translations/',
              function (err, pofiles) {
        if (err) {
+         return reject(err);
        }
        var vars = [[]];
        pofiles.forEach(function (file) {
          var loc = file.slice(0, -3);
          if ((file.slice(-3) === '.po') && (loc
              !== 'template')) {
            vars.push({tag: loc, language:
                translations[[loc]]});
          }
        });
+       return resolve(vars);
      });
+   });
  }
+ addTranslations(trans).then(jobComplete);
```

Listing 8. An example of an asynchronous callback after refactoring to promises using *Modify-original* strategy.

(4) cb_f **is *splittable***
This transformation requires a clearly identifiable success path that will be invoked when the (synthesized) Promise is fulfilled and also an error path to be invoked when this Promise is rejected. Therefore this precondition ensures that cb_f has a success path and an error path that do not interact with each other (i.e., that cb_f is splittable). For example, if cb_f is using the error-first protocol, the error parameter cannot be used on the success path and the data parameter cannot be used on the error path. This is because promises separate the success and error handlers, so any interaction between the two paths cannot be supported by a promises implementation.

(5) f **has exactly one** $async$
This is needed because only one promise will be returned after the transformation and the handler for a promise can only be invoked once. If more than one $async$ is invoked, a more complex refactoring is needed.

(6) **invocations of** cb_f **provide fewer than two arguments, or follow the error-first protocol**
This eliminates cases where more than one non-null argument is given to cb_f. This is a restriction of the current specification and implementation of promises in JavaScript, which only accepts one argument in both the resolve and reject handlers[8].

(7) f **is not contained in a third-party library**
This prevents library code from being refactored.

The limitation of modify-original is that it cannot transform more complex asynchronous callbacks that do not meet one or more of the above seven preconditions.

Transformation. PROMISESLAND implements the modify-original strategy by directly modifying the body of f. Below we work through an example of this strategy applied to function f in Listing 6. This function contains just a single $async$ call (to satisfy precondition (5) above). The modify-original strategy extends naturally to versions of the code where the $async$ call is surrounded by arbitrary synchronous code. The first step in modify-original is to create a new f' that returns a promise:

```
function f'() {
  return new Promise();
});
```

The Promise constructor takes one argument, namely the factory function for the promise. To build this, we declare cb'_{async}, an anonymous function that wraps the body of f:

```
function f'() {
  return new Promise(function (resolve, reject){
    async(function cb'async(error,data) {
      if(error) cbf(error, null);
      else cbf(null, data);
    });
  });
}
```

Next, we replace invocations of cb_f with invocations of resolve and reject. Invocations of cb_f that pass a non-null error argument are converted into invocations of reject. We look for arguments that use the error-first protocol or match the regular expression $e|err|error$ to find these invocations. All other invocations of cb_f are converted into invocations of resolve, which calls $succ$.

```
function f'() {
 return new Promise(function (resolve, reject){
   async(function cb'async(error,data) {
     if(error) reject(error, null);
     else resolve(null, data);
 });
}
```

Finally, in P' (the refactored program) we replace f with f'.

Listings 7 and 8 depict an asynchronous callback instance in a real-world JavaScript program, before and after it is refactored to promises by our modify-original technique, respectively. The refactored version of the function addTranslations

[8]Both Promise fulfillment and Promise rejection require only one resolution value because the fulfillment is similar to the return value of a function while the rejection of a Promise is similar to an exception thrown in a function, both of which are single values.

(Listing 8) does not accept a callback, and instead returns a promise. The original invocations of the callback (lines 6 and 15 in Listing 7) have been changed to reject and resolve (lines 7 and 16 in Listing 8) depending on whether an error occurred or not.

Strategy 2: Wrap-around

Because the modify-original strategy cannot transform asynchronous callbacks that do not satisfy one or more of the above preconditions, we also provide a strategy which (unlike modify-original) does not modify the body of f. This strategy is able to refactor a larger number of asynchronous callback functions than modify-original, but it produces more code by introducing a new function.

Preconditions. Candidates for the wrap-around refactoring must satisfy the following preconditions:

(1) $(cb_f = cb_{async}) \vee cb_f$ is invoked inside the closure of cb_{async}
(2) cb_f is not used in f except as described by the previous precondition
(3) f always returns undefined
(4) cb_f is *splittable*
(5) f has exactly one $async$
(6) invocations of cb_f provide fewer than two arguments, or follow the error first protocol
(7) f cannot be refactored by modify-original

Precondition (1) is the same as our precondition for identifying instances of f in Section V-B. Preconditions (2-6) are the same as the modify-original preconditions. Precondition (7) ensures that the modify-original strategy is selected first, because it produces more understandable code.

The preconditions for the wrap-around strategy are more relaxed than the preconditions for the modify-original strategy. This means that the wrap-around strategy can refactor more instances of asynchronous callback usage.

Transformation. In this strategy, we do not modify f. Instead, we wrap all of the calls to f inside a new function. We create this new function f', which creates and returns a Promise. A call to f is inserted into the body of the factory method for the promise:

```
1  function f'() {
2    return new Promise(function (resolve, reject) {
3      f();
4    });
5  }

7  function f(cb_f) {
8    async(function cb_async(error, data) {
9      if(error) cb_f(error, null);
10     else cb_f(null, data);
11   });
12 }
```

A new anonymous function is created as the continuation function for f. If cb_f follows the error-first protocol, the continuation function provides branches that direct the error parameter to reject and the data parameter to resolve:

```
1  function f'() {
2   return new Promise(function (resolve, reject) {
3    f(function(err,data){
4      if(err) return reject(err);
5      resolve(data);
6    });
7   });
8  }

10 function f(cb_f) {
11  async(function cb_async(error, data) {
12    if(error) cb_f(error, null);
13    else cb_f(null, data);
14  });
15 }
```

D. Transforming the callback function

By applying one of the two strategies, modify-original or wrap-around, we now have a new asynchronous function f' that returns a promise. We next transform all call sites of f to use the promise produced by f'. The first step is to identify call sites of f in the program. We rely on existing static analysis of TernJS [24], a type inference technique based on the work by Hackett and Guo [22] to determine the points-to relationships between call sites of f and the declaration of f.

Next, we convert all call sites to use f'. Consider c, a call site of f. The call site c has a callback function cb_f, which handles both successful and unsuccessful executions of f. However, f' requires a separate handler for successful and unsuccessful executions. From cb_f we derive two functions: the success handler $succ$ and the error handler err. $succ$ is the success-handling path of cb_f, while err is the error-handling path of cb_f. We therefore declare a success handler and an error handler for the promise. The code that is executed along the success path in cb_f is copied into $succ$, while all the code that is executed along the error path in cb_f is copied into err. Any conditional statements that cause control flow to branch to the success or error paths in cb_f are omitted from the handlers.

To determine the success path and error path of cb_f, we use a heuristic; we look for a conditional statement (e.g., an if) that checks if a parameter matching $e|err|error$ is non-null. We consider the branch where the parameter is null to be the success path, and consider the branch where the parameter is non-null to be the error path. This is based on the typical usage of the error-first protocol. Finally, in P' we replace cb_f with $succ$ and err. This is a simple heuristic, but we find that it is effective in practice (Section VII).

E. Transforming the call site

The last step in the refactoring process is to transform the call sites of f to invoke f' instead. First, if the wrap-around strategy was used, the name of f is changed to f'. If the modify-original strategy was used, the name remains unchanged.

Next, since f' no longer accepts a continuation function, the tool removes the cb_f argument. As the call to f' now produces a promise, it instead passes $succ$ and err to this promise: $f'()$.then($succ$, err);

In some cases, no err exists. Either because there was no error handling path in cb_f or one was not recognized by our

heuristics. In this case, in place of *err*, we insert a comment which recommends to the developer to create an error handler.

F. Flattening promise consumers

After a set of nested callbacks are converted into promises, the result is a set of nested promise consumers. Because [Promise].then also returns a Promise, we can improve readability by converting nested promises to a flat sequence of chained promises that are semantically equivalent. For example, Listing 9 has a set of nested promises that can be refactored to the chain of promises in Listing 10

```
getLocationDataNew('jackson').then(function (
    details) {
 getLongLatNew(details.address, details.country).
    then(function (longLat) {
  getNearbyATMsNew(longLat).then(function (atms) {
    console.log('Your nearest ATM is: ' + atms[0])
        ;
  });
 });
});
```

Listing 9. Nested promises.

```
getLocationDataNew('jackson').then(function (
    details) {
 return getLongLatNew(details.address, details.
    country);
}).then(function (longLat) {
 return getNearbyATMsNew(longLat);
}).then(function (atms) {
 return console.log('Your nearest atm is: ' + atms
    [0]);
});
```

Listing 10. Chained promises after they are flattened.

We have two preconditions for flattening promise consumers:

(1) $\forall$ variables v declared in $succ, v$ is not used inside a closure of $succ$

(2) only one nested promise is consumed inside $succ$

The first condition checks that no variable declared in $succ$ is also used in one of the asynchronous handlers declared in $succ$. This condition is necessary because if we add the handler for the nested promise through a promise chain, then v, which is declared in $succ$ will no longer be available to the nested promise's handler through closure. This is illustrated in Listing 11. We cannot flatten these nested promises because the parameter details is used by the success handler of getNearbyATMsNew.

The second condition checks that there is just one asynchronous call inside of $succ$ since promise chaining does not support executing multiple asynchronous functions in parallel.

If the two preconditions are met, to flatten a promise chain we perform two transformations. First, each handler is modified to return a promise. Second, for each handler that is not at the start of the chain, a new call to [Promise].then is created after the previous handler is registered. The handler is passed to the previous promise in the chain.

```
getLocationDataNew('jackson').then(function (
    details) {
  getLongLatNew(details.address, details.country
      ).then(function (longLat) {
    retunr getNearbyATMsNew(longLat).then(
        function (atms) {
      console.log('The closest ATM to ' +
          details.address + ' is: ' + atms[0]);
    });
  });
});
```

Listing 11. Nested promises which cannot be flattened

VI. Implementation

We have implemented our approach in a tool called PromisesLand. It supports both native promises, as well as a third party library implementation of promises called *Bluebird* [10]. PromisesLand is composed of two components: a static analyzer to search for refactoring opportunities, and a transformation engine to refactor the detected opportunities into promises. PromisesLand builds on prior JavaScript analysis tools, such as Esprima [25] to parse and build an AST, Estraverse [41] to traverse the AST, and Escope [3] for scope analysis. We also use TernJS [24], a type inference technique based on the work by Hackett and Guo [22], to query for function type arguments. As noted in Section V, TernJS is unsound, but is a good enough approximation for our purpose.

VII. Evaluation

The goal of our evaluation is to determine the efficacy of our approach. Specifically, we address the following three research questions:

RQ1: Can PromisesLand accurately identify instances of asynchronous callbacks to be converted?

We consider PromisesLand as an automated technique that a developer can use to first identify refactoring opportunities in the code. Therefore, we asses how accurately PromisesLand can find asynchronous callbacks in JavaScript code.

RQ2: Can PromisesLand correctly refactor asynchronous callbacks to promises?

A key factor determining adoption of a refactoring tool is confidence in its correctness [38]. We consider a refactoring correct if it preserves the program's behaviour after the transformation.

RQ3: Is PromisesLand efficient?

Refactoring tools that are slow face adoption challenges [32] in practice. We evaluate the efficiency of PromisesLand in detecting and transforming asynchronous callbacks.

We have made PromisesLand open source and all of our empirical data and results are available for download [4], [6].

A. Detection accuracy (RQ1)

To find out whether PromisesLand can accurately identify refactoring candidates in the form of asynchronous callbacks, we first manually inspect four subject systems (see Table I) to discover all asynchronous callbacks that can be converted to promises. This set of subject systems consists

TABLE I
DETECTION ACCURACY OF THE TOOL.

Subject System	LOC (JS)	Detected Instances	Refactored Instances	Precision (%)	Recall (%)
heroku-bouncer	947	7	6	100	85.7
moonridge	3,760	19	14	100	73.6
timbits	1,226	17	15	100	88.2
tingo-rest	238	4	4	100	100
Total	6,171	47	39	(avg) 100	(avg) 82.9

of heroku-bouncer,[9] a server-side middleware; moonridge,[10] an isomorphic library for MongoDB; timbits,[11] a client-side widget framework; and tingo-rest,[12] a REST-API wrapper for TingoDB. Because the manual inspection is time consuming, we included four systems. We believe these four are representative as they include server-side code, client-side code as well as isomorphic JavaScript that is executed both on the client-side and on the server-side.

We then use PROMISESLAND to find refactoring candidates to measure recall. We define recall as the percentage of asynchronous callbacks that PROMISESLAND detects, across all asynchronous callbacks that exist in the subject system.

Table I presents our results. The recall was 83% on average. PROMISESLAND missed 8/47 asynchronous callbacks. The reason is that our design is based on the premise that only if it can be guaranteed that all paths of a function execute the callback asynchronously, the callback can be considered to be semantically similar to a promise (and thus it becomes a refactoring candidate). To statically ensure that the callback is executed asynchronously and exactly once, we follow a conservative approach that can miss some of the potential candidates for conversion. This means that in practice, although PROMISESLAND detects and transforms most of the candidates, a few can be missed. We believe this can be improved further by using more advanced data-flow analysis techniques.

PROMISESLAND did not report any candidates that it could not correctly refactor. This is shown as 100% precision in Table I.

B. Refactoring correctness (RQ2)

In prior research, Brodu et al. [12] proposed a compiler-based technique to convert nested callbacks into a simpler specification of promises called Dues [11]. To evaluate PROMISESLAND, we select the subject systems used by Brodu et al. and compare our results to theirs. This set of subject systems consist of 64 Node.js modules and is expected to be representative of a majority of commonly used JavaScript modules. We measure how many asynchronous callback instances can be detected and converted to promises without leading to failures of the existing test cases of the subject systems.

[9] https://github.com/heroku/node-heroku-bouncer
[10] https://github.com/capaj/Moonridge
[11] https://github.com/postmedia/timbits
[12] https://github.com/lean-stack/node.tingo-rest

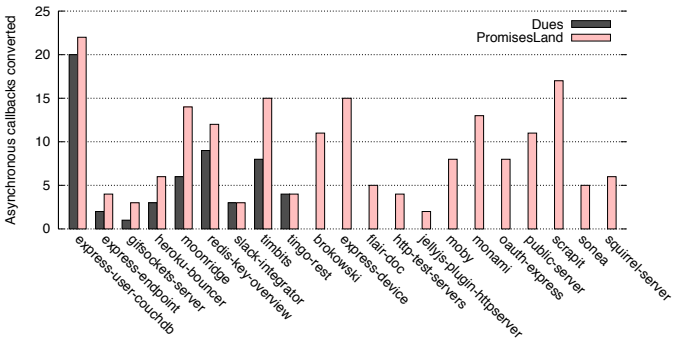

Fig. 2. Number of asynchronous callbacks converted into Dues by using the tool from [12]; and, the number of asynchronous callbacks converted into promises with PROMISESLAND.

We run PROMISESLAND on the 64 modules, which analyzes 438 JavaScript files and 108,615 lines of JavaScript code, to discover instances of asynchronous callbacks. From this list of asynchronous callbacks, we select those from projects with non-failing tests and those with at least one test case that executes them. There are 188 asynchronous callbacks from 21 subject systems that match these two criteria. This selection allows us to verify that behaviour is preserved after the refactoring step. We use test cases for this purpose because prior research [20] has shown that test cases are effective at providing an estimate of how reliable refactoring engines are for refactoring tasks on real software projects.

Next, refactoring is performed on the 188 asynchronous callbacks with PROMISESLAND. After PROMISESLAND refactors each asynchronous callback to a promise, we run the module's test suite to check if the original behaviour is preserved. For each refactoring that PROMISESLAND performed, the test suite passed successfully.

Figure 2 compares the results of our evaluation of PROMISESLAND against the technique proposed by Brodu et al. [12] (indicated as Dues). Across all subject systems, PROMISESLAND correctly transforms more asynchronous callbacks than the Dues transpiler. In total, the Dues transpiler converts 56 instances, while PROMISESLAND converts 188 instances (including those 56).

Out of the 188 converted instances, 73 are converted using the modify-original strategy; as it was the first strategy we attempted. The remaining 115 instances were converted using the wrap-around strategy. When detecting compatible continuations for refactoring, the Dues compiler restricted itself to choosing error-first callbacks only. However, PROMISESLAND does not have this constraint and determines the suitability for conversion by analyzing the body of the function itself. Therefore, our approach can select a larger set of asynchronous callbacks for safe conversion. Overall, we found that PROMISESLAND is able to refactor 235% more asynchronous callbacks than the tool proposed in [12].

These results show not only the ability of PROMISESLAND in detecting asynchronous callbacks, but also its correctness in transforming those callbacks into promises.

TABLE II
RUNNING TIME OF PROMISESLAND IN SECONDS PER PROGRAM.

Phase	Min	Max	Mean	Median
Async function detection	0.118	0.997	0.510	0.503
Promise creation conversion	0.102	0.495	0.293	0.289
Promise consumption conversion	0.114	0.468	0.273	0.292
Optimization and re-writing	0.138	0.951	0.610	0.584
All phases	0.969	2.569	1.686	1.645

C. Performance (RQ3)

Since refactoring tools are typically used in a development environment, refactoring must be completed quickly to not keep the developer waiting.

Table II shows the running time statistics for PROMISESLAND to complete each phase of the refactoring for a single program (from Figure 2) in seconds. The measurements were taken on a typical Linux system, containing Dual-core 2.16 GHz CPU and 4GB of RAM. In all cases the complete program refactoring finished in under 3 seconds. The last row of Table II shows the time taken for the complete refactoring process end-to-end. Since the migration from asynchronous callbacks to promises will be a one-time task in software maintenance, we believe the time taken by our technique is acceptable.

VIII. DISCUSSION

Wrap-around produces more complex code. It may be counter-intuitive that a refactoring strategy that produces more complex code, such as our wrap-around strategy, is useful. The benefits of this strategy are centered around the fact that this strategy does *not* modify the original function containing the asynchronous callback invocation. This is useful when:

- The function is in a library that cannot or should not be modified.
- The function has other clients that cannot be refactored.
- The function is too complex to refactor with other means (e.g., if it uses eval or other JS constructs that make the code difficult or impossible to analyze statically).

Evaluating PROMISESLAND. We evaluated the correctness of PROMISESLAND by running an application's tests after its code was refactored using the tool. This is a sanity check that the PROMISESLAND maintains program correctness. A more rigorous evaluation would require more formal techniques and is part of our future work.

Evaluating promises. Although at least some developers prefer promises over asynchronous callbacks, we do not know of any research that considers whether the use of promises improves JavaScript code quality. Our work contributes two refactoring techniques and a tool, PROMISESLAND, that implements these techniques. In our evaluation, we focus on features of the tool, such as its precision and recall. Empirical evaluation of the promises language feature itself and its implications for software quality and developer productivity remains an open problem.

IDE integration. By default PROMISESLAND refactors all asynchronous callbacks that it finds in the source code of an application, though it can be also run on a single source file. We believe PROMISESLAND can be integrated into common JavaScript IDEs to make it more easily accessible to developers, which forms part of our future work.

Backward Compatibility. Although all major JavaScript runtimes support promises, lack of backward-compatibility was a concern that we observed in discussions that we studied (Section IV). For example, one developer noted that *"[I] too believe promises are the future, but it seems that you need to make the users aware of what promise library they should use or native browser promises if supported."*[13] In other words, refactoring a library to use promises requires all clients of the library to change their code.

Async and await. Promises were initially specified in the ECMAScript6 specification. ECMAScript8 [5], which was released in June 2017, provides a new option for handling asynchrony in the form of the `async` and `await` keywords. These allow a linear programming style and permit traditional `try/catch` error handling, which is arguably more understandable than promises. However, our perspective is that, regardless of the underlying mechanism for managing asynchrony, the need for detecting and refactoring asynchronous callbacks will remain. The mechanisms described in this paper and implemented as part of PROMISESLAND are a step towards more powerful techniques. For example, Promises explicitly encode success and failure paths, which are implicit in the error-first protocol and are detected by our tool. With the techniques developed in this paper, when ECMAScript8 is widely adopted, we will be one step closer to automating the refactoring of JavaScript code to use `async` and `await`.

IX. CONCLUSION

It is difficult to imagine a useful JavaScript application that does not use asynchronous callbacks; these are used by applications to respond to GUI events, receive network messages, schedule timers, etc. Unfortunately, asynchronous callbacks present a number of software engineering challenges, including inability to properly catch and handle errors and callback nesting, which leads developers into "callback hell."

In this paper we presented two refactorings, modify-original and wrap-around, to refactor asynchronous callbacks into promises, a JavaScript language feature that resolves some of the issues with asynchronous callback. We implemented both refactorings as part of the PROMISESLAND tool and evaluated it on 21 large JavaScript applications. We found that PROMISESLAND correctly refactors asynchronous callbacks to promises, refactors 235% more callbacks than a tool from prior work, and runs in under three seconds on all of our evaluation targets.

PROMISESLAND is an open source tool. We made the tool and all of our empirical data and results available for download [4], [6].

[13] https://github.com/fixjs/define.js/issues/7

REFERENCES

[1] Can I use Promises?, 2017. http://caniuse.com/#feat=promises.

[2] The ECMAScript language specification, 2017. https://www.ecma-international.org/ecma-262/7.0/#sec-promise-objects.

[3] Escope, 2017. https://github.com/estools/escope.

[4] Motivation for moving to Promises. https://github.com/saltlab/PromisesLand/wiki/Motivation-for-moving-to-Promises, 2017.

[5] Status, process, and documents for ECMA262, 2017. https://www.ecma-international.org/ecma-262/8.0/#sec-async-function-objects.

[6] To the Promises Land: Refactoring Asynchrony in JavaScript. http://salt.ece.ubc.ca/software/promisland, 2017.

[7] S. Alimadadi, A. Mesbah, and K. Pattabiraman. Hybrid DOM-sensitive change impact analysis for JavaScript. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 321–345, 2015.

[8] S. Alimadadi, S. Sequeira, A. Mesbah, and K. Pattabiraman. Understanding JavaScript event-based interactions. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 367–377. ACM, 2014.

[9] E. Andreasen and A. Møller. Determinacy in static analysis for jQuery. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2014.

[10] Bluebird. Promise.promisify. http://bluebirdjs.com/docs/api/promise.promisify.html, 2017.

[11] E. Brodu. Due, 2017. https://github.com/etnbrd/due.

[12] E. Brodu, S. Frénot, and F. Oblé. Toward Automatic Update from Callbacks to Promises. In *Proceedings of the 1st Workshop on All-Web Real-Time Systems*, AWeS '15, pages 1:1–1:8, New York, NY, USA, 2015. ACM.

[13] B. Cavalier. Async programming part 2: Promises. 2013. http://blog.briancavalier.com/async-programming-part-2-promises/.

[14] D. Dig. Refactoring for asynchronous execution on mobile devices. *IEEE Software*, 32(6):52–61, 2015.

[15] A. Feldthaus, T. Millstein, A. Møller, M. Schäfer, and F. Tip. Tool-supported refactoring for JavaScript. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 119–138, New York, NY, USA, 2011. ACM.

[16] A. Feldthaus and A. Møller. Semi-automatic rename refactoring for JavaScript. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, pages 323–338. ACM, 2013.

[17] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Efficient construction of approximate call graphs for JavaScript IDE services. In *Proceedings of International Conference on Software Engineering (ICSE)*, pages 752–761. IEEE, 2013.

[18] K. Finley. Github has surpassed sourceforge and google code in popularity. 2011. http://readwrite.com/2011/06/02/github-has-passed-sourceforge.

[19] K. Gallaba, A. Mesbah, and I. Beschastnikh. Don't Call Us, We'll Call You: Characterizing Callbacks in JavaScript. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 247–256. IEEE Computer Society, 2015.

[20] M. Gligoric, F. Behrang, Y. Li, J. Overbey, M. Hafiz, and D. Marinov. Systematic testing of refactoring engines on real software projects. In G. Castagna, editor, *ECOOP 2013 – Object-Oriented Programming*, volume 7920 of *Lecture Notes in Computer Science*, pages 629–653. Springer Berlin Heidelberg, 2013.

[21] L. Gong, M. Pradel, M. Sridharan, and K. Sen. Dlint: Dynamically checking bad coding practices in JavaScript. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 94–105. ACM, 2015.

[22] B. Hackett and S.-y. Guo. Fast and precise hybrid type inference for JavaScript. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 239–250. ACM, 2012.

[23] Q. Hanam, F. Brito, and A. Mesbah. Discovering bug patterns in JavaScript. In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, page 13 pages. ACM, 2016.

[24] M. Haverbeke. Tern, 2017. https://github.com/marijnh/tern.

[25] A. Hidayat. Esprima, 2017. https://github.com/jquery/esprima.

[26] S. H. Jensen, P. A. Jonsson, and A. Møller. Remedying the eval that men do. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA, pages 34–44. ACM, 2012.

[27] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf. JSAI: A static analysis platform for JavaScript. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 121–132. ACM, 2014.

[28] Y. Lin, S. Okur, and D. Dig. Study and refactoring of android asynchronous programming. In *Proceedings of the International Conference on Automated Software Engineering, ASE*, pages 224–235, 2015.

[29] M. Madsen, F. Tip, and O. Lhoták. Static analysis of event-driven node.js JavaScript applications. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2015.

[30] F. Meawad, G. Richards, F. Morandat, and J. Vitek. Eval begone!: semi-automated removal of eval from JavaScript programs. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 607–620. ACM, 2012.

[31] A. Milani Fard and A. Mesbah. JSNose: Detecting JavaScript code smells. In *Proceedings of the International Working Conference on-Source Code Analysis and Manipulation (SCAM)*, pages 116–125, 2013.

[32] E. Murphy-Hill and A. Black. Breaking the barriers to successful refactoring. In *Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on*, pages 421–430, May 2008.

[33] H. V. Nguyen, C. Kästner, and T. N. Nguyen. Building call graphs for embedded client-side code in dynamic web applications. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 518–529. ACM, 2014.

[34] F. Ocariza, K. Pattabiraman, and A. Mesbah. Detecting inconsistencies in JavaScript MVC applications. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 325–335. ACM, 2015.

[35] M. Ogden. Callback hell, 2015. http://callbackhell.com.

[36] S. Rostami, L. Eshkevari, D. Mazinanian, and N. Tsantalis. Detecting Function Constructors in JavaScript. In *International Conference on Software Maintenance and Evolution*, ICSME, 2016.

[37] L. H. Silva, M. T. Valente, and A. Bergel. Refactoring Legacy JavaScript Code to Use Classes: The Good, The Bad, and The Ugly. In *Proceedings of the International Conference on Software Reuse*, ICSR, 2017.

[38] G. Soares, R. Gheyi, D. Serey, and T. Massoni. Making program refactoring safer. *Software, IEEE*, 27(4):52–57, 2010.

[39] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip. Correlation tracking for points-to analysis of JavaScript. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, pages 435–458. Springer, 2012.

[40] Stack Overflow. 2017 Developer Survey. http://http://stackoverflow.com/insights/survey/2017/, 2017.

[41] Y. Suzuki. Estraverse, 2017. https://github.com/estools/estraverse.

[42] TypeScript. Typescript, 2015. http://www.typescriptlang.org.

[43] R. von Behren, J. Condit, and E. Brewer. Why Events Are a Bad Idea (for High-concurrency Servers). In *Conference on Hot Topics in Operating Systems*, HOTOS, 2003.

[44] S. Wei and B. G. Ryder. State-sensitive points-to analysis for the dynamic behavior of JavaScript objects. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, pages 1–26. Springer, 2014.