Nessie: Automatically Testing JavaScript APIs with Asynchronous Callbacks (ICSE22)

Ellen Arteca
arteca.e@northeastern.edu
Northeastern University
USA

Sebastian Harner
harnersebastian@gmail.com
University of Stuttgart
Germany

Michael Pradel
michael@binaervarianz.de
University of Stuttgart
Germany

Frank Tip
f.tip@northeastern.edu
Northeastern University
USA

Motivation

- Previous algorithms for feedback-directed unit test generation iteratively create sequences of API calls by executing partial tests and by adding new API calls at the end of the test. However, these methods cannot deal with APIs that take callback functions as arguments.

Contributions

- The first automated test generator specifically aimed at APIs that accept callbacks to be invoked asynchronously.
- An algorithm for incrementally generating tests that not only sequence API calls but also nest them inside callbacks.
- Empirical evidence demonstrating that:
    - The approach is effective at exercising JavaScript APIs with asynchronous callbacks
    - It achieves modestly higher code coverage and finds more behavioral differences than the state of the art
    - It converges much more quickly than prior work when it comes to achieving a specific level of coverage or behavioral differences.

Movtivation Example

```
let fileName = ...;
exists(fileName,
  // asynchronously invoked callback function:
  (err, fileHandle) => {
    if (!err) {
      read(fileHandle,
        // another callback function:
        (err, jsonObj) => { ... })
    }
})
```

The call to **read** is nested in the callback that is passed to **exists**, to ensure that the read operation is executed after the **exists** operation has completed. Now suppose that the **read** operation contains a bug that is triggered in certain cases where a valid file-handle is passed (e.g., if the file's permissions do not permit read access), and suppose that we want to generate a test that invokes the **read** function to expose the bug. Since file-handle objects are created inside the library, it is unclear what the representation of these objects looks like without analyzing or executing the library code.
While it is possible for a test generator to create suitable file-handle objects using a purely random approach, the chances of successfully creating a valid file-handle would be small. Therefore, the most effective way to obtain a valid file-handle and expose the bug is to invoke **exists** with some callback function *f*, and invoke **read** with the file-handle that is passed to *f* as its second argument. That is, we would generate a test where a call to **read** is nested in the callback that is passed to **exists**, as in the above example.

Approach

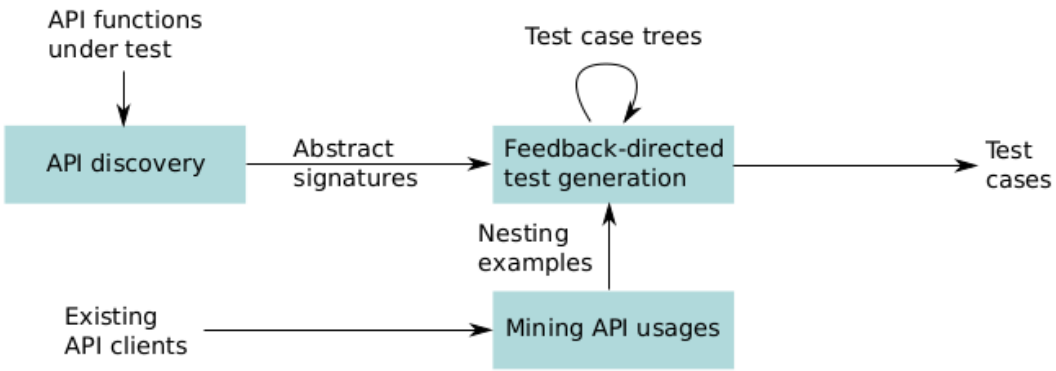


Figure 1: Overview of the approach.

API Discovery

Since JavaScript is dynamically typed, our approach needs to infer the signatures of functions as a prerequisite to generating effective tests.

Definition 1 (Abstract signature). An abstract signature for a function f is a tuple $(arg_1, ..., arg_n)$, where each arg_i is one of the following three kinds of elements:

- *async*: an argument is an asynchronously invoked callback
- *sync*: an argument is a synchronously invoked callback
- the _ symbol: any non-callback argument

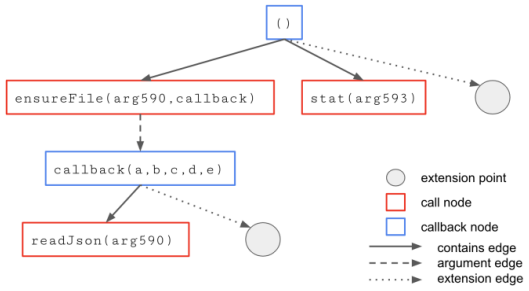
To discover signatures for a given API function under test, Nessie repeatedly invokes the function with randomly generated arguments. The approach alternates between generating calls with and without a callback argument, and passes different numbers of arguments.
Nessie collects non-erroneous executions and distinguishes three cases:

- A callback that executes after the test has executed is executed (async)
- a callback that executes before the API call returns is executed synchronously (sync)
- the callback is not executed or the test does not pass any callback (_)

Feedback Directed Test Generation

```
// Omitted for clarity:
//  * Try-catch around each call to an API function.
//  * Print statements to log arguments and return values.
//  * Print statements to log control flow.

let fs_extra = require("fs-extra");

var arg590 = "a/b/test";
var arg593 = null;

let r_126_0 = fs_extra.ensureFile(arg590, cb(a, b, c, d, e) =>
  let r_126_0_0 = fs_extra.readJson(arg590);
  return false;
));

let r_126_1 = fs_extra.stat(arg593);
```

(a) Generated test.



(b) Test case tree representation of the test, with extension points annotated.

Figure 2: Example of a generated test for the API functions of the `fs-extra` library and the corresponding test case tree.

Definition 2 (Test case tree). Let V be a map of variable names to non-callback values, called value pool, and S a map of function names to abstract signatures. Then, a *test case tree* is an ordered tree where nodes are either:

- a *call node* of the form $r = api(a_1, ..., a_k)$, meaning that function api is invoked with arguments $(a_1, ..., a_k)$ and yields return value r. If $s_i \in \{sync, async\}$ for some signature $(api \mapsto (s_1, ..., s_n)) \in S$, then a_i may be a callback function. Otherwise, $a_i \in V$ or it is a return value of another call, or
- a *callback node* of the form $cb(p_1, ..., p_k)$, meaning that callback function cb receives parameters $p_1, ..., p_k$.

Edges are either:

- a *contains edge* from a callback node to a call node, meaning that there is a call in the body of the callback function, or
- an *argument edge* from a call node to a callback node, meaning that a call is given a callback function as an argument.

Algorithm 1 Feedback-directed generation.

Input: Set F of API functions, map S from function names in F to their discovered signatures, mined nesting examples M
Output: Set of tests T

$T \leftarrow \emptyset$ ▷ Generated test case trees
$t \leftarrow$ empty test case tree
$E \leftarrow \{(t, root(t))\}$ ▷ Extension points
while $|T| <$ number of tests to generate **do**
    $(t, n) \leftarrow$randomly pick from E
    $f \leftarrow chooseFunction(t, n, F, M)$
    $args \leftarrow chooseArguments(t, n, f, S, M)$
    $t' \leftarrow$ extend t with call to $f(args)$ at n
    $feedback \leftarrow execute(t')$
    $T \leftarrow T \cup \{t'\}$
    if $noExceptions(feedback)$ **then**
        for each $n' \in extensionPoints(feedback)$ **do**
            $E \leftarrow E\{(t', n')\}$
return T

Mining API Usages

Nessi uses a static analysis for mining nesting examples from real-world uses of APIs by traversing the ASTs of existing API clients. This analysis was implemented in CodeQL, using its extensive facilities for static analysis.

The test generator uses the set of mined nesting examples in **chooseFunction**. When selecting a function to be nested in the callback of some function f, the set of nesting examples is consulted to find examples where f outer matches f. If such nesting examples exist, then one of the corresponding $f$inner functions is randomly selected to be invoked inside f's callback. Similarly, chooseArgu- ment consults the selected nesting example to determine which arguments (if any) to reuse from the outer function or from the surrounding callback, and at what position(s).
If no relevant mined nesting examples are available, an inner function is randomly selected.

Evaluation

Benchmarks

Table 1: Summary of projects used for evaluation.

Project	LOC	Cov. loading	Commit	Description
fs-extra	907	16.8%	6bffcd8	Extra file system methods
jsonfile	45	19.1%	9c6478a	Read/write JSON files in Node.js
node-dir	285	5.9%	a57c3b1	Common directory/file operations
bluebird	3.3k	23.7%	6c8c069	Performance-oriented promises
q	760	22.2%	6bc7f52	Promise library
graceful-fs	439	25.3%	c1b3777	Drop-in replacement for native `fs`
rsvp.js	579	16.4%	21e0c97	Tools for organizing async code
glob	845	11.0%	8315c2d	Shell-style file pattern matching
zip-a-folder	24	16.0%	5089113	Zip/tar utility
memfs	2.4k	29.1%	ec83e6f	In-memory file system

Baselines and Variants of the Approach

We compare Nessie against the state of the art approach **LambdaTester** (LT). Because the original LT does not support language features introduced in ECMAScript 6 and later, and because parts of the implementation are specific to their benchmarks, we re-implemented LT within our testing framework. To better understand the value of nesting and sequencing, we evaluate two variants of Nessie: NES (seq), which uses sequencing only, and NES (seq+nest), which uses both sequencingand nesting.

RQ1: Effectiveness of Automated API Discovery

Automated discovery finds 62% of documented signatures that expect callback arguments, and 38% of signatures without callback arguments. It also discovers some undocumented signatures, which in several cases reflect unexpected behavior.

Table 2: Abstract signatures categorized manually (M) and found by automated API discovery (A).

	Signatures *with* callbacks				Signatures *without* callbacks			
Project	M	A	Only M	Only A	M	A	Only M	Only A
fs-extra	21	88	3	70	45	361	21	337
jsonfile	4	8	0	4	8	12	4	8
node-dir	9	6	4	1	1	11	0	10
bluebird	25	22	7	4	29	68	26	65
q	16	27	7	18	57	155	19	117
graceful-fs	–	15	N/A	N/A	–	36	N/A	N/A
rsvp.js	3	7	0	4	10	31	3	24
glob	6	6	0	0	4	6	1	3
zip-a-folder	0	0	0	0	3	4	2	3
memfs	58	30	33	5	57	62	56	61

RQ2: Coverage Achieved by Generated Tests

Nessie achieves a higher coverage than the state of the art, and fewer tests are required to reach this coverage, in particular, when the approach uses both sequencing and nesting.

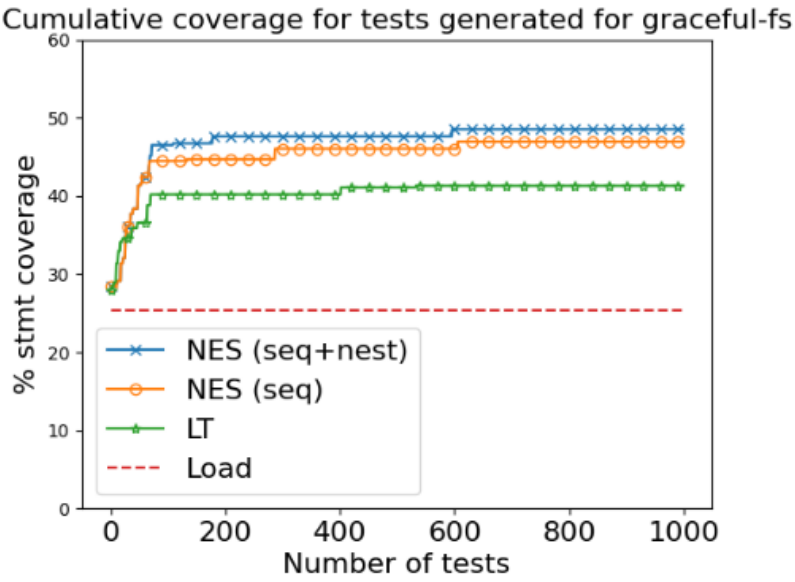


Figure 3: Cumulative coverage while generating 1,000 tests for graceful-fs.

RQ3: Finding Behavioral Differences during Regression Testing

Nessie finds many behavioral differences between versions of libraries, including accidentally introduced bugs, bug fixes, and API upgrades. These differences are found most quickly with generated tests that use both sequencing and nesting.

RQ4: Impact of Guidance by Mined Nesting Examples

Choosing mined nestings some of the time results in tests with higher coverage than those generated always or never using the mined nestings. The optimal parameter depends on the library under test, but 50% is an overall reasonable choice.

Table 6: Coverage after 1,000 tests at different levels of using mined nesting examples.

	Test coverage, using % mined nestings				
Project	**0%**	**25%**	**50%**	**75%**	**100%**
fs-extra	32.2%	33.6%	**37.2%**	33.0%	33.0%
jsonfile	**87.2%**	83.0%	**87.2%**	**87.2%**	**87.2%**
node-dir	32.5%	**33.6%**	32.5%	33.2%	33.2%
bluebird	45.7%	51.2%	48.0%	**55.1%**	48.4%
q	65.7%	66.4%	**67.2%**	66.3%	66.3%
graceful-fs	**48.5%**	**48.5%**	**48.5%**	**48.5%**	47.0%
rsvp.js	64.8%	65.0%	**66.0%**	58.2%	58.2%
glob	**36.1%**	**36.1%**	**36.1%**	**36.1%**	**36.1%**
zip-a-folder	24.0%	24.0%	**32.0%**	**32.0%**	24.0%
memfs	50.0 %	51.2%	**55.5%**	51.8%	51.8%

RQ5: Performance of Test Generation

With 15 to 30 seconds per 100 tests, the approach is efficient enough for practical use. Extending the set of mined nesting examples takes time proportional to the number of projects mined but is an up-front cost.