Reasoning about the Node.js Event Loop using Async Graphs

Haiyang Sun
Faculty of Informatics
Università della Svizzera italiana (USI)
Switzerland
haiyang.sun@usi.ch

Daniele Bonetta
Oracle Labs
USA
daniele.bonetta@oracle.com

Filippo Schiavio
Faculty of Informatics
Università della Svizzera italiana (USI)
Switzerland
filippo.schiavio@usi.ch

Walter Binder
Faculty of Informatics
Università della Svizzera italiana (USI)
Switzerland
walter.binder@usi.ch

***Abstract*—With the popularity of Node.js, asynchronous, event-driven programming has become widespread in server-side applications. While conceptually simple, event-based programming can be tedious and error-prone. The complex semantics of the Node.js event loop, coupled with the different flavors of asynchronous execution in JavaScript, easily leads to bugs. This paper introduces a new model called Async Graph to reason about the runtime behavior of applications and their interactions with the Node.js event loop. Based on the model, we have developed AsyncG, a tool to automatically build and analyze the *Async Graph* of a running application, and to identify bugs related to all sources of asynchronous execution in Node.js. AsyncG is compatible with the latest ECMAScript language features and can be (de)activated at runtime. In our evaluation, we show how AsyncG can be used to identify bugs in real-world Node.js applications.**

***Index Terms*—JavaScript, Dynamic Analysis, Event-driven Programming, AsyncG**

I. Introduction

Node.js [1] has become a major platform for server-side Web development. One key reason for its success is its non-blocking I/O runtime, backed by a high-level, event-driven, asynchronous programming model. Thanks to Node.js, such an event-driven programming model has emerged as a dominant programming abstraction for server-side Web development, and has recently also been adopted in other major platforms such as Java (via the new Reactive Streams API [2]), Swift [3], and Go [4].

Event-driven programming with an event loop [5] is conceptually simple: the event loop listens for events and triggers an event handler (e.g., callback functions) when an event occurs. The event loop in Node.js itself is executed by a single thread, but it can poll events from other threads (e.g., native threads for I/O), and thus makes Node.js capable of handling thousands of concurrent client requests.

Despite the simplicity of the model, the *exact* semantics of the Node.js event loop for different APIs is quite complex and can easily lead to bugs. For example, a simplified version of a common Node.js bug reported on StackOverflow (SO)[1] is depicted in Fig. 1. The problematic code line causing the bug is highlighted in red (-), while a possible fix replacing the buggy line is highlighted in green (+). The application creates an HTTP server to listen for incoming connections and performs some computation in the compute function. The developer splits the execution of the entire computation into multiple recursive steps, and uses process.nextTick to defer successive calls to compute to be executed in the "future". The overall goal is to avoid blocking the request processing of the server by scheduling some of the computations for a later iteration of the event loop, such that an incoming HTTP request can be handled in between. However, the server started by this code ends up in an infinite recursion of the function compute, and is unable to process any incoming request. The fix is to replace nextTick with setImmediate. While nextTick and setImmediate both provide ways to schedule a callback function for future execution, internally they interact with different event queues: tasks created with setImmediate are processed fairly with respect to incoming HTTP requests, while tasks created using nextTick are processed with the highest priority as soon as they are produced. Given such different scheduling semantics, the event loop will process *only* calls to compute, without allowing the HTTP server to process any incoming request.

In addition, developers can build complex execution patterns combining different asynchronous APIs such as simple task dispatch (e.g., nextTick, setImmediate, timers), non-blocking I/O, as well as those to manage asynchrony such as EventEmitter [6], Promise (introduced since ECMAScript 6 [7]) and async/await (introduced since ECMAScript 8 [8]). Because of the subtle difference in the scheduling semantics

Artifact available at: https://doi.org/10.5281/zenodo.2328360
[1] https://stackoverflow.com/questions/33330277

```
const http = require('http');
function compute() {
  performSomeComputation();
  // recursive nextTick blocking event loop
- process.nextTick(compute);
+ setImmediate(compute);
}
http.createServer((request, response) => {
  response.end('Hello World!');
}).listen(5000);
compute();
```

Fig. 1. Simplified SO-33330277 bug of a misuse of nextTick instead of setImmediate

and complicated execution patterns, debugging event-based, asynchronous applications is notoriously hard and tedious. Static analysis is used in [9], [10] to identify problematic code patterns in Node.js with significant limitations [11] due to the dynamic nature of JavaScript. Other tools visualizing callback executions [12]–[14] do not support all the language features (e.g., event emitters, promises, or async/await), and cannot reflect the complicated callback scheduling of the event loop, which is important to pinpoint API misuses due to misunderstanding the event loop.

In this paper, we introduce a graph-based model of asynchronous, event-based Node.js applications, called *Async Graph* (AG). The AG of an application models all callback scheduling in the Node.js event loop and can be used by developers or automated tools to reason about the actual execution of a Node.js application. Based on the model, we have developed a tool (called AsyncG) to automatically build the AG using instrumentation techniques. AsyncG can track all kinds of asynchrony sources, and is able to automatically identify certain kinds of bugs related to event scheduling in an application at runtime. To the best of our knowledge, AsyncG is the first tool to detect bugs in Node.js applications caused by the misuse of a *combination* of different event-based APIs. AsyncG is pluggable, and can be enabled/disabled at runtime. When enabled, it introduces an overhead that is compatible with that of an application debugger.

In summary, the paper makes the following contributions:

- We introduce a new model, Async Graph, to reason about the execution of event-driven Node.js applications.
- We implement AsyncG, an instrumentation-based tool which can build the AG of a running application at runtime.
- AsyncG is able to automatically analyze the AG of an application and report warnings about potential bugs and code smells.
- AsyncG can detect common Node.js event-related bugs, including well-known programming mistakes [10], [15] as well as newly-found bug patterns.

II. Background

In this section, we provide a detailed introduction to different kinds of asynchronous execution and the event-loop model of Node.js.

A. Sources of Asynchronous Execution

In essence, a Node.js application can be considered a set of JavaScript callback functions, executed by a single thread called *event loop*. The execution of callbacks is determined by the Node.js event loop based on the application state and its interaction with the underlying Operating System (OS). More precisely, callbacks can be executed as a consequence of one of the following two sources of asynchronous execution:

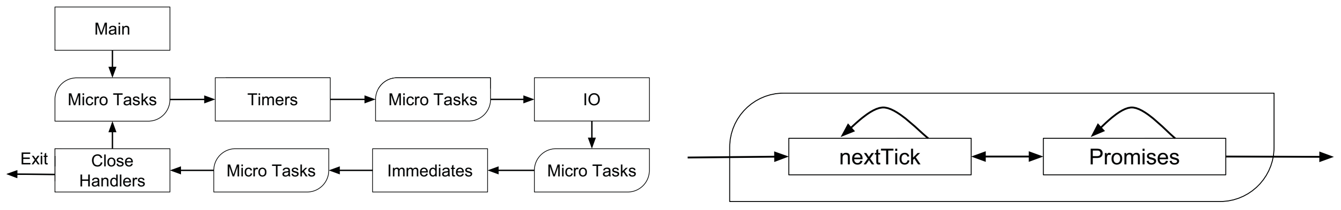
- Self-scheduling. The application has deferred the execution of a given callback function using a JavaScript or Node.js API including promises, process.nextTick(), and setImmediate. Depending on the API, the application has different guarantees on *when* a deferred callback will be executed.
- External scheduling. The application has registered a callback for execution upon an *event* happening in the OS. It can be I/O related, e.g., an application may register a callback to be executed whenever a new client connects to a TCP port. It can also be timing related, e.g., a callback can be registered to be executed after a given time (setTimeout). Such callbacks are scheduled by the OS, which interacts with the Node.js event loop by notifying it, providing event-specific data.

Besides these two sources of asynchronous execution, Node.js also offers the EventEmitter API [6] for explicit event emission on special objects called *emitters*. An emitter object can register callbacks (listeners) on named events, and such listeners will be called when such events are *emitted*. Event emission happens either via explicit function calls or via built-in (runtime) mechanisms. Events are widely used in I/O-related asynchronous APIs [6], e.g., the net.Server built-in object emits a "connect" event each time a new connection is established. Applications can simply register event listeners for "connect" events to handle incoming data.

It is common that asynchronous execution APIs are used together, leading to a *chain* of callback registrations, scheduling, and execution. A simple example is shown in the following code snippet:

```
http.createServer(function accept(req, res) {
  let body = [];
  req.on('data', function data(chunk) {
    body.push(chunk);
  }).on('end', function() {
    setImmediate(function defer(){
      res.end(process(body));
    }
  });
});
```

The code defines a simple HTTP server that accepts data from an incoming request and replies after processing the input data. The code exercises several APIs. First, createServer is an I/O scheduling API, which (internally) creates an emitter listening to the "request" event from the OS when an HTTP request is received. Second, every HTTP request is also modeled as an emitter to receive data from the connection, and after the data has been received (and stored in body), the



(a) Event-loop queues (b) Micro-task queues

Fig. 2. Callback execution phases in the Node.js event loop

"end" event is emitted. Then, the application defers the potentially heavy processing of the input data with self-scheduling (setImmediate), allowing other I/O events to be scheduled, and finally the server will respond with the processed data. Here, we see that multiple asynchronous APIs can be mixed with a processing *chain* (http-request → data receiving → setImmediate → data processing → response).

B. Node.js Event-loop Dispatch

External and self-scheduling callbacks are executed by the Node.js event loop following different event processing strategies that greatly depend on the API that is being used to register the callback. As shown in Fig. 2, the Node.js event loop internally handles several task queues, used by external and self-scheduling sources of asynchronous execution to *register* callbacks. These queues are *polled* by the event loop, and when an entry is found in the queue, the corresponding callback is executed by the JavaScript execution engine. The queues are not processed in round-robin order. Instead, the Node.js event loop performs multiple processing *phases* as depicted in Fig. 2(a). A Node.js application starts in a main phase, where the JavaScript engine executes the main application code, which can register callbacks and push new tasks into the queues. If there are no pending tasks in the current phase, the event loop will move to the next phase. Between each run of the event loop, Node.js checks whether it is waiting for some asynchronous I/O or timer events, and terminates if there are none [5]. The details of each phase are discussed below. A complete formal description of the Node.js event loop semantics can be found in [16].

- *Micro tasks*: Micro tasks have higher priority than all other tasks and can be scheduled between any other phases by the Node.js event loop. There are two kinds of microtasks: nextTick (callbacks scheduled via process.nextTick) and promise (callbacks scheduled via promise resolve or reject [7]). As shown in Fig. 2(b), nextTick tasks have higher priority than promise tasks, while both tasks can schedule each other.
- *Timers*: The timers phase processes callbacks scheduled by setTimeout once after a timeout, while setInterval schedules callbacks repeatedly (after each elapsed time interval).
- *Immediates*: The immediates phase deals with callbacks scheduled with the setImmediate API.
- *I/O*: External events that map to interactions with the OS are scheduled in the I/O phase. Examples of callbacks associated with this queue are functions to read data from a file, to accept an incoming connection, etc.
- *Close Handlers*: This phase deals with cleanup and low-priority operations, such as callbacks registered using the on('close') event handler [6]. Such events are typically related to external interactions with the OS (e.g., a socket disconnection), but are treated with the lowest priority by the event loop.

III. Motivation and Challenges

In this section, we motivate the need for a unified model to reason about asynchronous execution in Node.js, and we discuss the related challenges.

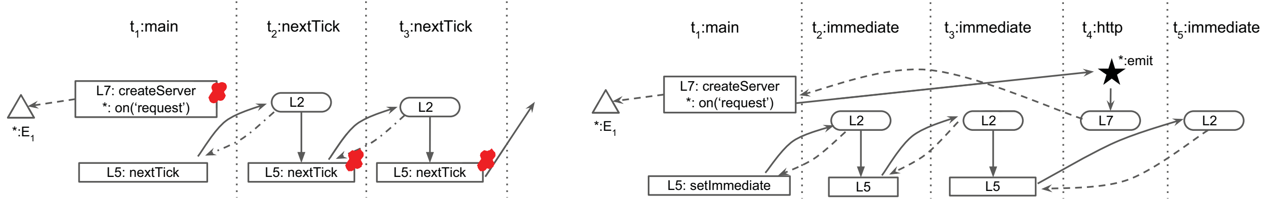
A. Motivation

The complex semantics of the Node.js event loop and the different user-level APIs enabling callback scheduling can easily result in bugs. Hence, our primary motivation is to help programmers understand the timing of the event-loop scheduling and find out the relations between callback registration and execution. As described in the example of Fig. 1, misusing an asynchronous API (e.g., due to lack of knowledge of the event-loop scheduling) can lead to program failures. Bugs can result from a scheduling of callbacks that is not expected by the programmer, as in the following code snippet.

```
1   let foo;
2   Promise.resolve({}).then( (v) => {
3     foo = v;
4   });
5   setTimeout(() => {
6     foo.bar = function() { /* ... */ };
7   }, 0);
8   process.nextTick(() => {
9     foo.bar();
10  });
```

This code snippet results in a program crash caused by the (wrong) assumption that the callbacks would be executed in the registration order: promise (L2) – setTimeout (L5) – nextTick (L8), whereas the real execution order of the callbacks is L8 – L2 – L5.

Another motivation is to find bugs related to promises and emitters, which are two widely used APIs in Node.js applications. Different from other asynchronous APIs, both promises and emitters have their own ways of scheduling callbacks.



(a) AG with recursive nextTick

(b) fixed AG using setImmediate

Fig. 3. AG for the code example in Fig. 1

Finding bugs related to promises requires knowledge of how they are chained [15], while emitters manage callbacks in named events and are widely used for handling I/O events. Understanding promises and emitters helps find bugs in server-side Node.js applications [10], [15].

B. Challenges

The ultimate goal of our research is to empower developers with a debugger to reason about the runtime behavior of Node.js applications, allowing them to *understand* the actual order of callback execution in an application, and to *identify* (potential) sources of bugs or code smells. Understanding the callback execution order is crucial, as a wrong execution order is the main source of bugs related to asynchronous execution [11], [17], [18]. There are four main challenges in developing such a tool:

- *Formalization*. The tool needs to represent asynchronous executions using a new abstraction, since traditional representations of program execution (such as call trees and application execution stacks) do not allow one to reason about callback scheduling and its relation with APIs such as promises and emitters.
- *Visualization*. A proper visual representation of the execution of asynchronous callbacks is crucial to clearly illustrate the event loop scheduling of an application execution.
- *Implementation*. The tool needs to be able to track all sources of asynchronous execution in an application (such as e.g. API calls and external events) and map them to the corresponding event queue. The implementation should be transparent to the application so that it causes no side-effects to the application execution.
- *Automatic Bug Detection*. The tool should be able to identify bugs which cannot be easily found with existing tools, especially the bugs which need understanding of the event loop scheduling and bugs involving promises and emitters.

Our research aims at solving these challenges. In the following section we introduce AG, corresponding to our proposed solution to both formalization and visualization, while Section V presents the implementation of our tool AsyncG and Section VI shows the bugs that can be detected.

IV. Async Graph (AG)

An Async Graph is a time-oriented graph that explicitly describes the asynchronous flow of execution in a Node.js application. Each node in the graph belongs to a specific event-loop *tick*, which encodes "when" a specific application event happens with regards to the global execution flow of the application. Each node is also related to a source-code location, which corresponds to "where" the event originates in the application such that readers of the graph can map each node to the originating code location.

A. Graph Structure

Event-loop *ticks* are displayed using vertical lines that "split" the graph into multiple blocks: each tick corresponds to a single execution of an event-loop phase as described in Section II-B. We name the ticks as $t_i : PhaseName$, where $t_1 : main$ is the first tick corresponding to the start of a program. Each tick may contain one or more nodes of the following types:

- □ – a *Callback Registration (CR)* node represents an API use which registers one or more callbacks to be executed. Depending on the API (self-scheduling or external), the callback may be executed instantly (e.g., the execution of a promise constructor), in the following micro-task tick (promise, nextTick), or in successive ticks of the event loop (I/O, timers, immediates).
- ○ – a *Callback Execution (CE)* node corresponds to the actual execution of a callback. A CE node is positioned in the event-tick block corresponding to where it happens.
- ★ – a *Callback Trigger (CT)* node stands for an API use that explicitly leads to the execution of a (previously registered) callback. A CT node can be either an event emission (using the `emitter` API) or a promise action (resolve or reject).
- △ – an *Object Binding (OB)* node corresponds to the creation of a promise or emitter object.

Two different kinds of edges are possible between node α and node β in the graph:

- A direct edge (expressed as $\alpha \rightarrow \beta$) means that α *causes the execution of* β:
  - □|★ → ○: a CE is always caused by its registration (□ → ○), while a CE can also be caused by a CT (★ → ○) in the case of a promise resolve/reject action or event emission.

  - $\bigcirc \rightarrow$: this “happens-in” edge is used to show the relation between a CE and any nodes that are executed during this CE.
- A dashed connection (expressed as $\alpha \dashleftarrow \beta$) describes the following relations between α and β:
  - $\square \dashleftarrow \bigcirc$: this edge shows the binding between a CR and the corresponding CE. With this edge, $\square \rightarrow \bigcirc$ for emitters and promises can be inferred from the chain $\bigstar \rightarrow \bigcirc \dashrightarrow \square$ and thus can be omitted in the graph.
  - $\triangle \overset{\text{relation}}{\dashleftarrow}$: the edge shows the “relation” in the label between a node with a promise or emitter OB ($\triangle$). For example, adding a new listener for an emitter object will be represented by $\triangle \overset{\text{connection}}{\dashleftarrow} \square$, where *connection* is the name of the event. Relations between a promise created by another promise via APIs (such as then,catch,all and race) can be shown in edges like $\triangle \overset{\text{p}}{\dashleftarrow} \triangle$, where p is the name of the promise API. In addition, $\triangle \overset{\text{link}}{\dashleftarrow} \triangle$ is used to join the promise returned from a then callback to the corresponding promise chain.

B. Examples

AGs describe both the asynchronous execution flow of an application (i.e., *when* a callback is executed), as well as the *relations* between callbacks (i.e., *why* a callback is executed, and *what* causes its execution). Such information is crucial to understand bugs related to the scheduling of asynchronous code in Node.js. In the following sections, we demonstrate how bugs can be identified using an AG. For clarity, each node in the graph has a name starting with the line number of the associated source-code location (e.g., L_3, or $*$ for internal libraries). Warnings are highlighted with ✗ next to the node.

1) Example 1: The AG for code in Fig. 1 tested with a client sending new requests is depicted in Fig. 3(a). As the graph grows infinitely due to the recursive nextTick bug, we only show the first 3 ticks which suffice to illustrate the problem.

In t_1, CR ($\square - L7 : createServer$) registers the callback to handle incoming connections. In Node.js, http.createServer registers the callback with an internal event emitter ($* : E_1$) on the named event “request”. As the callback is never executed although there are incoming connections, a warning of *Dead Listener* is reported. Warnings about recursive nextTick are given in the following ticks starting from t_2, where nextTick is used for registering the same function in a nextTick phase.

The AG for a fixed version of the application is shown in Fig. 3(b). Using setImmediate instead of process.nextTick, the callback will be scheduled for execution in an immediate phase (e.g., t_2, t_3 and t_5). Unlike the micro task introduced in Section II which is always scheduled first, the immediate queue will allow I/O events to be scheduled in between, e.g., createServer is scheduled in t_3 when there is an incoming connection.

2) Example 2: The second example corresponds to a more complicated application combining emitters and promises. The code with the fix and the AGs are shown in Fig. 4 and Fig. 5.

```
1  var ee = new EventEmitter();
2  var p = new Promise(
3    resolve => { resolve(0); }
4  );
5
6  //resolved in a different loop
7  p.then(() => {
8    //unused listener
9    ee.on('foo', () => {
10       //...
11   })
12 -}); //missing exception handler
12 +}).catch((err)=>{});
13
14 //dead emit
15 -ee.emit('foo');
15 +setImmediate(()=> {ee.emit('foo')});
```

Fig. 4. Example 2: combination of promises and emitters

The bug in the example results from the misunderstanding of the implicit event-loop scheduling of promises: although the promise is constructed ($\square - L2$) and resolved ($\bigstar - L3$) in t_1, the reaction of the promise registered via then ($\bigcirc - L7$) is scheduled in the successive tick. As a result, the “foo” event is emitted ($\bigstar - L15$) for the emitter ($\triangle - L1$) before the listener ($\square - L9$) is registered. Thus, a dead-emit error ($L15$) and a dead-listener warning ($L9$) can be automatically found and highlighted in the graph.

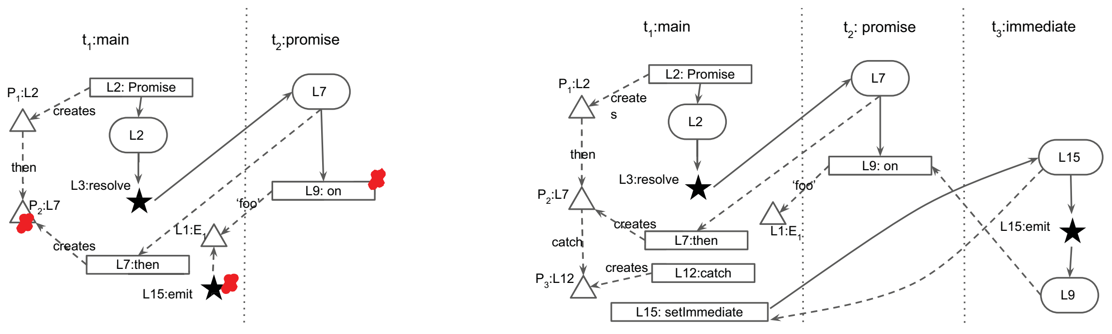
As shown in Fig. 4, the fix for this bug is to delay the event emission using setImmediate, which has a lower priority than the promise (micro-task) queue. Fig. 4 also illustrates the ability of AGs to model the promise chain [15] with object-binding nodes. Promise P_2 is chained from another promise P_1 via $P_1.then$ in line 7. The resolve action ($\bigstar - L3$) reveals that the value is passed from p_1 to p_2 during the CE ($\bigcirc - L2$) scheduled instantly after the CR($\square - L2$) of the promise constructor. As shown in Fig. 4(a), a missing-exception-handling warning is reported for p_2, as there is no catch or any exception handler registered at the end of a promise chain. AsyncG analyzes the structure of the promise chain and is able to raise such warnings without the need to have an actual exception thrown during the execution.

V. Building AG with AsyncG

Here, we describe AsyncG, an automatic debugging tool that dynamically builds and analyzes the AG of a running application to identify potential bugs. AsyncG relies on runtime instrumentation of a Node.js application. First, we introduce the instrumentation technique we are using. Then, we discuss the main steps to build an AG and visualize it.

A. Instrumentation

AsyncG is based on NodeProf [19], an instrumentation framework for Graal.js [20]. Using NodeProf instead of a source-code instrumentation framework such as Jalangi [21] or



(a) AG with promise and listener bugs

(b) fixed AG with setImmediate

Fig. 5. AGs for the code example in Fig. 4

API rewriting [12], [13], [22] has several benefits: (1) Node-Prof directly instruments the internal Abstract Syntax Tree (AST) representation in the JavaScript engine, and is transparent to the application, whereas other approaches need to wrap the original source code, resulting in extra frames on the call stack that are observable by the application. (2) In contrast to other approaches, NodeProf supports the instrumentation of internal libraries of Node.js, allowing us to track the use of asynchronous APIs within library code. (3) NodeProf supports the latest Node.js features, such as e.g. async/await, which are currently not covered by any other framework.

B. Algorithms for Building AG

AsyncG uses NodeProf to intercept all the function calls of the monitored Node.js application. To this end, AsyncG executes the callback functionEnter(func, receiver, args), before each function call (where func is the function being called, receiver is the receiver object, and args is the argument array). The callback functionExit(func, retVal) is executed at the end of a function call (where retVal is the function's return value). The algorithms below show how AsyncG processes function calls to build the AG. Due to space limitations, we only show the main algorithms for identifying event-loop ticks, asynchronous callback registration, and mapping a callback execution to the corresponding callback registration.

Algorithm 1 Shadow stack

```
 1: procedure FUNCTIONENTER(func, receiver, args)
 2:     if sstack.isEmpty() then
 3:         type ← getIterType(func)
 4:         curTick ← new Tick(curTick.idx+1, type)
 5:     sstack.push(func)
 6: procedure FUNCTIONEXIT(func, retVal)
 7:     popped ← sstack.pop()
 8:     assertEqual(poped, func)
 9:     if sstack.empty() and !curTick.nodes.empty() then
10:         graph.appendIter(curTick)
```

1) Identifying Event-loop Ticks: Thanks to the ability to instrument Node.js internal libraries, AsyncG is able to identify the start of an event-loop tick by maintaining a shadow stack, based on the fact that a new event-loop tick t_i starts only when such a shadow stack is empty. As shown in Algorithm 1, the shadow stack is stored in sstack, and the current active tick is stored in curTick, which are global variables exposed also to the other algorithms. If AsyncG is enabled in the middle of the run when the real stack might not be empty, it refers to the current stack trace provided by Node.js, e.g., via console.trace, and finds out when the current tick will finish. Then it will construct the shadow stack from the following tick.

Algorithm 2 Callback registration

```
1: procedure FUNCTIONENTER(func, receiver, args)
2:     if isAsyncAPI(func) then
3:         template ← getAsyncTemplate(func)
4:         regInfo ← template.process(func,receiver,args)
5:         crNode ← createCR(regInfo, □)
6:         curTick.addNode(crNode)
7:         foreach cb ∈ regInfo.cbs do
8:             $L^{cb}_{pending}$.add(crNode)
9:         end foreach
```

2) Callback Registration: Algorithm 2 shows the operations performed by AsyncG for each callback registration. Function isAsyncAPI (line 2) is used to check whether func is an asynchronous API. The common information needed for callback registration is similar, including: (1) which argument refers to the callback function; (2) in which event-loop phase will the callback be scheduled; (3) will the callback be scheduled exactly once (e.g., setImmediate) or can it be scheduled multiple times (e.g., emitter.on); (4) is the function call bound to an emitter or promise object. In AsyncG, we implement different templates for different APIs to fetch such information (lines 3 and 4). Then, a CR (□) node crNode is created (line 5) and included in the current tick curTick of the graph (line 6). Finally, the newly created CR node is pushed to a pending list $L^{cb}_{pending}$, which will be used in Algorithm 3 to map a CE to the corresponding CR.

3) Callback Execution: The last part of the graph building process is to map a callback execution to the corresponding registration, as shown in Algorithm 3. A *context validator* is applied to check if the current execution context (e.g., the type of the current tick, current shadow stack, emitters or promises bound to this function call) matches any pending callback registration in the list $L^{func}_{pending}$. If such a valid registration node crNode exists, it means the current function call is registered by crNode. Then, a new CE node ceNode is created and included in curTick, and an edge (⇠) is created starting from ceNode to crNode showing the registration relation. Another edge (→) to ceNode (showing the causal relation) is also created from crNode or the trigger node (★), if any. Finally, crNode will be removed from $L^{func}_{pending}$ if its callback is supposed to be executed only once.

Algorithm 3 Callback execution and registration mapping

```
1:  procedure FUNCTIONENTER(func, receiver, args)
2:      foreach crNode ∈ $L^{func}_{pending}$ do
3:          if validator.isValid(crNode, sstack, func, receiver, args) then
4:              ceNode ←createCE(crNode)
5:              curTick.addNode(ceNode, ◯)
6:              createEdge(ceNode, crNode, ⇠ )
7:              ctNode = validator.getTriggerNode(crNode)
8:              if ctNode ≠ null then
9:                  curTick.addNode(ctNode, ★)
10:                 createEdge(ctNode, ceNode, →)
11:             else
12:                 createEdge(crNode, ceNode, →)
13:             if crNode.scheduleOnce() then
14:                 $L^{func}_{pending}$.remove(crNode)
15:             break
16:     end foreach
```

C. Visualization

AsyncG can visualize the AG using the DOT language [23], or with a JavaScript library such as D3.js [24] to show the graph in the browser. The graphs shown in Section IV-B are equivalent to the ones generated by AsyncG, but have been manually adjusted to better fit in this paper.

VI. Bug Detection with AsyncG

AsyncG builds and analyzes the AG of any running application. Some bugs can be identified automatically by analyzing the AG on the fly while other problematic code patterns may need application-specific knowledge to be detected, and AsyncG provides a visual representation of the AG to support debugging.

A. Automatic Bug Detection

AsyncG can automatically detect bugs that belong to the following categories:

- *Scheduling bugs* related to misuse of event-scheduling APIs.
- *Emitter bugs* related to emitter objects.
- *Promise bugs* related to promises and async/await.

A description of bugs in each category is given in the following subsections.

1) Scheduling Bugs:

a) Recursive Micro-tasks: As micro-tasks (nextTick, promise) have a higher priority than any other task, recursively scheduling a micro-task can lead to a scenario where callbacks in other phases would never get scheduled. E.g., this kind of bug occurs in line 5 of the example in Fig. 1. It can be identified by tracking the consecutively scheduled micro-tasks and reporting when a cycle is detected (e.g., recursively scheduling the same callback using the process.nextTick API).

b) Mixing Similar APIs: APIs such as process.nextTick, setTimeout(0), or setImmediate are similar but have different scheduling priorities. Mixing the use of these APIs without knowing their slightly different semantics can lead to unexpected behaviors such as the code shown in Section III. AsyncG reports warnings when uses of such APIs within the same tick can lead to a wrong event execution order.

c) Unexpected Timeout Execution Order: Depending on the internal state of the event loop, setTimeout callbacks may be scheduled with an order that is different from the one expected by the developer. For example, consider the following callbacks:

```
setTimeout(foo, 101);
setTimeout(bar, 100);
```

The programmer might expect foo to be scheduled after bar with a higher wait time. However, this is not always the case, because the timer phase may be scheduled after both callbacks have timed out, and the earlier registered callback (foo) will be executed first. The proper way to schedule such callbacks would be to use the same timeout value and make sure the registration of foo comes after bar. AsyncG can report a warning when two setTimeout calls are used in the same tick and the callback registered with a higher timeout is scheduled before any other.

2) Emitter Bugs: AsyncG automatically identifies the following types of bugs related to emitters. While *Dead Emits* and *Dead Listeners* have been introduced in [10], the other problematic code patterns are new, and are identified only by AsyncG.

a) Dead Listeners: This code pattern happens when a developer registers a listener on an emitter that never emits such an event. AsyncG can provide a warning about a dead listener to reveal unexecuted listener callbacks. For example, this kind of bug happens in line 9 of the example in Fig. 4.

b) Dead Emits: When an event is emitted while no listeners have been registered for it, AsyncG reports a bug. This happens when an event has been emitted without a listener, or before the listener is added to the emitter, or the emitted event is illegal. For example, this kind of bug happens in line 15 of the example in Fig. 4.

c) Invalid Listener Removal: The e.removeListener API can be used to remove a given listener from an emitter object e. As discussed on SO-10444077 , a common mistake is that a programmer may try to remove an event listener passing a wrong function, which appears to be the same as the function to be removed. AsyncG detects such invalid listener removals.

d) Duplicate Listeners: This is a warning reported when the same function is registered as a listener for the same event multiple times. It is unlikely developers would intend to do so, and AsyncG warns developers of this bug.

e) Add Listener within Listener: AsyncG reports a warning when an event listener l_a is registered during the execution of another listener l_b of the same emitter. l_a could be lost (never registered) if l_b is never executed. A typical example can be found in SO-17894000.

3) Promise Bugs: AsyncG detects all promise-related bugs introduced in [15]. In addition, AsyncG detects promise bugs related to async/await.

a) Dead Promise: AsyncG detects dead promise objects, namely promises that are never resolved or rejected during the current execution.

b) Missing Reaction: AsyncG detects a promise that is resolved or rejected without any promise reaction (e.g., then, catch, await).

c) Missing Exceptional Reject Reaction: Exceptions that are thrown when executing a promise must be caught by registering an exception handler in the promise chain. AsyncG finds missing exception handlers in promise chains even when the execution does not throw any exception, by always checking that all promise chains end with a reject reaction. This kind of bug happens e.g. in line 12 in Fig. 4.

d) Missing Return: As the return value of the callback registered by Promise.then will be used to resolve the next promise in the chain, not explicitly returning in such callbacks would return a default JavaScript undefined object. AsyncG automatically detects such bugs by checking the return value of the reaction of a promise.

e) Double Resolve or Reject: Once a promise is resolved or rejected, further resolving or rejecting this promise will not take any effect. AsyncG detects such promises with double resolve or reject.

B. Bug Detection Using the AG

Some patterns are not necessarily leading to a bug, and more information is required to debug the root cause. Such bugs can be manually detected by checking the AG produced by AsyncG.

1) Expecting Callbacks to Run Synchronously: A common mistake of asynchronous programming with callbacks is that a programmer can call an asynchronous API and then do something else immediately after the asynchronous call. As the callback will be executed in the successive event-loop ticks, the program may not work as expected. As the execution order can be inferred with the AG, AsyncG reveals such bugs by visualizing the event-loop scheduling.

2) Broken Promise Chain / Unnecessary Promise: Although an application may not show any of the aforementioned bug patterns, its promises may be chained in a wrong way. Users can refer to the promise chain built by AsyncG to identify such bugs.

TABLE I
DETECTED BUGS

Bug name	Categories
SO-38140113	Dead Emits
SO-32559324	Dead Emits
SO-33330277	Recursive Micro Tasks
SO-30515037	Recursive Micro Tasks
SO-50996870	Brokend Promise Chain
SO-28830663	Mixing Similar APIs
SO-30724625	Dead Emits
SO-43422932	Missing Reaction
SO-10444077	Invalid Listener Removal
SO-45881685	Duplicate Listeners
SO-31978347	Expect Sync Callback
GH-vuex-2[2]	Missing Return In Then
GH-flock-13[3]	Missing Exceptional Reaction
GH-npm-12754[4]	Recursive Micro Tasks

VII. EVALUTION

In this section, we first discuss our experience with bug detection using AsyncG. Then, we evaluate the performance of AsyncG on the Node.js server-side benchmark AcmeAir [25].

A. Case Study

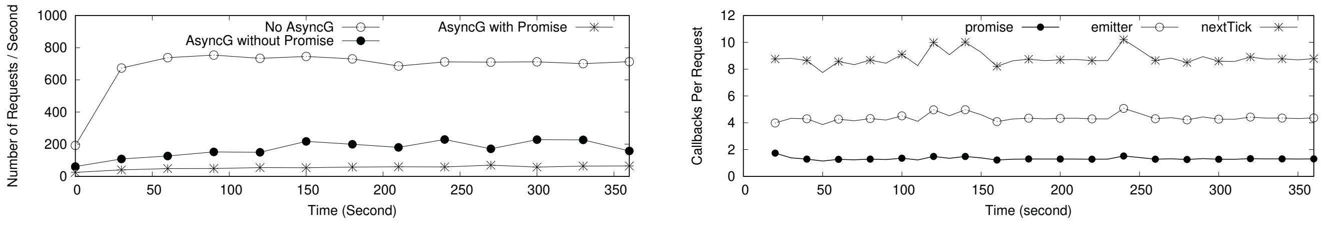
We validate AsyncG with common mistakes of programmers, including 33 questions from SO reported in previous work [10], [15], and 11 other SO questions plus 3 GitHub (GH) issues as shown in Table I. AsyncG locates the cause of these bugs and gives detailed warnings referring to the corresponding code locations. Below we discuss a subset of bugs identified by us:

GH-npm-12754: This Github issue reports a recursive nextTick bug similar to the one in Fig. 1. Such a bug can be detected by AsyncG and the fix is to replace process.nextTick with setImmediate.

SO-28830663: The programmer has problems distinguishing the difference between calling a function directly, using nextTick, or using setImmediate. AsyncG is able to build the AG of the provided code and illustrates the different event-loop ticks in which the callback is executed.

SO-17894000: This is a bug case which adds a listener within another listener when using the network APIs. The problem in the code is that the listener for the “close” event is defined within the “data” event listener. If the connection closes before any data has arrived, the “close” listener will never be executed (as it has not been registered). AsyncG

[2]https://github.com/takefumi-yoshii/vuex-aggregate/issues/2
[3]https://github.com/gradealabs/flock/issues/13
[4]https://github.com/npm/npm/issues/12754



(a) Throughput

(b) Number of callback executions per request

Fig. 6. Running AcmeAir with AsyncG

automatically detects such a pattern.

SO-38140113: The programmer provides two similar code snippets and asks why their behaviors are different. Within both snippets, the programmer defines a constructor function MyEmitter extending EventEmitter and emits an event by calling this.emit("e") within the constructor. Then, the developer defines a listener for the "e" outside the constructor. The difference between the snippets is that the first one directly calls this.emit("e"), while the second calls it inside a nextTick callback. AsyncG detects that in the first case, the this.emit("e") is a dead emit because it is called before the listener is registered. The second snippet does not suffer from this bug because the event emission happens in the following tick after the listener registration.

SO-50996870: The programmer uses promises for database accesses. However, the promise chain is broken due to a missing return in one of the reaction callbacks. AsyncG automatically locates the reaction callback without return, and pinpoints it in the promise chain.

SO-43422983: The programmer forgets to add the keyword await when calling the async function which fetches a JSON object after a timeout. Thus, the promise object instead of the JSON value to be resolved by the promise is used by mistake for further processing. AsyncG is able to report a missing reaction bug for the promise returned by the async function which is never resolved, rejected, or awaited.

GH-vuex-2: The programmer creates a list of functions, each of them executes multiple asynchronous tasks and creates a single promise. However, the programmer forgot to use such promises. AsyncG reports the bug by locating these promises without any reaction.

B. AcmeAir Benchmark

We evaluate the overhead of AsyncG with the Node.js server-side benchmark AcmeAir [25]. AcmeAir is a flight booking system with a server-side backend implemented in Node.js. AcmeAir is a good benchmark to evaluate AsyncG, as it mixes the use of different asynchronous APIs. By default, AcmeAir uses no promises. To evaluate the promise feature of AsyncG, we slightly modify AcmeAir's source code to use the promise-version interface for mongodb [27] access. The measurements are collected with the JMeter [28] test suite of AcmeAir simulating realistic workloads on the server. Our evaluation is performed on an Intel(R) Xeon(R) E5-2680 CPU with 8 physical cores (16 virtual cores) running at 2.7 GHz, equipped with 128 GB RAM, running Ubuntu Server release 16.04 (kernel version 4.4.0-112-generic). AsyncG uses NodeProf (August 2018) [29] with GraalVM 1.00-rc5 [30].

We evaluate the performance by measuring the throughput of the server in terms of the number of client requests processed per second. The result is shown in Fig. 6(a). We can see that running AsyncG tracking all asynchronous APIs makes the server 10 times slower, while excluding promise-related features introduces around 2 times overhead. As the server is heavily exercising asynchronous APIs, the overhead caused by the dynamic analysis to build and analyze the AG is significant. Such performance overhead is expected, and we consider it comparable with the overhead of other debugging tools. Moreover, such overhead has to be paid only when AsyncG is enabled. Since AsyncG is pluggable, it can be enabled/disabled at runtime. Once disabled, AsyncG introduces no overhead thanks to NodeProf [19].

Fig. 6(b) shows how frequently asynchrounous APIs are used with the average number of callback executions per client request for the most used asynchronous APIs: process.nextTick, emitter, and promise. On average, a client request requires 1.31 promises for database accesses, 4.31 emitter executions for I/O, and about 8.70 nextTick executions during the database access and I/O.

VIII. Related Work

In this section, we differentiate our work from other related approaches.

A formalization of the semantics of the Node.js event loop is presented in [16]. Our work relies on such a formalization, and AsyncG can be used to generate a visual representation of such a model for a given Node.js application.

Other graph-based visualizations of Node.js applications have been proposed. In [15], the authors propose *Promise Graphs* as a model to reason about Node.js promises. Based on promise graphs, PromiseKeeper [26] is a tool that can automatically find promise-related bugs. Compared to such approaches, our work is not limited to promises, and is able to capture all sources of asynchronous executions in Node.js.

TABLE II
COMPARISON WITH RELATED WORK

Work	Methods	Event Loop	Emitter	Promise	Async/ Await	Tool Availability	Full Coverage	Automatic Bug Detection
Semantics [16]	Modelling	Y	N	N	N	/	/	N
PromiseKeeper [26]	Dynamic	N	N	Y	N	Y	N	Y
Radar [10]	Static	N	Y	N	N	N	Y	Y
Clematis [22]	Dynamic	N	N	N	N	Y	N	N
Sahand [12]	Dynamic	N	N	N	N	Y	N	N
Domino [13]	Dynamic	N	N	Y	N	N	N	N
Jardis [14]	Dynamic	N	Y	Y	N	Y	Y	N
AsyncG	Dynamic	Y	Y	Y	Y	Y	Y	Y

In [10] the tool Radar is introduced. Radar uses static analysis to build an *event-based call graph* to model the event-driven control-flow of asynchronous JavaScript applications. The model does not differentiate asynchronous API uses according to the event-loop semantics. In addition, as JavaScript is a dynamic language, static analysis approach like Radar can over-estimate potential interleavings and face the problem of state explosion in larger applications.

Clematis [22] is the first tool using dynamic analysis for visualizing the event-based interactions in JavaScript web application. Compared to AsyncG, Clematis focuses only on client-side applications without covering the complexity of the Node.js event loop. Sahand [12] and Domino [13] are similar tools to visualize interactions between client and server JavaScript applications. They rewrite (wrap) some of the JavaScript APIs to track asynchronous callback execution and show the interactions between client and server with timelines in their graph. Jardis [14] is a debugger extension for Visual Studio Code IDE [31]. Jardis provides only call-stack information within the IDE. These tools cannot tell different event-loop phases in their graph to find bugs related to the complicated event-loop scheduling, and they do not model the promises and emitters as we do in AsyncG. In addition, none of these tools is able to instrument the Node.js internal libraries and thus cannot detect internal asynchronous executions.

A detailed comparison of the tools is shown in Table II. In summary, (1) We not only provide a model for the asynchronous events in Node.js, but also a tool implementing the model. (2) AsyncG is a dynamic analysis tool which is able to deal with the dynamic nature of JavaScript. (3) Compared to existing tools, AsyncG is able to visualize the event-loop scheduling and supports more APIs such as emitter, promise and async/await.

IX. CONCLUSION

In this paper, we have introduced Async Graph, a novel model that allows one to reason about the asynchronous control flow of Node.js applications. We have introduced AsyncG, a tool that automatically builds the AG of a Node.js application at runtime. AsyncG is a debugger that helps developers understand the asynchronous control flow of Node.js applications. AsyncG analyzes the built AG to automatically detect bugs related to all sources of asynchronous execution.

In ongoing research, we are extending AsyncG with data flow analysis to automatically detect race conditions caused by non-deterministic event ordering in Node.js.

ARTIFACT APPENDIX

A. Abstract

In this artifact, we provide AsyncG both as a binary executable integrated with a customized GraalVM (which includes our open-source dynamic instrumentation framework NodeProf) and as an open-source release, together with the scripts to reproduce the data for the graphs in the paper. We have also set up a website to visualize the generated Async Graphs (AGs) based on the output of AsyncG. The website not only includes a few examples illustrating the bug categories mentioned in the paper, but also allows one to upload the output generated by AsyncG when analyzing any custom Node.js application. The open-source release and more detailed information can be found on GitHub: https://github.com/Haiyang-Sun/AsyncG.

B. Artifact Check-list

- **Program:** We provide AsyncG both as a binary and as an open-source release. As mentioned in the paper, the AcmeAir benchmark has been slightly modified to use the promise interface of mongodb. The code is available on GitHub: https://github.com/Haiyang-Sun/acmeair-nodejs.
- **Binary:** We build the AsyncG implementation together with the GraalVM [30]. This customized GraalVM, which also includes our instrumentation framework NodeProf [19], [29], can run AsyncG by simply setting some environment variables. The user can run AsyncG for any Node.js program, producing an AG. The AG can be visualized in our tool's website: https://asyncgraph.github.io/. The website is fully static, so the user may download it and deploy it on his own server.
- **Data Set:**
  1) We provide a list of code examples (including two examples similar to those used in the paper) with generated AGs on our website. The user may reproduce the same AG by running the example code with AsyncG and submitting the output log to the website.
  2) For the AcmeAir benchmark, we use the default data set from the AcmeAir benchmark driver to run with JMeter.
- **Run-time Environment:**

1) System: The provided AsyncG binary is for Linux (Ubuntu 16.04 or Ubuntu 18.04 is recommended) x86-64.
2) To run our script for the AcmeAir benchmark, the following software has to be installed: curl, mongodb, gnuplot.
3) JMeter 4.0 is used to run the AcmeAir benchmark driver.

- **Hardware:** A machine equiped with at least 16 GB memory is recommended. (We have repeated our experiments with three different machines with different specs.)
- **Execution:** A script is given to run AsyncG on any Node.js source code to dump the AG into a log file. The log file can be uploaded to our website for visualization. The website includes several pre-configured examples which can be evaluated directly online. To run the AcmeAir benchmark, the user has to start the server first, and then use JMeter to generate the workload input. To reproduce Fig. 6(a), the user has to measure three different settings (each will take about 10 minutes), and integrate the measurements into a single figure. To reproduce Fig. 6(b), the user needs the API usage information dumped by the server process and the JMeter log with detailed HTTP client request.
- **Metrics:** The throughput shown in Fig. 6(a) is the number of requests processed per second, and the API usage shown in Fig. 6(b) is the number of API callback executions per client request.
- **Output:** The graphs for the example code are dumped into logs and can be visualized on the website. Our script will automatically execute the benchmark and generate Fig. 6(a) and Fig. 6(b).
- **Experiments:** The evaluation is divided into two parts: the first part shows the AGs generated by AsyncG; the second part is to reproduce Fig. 6.
- **Disk Space Required:** At least 2 GB of disk space is recommended to store the binary and the dumped log.
- **Time Needed to Prepare the Workflow:** About 10 minutes, depending on the network speed.
- **Time Needed to Complete the Experiments:** About half an hour.
- **Public Availability:** Yes. The tool is open-source and more detailed information can be found on GitHub: https://github.com/Haiyang-Sun/AsyncG.

C. Description

1) How Delivered: Information and scripts needed to run AsyncG are available in our GitHub repository: https://github.com/Haiyang-Sun/AsyncG.

2) Hardware Dependencies: We recommend testing on a machine with at least 16 GB memory.

3) Software Dependencies: We evaluate our script on our servers running Ubuntu 16.04 or 18.04 x86-64. The following software needs to be installed: *curl*, *mongodb*, *JDK 8*, and *gnuplot*. Other dependencies such as JMeter will be downloaded by the script *setup.sh* (included in our GitHub repository).

4) Data Sets: The default data set of the AcmeAir benchmark is used to measure throughput. A set of code examples can be found in *code-examples* in our repository.

D. Installation

```
sudo apt-get install curl mongodb gnuplot
git clone https://github.com/Haiyang-Sun/AsyncG.
    git
cd AsyncG
./setup.sh
```

E. Experiment Workflow

The visualization of a list of code examples can be found on our website: https://asyncgraph.github.io/. The code examples can be found in *code-examples* in our repository. To reproduce the AGs generated for these examples, simply run *./scripts/runExamples.sh*. The resulting logs together with the source code in the same folder can be uploaded and visualized on our website.

To reproduce Fig. 6(a) and Fig. 6(b), simply run *./scripts/figure6.sh*. The script will run the AcmeAir benchmark with the three different settings. The raw logs are saved in *./logs* (starting with three settings' names, i.e., baseline, nopromise, withpromise). Each setting will need about 10 minutes to finish, so the overall execution of this script will be about 30 minutes. Finally, the script will use the logs generated to plot the two figures in eps format and save them in the *figures* directory.

F. Evaluation and Expected Result

The website can visualize the AGs as described in the paper. The website not only shows the AGs, but also highlights some of the warnings.

The throughput figure generated (*figures/figure6a.eps*) should reveal the peak throughput with a slowdown of around 10x when using AsyncG with the promise feature enabled. Please note that we have added some extra features to AsyncG and upgraded the version of GraalVM. In addition, hardware configurations may affect the performance experiments. Hence, the performance numbers can be somewhat different from the paper.

Fig. 6(b) should reveal similar numbers of the asynchrnous events per request as in the paper.

We also include the figures of the results obtained by us on three different machines in the folder *provided_results* of our GitHub repository:

1) *x1carbon* has 16 GB memory and an Intel 4-core CPU i7-8650U (1.9 GHz) running Ubuntu 18.04 x86-64 Desktop LTS.
2) *p51* has 16 GB memory and an Intel 4-core CPU E3-1505M (3.0 GHz) running Ubuntu 18.04 x86-64 Desktop LTS.
3) *dellserver* has 128 GB memory and an Intel 8-core CPU E5-2680 (2.7 GHz) running Ubuntu 14.04 Server LTS.

G. Experiment Customization

The customization of the AcmeAir benchmark can be found on GitHub: https://github.com/acmeair/acmeair-driver. The user can run AsyncG with any valid Node.js code, upload the log, and check the AG as visualized on the website.

Acknowledgement

The work presented in this paper was supported by Oracle (ERO project 1332). We thank the VM Research Group at Oracle for their support. Oracle, Java, and HotSpot are trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

References

[1] "About - Node.js," 2018. [Online]. Available: https://nodejs.org/en/about/
[2] "Reactive Streams," 2018. [Online]. Available: http://www.reactive-streams.org/
[3] "Swift.org - Welcome to Swift.org," 2018. [Online]. Available: https://swift.org/
[4] "The Go Memory Model - The Go Programming Language," 2018. [Online]. Available: https://golang.org/ref/mem
[5] "The Node.js Event Loop, Timers, and process.nextTick()," 2018. [Online]. Available: https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/
[6] "Events," 2018. [Online]. Available: https://nodejs.org/api/events.html
[7] "ECMAScript 2015 Language Specification (ECMA-262 6th Edition)," 2015. [Online]. Available: https://www.ecma-international.org/ecma-262/6.0/
[8] "ECMAScript 2017 Language Specification (ECMA-262 8th Edition)," 2017. [Online]. Available: https://www.ecma-international.org/ecma-262/8.0/

[9] Y. Zheng, T. Bao, and X. Zhang, "Statically locating web application bugs caused by asynchronous calls," in *Proceedings of the 20th international conference on World wide web*, 2011, pp. 805–814.

[10] M. Madsen, F. Tip, and O. Lhoták, "Static analysis of event-driven node.js javascript applications," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2015, pp. 505–519.

[11] E. Andreasen, L. Gong, A. Møller, M. Pradel, M. Selakovic, K. Sen, and C.-A. Staicu, "A survey of dynamic analysis and test generation for javascript," *ACM Computing Surveys (CSUR)*, vol. 50, no. 5, p. 66, 2017.

[12] S. Alimadadi, A. Mesbah, and K. Pattabiraman, "Understanding asynchronous interactions in full-stack javascript," in *IEEE/ACM 38th International Conference on Software Engineering*, 2016, pp. 1169–1180.

[13] D. Li, J. Mickens, S. Nath, and L. Ravindranath, "Domino: Understanding wide-area, asynchronous event causality in web applications," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, 2015, pp. 182–188.

[14] E. T. Barr, M. Marron, E. Maurer, D. Moseley, and G. Seth, "Time-travel debugging for javascript/node.js," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 1003–1007.

[15] M. Madsen, O. Lhoták, and F. Tip, "A model for reasoning about javascript promises," *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, vol. 1, pp. 86:1–86:24, 2017.

[16] M. C. Loring, M. Marron, and D. Leijen, "Semantics of asynchronous javascript," in *Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages*, 2017, pp. 51–62.

[17] V. Raychev, M. Vechev, and M. Sridharan, "Effective race detection for event-driven programs," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, 2013, pp. 151–166.

[18] J. Davis, A. Thekumparampil, and D. Lee, "Node.fz: Fuzzing the server-side event-driven architecture," in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 145–160.

[19] H. Sun, D. Bonetta, C. Humer, and W. Binder, "Efficient dynamic analysis for node.js," in *Proceedings of the 27th International Conference on Compiler Construction*, 2018, pp. 196–206.

[20] T. Würthinger, C. Wimmer, C. Humer, A. Wöß, L. Stadler, C. Seaton, G. Duboscq, D. Simon, and M. Grimmer, "Practical partial evaluation for high-performance dynamic language runtimes," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, pp. 662–676.

[21] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: A selective record-replay and dynamic analysis framework for javascript," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 488–498.

[22] S. Alimadadi, S. Sequeira, A. Mesbah, and K. Pattabiraman, "Understanding javascript event-based interactions," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 367–377.

[23] "The DOT language," 2018. [Online]. Available: https://www.graphviz.org/doc/info/lang.html

[24] "D3.js, Data-Driven Documents," 2018. [Online]. Available: https://d3js.org/

[25] "Acme Air," 2018. [Online]. Available: https://github.com/acmeair/acmeair-nodejs

[26] S. Alimadadi, D. Zhong, M. Madsen, and F. Tip, "Finding broken promises in asynchronous javascript programs," *OOPSLA*, Oct. 2018.

[27] "MongoDB," 2018. [Online]. Available: https://www.mongodb.com/

[28] "Apache JMeter," 2018. [Online]. Available: http://jmeter.apache.org/

[29] "NodeProf Github," 2018. [Online]. Available: https://github.com/Haiyang-Sun/nodeprof.js

[30] "GraalVM," 2018. [Online]. Available: https://www.graalvm.org/

[31] "Visual Studio Code - Code Editing. Redefined," 2018. [Online]. Available: https://code.visualstudio.com/