NodeMOP: Runtime Verification for Node.js Applications

Filippo Schiavio
Haiyang Sun
filippo.schiavio@usi.ch
haiyang.sun@usi.ch
Università della Svizzera Italiana
(USI), Switzerland

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

Andrea Rosà
Walter Binder
andrea.rosa@usi.ch
walter.binder@usi.ch
Università della Svizzera Italiana
(USI), Switzerland

ABSTRACT

Node.js has become one of the most popular frameworks for general-purpose and server-side application development in JavaScript. However, due to its dynamic, asynchronous, event-driven programming model, Node.js applications are considered error-prone, and their correctness is hard to verify. Monitoring-Oriented Programming (MOP) is a Runtime Verification (RV) paradigm that aims at improving the safety and reliability of a software system. To the best of our knowledge, no practical RV framework targets JavaScript and Node.js applications.

In this paper, we introduce NodeMOP, a novel RV framework for JavaScript that allows one to apply RV to Node.js applications. Using NodeMOP, we have formalized two properties related to popular asynchronous APIs based on the Node.js documentation, one from the file-system module and the other from the HTTP module. NodeMOP also supports error recovery by allowing developers to define custom handlers in case of property violations. We showcase NodeMOP with our specified properties on examples of Node.js API misuse. We also evaluate the overhead of NodeMOP with benchmarks based on the introduced examples.

CCS CONCEPTS

• **Software and its engineering** → **Correctness**; • **General and reference** → Verification;

KEYWORDS

Runtime Verification, Monitoring-Oriented Programming, JavaScript, Node.js, Dynamic Analysis, Software Verification, Self-Healing Systems.

ACM Reference Format:
Filippo Schiavio, Haiyang Sun, Daniele Bonetta, Andrea Rosà, and Walter Binder. 2019. NodeMOP: Runtime Verification for Node.js Applications. In *The 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19), April 8–12, 2019, Limassol, Cyprus.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3297280.3297456

1 INTRODUCTION

With the growing popularity of Node.js [16], JavaScript has become one of the most popular programming languages, and its popularity goes beyond client-side web development, towards server-side and general-purpose applications (e.g., databases, desktop and mobile applications, Internet of Things, etc.).

Despite its popularity, programming in JavaScript is known to be error-prone [1]. The asynchronous, event-driven, programming model of Node.js often leads to subtle bugs and unexpected application behaviors. Though executed in a single thread, JavaScript code is not free of nondeterminism and race conditions [22, 36]. Furthermore, even though the JavaScript syntax is simple and easy to learn, the language semantics is complex and often counter-intuitive [2, 28]. The JavaScript language is highly dynamic (e.g., developers can evaluate code at runtime, use several forms of meta-programming, etc.) and permissive (e.g., no-crash philosophy [7], automatic type coercion [37], etc.) Although such properties are often considered as advantages by JavaScript developers, they can also easily hide errors [7]. Furthermore, thanks to the NPM module ecosystem [34], developers often build complex applications by using hundreds of existing, ready-to-use, third-party software components. Unfortunately, such appealing modularity often leads to issues, as developers need to rely on the correctness of such third-party code that may introduce unexpected bugs. Based on these considerations, we believe it is crucial to have a system that allows Node.js developers to verify the correctness of programs and modules they depend on.

Runtime Verification (RV) is a dynamic software analysis technique that deals with how to verify that the observable behaviour of a program execution satisfies certain (given) formal correctness properties [27]. Runtime execution monitoring is used in RV to detect the violations of such properties, as well as to take appropriate *reactions* to avoid system failures (e.g., by adopting error recovery strategies). Monitoring-Oriented Programming (MOP) [10, 12] is a popular approach to RV where developers are required to specify complex correctness properties about certain aspects of an application. Such properties can be specified using different formalisms (which may vary between MOP implementations), together with some executable code to be executed when a running application violates (or adheres to) the given specification. MOP frameworks usually allow one to define formal specifications with the support of a domain-specific language, and rely on instrumentation techniques to verify that applications do not violate such specifications. As an example, the JavaMOP RV framework [11, 12] allows the developer to define specifications based on different formalisms, namely: finite state machines (FSM), regular expressions (RE), linear temporal logic (LTL), and string rewriting systems (SRS).

Existing MOP frameworks target statically-typed languages, and rely on instrumentation techniques to weave the original application code with a layer that verifies the given properties at runtime, eventually triggering appropriate error recovery actions in order to prevent system failures. As an example, MOP tools that target the Java programming language weave the monitoring code at compilation time[1] using the AspectJ weaver [23], an Aspect-Oriented Programming (AOP) [15] framework for the Java Virtual Machine (JVM).

Given the wide adoption of JavaScript in server-side applications, we believe that a MOP framework for Node.js could bring several benefits to new and existing applications. Example usages of a MOP framework for Node.js could include enforcing coding practices, web services self-healing [8, 9], as well as automatic bug detection [21]. Moreover, monitoring the execution of Node.js applications can ensure security constraints that are challenging to verify, e.g., dynamic taint analysis [33].

In this paper, we introduce NodeMOP, a practical MOP framework for Node.js applications. NodeMOP can be used to perform RV of existing Node.js applications, and supports the full JavaScript specification, allowing developers to specify monitors targeting all Node.js built-in APIs. NodeMOP is implemented on top of NodeProf [39], an efficient dynamic program analysis framework for Node.js based on Graal.js [26, 45], an ECMA2018-compatible JavaScript engine included in the GraalVM [25] polyglot language runtime with full support for Node.js applications. Furthermore, NodeMOP offers a set of high-level, JavaScript-specific, execution events that can be monitored, and allows developers to define custom actions to be executed when a property specification is violated, enabling automatic recovery from a detected flawed code pattern to prevent a failure.

This paper makes the following contributions:

- We introduce NodeMOP, the first practical MOP framework supporting event-driven programming in JavaScript and Node.js applications.
- We define two correctness properties derived from the Node.js documentation [17, 18] and we provide NodeMOP monitor implementations that verify such properties, together with an error recovery strategy.
- We provide a preliminary evaluation of NodeMOP, conducted on a set of benchmarks including both correct and flawed code patterns, showing that the lowest slowdown factor we have collected is 1.039x and the average is 1.261x.

This paper is structured as follows. Section 2 provides background information on RV and MOP. Section 3 motivates the need of NodeMOP and presents two use cases. Section 4 discusses the NodeMOP API and provides a monitor implementation for the introduced examples. Section 5 shows how monitors can be used to handle a potential application crash through specific recovery actions and presents preliminary evaluation results. Section 6 compares NodeMOP with related works. Finally, Section 7 discusses future research directions and concludes.

[1]JavaMOP can also weave the code at load-time using a Java Agent; however, such approach has worse performance than compilation-time weaving [44].

2 BACKGROUND

Runtime Verification (RV). RV is a dynamic software analysis technique that aims at improving software safety and reliability, complementing static verification and testing. RV dynamically verifies whether a trace (i.e., the observed runtime behavior of a running program) conforms to a given property (formally specified using one or more formalisms). Such properties are verified by means of monitors, automatic code generation, and instrumentation. A *monitor* is a runtime event listener which is generated by the MOP framework according to a given specification and is automatically attached to the monitored code by means of instrumentation. In this way, monitors are able to observe all runtime values and events that are required in order to verify the correct behaviour of the application.

Parametric Runtime Verification. Parametricity [32] is an important property of RV systems, meaning that traces are sliced [13] based on parametric properties of the specification (i.e., properties that are bound to values intercepted at runtime). Specifically, parametric RV systems maintain a separate trace for each parameter binding and verify that each trace, taken singularly, verifies the specification.

JavaScript Instrumentation. Runtime instrumentation is used in RV frameworks to generate correct monitors and check for runtime events efficiently. NodeMOP is built on NodeProf [39], a Node.js instrumentation framework for GraalVM [25] that provides the following properties:

1. *Performance.* In contrast to other approaches based on source-code transformations (e.g., Jalangi [38] or Java ASM [42]), NodeProf's instrumentation overhead is proportional to the size of the instrumented code. Monitoring a few selected functions leads to minimal runtime overhead.
2. *Coverage.* NodeProf is able to cover all executed code, including Node.js internal code. In this way, NodeMOP can monitor APIs related to file system, networking, etc.
3. *ECMAScript features supported.* NodeProf is based on Graal.js and is up-to-date with the latest ECMA2018 features [43].

3 MOTIVATING EXAMPLE

Manually verifying the correctness of asynchronous programs can be hard and tedious. Hence, it is crucial to provide an RV framework for JavaScript and Node.js applications to enforce the proper usage of APIs. In this section, we discuss two examples of improper usage of Node.js asynchronous APIs, and we use them throughout the paper to illustrate the design of NodeMOP, showing how RV can be applied to prevent runtime failures. The two problematic examples could both be fixed by ensuring that the Node.js asynchronous APIs are used as prescribed by the Node.js documentation. In other words, both issues would be solved if both examples would respect a formally-specified set of execution constraints for such Node.js APIs.

3.1 Example 1: Asynchronous file writes

The code in Figure 1 attempts to write a sequence of numbers to a file. It first opens a file descriptor (line 4) and then uses the

```
1 const fs = require('fs')
2 const limit = 2**24;
3
4 fs.open(tmp, 'w', (err, fd) => {
5   if (! err )
6     for (let i=0; i<limit; i++)
7       fs.write(fd, i+'\n', () => {})
8   fs.close(fd, () => {})
9 })
```

Figure 1: Flawed `fs.write` usage [6].

Node.js asynchronous `fs.write` API multiple times to write each number (line 7). The code is problematic according to the Node.js API documentation of the file-system module [17], stating that "it is unsafe to use `fs.write` multiple times on the same file without waiting for the callback" [6].

In detail, the code in Figure 1 presents multiple issues:

(1) Due to the interaction with the OS, the order of the asynchronous `write` operations is unpredictable, and thus the program's behaviour is nondeterministic.
(2) The asynchronous `close` operation may be executed before the last `write`. Such out-of-order execution would raise a runtime exception that might crash the Node.js process and thus lead to an abrupt service termination.
(3) The program might also crash due to an out-of-memory error, because too many asynchronous calls are registered on the Node.js' event loop.

Note that the second and the third of the above issues are critical even if the first one (i.e., nondeterministic order of writes) might not be considered problematic for some specific applications (e.g., such order may not be important in cases where the order in which values are saved is not important). This simple example highlights some of the issues that Node.js developers face. Here, enforcing and verifying a correct usage of the `fs` API [17] would help developers identify potential bugs.

Using NodeMOP, developers can implement monitors to detect the flawed code patterns. Moreover, they can specify an error recovery strategy for each pattern. Specifically, a recovery monitor could solve the issue in Example 1 by automatically invoking the synchronous version of `fs.write` (i.e., `fs.writeSync`) in place of `fs.write`. A property specification in NodeMOP could then be used to detect asynchronous invocations of the `fs.write` function targeting the same file descriptor[2] and correlate the respective callbacks together on the same monitor instance.

More formally, the correct API usage pattern for writing the same file multiple times can be formalized with the following regular expression:

$$(\mathit{write}\ \mathit{writeCB})^*$$

where the *write* event represents the invocation of `fs.write`, and *writeCB* represents the execution of the `fs.write` callback.

[2]The monitor can be extended to handle multiple file descriptors pointing to the same file (i.e., symlinks). We omit the code for such monitor due to space constraints.

```
1 function sendRequest() {
2   http.get('http://localhost:8080', (res) => {
3     const { statusCode } = res;
4     if (statusCode !== 200) {
5       console.error("Request Failed");
6     }
7   });
8 }
```

Figure 2: Flawed `http.get` usage.

Note that the provided regular expression by itself is not expressive enough to model the correct behavior in case that an application writes to two or more different files. For example, consider the following sequence of file write operations:

$$\mathit{write}(1)\ \mathit{write}(2)\ \mathit{writeCB}(1)\ \mathit{writeCB}(2)$$

where 1 and 2 denote different file descriptors that point to different files. Without taking into account the parameter value, the provided RE rejects the above string, since:

$$\mathit{write}\ \mathit{write}\ \mathit{writeCB}\ \mathit{writeCB} \notin (\mathit{write}\ \mathit{writeCB})^*$$

To capture the correct behaviour, the RV implementation needs to be parametric with respect to the function arguments [12]. As introduced in Section 2, parametric traces are sliced based on the binding parameters and verified singularly. Taking into account the parametric property, the sequence above generates two equals strings (*write writeCB*) and both of them satisfy the specification $(\mathit{write}\ \mathit{writeCB})^*$. The next section describes how these parametric properties can be intercepted with NodeMOP.

3.2 Example 2: HTTP client response

The code in Figure 2 performs an HTTP GET request (line 2), and verifies that the request has succeeded by checking that the HTTP response status code is 200 (line 4). The application in the figure contains a potential memory leak. As mentioned in the Node.js `http` module API documentation [18]: "if a 'response' event handler is added, then the data from the response object must be consumed [...]. Also, until the data is read it will consume memory that can eventually lead to a 'process out of memory' error." Indeed, Figure 2 is an example of such a flawed code pattern. The underlying problem is that the garbage collector cannot clean the data buffer, causing a memory leak.

This example of flawed code could be fixed in different ways. Specifically, the Node.js API documentation says that there are three different ways for reading the data from an HTTP response within the respective callback: adding the `.on("data")` event listener or calling either `.read()` or `.resume()` [18]. Therefore, NodeMOP needs to be able to detect multiple API usage patterns. This can be formalized in NodeMOP with the following regular expression:

$$\mathit{req}\ \ \mathit{reqCbPre}\ (\mathit{read} \mid \mathit{resume} \mid \mathit{listener})^+\ \ \mathit{reqCbPost}$$

where *req* is the event associated with the asynchronous function call to `http.get`; *reqCbPre* and *reqCbPost* are the enter and exit events of the related callback execution, and *read*, *resume*, *listener* are the events related to the three different ways of reading the response data as described in the Node.js documentation.

In order to recover from the issues caused by executing a flawed code pattern, a monitor implementation can automatically execute method `resume()` on the response object when a flawed code pattern is detected.

4 API DESCRIPTION AND USAGE EXAMPLES

Here, we first introduce the NodeMOP syntax for event and monitor definition. Then, we show how to define a monitor in NodeMOP that can detect and recover from the flawed code patterns presented in the previous section.

4.1 NodeMOP API

The events that NodeMOP is able to track are related to function invocation and execution. NodeMOP monitors can be developed targeting the following runtime events:

- `invokeFunPre`: executed on the call site, before function call;
- `invokeFunPost`: executed on the call site, after function call;
- `execFunPre`: executed before the function body;
- `execFunPost`: executed after the function body;
- `invokeAsyncFunPre`: variation of `invokeFunPre` that allows tracking callbacks;
- `invokeAsyncFunPost`: variation of `invokeFunPost` that allows tracking callbacks;
- `execCallBackPre`: variation of `execFunPre` that allows tracking the callback executions of asynchronous functions which are tracked with `invokeAsyncFun[Pre|Post]`;
- `execCallBackPost`: variation of `execFunPost` that allows tracking the callbacks execution of asynchronous functions which are tracked with `invokeAsyncFun[Pre|Post]`.

Except for callback executions events (i.e., `execCallBackPre` and `execCallBackPost`), all event tracking APIs in NodeMOP take two arguments, namely the function object that shall be instrumented (as a shortcut, it can be a single function or an array of functions), and an object which specifies different optional attributes, as listed below:

- `id`: function object which uniquely identifies the monitor instance that shall receive the event, based on the intercepted function's arguments;
- `target`: function object which identifies the set of monitor instances that shall receive the event, based on the intercepted receiver of the call;
- `condition`: function object which returns `true` if the event has to be fired;
- `action`: function object which defines an optional action to be executed when the event is fired.

The `id` and `target` attributes are provided to manage parametric properties. The next subsection exemplifies how the attributes `id` and `target` can be used to intercept the parametric properties.

The callback execution events (`execCallBackPre` and `execCallBackPost`) are special cases. Instead of a function object they take a string as first argument, specifying the name of the related asynchronous invocation event. This design choice allows to track all the callback executions even without a reference to the function that has to be woven, e.g., anonymous functions directly passed as callback parameter. The second argument is an object which defines the `condition` and `action` functions as introduced above. Note that callback events do not need to specify the `id` and `target` functions, because the monitor instances which shall receive the events can be obtained from the related asynchronous invocation event.

```
<RE Syntax> ::=
    <Event Name>
  | <RE Syntax> "*"              // zero or more
  | <RE Syntax> "+"              // one or more
  | <RE Syntax> <RE Syntax>      // concatenation
  | <RE Syntax> "|" <RE Syntax>  // union
  | "(" <RE Syntax> ")"          // grouping
```

Figure 3: Grammar defining the NodeMOP RE syntax. Operators are listed in descending order with respect to their precedence

Monitor definitions in NodeMOP consist of a JavaScript object with the following attributes:

- `events`: an object in which each attribute represents an event. The attribute name defines the event name and the corresponding value is an event definition as explained above.
- `re`: monitor specification as RE, defined by the grammar in Figure 3;
- `fsm`: monitor specification as FSM, defined as a JavaScript object with two attributes: `transitions` and `finals`. FSM states are represented as integers, the final states as an array of integers and the transitions as an array of objects in the form `{event_name: next_state}`, where the *nth* position of the array specifies the outgoing transitions from the state *n*;
- `fail`: function object defining the action to be executed if the specification is violated;
- `match`: function object defining the action to be executed if the specification is validated.

Among the attributes listed above only `events` is mandatory. Furthermore, only one specification can be defined, either `re` or `fsm`. It is also possible not to define a specification, in this case the user can manage the monitor behavior by defining JavaScript function using the `action` attribute in the definition of `events`.

Both functions `fail` and `match` may return an object that allows the monitor to alter the original application control flow. In this way, a monitor can skip the execution of the original instrumented function that causes a violation or validation of the specification, or return a different value (only if the execution has been skipped). These features can be used by returning an object with the attributes `{skip: true}` and, optionally, `{ret: <value>}`. Developers can also manually dispose monitor instances if they are not useful anymore, by calling `monitor.destroy()` within event actions or within functions `fail` and `match`.

4.2 Usage Examples

Two examples of monitors are provided in Figure 4 and Figure 5. The code examples show how to define a NodeMOP monitor for detecting and recovering from the flawed code patterns introduced in the previous section.

In this paragraph we describe the NodeMOP monitor implementation reported in Figure 4, which can detect and recover the problematic code pattern introduced in Example 1. In order to monitor such code pattern, two events need to be tracked, i.e., the asynchronous invocation of function `fs.write` (line 5) and its associated callback (line 9). Furthermore, the monitored events need to be parametric on the file descriptor (i.e., the first argument of the function `fs.write`). This parametric property is identified using the `id` attribute (line 7), which defines a function that returns the first argument of the function `fs.write`, that is the file descriptor. The function `fail` (lines 15-29) defines the error recovery strategy. As introduced in Section 3.1, the flawed code is recovered by invoking function `fs.writeSync` instead of function `fs.write`. Furthermore, the original callback must be called to preserve the original application semantics. Such callback, obtained from the original function invocation arguments (line 17), is manually called after the invocation of `fs.writeSync` (line 21).

The problematic code pattern introduced in Example 2 is detected and recovered with the monitor shown in Figure 5 and described in this paragraph. As introduced in Section 3.2, the events that have to be tracked are the invocation of the asynchronous function `http.get` (line 6), the beginning (line 8) and the end (line 27) of the execution of the related callback, and the invocation of functions `read` (line 14), `resume` (line 17) and `on('data')` (line 21) on the response object. In this example, the parametric property is the response object, that is passed as first argument to the callback function. Such argument is intercepted and stored using the `action` attribute (lines 9-11). Considering that such response object is the receiver of calls `read`, `resume` or `on('data')`, the monitor implementation should intercept it using the `target` attribute in place of `id` (lines 15, 18 and 24).

```
1  SafeFileWriteAsync = {
2
3    events: {
4      // Definition of the parametric event write
5      write: invokeAsyncFunPre(fs.write, {
6        // Parametric property: file descriptor
7        id: ({args}) => args[0]
8      }),
9      writeCB: execCallBackPre("write")
10   },
11
12   re: "(write writeCB)*",
13
14   // Error recovery action:
15   fail: function({event, args}) {
16     if (event.name === "write"){
17       const callback = args.pop();
18       var err, written;
19       try{
20         // Execute the synchronous API version
21         written = fs.writeSync(...args)
22       } catch(exception) {
23         err = exception;
24       }
25       callback(err, written);
26       // Skip the original invocation
27       return {skip: true}
28     }
29   }
30 }
```

Figure 4: Monitor implementation for detecting and recovering the flawed `fs.write` usage pattern introduced in Example 1.

5 EVALUATION

In this section, we provide a preliminary evaluation of NodeMOP based on the two case studies discussed in Section 3. We first show the effectiveness of NodeMOP to recover errors from flawed code, and then we evaluate the NodeMOP instrumentation overhead. All measurements are performed on an Intel(R) Xeon(R) CPU E3-1505M v6 @ 3.00GHz with 16GB RAM, running Ubuntu Desktop 18.04 with NodeProf (September 2018) on GraalVM vm-1.0.0-rc3 CE.

5.1 Error Recovery

When executing the flawed Example 1, the virtual machine often crashes after several minutes with an out-of-memory error. NodeMOP can detect such memory leaks related to API misusages, leading to a correct program termination.

Regarding Example 2, when the HTTP request completes each allocated object should become unreachable, so the garbage collector should be able to bring back the amount of memory usage almost at the same level as when the application started. Figure 6 shows the Node.js heap usage over time when executing multiple times the code presented in Example 2. Each steep descent in the figure represents the garbage collector activity. As the figure shows, the garbage collector cannot reclaim the memory used by completed HTTP requests, leading to a process out-of-memory error.

As can be seen from the monitor code (i.e., the `fail` function at line 26), NodeMOP can be used to recover from the flawed API usage by calling `resume()` on the `response` object, causing leaked (open) connections to be correctly disposed. Figure 7 shows the memory usage of the application introduced in Example 2 executed together with the NodeMOP monitor. As the picture shows, the monitor is able to recover from the developer's mistake, allowing the garbage collector to reclaim the whole memory when activated.

5.2 Instrumentation Overhead

To assess the instrumentation overhead imposed by NodeMOP monitors, we derive two benchmarks from the flawed applications, and measure the overall execution using the following benchmark configurations:

(1) `Fixed`: Uninstrumented application, manually fixed to remove the problematic code pattern. We use this configuration as a baseline for our experiments;

```
const {get,IncomingMessage,ClientRequest} = require('http');

HttpResponseMemoryLeak = {
  events: {
    // Track the invocation of the asynchronous function http.get (the event named req)
    req:  invokeAsyncFunPre(get),
    // Track the beginning of the execution of the http.get function callback (the event named 'req')
    reqCbPre: execCallBackPre('req', {
      action: ({monitor, args}) => { // Define the action to be executed when the event is tracked
        monitor.data.response = args[0]; // Store the intercepted http response object to identify
            the monitor that shall receive the read, resume and listener events
      }
    }),
    // Track the invocation of the functions read and resume applied to the response object
        previously stored
    read: execFunPre(IncomingMessage.prototype.read, {
      target: ({monitor}) => monitor.data.response
    }),
    resume: execFunPre(IncomingMessage.prototype.resume, {
      target: ({monitor}) => monitor.data.response
    }),
    // Track the listener registration on the response object
    listener: execFunPre(ClientRequest.prototype.on, {
      // Condition that filters only the registration of the listener named 'data'
      condition: ({args}) => args[0] === 'data',
      target: ({monitor}) => monitor.data.response
    }),
    // Track the end of the execution of the http.get function callback (the event named 'req')
    reqCbPost: execCallBackPost('req')
  },

  re: "req reqCbPre (read|resume|listener)+ reqCbPost",

  fail: function({monitor}) { // Error recovery action:
    monitor.data.response.resume();
    monitor.destroy() //  Dispose the related monitor instance
  },

  match: function({monitor}) { // Action to be performed if the specification is validated
    monitor.destroy() //  Dispose the related monitor instance
  }
}
```

Figure 5: Monitor implementation for detecting and recovering the flawed `http.get` usage pattern introduced in Example 2.

(2) `Fixed+MOP`: Same application as the `Fixed` configuration, monitored with NodeMOP. Since the application is fixed this configuration allow us to measure the instrumentation overhead introduced by the property verification;
(3) `Flawed+MOP`: Flawed applications as reported in Example 1 and Example 2, monitored and recovered with NodeMOP. This configuration allow us to measure the instrumentation overhead together with the execution time of the recovery operation.

We evaluate the instrumentation overhead as slowdown factor, i.e., the ratio between the execution time of the instrumented configurations and the `Fixed` configuration. As the flawed versions eventually lead to VM crashes, their execution time cannot be measured without the error recovering operation provided by NodeMOP.

As introduced in Section 3, we fixed the flawed code introduced in Example 1 by replacing the call to `fs.write` with a call to `fs.writeSync`. With this modification, no monitor instances are allocated by NodeMOP, because the property specification

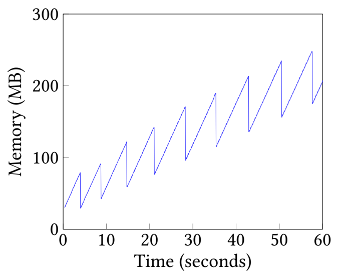

Figure 6: Memory usage of the flawed http request introduced in Example 2. Steep descents represent garbage collector activity.

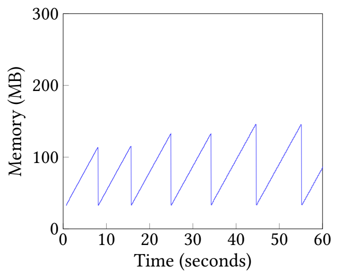

Figure 7: Memory usage of flawed http-request introduced in Example 2 recovered with NodeMOP. Steep descents represent garbage collector activity.

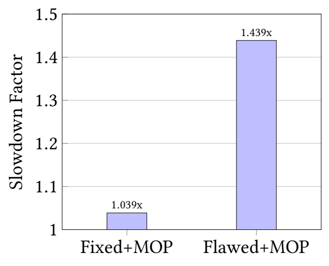

Figure 8: NodeMOP slowdown for Example 1.

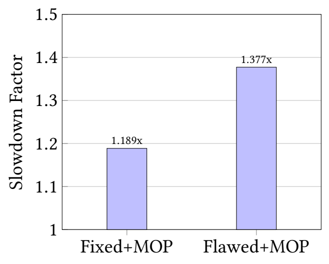

Figure 9: NodeMOP slowdown for Example 2.

traces all invocations to `fs.write`, but does not monitor calls to `fs.writeSync`. Figure 8 shows the performance of the `fs.write` example as the mean of the slowdown factors collected in 20 runs. The slowdown factor introduced by the `Fixed+MOP` version is as low as 1.039x. This is expected, since NodeMOP does not instrument functions that are not targeted by its formal specification. When the flawed application is executed, the slowdown factor introduced by the NodeMOP instrumentation together with recovery operations is 1.439x.

Recovering the flawed code in Example 2, a monitor instance is created for each request and destroyed with the `.destroy()` function call[3]. In this case NodeMOP instruments calls to the `http` APIs. Therefore, the slowdown factor introduced by the `Fixed+MOP` version is 1.189x and 1.377x by the Flawed+MOP version.

Hence, our initial evaluation shows that NodeMOP allows developers to monitor applications introducing an average slowdown factor of 1.114x and to recover from flawed applications introducing an average slowdown factor of 1.408x. Although the collected instrumentation overhead is significant we do not consider it prohibitive for most applications.

6 RELATED WORK

Prevailing RV frameworks rely on AOP [24] with AspectJ [23] to weave the monitoring logic into the observed program. Because AspectJ supports weaving of Java source code and/or bytecode only, the aforementioned RV frameworks are limited to Java [11] or Android [40, 41] applications, or to code that is compiled to Java bytecode (e.g., Scala). More advanced approaches are presented in the literature to support runtime verification in distributed [29, 31] and multi-language [30] systems. Even in these approaches, the target languages are compiled and statically typed, so the instrumentation is based on compilation-time weaving techniques. Hence, prevailing RV frameworks are not applicable to dynamically typed languages. Although RV is very important in such a context due to complex (and error-prone) programming models, frameworks weaving monitoring code at compilation-time cannot help developers since they are limited to compiled languages.

Runtime verification has been largely and successfully applied in the context of web services [14, 19, 20]. Instead of monitoring the JavaScript control flow, existing approaches provide a MOP specification in order to verify that the data exchanged between clients and servers (e.g., XML or JSON messages) satisfy a given protocol.

In a recent work, Ancona et al. [6] introduced the first RV tool that target Node.js, based on the parametric trace expressions model [5], an extension to trace expressions [3, 4] expressly designed for parametric RV. Although the formal approach is elegant and the parametric trace expression formalism is well suited for RV, the proposed tool is limited to error detection and it does not offer error recovery strategies that are usually integrated in MOP tools, like NodeMOP. Furthermore, their tool is based on Jalangi, which is known to be inefficient compared to NodeProf [39], besides not supporting the latest ECMA2018 features [43]. To the best of our knowledge, NodeMOP is the first MOP tool that targets the JavaScript language and Node.js applications.

[3]To reduce the instrumentation overhead it is possible to reuse monitor instances instead of instantiate and destroy one monitor for each request. We omit the code for such more complex monitor definition due to space constraints.

7 CONCLUSIONS

In this paper, we have introduced NodeMOP, the first practical MOP framework for Node.js applications. NodeMOP is built on top of NodeProf, an efficient dynamic analysis framework for Node.js. NodeMOP allows the developer to specify properties in an intuitive way. NodeMOP supports error recovery procedures, allowing developers to define custom actions to be executed when a specification is violated. Preliminary evaluation results show that the slowdown factor introduced by NodeMOP when applied to a correct application is 1.039x if the application does not use monitored functions and 1.439x if it does. Moreover, the slowdown factors that we collected by recovering flawed applications with NodeMOP are 1.377x and 1.439x.

In ongoing research, we are extending NodeMOP with more formalisms (e.g., context-free grammars, linear temporal logic, string rewriting system). We also plan to improve NodeMOP by defining a high-level domain-specific language for JavaScript code weaving and for MOP specifications, extending the general MOP syntax [35].

ACKNOWLEDGMENTS

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] Christoffer Quist Adamsen, Anders Møller, Rezwana Karim, Manu Sridharan, Frank Tip, and Koushik Sen. 2017. Repairing Event Race Errors by Controlling Nondeterminism *(ICSE)*. IEEE Press, Piscataway, NJ, USA, 289–299.

[2] Saba Alimadadi, Sheldon Sequeira, Ali Mesbah, and Karthik Pattabiraman. 2014. Understanding JavaScript Event-based Interactions *(ICSE)*. ACM, New York, NY, USA, 367–377.

[3] Davide Ancona, Viviana Bono, and Mario Bravetti. 2016. *Behavioral Types in Programming Languages*. Now Publishers Inc., Hanover, MA, USA.

[4] Davide Ancona, Angelo Ferrando, and Viviana Mascardi. 2016. Comparing Trace Expressions and Linear Temporal Logic for Runtime Verification. In *Essays Dedicated to Frank De Boer on Theory and Practice of Formal Methods - Volume 9660*. Springer-Verlag, Berlin, Heidelberg, 47–64.

[5] Davide Ancona, Angelo Ferrando, and Viviana Mascardi. 2017. Parametric Runtime Verification of Multiagent Systems. In *AAMAS '17*. 1457–1459.

[6] Davide Ancona, Luca Franceschini, Giorgio Delzanno, Maurizio Leotta, Marina Ribaudo, and Filippo Ricca. 2017. Towards Runtime Monitoring of Node.js and Its Application to the Internet of Things. In *ALP4IoT*. 27–42.

[7] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. 2017. A Survey of Dynamic Analysis and Test Generation for JavaScript. *ACM Comput. Surv.* 50, 5, Article 66 (Sept. 2017), 36 pages.

[8] Walter Binder, Daniele Bonetta, Cesare Pautasso, Achille Peternier, Diego Milano, Heiko Schuldt, Nenad Stojnic, Boi Faltings, and Immanuel Trummer. 2011. Towards self-organizing service-oriented architectures. In *SERVICES 2011*. IEEE, 115–121.

[9] Daniele Bonetta, Achille Peternier, Cesare Pautasso, and Walter Binder. 2010. A multicore-aware runtime architecture for scalable service composition *(APSCC '10)*. IEEE, 83–90.

[10] Feng Chen and Grigore Roşu. 2003. Towards Monitoring-Oriented Programming: A Paradigm Combining Specification and Implementation *(Electronic Notes in Theoretical Computer Science. RV '03)*, Vol. 89(2). Elsevier, 108–127.

[11] Feng Chen and Grigore Roşu. 2005. Java-MOP: A Monitoring Oriented Programming Environment for Java. In *TACAS '05 (LNCS)*, Vol. 3440. Springer-Verlag, 546–550.

[12] Feng Chen and Grigore Roşu. 2007. MOP: An Efficient and Generic Runtime Verification Framework. In *OOPSLA*. ACM press, 569–588.

[13] Feng Chen and Grigore Roşu. 2009. Parametric Trace Slicing and Monitoring *(TACAS '09)*. Springer-Verlag, Berlin, Heidelberg, 246–261.

[14] Normann Decker, Franziska Kühn, and Daniel Thoma. 2014. Runtime Verification of Web Services for Interconnected Medical Devices *(ISSRE '14)*. 235–244.

[15] Tzilla Elrad, Robert E. Filman, and Atef Bader. 2001. Aspect-oriented Programming: Introduction. *Commun. ACM* 44, 10 (Oct. 2001), 29–32.

[16] Node.js Foundation. 2018. About | Node.js. https://nodejs.org/en/about/

[17] Node.js Foundation. 2018. File System | Node.js v8.11.3 Documentation. https://nodejs.org/docs/latest-v8.x/api/fs.html

[18] Node.js Foundation. 2018. HTTP | Node.js v8.11.3 Documentation. https://nodejs.org/docs/latest-v8.x/api/http.html

[19] Sylvain Hallé, Tevfik Bultan, Graham Hughes, Muath Alkhalaf, and Roger Villemaire. 2010. Runtime Verification of Web Service Interface Contracts. *IEEE Computer* 43, 3 (2010), 59–66.

[20] Sylvain Hallé and Roger Villemaire. 2010. Runtime Verification for the Web. In *Runtime Verification*, Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Roşu, Oleg Sokolsky, and Nikolai Tillmann (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 106–121.

[21] Quinn Hanam, Fernando S. de M. Brito, and Ali Mesbah. 2016. Discovering Bug Patterns in JavaScript *(FSE 2016)*. 144–156.

[22] Casper S. Jensen, Anders Møller, Veselin Raychev, Dimitar Dimitrov, and Martin Vechev. 2015. Stateless Model Checking of Event-driven Applications *(OOPSLA)*. ACM, New York, NY, USA, 57–73.

[23] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. 2001. An Overview of AspectJ *(ECOOP)*. Springer-Verlag, London, UK, UK, 327–353.

[24] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-Oriented Programming. In *ECOOP*. 220–242.

[25] Oracle Labs. 2018. GraalVM. https://www.graalvm.org/

[26] Oracle Labs. 2018. graalvm/graaljs: A Javascript implementation built on GraalVM. https://github.com/graalvm/graaljs

[27] Martin Leucker and Christian Schallhart. 2009. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming* 78, 5 (2009), 293 – 303. The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS).

[28] Magnus Madsen, Ondřej Lhoták, and Frank Tip. 2017. A Model for Reasoning About JavaScript Promises. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 86 (2017), 24 pages.

[29] S. Malakuti, M. Aksit, and C. Bockisch. 2011. Distribution-Transparency in Runtime Verification. 328–335.

[30] S. Malakuti, C. Bockisch, and M. Aksit. 2009. Applying the Composition Filter Model for Runtime Verification of Multiple-Language Software *(ISSRE '09)*. 31–40.

[31] Somayeh Malakuti Khah Olun Abadi, Jong Hyuk Park, Mohammad Obaidat, Mehmet Aksit, and Christoph Bockisch. 2011. Runtime Verification in Distributed Computing. *Journal of convergence* 2, 1 (30 6 2011), 1–10.

[32] P. O. Meredith, D. Jin, F. Chen, and G. Rosu. 2008. Efficient Monitoring of Parametric Context-Free Patterns *(ASE '08)*. IEEE Computer Society, Washington, DC, USA, 148–157.

[33] James Newsome and Dawn Xiaodong Song. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software. In *NDSS*.

[34] npm Inc. 2018. npm. https://www.npmjs.com/

[35] Formal Systems Laboratory (FSL) of the Department of Computer Science at the University of Illinois. 2018. MOP4 Syntax. http://fsl.cs.illinois.edu/index.php/MOP4_Syntax

[36] Boris Petrov, Martin Vechev, Manu Sridharan, and Julian Dolby. 2012. Race Detection for Web Applications. *SIGPLAN Not.* 47, 6 (June 2012), 251–262.

[37] Michael Pradel and Koushik Sen. 2015. The Good, the Bad, and the Ugly: An Empirical Study of Implicit Type Conversions in JavaScript. In *ECOOP*, John Tang Boyland (Ed.), Vol. 37. Dagstuhl, Germany, 519–541.

[38] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: A Selective Record-replay and Dynamic Analysis Framework for JavaScript *(ESEC/FSE)*. ACM, New York, NY, USA, 488–498.

[39] Haiyang Sun, Daniele Bonetta, Christian Humer, and Walter Binder. 2018. Efficient Dynamic Analysis for Node.Js *(CC)*. ACM, New York, NY, USA, 196–206.

[40] Haiyang Sun, Alexander North, and Walter Binder. 2017. Multi-Process Runtime Verification for Android *(APSEC '17)*. 701–706.

[41] Haiyang Sun, Andrea Rosà, Omar Javed, and Walter Binder. 2017. ADRENALIN-RV: Android Runtime Verification Using Load-Time Weaving *(ICST '17)*. 532–539.

[42] ASM Team. 2018. ASM. https://asm.ow2.io/

[43] ECMAScript Team. 2018. ECMAScript®2018 Language Specification. https://www.ecma-international.org/ecma-262/9.0/index.html

[44] JavaMOP Team. 2018. Javamop/Usage.md at master - runtimeverification/javamop. https://github.com/runtimeverification/javamop/blob/master/docs/Usage.md

[45] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical Partial Evaluation for High-performance Dynamic Language Runtimes. *SIGPLAN Not.* 52, 6 (June 2017), 662–676.