# Parametric runtime verification of Node.js applications with trace expressions

Davide Ancona[a], Luca Franceschini[a], Giorgio Delzanno[a], Maurizio Leotta[a], Marina Ribaudo[a], and Filippo Ricca[a]

a DIBRIS, University of Genova, Italy

**Abstract** Node.js is a runtime framework that gained popularity in the last years as it allows rapid development of lightweight servers in JavaScript; more recently, it has been successfully employed to support programming for the Internet of Things. While the asynchronous, single-thread execution model on which Node.js is based turns out to be an effective way to efficiently handle big volumes of data and requests, it also makes quite challenging to ensure the correct behavior of code heavily based on nested callbacks which are executed asynchronously. To this aim, runtime verification can be a valuable support to tackle such a complex task, and to complement other software verification techniques based on static analysis or testing. Furthermore, runtime verification of Node.js can be fruitfully exploited for ensuring the correctness of Internet of Things systems, a challenge which has been barely explored so far.

Trace expressions have been initially devised as a formalism for runtime verification of protocol-compliance adopted in widespread multiagent system platforms. Key features of trace expressions are their support for parametricity, which allows definition of specifications depending on values that can be known only at runtime when events are traced, and their independence from a particular language and system, obtained through the notion of event domain and type.

In this work we show how trace expressions can be employed in the context of runtime verification for Node.js applications to specify the expected behavior, and we present the implementation of a corresponding prototype tool allowing automatic monitoring of the correct use of the features offered by widespread Node.js libraries, as `http`. The implementation is based on the Jalangi2 framework for JavaScript code instrumentation, required to dynamically trace events, and on SWI-Prolog for supporting trace expressions and the inference engine for their verification. The tool offers a simple HTTP interface based on JSON to allow runtime verification of distributed and heterogeneous components.

Preliminary experiments with the popular framework Express show that the tool is able to support runtime verification of real Node.js code.

**ACM CCS 2012**

- **General and reference** → *Computing standards, RFCs and guidelines*;
- **Applied computing** → **Publishing**;

**Keywords** parametric runtime verification, trace expressions, Jalangi2, Node.js, SWI-Prolog

The Art, Science, and Engineering of Programming

**Perspective** The Empirical Science of Programming

**Area of Submission** Program verification, testing and debugging

© D. Ancona, L. Franceschini, G. Delzanno, M. Leotta, M. Ribaudo, F. Ricca
This work is licensed under a "CC BY 4.0" license.
Submitted to *The Art, Science, and Engineering of Programming*.

## 1 Introduction

*Node.js* is a runtime environment that allows JavaScript programs to be executed outside a browser. The framework gained popularity in the last years as it allows developers to successfully employ JavaScript for server-side programming.

At its core, Node.js is based on an asynchronous model of computation: I/O requests are non-blocking and they immediately return to the caller, scheduling the request for later execution. Furthermore, while I/O operations can be parallelized by the underlying system, the JavaScript code always runs on a single thread. This asynchronous, single-thread execution model turned out to be an effective way to handle big volumes of data and huge amounts of (simultaneous) requests [25, 32], thus making Node.js a suitable choice to implement efficient servers without explicitly dealing with concurrency and parallelism.

On the other hand, asynchronous programming heavily relies on the notion of callback, which can lead to many levels of nested blocks and obfuscate the actual execution flow, making the code hard both to read and to debug. When an asynchronous request is made a callback (i.e., a function) is expected, and it will be executed when the result of the request will be available. This, together with the non-deterministic environments in which Node.js is employed, and the dynamic nature of JavaScript itself, makes it challenging to ensure correct program behavior with traditional approaches like static/formal verification techniques and testing.

*Runtime verification* [26] is a software analysis approach in which a running system is observed by monitors that perceive relevant events and use their associated information to verify them against a given (formal) specification of the expected behavior. When efficiently implemented, the verification process can even run after deployment in order to monitor real scenarios.

As preliminary results suggest [9], runtime verification can be fruitfully used to monitor Node.js systems as a complement to formal verification and testing. The approach suggested in [9] to implement a monitoring system for JavaScript and Node.js is based on code instrumentation, to allow suitable (and automatic) modification of the original code for appropriate reaction upon occurrences of interesting events.

We propose *trace expressions* [7] as a formalism to write down formal specifications. They are more expressive than other formalisms commonly adopted for runtime verification, as attributed context-free grammars [14] and $\text{LTL}_3$ [6]. Though they were initially devised to verify protocol-compliance in multi-agent systems, existing work and preliminary results show that trace expressions can be used as general specification formalism for parametric runtime verification [5, 9]; the expressive power and the algebraic properties of the underlying operators have been partially investigated [5, 7]. A key feature of trace expressions is parametricity: correctness of execution may depend on the actual values associated with events at runtime. This makes the formalism very expressive and allows runtime verification of complex properties. Another interesting characteristic is their independence from the monitored system and underlying programming language, obtained through the abstraction of event type.

In this work we show how trace expressions can be successfully used in the context of runtime verification for Node.js applications, to formally specify the correct use of the features exported by a Node.js module as can be derived from its documentation, and to dynamically verify such a specification. This is the core idea of this work and the main novelty, since (runtime) verification of Node.js has not been studied before.

A corresponding prototype tool[1] has been developed based on the Jalangi2[2] framework and SWI-Prolog.[3] The former is employed for performing instrumentation of JavaScript code, to be able to dynamically trace all those events that constitute what we call an *event domain*, that is, the set of all events relevant to certain kinds of properties; all examples shown in this paper refer to a unique event domain consisting of calls and returns from asynchronous functions and their corresponding callbacks. Hence, a unique instrumentation was required to automatically monitor and verify different properties, each specified by a particular trace expression. SWI-Prolog is the underlying language for supporting parametric trace expressions and implementing the inference engine for their verification; it provides a natural support to cyclic terms, needed for recursive trace expressions, and coinductive programming with the `coinduction`[4] module.

The tool offers a simple HTTP interface based on JSON to allow runtime verification of distributed and heterogeneous components; in this way, the tool is ready to support runtime verification in the context of the Internet of Things.

To experiment with the tool, we have inspected the documentation of the standard `http` module, one of the most frequently used in Node.js, to mine specifications expressible with parametric trace expressions, with the aim of dynamically verifying the correct use of `http` in Node.js code. Promising preliminary tests with Express, one of the most popular framework for web applications built on `http`, show that our prototype tool is able to support runtime verification of real Node.js code.

**Outline** Section 2 motivates the choice of Node.js, and together with Section 3 they give the necessary background about Node.js and trace expressions. Section 4 is a study of the state of the art for runtime verification techniques, while Section 5 shows how it can help to detect bugs in Node.js programs with motivating examples. Section 6 presents the implementation of our framework for runtime verification of Node.js based on trace expressions. Section 7 reports on some preliminary experiments conducted with Node.js programs based on the standard module `http` and the widely adopted framework Express. Finally, Section 8 draws conclusions and discusses future research directions.

[1] The implementation and a script for running all tests reported in this paper are available at http://www.disi.unige.it/person/AnconaD/NodeRiVe/artifact.zip.
[2] https://github.com/Samsung/jalangi2.
[3] http://www.swi-prolog.org.
[4] http://www.swi-prolog.org/pldoc/doc/_SWI_/library/coinduction.pl.

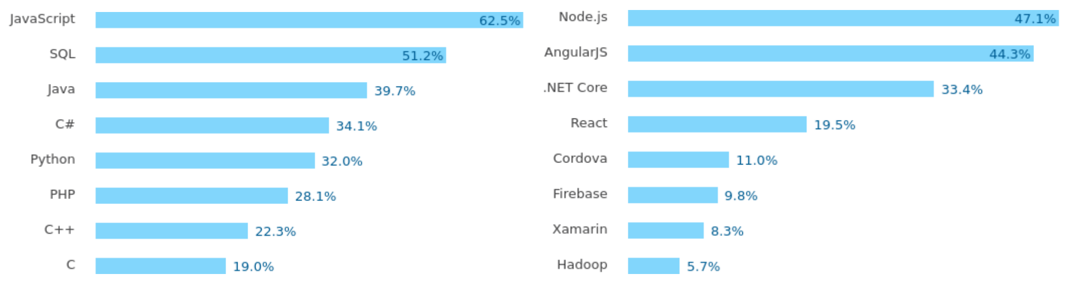


**Figure 1** Stack Overflow Developer Survey 2017, https://insights.stackoverflow.com/survey/2017. Most used programming languages (on the left) and frameworks (on the right).

## 2 Node.js

**Node.js** In 2009 Node.js[5] was released, and since then its popularity quickly raised to the point that it now appears to be the single most used framework, according to Stack Overflow Developer Survey 2017 (Figure 1). Node.js is a JavaScript runtime environment expressly devised for running JavaScript code outside web browsers, thus giving developers the opportunity to use the language to write server-side code as well.

Node.js is based on Chrome V8, a highly optimized JavaScript engine developed for the Google Chrome web browser. Furthermore, developers can count on a huge package ecosystem[6] with a repository of hundreds of thousand modules that can be reused. More recently, Node.js has become relevant for the Internet of Things as well, both because of the availability of packages for device management and its asynchronous model, which is well suited for distributed environments. Node-RED[7] is a tool based on Node.js to support flow-based programming for the Internet of Things to connect together devices at the edge of the network.

**Asynchronous Computation** In Node.js, connection to databases, HTTP requests, access to the file system and every other I/O intensive operation is by default *asynchronous*, meaning the actual execution is postponed until a later time, and the program can go on without waiting for completion. Computations needing the results of I/O operations are modeled by *callbacks*: when an asynchronous function is invoked, another function (the callback) is given as an additional argument, and it will be executed once the results will be available.

Figure 2 shows the different programming patterns using synchronous and asynchronous calls. On the left, the program is blocked until the result of syncIO is returned, and then it is passed to function use. On the right, asyncIO *immediately returns* and the I/O operation is scheduled for execution together with the callback use; when the result will be available, the callback will be executed receiving such result as an

[5] https://nodejs.org/
[6] https://www.npmjs.com/
[7] https://nodered.org/.

argument. Node.js heavily takes advantage of the right pattern (for the sake of brevity, error handling is ignored in the example).

```
function use(data) { ... }
let res = syncIO(...);
use(res);
```

```
function use(data) { ... }
asyncIO(..., use);
```

**Figure 2** Difference between synchronous and asynchronous I/O.

**Event Loop** The core of Node.js is the *event loop*, represented in Figure 3. Asynchronous requests are processed from a simple loop, and the corresponding I/O intensive operations are scheduled for execution together with their callbacks. The event loop is run after the whole Node.js script is evaluated.

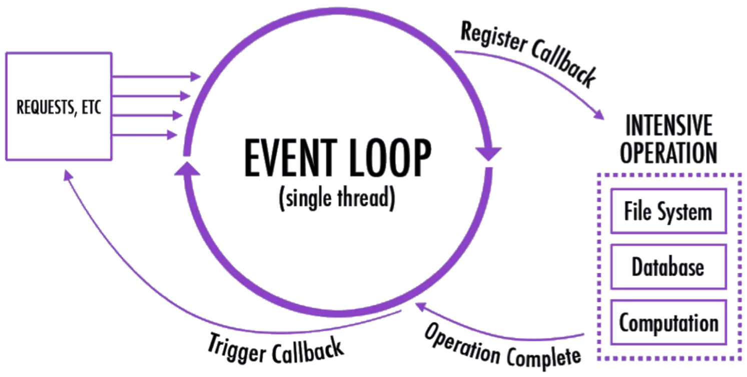


**Figure 3** Node.js execution model representation based on the event loop.

Note that callbacks themselves can make more asynchronous requests to the event loop, and those requests will receive callbacks, and so on... Indeed, in real programs callbacks and I/O operations are usually nested into each other as a synchronization mechanism, making it hard to understand (and debug) the code.

Even if it may seem counterintuitive at first, this execution model based on asynchronous I/O and the event loop scales very well to a lot of concurrent operations, without the need for the programmer to explicitly deal with concurrency and parallelism [25, 32]. At the system level, different I/O operations can be executed in parallel.

## 3 Trace Expressions

Trace expressions were initially developed in the context of multi-agent systems as "global types" [4]. The aim was to obtain a formalism expressive enough to check messages exchanged by autonomous agents for compliance against a given interaction protocol [2, 16]. Global types, and thus trace expressions, derive from the notion of behavioral type, for which a survey can be found in [1].

Later on, with the introduction of the notion of event type, trace expressions have become system and language agnostic, so that they can be exploited for runtime verification in several different contexts.

**Events** Trace expressions can be used to specify the possible correct behavior of a system, w.r.t. some property that needs to be verified. To this aim, *events* are defined as all the relevant observations that can be made on the system. Examples include the execution of I/O operations, the triggering of a Node.js callback, invocation of functions… A fixed set $\mathcal{E}$ of events is assumed.

An *event trace* $e_1 \ldots e_n \ldots$ is a (possibly infinite) sequence of events. Using formal language notation, the set of traces can be denoted as $\mathcal{E}^\infty = \mathcal{E}^* \cup \mathcal{E}^\omega$ (the union of all finite and infinite traces, respectively). Intuitively a trace encodes a run of the system under test, or at least the relevant parts of its execution.

As an example, consider the monitoring of a program writing on files. The set of relevant events could be, for instance, the three common operations *open*, *write* and *close*, where *fd* is the unique file descriptor:

$$\mathcal{E} = \bigcup_{fd \in \mathbb{N}} \{open(fd), write(fd), close(fd)\} \tag{1}$$

An example of event trace over the set $\mathcal{E}$ would be the following:

$$open(42)\ write(42)\ write(42)\ close(42) \tag{2}$$

**Event Types** On the top of events, a language $\mathcal{ET}$ of *event types* is defined. Event types are generally terms allowed to contain variables, and their language is not fixed in order to make trace expressions more flexible and easily adaptable to different domains.

Together with event types a function *match* is assumed to be given, with the following semantics. Given an event $e$ and an event type $\vartheta$, $match(e, \vartheta) = \sigma$ if and only if $e$ matches $\vartheta$ with the computed substitution $\sigma$; substitutions on terms have the usual meaning.

In the rest of the paper, we consider different examples of event domains and, thus, event type languages. We do not expect considerable expressivity is needed in this step, and as long as matching and substitution functions are defined, all properties still hold. Since we will use standard inductive terms over some (implicitly given) signature and a (enumerable) set of variables, the usual definition of substitution apply.

Considering again the previous files event domain (1), a sensible event type matching all write operations would be $write(\mathsf{x})$, where $\mathsf{x}$ is a variable. A sensible definition of matching would lead to the following:

$$match(write(42), write(\mathsf{x})) = \{\mathsf{x} \mapsto 42\}$$

**Trace Expressions** Finally, a *trace expression* identifies a set of traces corresponding to correct system behaviors. It is built on top of event types and the following operators:

- $\varepsilon$ (*empty trace*): the singleton set $\{\varepsilon\}$ containing the empty event trace $\varepsilon$;

- $\vartheta : \tau$ (*prefix*): the set of all traces whose first event $e$ matches the event type $\vartheta$, and the remaining part is a trace of $\tau$;
- $\tau_1 \cdot \tau_2$ (*concatenation*): the set of all traces obtained by concatenating the traces of $\tau_1$ with those of $\tau_2$;
- $\tau_1 \wedge \tau_2$ (*intersection*): the intersection of the traces of $\tau_1$ and $\tau_2$;
- $\tau_1 \vee \tau_2$ (*union*): the union of the traces of $\tau_1$ and $\tau_2$;
- $\tau_1 \mid \tau_2$ (*shuffle*, a.k.a. *interleaving*): the set obtained by shuffling the traces of $\tau_1$ with the traces of $\tau_2$;
- $\langle \mathsf{x};\ \tau \rangle$ (*binder*): it binds the free occurrences of $\mathsf{x}$ in $\tau$;
- $\vartheta \gg \tau$ (*filter*): denoting the set of all traces contained in $\tau$, when they are deprived af all events that do not match $\vartheta$ (theoretically, this operator can also be derived from the others).

Trace expressions are regular terms (a.k.a. cyclic) [20], thus there is no need for an explicit recursion operator.

For instance, the following trace expression $\tau$ specifies the correct use of the file descriptor 42:

$$\begin{aligned} \tau &= \varepsilon \vee (open(42) : \tau') \\ \tau' &= (write(42) : \tau') \vee (close(42) : \varepsilon) \end{aligned}$$

In the first line, *open* is forced to be the first operation, if any. After that, in $\tau'$, either there will be write operations or the file will be closed and no more writes will be allowed.

**Parametric Trace Expressions** The trace expression above can only verify the correct use of a single file descriptor, but with variables and binders [8] it is possible to write a *parametric* specification that solves the problem for any number of files:

$$\tau = \varepsilon \vee \langle \mathsf{fd};\ open(\mathsf{fd}) : (\tau \mid \tau') \rangle \tag{3}$$

$$\tau' = (write(\mathsf{fd}) : \tau') \vee (close(\mathsf{fd}) : \varepsilon) \tag{4}$$

When the first event $open(x)$ will occur, it will match the prefix computing the substitution $\{\mathsf{fd} \mapsto x\}$. At this point the shuffle operator is crucial: following events are allowed to belong either to $\tau$ (operations on new files, in the correct order) or to $\tau'\{\mathsf{fd} \mapsto x\}$ (note the substitution!) where further operations on $x$ will be checked.

Note that we are not monitoring the correct behavior of the underlying operating system, namely, that the same file descriptor is not assigned to many opened files simultaneously. Rather, the specification ensures that the API offered by such system is correctly used.

**Operational Semantics** The semantics of trace expressions is given by the labeled transition system in Figure 4. $\tau \overset{e}{\rightarrowtail} \tau'; \sigma$ holds iff the trace expression $\tau$ accepts the event $e$ with the substitution $\sigma$ and rewrites to $\tau'$. For instance, considering again trace expressions in Equations (3) and (4), the following transition is valid:

$$\tau \xrightarrow{open(1)} \tau \mid \tau'' \quad \text{with } \tau'' = (write(1) : \tau'') \vee (close(1) : \varepsilon)$$

$$\text{(main)}\,\frac{\tau \overset{e}{\rightarrowtail} \tau';\emptyset}{\tau \xrightarrow{e} \tau'} \qquad \text{(prefix)}\,\frac{}{\vartheta:\tau \overset{e}{\rightarrowtail} \tau;\sigma}\ \sigma{=}\mathit{match}(e,\vartheta) \qquad \text{(and)}\,\frac{\tau_1 \overset{e}{\rightarrowtail} \tau_1';\sigma_1 \quad \tau_2 \overset{e}{\rightarrowtail} \tau_2';\sigma_2}{\tau_1 \wedge \tau_2 \overset{e}{\rightarrowtail} \tau_1' \wedge \tau_2';\sigma}\ \sigma{=}\sigma_1{\cup}\sigma_2$$

$$\text{(or-l)}\,\frac{\tau_1 \overset{e}{\rightarrowtail} \tau_1';\sigma}{\tau_1 \vee \tau_2 \overset{e}{\rightarrowtail} \tau_1';\sigma} \qquad \text{(or-r)}\,\frac{\tau_2 \overset{e}{\rightarrowtail} \tau_2';\sigma}{\tau_1 \vee \tau_2 \overset{e}{\rightarrowtail} \tau_2';\sigma} \qquad \text{(shuffle-l)}\,\frac{\tau_1 \overset{e}{\rightarrowtail} \tau_1';\sigma}{\tau_1 \mid \tau_2 \overset{e}{\rightarrowtail} \tau_1' \mid \tau_2;\sigma}$$

$$\text{(cat-l)}\,\frac{\tau_1 \overset{e}{\rightarrowtail} \tau_1';\sigma}{\tau_1 \cdot \tau_2 \overset{e}{\rightarrowtail} \tau_1' \cdot \tau_2;\sigma} \qquad \text{(cat-r)}\,\frac{\tau_2 \overset{e}{\rightarrowtail} \tau_2';\sigma}{\tau_1 \cdot \tau_2 \overset{e}{\rightarrowtail} \tau_2';\sigma}\ \epsilon(\tau_1) \qquad \text{(shuffle-r)}\,\frac{\tau_2 \overset{e}{\rightarrowtail} \tau_2';\sigma}{\tau_1 \mid \tau_2 \overset{e}{\rightarrowtail} \tau_1 \mid \tau_2';\sigma}$$

$$\text{(var-t)}\,\frac{\tau \overset{e}{\rightarrowtail} \tau';\sigma}{\langle \mathsf{x};\ \tau\rangle \overset{e}{\rightarrowtail} \sigma\tau';\sigma_{\setminus \mathsf{x}}}\ \mathsf{x}{\in}\mathit{dom}(\sigma) \qquad \text{(var-f)}\,\frac{\tau \overset{e}{\rightarrowtail} \tau';\sigma}{\langle \mathsf{x};\ \tau\rangle \overset{e}{\rightarrowtail} \langle \mathsf{x};\ \tau'\rangle;\sigma}\ \mathsf{x}{\notin}\mathit{dom}(\sigma)$$

$$\text{(filter-t)}\,\frac{\tau \overset{e}{\rightarrowtail} \tau';\sigma}{\vartheta \gg \tau \overset{e}{\rightarrowtail} \vartheta \gg \tau';\sigma}\ \sigma{=}\mathit{match}(e,\vartheta) \qquad \text{(filter-f)}\,\frac{}{\vartheta \gg \tau \overset{e}{\rightarrowtail} \vartheta \gg \tau}\ \nexists\sigma{=}\mathit{match}(e,\vartheta)$$

$$(\epsilon\text{-empty})\,\frac{}{\epsilon(\varepsilon)} \qquad (\epsilon\text{-var})\,\frac{\epsilon(\tau)}{\epsilon(\langle \mathsf{x};\ \tau\rangle)}$$

$$(\epsilon\text{-or-l})\,\frac{\epsilon(\tau_1)}{\epsilon(\tau_1 \vee \tau_2)} \qquad (\epsilon\text{-or-r})\,\frac{\epsilon(\tau_2)}{\epsilon(\tau_1 \vee \tau_2)} \qquad (\epsilon\text{-others})\,\frac{\epsilon(\tau_1) \quad \epsilon(\tau_2)}{\epsilon(\tau_1 \mathit{op}\ \tau_2)}\ \mathit{op}{\in}\{\mid,\cdot,\wedge\}$$

**Figure 4** Transition system for parametric trace expressions.

Transition rules depend on predicate $\epsilon(-)$ which checks for termination, i.e., $\epsilon(\tau)$ holds only if $\tau$ accepts the empty trace $\varepsilon$. More generally, a trace expression $\tau$ accepts a (possibly infinite) event trace $e_1e_2\ldots$ iff there exists a (possibly infinite) reduction $\tau \xrightarrow{e_1} \tau' \xrightarrow{e_2} \cdots$.

The operational semantics rule for the intersection operator depends on the side condition $\sigma = \sigma_1 \cup \sigma_2$. Such equality holds when $\sigma_1$ and $\sigma_2$ coincide on $\mathit{dom}(\sigma_1) \cap \mathit{dom}(\sigma_2)$.

The top-level rule (main) expects the computed substitution to be empty. This corresponds to the fact that valid trace expressions, when considered as a whole, are supposed not to have free variables. However, when a variable is introduced by a binder, it will be substituted in the trace expression (rule (var-t)), and removed from the computed substitution. Finally, (var-f) handles the case in which a parametric trace expression accepts an event but the matching does not instantiate a variable (for instance, because its value will only be discovered observing further events).

Trace expressions semantics has been implemented in SWI-Prolog. The logic programming paradigm is well suited for the implementation of inference system, since inference rules can be translated almost directly to logic clauses. Furthermore SWI-Prolog offers native support to cyclic terms, therefore recursive trace expressions can be easily encoded. Support for programming with cyclic terms is based on coinductive logic programming [31], which is supported by the SWI-Prolog library coinduction.

## 4 Runtime Verification

**Context** Historically, different approaches has been proposed to solve the problem of verifying the behavior of software systems.

On one hand *formal methods* has been proposed, including theorem proving [13], a semi-automated class of techniques inspired by mathematical proofs, and model checking [19], which aims to automatically verify a suitably small model of the system against expected properties. These techniques have been extensively studied in the literature and are quite mature. However, they require either a deep theoretical knowledge or a simplified model of the system, and this may not be the case for many real-world scenarios.

On the other hand, software *testing* [28] has been successfully employed in industry, to find bugs by running (part of) the system on input whose correct output is known. Testing proved itself useful by working on many different levels of abstraction, from unit-test of small code snippets to black-box testing of complex systems as a whole.

More recently another approach to the problem has been proposed, that is, *Runtime Verification* [26]. First, a *monitor* is attached to the system under test in order to observe all relevant events. The result of this is said to be a *trace*, and it encodes the observed run. Then, the trace is verified against correctness properties expressed in a suitable formalism.

**Word Problem** Runtime verification is often considered a *lightweight* verification technique, since it aims at analyzing only a *single run* of the system, and not all the possible execution paths (as it is customary, for instance, in model checking). If we think of correctness properties as some set $S$ of traces satisfying them, the problem tackled by runtime verification is clearly easier than the one solved by static techniques. Essentially it is a *word problem* (check whether a given trace belongs to $S$) rather than an *inclusion* problem (check whether the set of all possible traces is included in $S$).

**Expressiveness** Part of the complexity comes from the expressiveness of the specification language in use, and different formalisms have been proposed, including regular expressions, context-free grammars with attributes [14, 15], and *Linear Temporal Logic (LTL)* [29] (and its variations), a modal logic where modalities refer to time which is one of the most commonly used specification formalism. While LTL deals with infinite traces, when doing runtime verification traces are always finite for obvious reasons. Nonetheless, it is useful to verify non-terminating systems: in such cases, only finite trace prefixes are available. In order to deal with this, $LTL_3$ has been proposed as a three-valued LTL [12], with the additional truth value encoding an inconclusive verification result.

Though they are not comparable to LTL (meaning none of them is more expressive than the other), trace expressions *are* more expressive than $LTL_3$ [7], which is very relevant for runtime verification purposes since the formalism has explicitly been devised for that [12].

**Monitoring Techniques** Besides specification, a crucial point of runtime verification is the monitoring part, which can either be *offline* or *online*. With the former approach, the execution is observed and, in the end, a trace is produced as a result; then it is checked against the specification. The latter approach can offer more possibilities, since events are observed by the monitor as soon as they occur, and the updated trace is checked in real-time. While this can slow down the execution, it offers some advantages. Not only errors are caught (almost) as soon as they manifest, but it is also possible to take *recovery actions* [3] in case monitoring is efficient enough to be performed even after deployment.

Programs implementing such behaviors have to be aware of the monitor and to interact with it. Different patterns and paradigms have been proposed for this, see *Monitor-Oriented Programming (MOP)* [18], a framework for runtime verification of Java programs, or *Runtime Reflection* [11], an architectural pattern designed for monitor-aware systems able to diagnose problems and mitigate issues.

Another critical aspect of runtime verification systems is the generation of the monitor itself, especially since it is desirable to automatically generate them from the high-level formal specification, thus making their adoption and application easier. Such a monitor should avoid to interfere with the program behavior as much as possible, in order for the verification to be effective. This is very important when the system under test has to operate in real-time, and/or when the property to be checked contains time constraints itself.

**Applications** Runtime verification can also be seen as a complement to more traditional approaches, like formal methods and testing, either for inherently dynamic properties that otherwise would be hard to verify, or for (safety-)critical systems that needs to be constantly checked, even *after deployment* and not only during the development cycle. Runtime verification can be used in conjunction with testing in order to generate test cases, as shown in [10]. As a note, however, neither testing nor runtime verification are *complete*. In other words, they cannot prove that a program satisfy some property for every input.

Runtime verification has been successfully applied to many other contexts, including *web services and applications* [23, 24], *multi-agent systems* [4], *object-oriented languages* [14, 15, 17, 27] and the *Internet of Things* [21, 33].

**Node.js** In the context of runtime verification, we only found one work that has been applied to Node.js applications [30]. The approach is based on DTrace [22], a low-level dynamic tracing framework that allows to extract a trace from a running system. However, there are important differences both in the implementation and in the goals. The monitors produces in the cited work exploit LTL [29] as a specification formalism, thus parametric verification is not supported, but as we show in the examples this is an important feature that increases expressiveness. Moreover, since one of our goals is to target the Internet of Things, we chose a distributed approach with a remote monitoring server. On the other hand, DTrace is not distributed and works at a lower level, thus being more efficient.

For a more in-depth, authoritative survey of runtime verification see [26].

## 5 Monitoring examples

### 5.1 File System

The fs standard module for Node.js deals with the file system, and its typical uses are good examples of code that would benefit from runtime verification techniques.

Consider the snippet in Figure 5, writing to a file all numbers from 0 to the constant LIMIT in any order (the ** operator stands for exponentiation). As it is customary in Node.js, all I/O operations are asynchronous, starting from the open function which, upon completion, will execute the callback (err, fd) => { ... }. The convention in Node.js is to pass errors (if any) to callbacks as first argument. fd is the file descriptor of the opened file.

```
const fs = require('fs');

const LIMIT = 2**8;

fs.open('tmp.txt', 'w', (err, fd) => {
 if (!err)
 for (let i=0; i<LIMIT; i++)
   fs.write(fd, i+'\n', () => {});

 fs.close(fd, () => console.log('bye'));
});
```

**Figure 5** An example of *incorrect* usage of the Node.js fs module.

The core of the program is a simple loop making a write request (with an empty callback, () => {}) for each number. After that, a request to close the file is generated, and a callback printing 'bye' is provided.

While the code may look fine to someone not familiar with asynchronous I/O and Node.js, it is actually flawed, but it will work up to considerable values of LIMIT. Depending on the Node.js implementation and the hardware resources, the code will fatally crash with some memory error if the number gets too high, though the amount of used memory is expected to be constant.

From the documentation, it turns out that a second write to the same file should only be called *after* the callback of the previous one has been called. This constraint however is not checked by the library for performance reasons.

Recalling the event-loop mechanism previously presented, first *all* the asynchronous `write` requests will be queued, and only *after* this they will be fulfilled. This explains the documented requirement and gives a possible cause for the error: the asynchronous I/O queue will grow linearly with LIMIT.

Trace expressions can be used to formally specify (and later verify) the requirement. Consider for instance the following domain (in this simple case, events and event types coincides):

$$\mathscr{E} = \{open, write, callback, close\}$$

A trace expression specifying the correct order of operations would then be:

$$\begin{aligned} \tau &= \varepsilon \lor (open : callback : \tau') \\ \tau' &= (write : callback : \tau') \lor (close : callback : \varepsilon) \end{aligned}$$

Note that the recursive occurrence of $\tau'$ is guarded by a *write* and a *callback*, thus forcing them to be alternating.

The code snippet we discussed would not satisfy the trace expression *as soon as the second write is called*, thus it will not be necessary to reach some system-dependent high number of requests for the bug to be detected.

Assuming the monitor can match asynchronous requests and their callbacks (for instance by generating a unique identifier for each request), a more precise specification would also be able to check that the correct callbacks are called. Then events will include this new information:

$$\mathscr{E} = \bigcup_{id \in \mathbb{N}} \{open(id), write(id), close(id), callback(id)\}$$

This time, however, parametric runtime verification is needed, and variables and event types are required. We define event types to be event terms over a set of variables $\mathscr{X}$ used as placeholders for request identifiers (the *match* will simply compute the appropriate substitution):

$$\begin{aligned} \mathscr{X} &= \{\mathsf{id}_1, \mathsf{id}_2, \dots\} \\ \mathscr{E}\mathscr{T} &= \bigcup_{\mathsf{x} \in \mathscr{X}} \{open(\mathsf{x}), write(\mathsf{x}), close(\mathsf{x}), callback(\mathsf{x})\} \end{aligned}$$

Finally, the parametric trace expression is given ($\tau$, $\tau_w$ and $\tau_c$ deal with opening, writing to and closing the file, respectively):

$$\begin{aligned} \tau &= \varepsilon \lor \langle \mathsf{id}_1;\ open(\mathsf{id}_1) : callback(\mathsf{id}_1) : \tau_w \rangle \\ \tau_w &= \langle \mathsf{id}_2;\ write(\mathsf{id}_2) : callback(\mathsf{id}_2) : \tau_w \rangle \lor \tau_c \\ \tau_c &= \langle \mathsf{id}_3;\ close(\mathsf{id}_3) : callback(\mathsf{id}_3) : \varepsilon \rangle \end{aligned}$$

Note that different variable names are only used for the sake of clarity, even if using the same variable in every binder would work since trace expression semantics correctly handles the shadowing of variables in nested scopes.

Identifiers are crucial to match asynchronous operations with their callbacks: after event $open(\mathsf{id}_i)$ occurs, meaning an asynchronous call to open has been made and a callback for it has been registered, event $callback(\mathsf{id}_i)$ will occur when *that* callback will be executed.

The trace expressions above are limited to a single file. If monitoring a collection of concurrently modified files is required, more parameters are necessary, since the monitor needs to track functions arguments[8] in order to retrieve file descriptors:

$$\begin{aligned}
\tau &= \varepsilon \vee \langle \mathsf{id}_1;\ open(\mathsf{id}_1) : (\tau' \mid \tau)\rangle \\
\tau' &= \langle \mathsf{fd};\ callback(\mathsf{id}_1, \mathsf{fd}) : \tau_w\rangle \\
\tau_w &= \langle \mathsf{id}_2;\ write(\mathsf{id}_2, \mathsf{fd}) : callback(\mathsf{id}_2) : \tau_w\rangle \vee \tau_c \\
\tau_c &= \langle \mathsf{id}_3;\ close(\mathsf{id}_3, \mathsf{fd}) : callback(\mathsf{id}_3) : \varepsilon\rangle
\end{aligned}$$

The shuffle operator allows to concurrently monitor the use of the opened file as well as other files.

## 5.2 Web Requests

Node.js is typically used as a server-side JavaScript environment in web development. Because of this, its library offers many features that greatly simplify the task of setting up a web server. For instance, it only takes a few lines of code to obtain a (very simple) HTTP server, as shown in Figure 6.

```
const http = require('http');

const server = http.createServer((request, response) => {
  request.on('data', chunk => console.log('some data received'));
  request.on('end', () => response.end('ok'));
});

server.listen(80);
```

**Figure 6** A basic web server in Node.js based on the http module.

The web server is created by passing a callback that will be invoked on every incoming request. Such a function will receive both the request and the response object through which it can reply.

For each request, first a callback logging a message for every received chunk of data is registered, then a response is sent back to the client when the whole request has been received. Finally, the server start listening at the standard HTTP port 80.

Now consider the (buggy) web client in Figure 7, where seven bytes of data are sent ('hey!', 'bye') but ten are declared in the 'content-length' field.

[8] For the sake of brevity, from now on we will omit the explicit definition of the employed event domains.

```
const http = require('http');

const options = {
  method: 'POST',
  headers: {
    'content-length': 10  // BUG! size too high
  }
};

const req = http.request(options, res => {
  res.on('data', console.log);
});

req.write('hey!');
req.end('bye');
```

**Figure 7** An *incorrect* web client sending a request with the wrong content-length value.

The logging callback (console.log) is scheduled for execution after receiving the response from the server. However, it will not be called since the server will be waiting for more data, and no reply will be sent.

The 'content-length' is *not* checked by the Node.js runtime against the actual length of the content, as it is clearly stated in the documentation. Trace expressions can be used again to specify the correct behavior.

The monitor needs to observe the creation of the ClientRequest object returned by function http.request as well as the use of its methods write and end. This time the focus is not on the order but rather on the arguments of functions.

The event domain will include all relevant information:

$$\mathcal{E} = \bigcup_{\substack{data \;\in\; \mathbb{B}^* \\ cl, id \;\in\; \mathbb{N}}} \{request(id, cl), write(id, data), end(id, data)\}$$

All operations are associated with a unique request identifier (at the Node.js level, this corresponds to the ClientRequest object), furthermore the request is associated with the declared 'content-length' *cl*, while *write* (and *end*) are associated with *data*, the sequence of bytes being sent.

The following is a possible trace expression specification of the correct behavior:

$$\begin{aligned}
\tau &= \varepsilon \vee \langle \mathsf{id};\ \langle \mathsf{cl};\ request(\mathsf{id}, \mathsf{cl}) : (\tau_w \mid \tau)\rangle\rangle \\
\tau_w &= \langle \mathsf{cl}';\ write(\mathsf{id}, \mathsf{cl}, \mathsf{cl}') : \tau'_w\rangle \vee \tau_e \\
\tau'_w &= \langle \mathsf{cl};\ write(\mathsf{id}, \mathsf{cl}', \mathsf{cl}) : \tau_w\rangle \vee \tau'_e \\
\tau_e &= end(\mathsf{id}, \mathsf{cl}) : \varepsilon \\
\tau'_e &= end(\mathsf{id}, \mathsf{cl}') : \varepsilon
\end{aligned}$$

The event types above have the following semantics:

- $request(\mathsf{id}, \mathsf{cl})$: a request object identified by id has been created, with a declared 'content-length' cl;

- $\mathit{write}(\mathsf{id},\mathsf{cl},\mathsf{cl}')$: `write` method has been invoked on request $\mathsf{id}$, with $\mathsf{cl}$ bytes still to send, and after the invocation the amount of missing data is $\mathsf{cl}'$;
- $\mathit{end}(\mathsf{id},\mathsf{cl})$: `end` has been invoked on request $\mathsf{id}$, and the last $\mathsf{cl}$ bytes are sent (besides finishing the request, `end` can also send a final chunk of data).

The shuffle operator, together with the unique identifier of each request, allows monitoring of multiple concurrent requests. This is similar to the previous example on files.

Variables $\mathsf{cl}$ and $\mathsf{cl}'$ are swapped between $\tau_w$ and $\tau'_w$, as two variables are needed to keep track of the number of bytes left before and after each write.

Here the *match* function plays a crucial role and it is worth showing, since in this case event types *write* and *end* are not a trivial abstraction over events:

$$\mathit{match}(\mathit{request}(\mathit{id},\mathit{cl}),\mathit{request}(\mathsf{id},\mathsf{cl})) = \{\mathsf{id} \mapsto \mathit{id}, \mathsf{cl} \mapsto \mathit{cl}\}$$

$$\mathit{match}(\mathit{write}(\mathit{id},\mathit{data}),\mathit{write}(\mathit{id}',\mathit{cl},\mathsf{cl}')) = \{\mathsf{cl}' \mapsto \mathit{cl} - \mathit{size}(\mathit{data})\} \text{ iff } \begin{array}{l} \mathit{id} = \mathit{id}', \\ \mathit{size}(\mathit{data}) \leq \mathit{cl} \end{array}$$

$$\mathit{match}(\mathit{end}(\mathit{id},\mathit{data}),\mathit{end}(\mathit{id}',\mathit{cl})) = \emptyset \text{ iff } \mathit{id} = \mathit{id}',\ \mathit{size}(\mathit{data}) = \mathit{cl}$$

At the beginning, both variables $\mathsf{id}$ and $\mathsf{cl}$ are free and their values are discovered when event *request* occurs. On the other end, upon writing, the current amount of data that still has to be sent is known, but the updated one has to be computed based on the size of *data*. Finally, *end* event is expected to send the exact amount of data left, and no further substitution is needed.

## 6 Implementation

We have chosen to implement a monitoring system for Node.js based on *code instrumentation*. The core idea is to modify the original source code of the program to be monitored in a way that allows observation of relevant events. Once an event is observed, the monitor can verify it against the specification, eventually acting in case of erroneous behavior.

The tool we have employed is *Jalangi2*. It allows addition of arbitrary code before and after basically every JavaScript operation: access to fields, declarations, functions and methods invocation, among others. Jalangi2 also gives access to information about the operation itself, as the arguments of a function call or the target object of a field access.

Thanks to these features it is possible to determine when asynchronous calls and their callbacks are executed, as well as matching them. However, implementing a prototype for Node.js runtime verification is not trivial, and requires to address some issues, as explained below:

- Anonymous functions are extremely common in JavaScript, thus function names are not a reliable way to identify them.
- Matching asynchronous functions with their callback needs some bookkeeping: the same asynchronous function can be used multiple times with different arguments and callbacks, and vice versa. In order to deal with this problem, our prototype

dynamically wraps callbacks *at runtime* and generate a unique identifier for each one of them.
- The prototype only tracks the program, but does not directly interact with the Node.js runtime. Since the execution goes back and forth between the event loop and the program, tracking the flow is not easy.
- Many different patterns arise in Node.js programming, and they need complex code instrumentation. For instance, the file system module functions expect a file descriptor as an argument, while HTTP requests are handled through methods invoked on objects encoding the requests.

The current implementation of trace expression semantics is coded in SWI-Prolog, since logic programming is a natural fit for inference rules and transition systems, and cyclic terms are supported by the `coinduction` library [31].

Our monitoring system has two main parts. On one hand, an instrumented Node.js program is executed, and the additional code observes relevant events. On the other hand, a Prolog server encoding the trace expression specification is running and validating events. These two parts communicate through a HTTP interface, and events are exchanged in the JSON format. The architecture is represented in Figure 8. Further details on the approach and the implementation with Jalangi2 can be found in the related paper [9].

Though JSON is a natural fit for JavaScript objects serialization, some important aspects need to be handled. Objects containing circular references, for instance, are quite common in Node.js modules, but they cannot be encoded in JSON, and the JavaScript standard library serialization (`JSON.stringify`) throws an error on such objects. Furthermore, JavaScript supports getters, which are special functions invoked when a property is accessed. During experiments, we found that some modules set getter functions before their code can actually be correctly executed, and this again breaks standard serialization since getters are invoked in the process. For these reasons, we had to modify the JSON serialization process for it to work in real Node.js scenarios, detecting and handling circular references and getters as special cases.

This work is a preliminary step towards runtime verification of Internet of Things applications, both because Node.js is emerging as a standard framework for IoT development, and the implementation through an HTTP server offers a natural support to verification of distributed systems, where (instrumented) devices can send events to a monitoring server.

## 7 Experiments

In this section we report on our preliminary experiments conducted with the prototype tool whose implementation has been described in Section 6. All these experiments can be easily reproduced by downloading our project at http://www.disi.unige.it/person/AnconaD/NodeRiVe/artifact.zip.

Since one of the main use of Node.js is the development of Web applications, we have focused on runtime verification of code based on the `http` module.

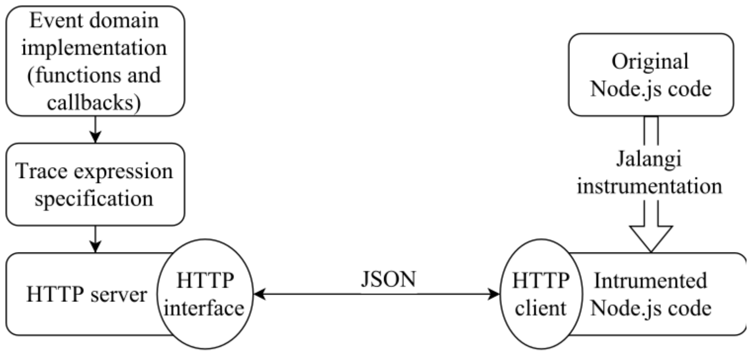


**Figure 8** Monitoring architecture for Node.js exploiting SWI-Prolog server with an HTTP interface.

First, the documentation of the module[9] has been inspected with the purpose of mining specifications of the features provided by http. Several of them can be easily translated in corresponding trace expressions that have been employed in our tool to automatically monitor correct use of the http module by client code.

Six constraints that are not enforced by the library have been identified on the use of the functions exported by http; correspondingly, six trace expressions have been derived to dynamically check such constraints and detect the related bugs. In each trace expression we use the filter operator to restrict the event domain to functions that are relevant for the specification.

The tool has been tested in two different ways. At a first stage, simple server and client Node.js applications based on http have been developed, and violations of the module specifications have been deliberately introduced in the code, to verify that the tool is effectively able to detect illegal use of the http features.

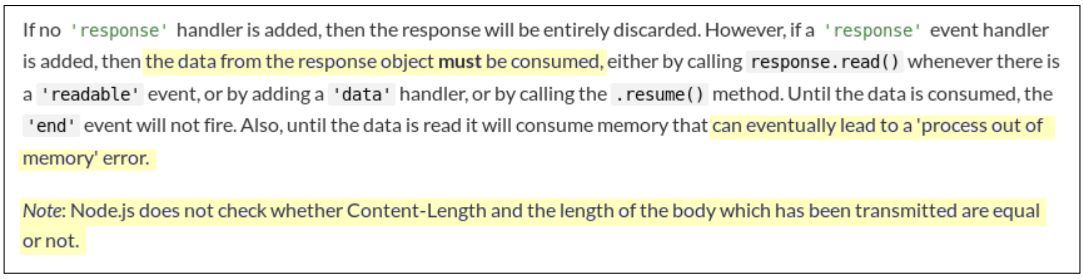
Then, at a second stage, the tool has been experimented with real and widely used code based on http. To this aim, the Web application framework Express[10] has been selected for our test: it offers a large number of HTTP utility methods and middleware functions for more rapid, efficient and robust development of HTTP APIs, and, for these reasons, several popular Node.js frameworks and Node.js Web applications are built on top of it.

### 7.1 Mining specifications from the http documentation

The documentation of the standard Node.js http module has been carefully inspected to find illegal uses of the provided features. As an example, Figure 9 shows an excerpt concerning http.ClientRequest; objects are internally created with this constructor and returned from http.request(). The highlighted sentences reveal two possible issues:

- in case the client defines a handler for the server response, data from the response must be consumed, otherwise a 'process out of memory' error can occur;

[9] Available at https://nodejs.org/api/http.html.
[10] See https://expressjs.com/.


If no `'response'` handler is added, then the response will be entirely discarded. However, if a `'response'` event handler is added, then the data from the response object **must** be consumed, either by calling `response.read()` whenever there is a `'readable'` event, or by adding a `'data'` handler, or by calling the `.resume()` method. Until the data is consumed, the `'end'` event will not fire. Also, until the data is read it will consume memory that can eventually lead to a 'process out of memory' error.

*Note*: Node.js does not check whether Content-Length and the length of the body which has been transmitted are equal or not.

**Figure 9** Excerpt from the http documentation at https://nodejs.org/api/http.html.

- unchecked length of the body of a request/response can block the communication, or data can be lost, as already shown in Section 5.2.

In the first case, the correct behavior can be captured by the following trace expression $\tau$ defined as follows:

$$\tau = \langle \mathsf{id};\, \mathit{onData}(\mathsf{id}) : ((\mathit{onEnd}(\mathsf{id}) : \varepsilon) \mid \tau) \rangle$$

where the variable id corresponds to the id of the specific response, whereas the event types *onData* and *onEnd* capture the handling of the 'data' and 'end' events, respectively. As shown in previous examples, combination of shuffle and recursion is essential to be able to deal with an arbitrary number of different responses which can be received by a client.

In the second case, the trace expression which is used for checking the correct behavior of code dealing with http.ClientRequest objects has already been presented in Section 5.2.

Besides the two properties discussed above, the following four additional specifications have been mined from the documentation and expressed with corresponding trace expressions:

- depending on the request received by the client or the response sent by the server, there are cases when the response must not contain a body, hence a server which tries to include a body in such situations does not behave correctly. The different cases require different treatments and trace expressions, as follows:
  - when the method of the request sent by the client is HEAD;
  - when the response of the server has status code 204 (no content);
  - when the method of the request sent by the client is a conditional GET and access is allowed, and the response server has status code 304 (not modified) because the requested document has not been modified.
- the method writeHead to write the head of a server response must only be called once on a message and it must happen before the method end is called to signal to the server that the response is complete. If this rule is not followed, then a default header, most likely different from the intended one, is written in the response.

While some of the mined specifications directly depend from the specific implementation of http, others are more related to the specification of the HTTP protocol; indeed, other trace expressions could be mined by looking at the standard specification of

HTTP, to dynamically check that a server (resp. client) implementation in Node.js verifies the HTTP requirements.

### 7.2 Testing the tool with simple applications

The tool has been tested with the trace expressions of the specifications presented in the previous section by performing runtime verification of simple servers/clients implemented in Node.js. For all specifications both correct and incorrect code has been developed, to verify the absence of both false negatives and positives.

Consider for instance the last specification presented in Section 7.1: the header of a server response must be written before calling the `end` method which finalizes the response.

A very simple server verifying the specification is the following one:

```
const http = require('http');

const server = http.createServer((req, res) => {
  // preparing response...
  res.writeHead(200);
  res.end(() => console.log('response sent'));
});

server.listen(80);
```

As expected, in this case the tool does not report any anomalous behavior. However, if we modify the code above as follows

```
const http = require('http');

const server = http.createServer((req, res) => {;
  res.end(() => console.log('response sent'));
  // BUG! writing header after calling end() method
  res.writeHead(204);
});

server.listen(80);
```

and monitor the server with our tool, then we get an error message associated with the following event, corresponding to a call to `writeHead`:

```
{ event: 'func_pre',
  name: 'writeHead',
  id: 12,
  res: undefined,
  args: [ 204 ],
  targetId: 10,
  resultId: undefined }
```

Similar examples have been used to test our tool with the other specifications.

### 7.3 Testing the tool with Express

After the simple experiments conducted in the first phase as described in the previous section, a natural next step towards the assessment of our tool consisted in testing popular real Node.js code based on `http`.

To this aim, the three components of Express depending on http (application.js, request.js and response.js) have been instrumented with Jalangi2, for a total of almost 1K SLOC. As for the other tests of Section 7.2, we have reused the same Jalangi2 instrumentation and trace expressions defined in Section 7.1.

Then the use of http by Express has been dynamically monitored with our tool through simple examples of server applications written with Express as the following one:

```
const express = require('express');
const morgan = require('morgan');
const responseTime = require('response-time');
const serveIndex = require('serve-index');

const path = process.cwd();
const app = express();

app.use(morgan('combined'));
app.use(responseTime());
app.use('/', express.static(path), serveIndex(path, {'icons': true}));
app.get('/*', (req, res) => {
  res.writeHead(404, {'Content-Type': 'text/html'});
  res.end('Not found\n');
});

app.listen(80);
```

Although the application contains only a dozen of source lines of code, the functionalities of the server are not so trivial, thanks to Express and the associated middleware morgan, response−time, serve−static and serve−index.

The morgan module creates a logger middleware function that allows the server to log (on the standard output, in this example) information about the received requests, while the response−time module creates a middleware that records the response time of the HTTP server, that is, the elapsed time from when a request enters the middleware to when the headers are written out to the client.

The serve−static middleware serves files from within a given root directory (in this case, the same directory from which the server has been launched), while serve−index serves an index (with displayed icons in this case) of the directory of the server based off the URL value of the request. In practice, the server allows inspection of the directory from which it has been launched, included all its subdirectories, and downloading of all files rooted at it.

Finally, the last call app.get deals with the case when no directory or file could be found. The tool has been tested with the server above, and a Node.js client sending requests with random paths based on its root directory at the frequency of three requests per second. The tool was able to monitor all relevant http events triggered in the above mentioned Express components used through the server, showing that our instrumentation and monitoring system can deal with code bases of considerable size using all the main features of the programming language. As expected, though, Express it is correct w.r.t. the previously described constraints on the use of the http module.

### 7.4 Benchmarks

| Language | Result | Factor |
|---|---:|---:|
| Results on Microsoft CLR | | |
| C# | 6.94 ms | 1.00x |
| RPython | 7.25 ms | 1.04x |
| IronPython | 1675.00 ms | 241.35x |
| Results on Mono | | |
| C# | 4.19 ms | 1.00x |
| RPython | 9.63 ms | 2.30x |
| IronPython | 1509.41 ms | 360.24x |
| Results on JVM | | |
| Java | 1.77 ms | 1.00x |
| RPython | 2.10 ms | 1.18x |
| Jython | 2918.90 ms | 1641.80x |

**Table 1** Benchmark results

## 8 Conclusion

We have presented the implementation of a prototype tool able to support parametric runtime verification of Node.js applications in a distributed environment.

The tool is based on parametric trace expressions, and exploits Jalangi2 for tracing events via code instrumentation, and SWI-Prolog for manipulating trace expressions and implementing the inference engine for verifying them.

Preliminary experiments have been conducted on http, one of the most popular standard Node.js modules, by mining from its documentation several specifications later expressed with trace expressions.

The mined specifications have been dynamically verified with the tool on several simple examples of HTTP servers and clients, and with a couple of more significant examples of servers built on top of Express; these last tests required code instrumentation of the Express components built on top http for a total of almost 1K of SLOC, and have shown that the prototype tool is able to support runtime verification of real Node.js code. To our knowledge, no other currently available tool allows parametric runtime verification of Node.js applications in a distributed environment.

With respect to out previous work [9], here we (a) presented a more complete set of examples, including HTTP interactions and the Express framework; (b) enhanced our implementation in order to keep track of target objects in method invocations (necessary for monitoring HTTP interactions) and to deal with cyclic objects; (c) optimized our monitoring system by making it asynchronous, so that it fits in the Node.js execution model without stopping the execution of the program everytime an event is observed; (d) measure our performance with benchmarks. More details can be found in the following subsections.

Although the reported experiments are promising, a considerable amount of work is still required for making our reaserach prototype a tool usable in practice.

### 8.1 Experiments

Runtime verification of code based on Express is an interesting result for the tool, nevertheless at the moment we are not able to determine to which degree our examples are able to cover uses of http features in the relevant components of Express.

To this aim, it would be useful to experiment the tool with real Node.js applications based on Express, and try to apply to our case techniques for code coverage [10] used in software testing.

On the side of specification mining, there are still other interesting properties that can be expressed an dynamically verified with parametric trace expressions which we are considering to mine both from the documentation of Node.js http module, and from the specification of HTTP. Furthermore, there is a number of other widely used Node.js modules and frameworks as async and jquery for which specification mining and runtime verification would be quite useful.

### 8.2 Performance

At the moment we have not focused on performance issues, because our first target was to have a proof of concept that it is possible to implement a tool for parametric runtime verification for Node.js applications in a distributed environment. As long as monitoring is used for software verification before deployment, problems with performance degradation are less stringent; however, the current implementation of our prototype still requires several optimizations to support runtime verification in practice.

A first practical way to obtain a speed up of the monitored application is to allow asynchronous communication between the instrumented code and the monitor; indeed, the current implementation is based on synchronous communication, because this is a simple way to guarantee that the monitor always receives events in the correct order. Asynchronous communication is more complex to implement and can be achieved in different ways; anyway, the speed up obtained by this solution depends on the responsiveness of the monitoring server, which cannot be easily estimated without experimental data, because it depends on the specifically employed trace expression for runtime verification.

Another useful optimization consists in reducing the network traffic between the monitored application and the monitor. This can be effectively achieved in the following ways.

The current instrumentation implemented with Jalangi2 sends to the monitor all events of the domain; in our case, all calls and returns from functions/callbacks. However, the instrumentation can be configured so that it sends only those events of the domain which are relevant for the verification.

We have discussed in Section 6 that problems with getters and circular objects prevented us to use JSON.stringify to serialize values of the events to be sent to the monitor. The customized version of stringify we have implemented for the tool can be optimized in several ways at the cost of a more complex code; for instance, objects are always serialized at arbitrary depth (up to circularity, of course), but for all kinds of verifications we have tested with our experiments, it is sufficient to serialize objects at fixed depth not larger than 4 levels.

Finally, to optimize connection time, it would be worth sending sequence of events rather than one event at time as happens in the current implementation.

### 8.3 Error reporting and IDE

So far error reporting has been neglected, and in case of failure the only available information regards the unexpected event that has been traced; clearly, more informative error messages are required to allow the user to understand and locate more easily the detected problem.

Jalangi2 already supports information about the location of the original code that has been instrumented, hence it should not be difficult to display the line of code and name of the file where the faulty event has originated.

For error messaging, the SWI-Prolog implementation already supports identification of all trace expressions that constitute a certain specification, hence associating specific error messages with them would be a simple extension.

Finally, at the moment, trace expressions must be written directly in Prolog and, although the used syntax is similar to that defined in the paper, the interpreter adopts non standard rules to manage operator precedences, and does not statically detect ill-formed trace expressions.

To this aim, we are developing in *Xtext*[11] an IDE to write specifications with trace expressions at a higher level, and to automatically generate the corresponding Prolog code. *Xtext* allows such an IDE to be integrated as an Eclipse plugin with support for syntax-highlighting, auto-completion, and a number of static checks useful for early error detection of the written specification.

## References

[1] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. "Behavioral Types in Programming Languages". In: *Foundations and Trends in Programming Languages* 3.2-3 (2016), pages 95–230.

[2] Davide Ancona, Daniela Briola, Amal El Fallah-Seghrouchni, Viviana Mascardi, and Patrick Taillibert. "Efficient Verification of MASs with Projections". In:

[11] https://eclipse.org/Xtext/

*Engineering Multi-Agent Systems - Second International Workshop, EMAS 2014, Paris, France, May 5-6, 2014, Revised Selected Papers*. 2014, pages 246–270.

[3] Davide Ancona, Daniela Briola, Angelo Ferrando, and Viviana Mascardi. "Global protocols as first class entities for self-adaptive agents". In: *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*. International Foundation for Autonomous Agents and Multiagent Systems. 2015, pages 1019–1029.

[4] Davide Ancona, Sophia Drossopoulou, and Viviana Mascardi. "Automatic Generation of Self-monitoring MASs from Multiparty Global Session Types in Jason". In: *Declarative Agent Languages and Technologies X - 10th International Workshop, DALT 2012, Valencia, Spain, June 4, 2012, Revised Selected Papers*. 2012, pages 76–95.

[5] Davide Ancona, Angelo Ferrando, Luca Franceschini, and Viviana Mascardi. "Parametric Trace Expressions for Runtime Verification of Java-Like Programs". In: *Proceedings of the 19th Workshop on Formal Techniques for Java-like Programs*. FTFJP'17. Barcelona, Spain: ACM, 2017, 10:1–10:6.

[6] Davide Ancona, Angelo Ferrando, and Viviana Mascardi. "Comparing Trace Expressions and Linear Temporal Logic for Runtime Verification". In: *Theory and Practice of Formal Methods - Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*. 2016, pages 47–64.

[7] Davide Ancona, Angelo Ferrando, and Viviana Mascardi. "Comparing trace expressions and linear temporal logic for runtime verification". In: *Theory and Practice of Formal Methods*. Springer, 2016, pages 47–64.

[8] Davide Ancona, Angelo Ferrando, and Viviana Mascardi. "Parametric Runtime Verification of Multiagent Systems". In: *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2017, São Paulo, Brazil, May 8-12, 2017*. 2017, pages 1457–1459.

[9] Davide Ancona, Luca Franceschini, Giorgio Delzanno, Maurizio Leotta, Marina Ribaudo, and Filippo Ricca. "Towards Runtime Monitoring of Node.js and Its Application to the Internet of Things". In: *Proceedings of the 1st Workshop on Architectures, Languages and Paradigms for IoT*. To appear. 2017.

[10] Cyrille Artho, Howard Barringer, Allen Goldberg, Klaus Havelund, Sarfraz Khurshid, Mike Lowry, Corina Pasareanu, Grigore Roşu, Koushik Sen, Willem Visser, et al. "Combining test case generation and runtime verification". In: *Theoretical Computer Science* 336.2-3 (2005), pages 209–234.

[11] Andreas Bauer, Martin Leucker, and Christian Schallhart. "Model-based runtime analysis of distributed reactive systems". In: *Software Engineering Conference, 2006. Australian*. IEEE. 2006, 10–pp.

[12] Andreas Bauer, Martin Leucker, and Christian Schallhart. "Runtime verification for LTL and TLTL". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 20.4 (2011), page 14.

[13] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[14] Frank S de Boer and Stijn de Gouw. "Combining monitoring with run-time assertion checking". In: *Formal Methods for Executable Software Models*. Springer, 2014, pages 217–262.

[15] F.S. de Boer, S. de Gouw, E.B. Johnsen, A. Kohn, and P. Y. H. Wong. "Run-Time Assertion Checking of Data- and Protocol-Oriented Properties of Java Programs: An Industrial Case Study". In: *Trans. Aspect-Oriented Software Development* 11 (2014), pages 1–26.

[16] Daniela Briola, Viviana Mascardi, and Davide Ancona. "Distributed Runtime Verification of JADE Multiagent Systems". In: *Intelligent Distributed Computing VIII - Proceedings of the 8th International Symposium on Intelligent Distributed Computing, IDC 2014, Madrid, Spain, September 3-5, 2014*. 2014, pages 81–91.

[17] Mark Brörkens and Michael Möller. "Dynamic Event Generation for Runtime Checking using the JDI11This work was partially funded by the German Research Council (DFG) under grant OL 98/3-1." In: *Electronic Notes in Theoretical Computer Science* 70.4 (2002), pages 21–35.

[18] Feng Chen and Grigore Rosu. "Java-MOP: A Monitoring Oriented Programming Environment for Java." In: *TACAS*. Volume 3440. Springer. 2005, pages 546–550.

[19] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[20] Bruno Courcelle. "Fundamental Properties of Infinite Trees". In: *Theor. Comput. Sci.* 25 (1983), pages 95–169.

[21] Laura González, Javier Cubo, Antonio Brogi, Ernesto Pimentel, and Raúl Ruggia. "Run-time verification of behaviour-aware mashups in the internet of things". In: *European Conference on Service-Oriented and Cloud Computing*. Springer. 2013, pages 318–330.

[22] Brendan Gregg and Jim Mauro. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*. Prentice Hall Professional, 2011.

[23] Sylvain Hallé, Tevfik Bultan, Graham Hughes, Muath Alkhalaf, and Roger Villemaire. "Runtime verification of web service interface contracts". In: *Computer* 43.3 (2010).

[24] Sylvain Hallé and Roger Villemaire. "Runtime Verification for the Web". In: *Runtime Verification*. Springer. 2010, pages 106–121.

[25] K. Lei, Y. Ma, and Z. Tan. "Performance Comparison and Evaluation of Web Development Technologies in PHP, Python, and Node.js". In: *2014 IEEE 17th International Conference on Computational Science and Engineering*. Dec. 2014, pages 661–668. DOI: 10.1109/CSE.2014.142.

[26] Martin Leucker and Christian Schallhart. "A brief account of runtime verification". In: *The Journal of Logic and Algebraic Programming* 78.5 (2009), pages 293–303.

[27] Michael Martin, Benjamin Livshits, and Monica S Lam. "Finding application errors and security flaws using PQL: a program query language". In: *ACM SIGPLAN Notices*. Volume 40. 10. ACM. 2005, pages 365–383.

[28] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[29] Amir Pnueli. "The temporal logic of programs". In: *Foundations of Computer Science, 1977., 18th Annual Symposium on*. IEEE. 1977, pages 46–57.

[30] Carl Martin Rosenberg, Martin Steffen, and Volker Stolz. "Leveraging DTrace for Runtime Verification". In: *Runtime Verification: 16th International Conference, RV 2016, Madrid, Spain, September 23–30, 2016, Proceedings*. Edited by Yliès Falcone and César Sánchez. Cham: Springer International Publishing, 2016, pages 318–332. ISBN: 978-3-319-46982-9. DOI: 10.1007/978-3-319-46982-9_20. URL: https://doi.org/10.1007/978-3-319-46982-9_20.

[31] Luke Simon, Ajay Mallya, Ajay Bansal, and Gopal Gupta. "Coinductive Logic Programming". In: *Logic Programming, 22nd International Conference, ICLP 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*. 2006, pages 330–345.

[32] S. Tilkov and S. Vinoski. "Node.js: Using JavaScript to Build High-Performance Network Programs". In: *IEEE Internet Computing* 14.6 (Nov. 2010), pages 80–83. ISSN: 1089-7801. DOI: 10.1109/MIC.2010.145.

[33] Arild B Torjusen, Habtamu Abie, Ebenezer Paintsil, Denis Trcek, and Åsmund Skomedal. "Towards run-time verification of adaptive security for IoT in eHealth". In: *Proceedings of the 2014 European Conference on Software Architecture Workshops*. ACM. 2014, page 4.

## About the authors

**Davide Ancona** davide.ancona@unige.it.

**Luca Franceschini** luca.franceschini@dibris.unige.it.

**Giorgio Delzanno** giorgio.delzanno@unige.it.

**Maurizio Leotta** maurizio.leotta@unige.it.

**Marina Ribaudo** marina.ribaudo@unige.it.

**Filippo Ricca** filippo.ricca@unige.it.