Towards Runtime Monitoring of Node.js and Its Application to the Internet of Things

Davide Ancona, Luca Franceschini, Giorgio Delzanno, Maurizio Leotta
Marina Ribaudo, Filippo Ricca
DIBRIS, University of Genoa
{*name.surname*}*@unige.it, luca.franceschini@dibris.unige.it*

In the last years Node.js has emerged as a framework particularly suitable for implementing lightweight IoT applications, thanks to its underlying asynchronous event-driven, non blocking I/O model. However, verifying the correctness of programs with asynchronous nested callbacks is quite difficult, and, hence, runtime monitoring can be a valuable support to tackle such a complex task.

Runtime monitoring is a useful software verification technique that complements static analysis and testing, but has not been yet fully explored in the context of Internet of Things (IoT) systems. Trace expressions have been successfully employed for runtime monitoring in widespread multiagent system platforms. Recently, their expressive power has been extended to allow parametric specifications on data that can be captured and monitored only at runtime. Furthermore, they can be language and system agnostic, through the notion of event domain and type. This paper investigates the use of parametric trace expressions as a first step towards runtime monitoring of programs developed in Node.js and Node-RED, a flow-based IoT programming tool built on top of Node.js. Runtime verification of such systems is a task that mostly seems to have been overlooked so far in the literature.

A prototype implementing the proposed system for Node.js, in order to dynamically check with trace expressions the correct usage of API functions, is presented. The tool exploits the dynamic analysis framework Jalangi for monitoring Node.js programs and allows detection of errors that would be difficult to catch with other techniques. Furthermore, it offers a simple REST interface which can be exploited for runtime verification of Node-RED components, and, more generally, IoT devices.

1 Introduction

Node.js[1] is a platform for server-side JavaScript applications which is based on an event-driven non blocking I/O model, expressly designed to support easy and rapid development of lightweight servers able to manage a considerable number of simultaneous requests. For these features, and the huge module open ecosystem and large associated community of developers, Node.js has emerged as a convenient platform for developing IoT applications.

Node-RED[2] is a visual tool built on top of Node.js to easily wire together mobile devices, sensors, and, more in general, real-world events, and to add the necessary logic to integrate them with databases, messaging systems, and cloud platforms. Thanks to its lightweight implementation based on Node.js, it can run on devices at the edge of the network, such as the Raspberry Pi[3]. Node-RED turns out to be extremely useful also for implementing mocks for sensors and mobile devices for acceptance testing of IoT systems [16]. For all these reasons, it is an ideal solution for rapid and effective development of innovative solutions to the challenges of the IoT technical revolution.

[1] `https://nodejs.org`
[2] `https://nodered.org/`
[3] `https://www.raspberrypi.org/`

D. Pianini and G. Salvaneschi (Eds.): First workshop on Architectures, Languages and Paradigms for IoT
EPTCS 264, 2018, pp. 27–42, doi:10.4204/EPTCS.264.4

© Ancona, Franceschini, Delzanno, Leotta, Ribaudo & Ricca
This work is licensed under the
Creative Commons Attribution License.

To fully exploit its characteristics, Node.js strongly relies on asynchronous and continuation passing style programming: I/O operations are performed through calls to asynchronous functions where a callback must be passed to specify how the computation continues once the called I/O operation completed asynchronously [4]. Indeed, the Node.js execution model consists of a main *event loop* which is run on a single-threaded process. Despite this fact, with asynchronous programming very subtle bugs can occur as in concurrent programs. Furthermore, the dynamic nature of JavaScript makes error detection even harder. Finally, in IoT scenarios which typically rely on the interaction of several distributed components, ensuring the correctness of a system becomes a challenging task [16].

Runtime verification [17] is a lightweight verification technique complementing software testing and formal verification. Dynamic checking of the correct behavior of a system is performed by a *monitor* which verifies that the trace of events captured at runtime is compliant with the expected behavior expressed with a specification formalism, thus essentially solving a “word problem” (that is, deciding whether a word is included in a language). This approach has been successfully employed in the context of distributed applications such as Web applications [14, 15] and multi-agent systems [5], and of real-time systems [12]. Runtime verification avoids the complexity of traditional formal verification approaches at the expenses of less coverage by analyzing only a finite set of execution traces, and works directly with the actual system, thus scaling up very well [6]. As opposite to software testing, runtime verification offers the possibility to recover from detected violations with suitable handlers [3].

Trace expressions have been inspired by previous work on behavioral types [5, 2] to specify and verify at runtime the correctness of interaction protocols in multiagent systems. They have been successfully adopted for several purposes and applications in the context of different multiagent system platforms [10]. Recently, they have been extended [7] to allow specifications to be *parametric* [18] in data that can be captured and monitored only at runtime. Thanks to this extension, specifications which, for instance, depend on the values exchanged by objects through methods, or on the dynamically evolving collection of objects or resources available at runtime, can be suitably modeled, and the number of correct programming patterns that can be specified is significantly enlarged.

This paper investigates the use of trace expressions as a first step towards runtime monitoring of programs developed in Node.js and Node-RED, a flow-based IoT programming tool built on top of Node.js. By inspecting the Node.js API reference documentation, we have identified several patterns that require calls to asynchronous functions and, possibly, to their corresponding callbacks, to occur in a correct order to prevent the application to exhibit unsafe and unpredictable behavior. Such patterns can be succinctly defined by means of trace expressions. The tool exploits the dynamic analysis framework Jalangi [19] for monitoring Node.js programs. Furthermore, it offers a simple REST interface which can be exploited for runtime verification of Node-RED components. To the best of our knowledge, no other similar approaches for runtime verification of Node.js systems have been proposed in the literature.

After providing the necessary background concerning trace expressions in Section 2, we turn to consider examples of Node.js code with subtle bugs and show how runtime monitoring through trace expressions can help to detect such bugs (Section 3). The implementation details of our prototype tool are discussed in Section 4, together with examples of its use for monitoring Node.js applications and Node-RED flows. Finally, Section 5 concludes and outlines directions for future work.

2 Parametric trace expressions

A trace expression defines the set of all possible event traces that can be correctly observed during the execution of the program under monitoring, thus serving as a specification. Its semantics is defined by a

labeled transition system where labels correspond to the events of the system that have to be monitored: if a trace expression τ rewrites with event e into a new trace expression τ', this means that the occurrence of e corresponds to a correct behavior according to τ, and τ' represents the new state to be monitored in the continuation of the system; otherwise, an anomalous behavior is detected.

The rewriting rules defining the semantics of trace expressions can be directly turned into an algorithm for event trace recognition. In this way, a monitor for dynamically verifying the correct behavior of a system specified by a trace expression τ can be simply obtained from τ, and from the implementation of the labeled transition system.

To make trace expressions language and system agnostic, the notions of *event domain* and *type* are introduced. Abstractly, an event domain consists of a set of events $\mathcal{E}$ together with their semantics and structure. More operationally, an event domain corresponds to the infrastructure required for capturing and conveniently representing at runtime the set of events that are emitted during the execution of the system, and that have to be monitored to ensure its correct functioning. Typically, an event domain is implemented through code instrumentation.

As we will see in the rest of the paper, a possible example of an event domain supporting Node.js consists of calls to asynchronous functions and the execution of their associated callbacks. Such events have useful information associated with them, like the name of the functions that is being invoked (if not anonymous), its parameters and the returned value. Typical events for an event domain supporting Node-RED correspond to messages sent with several protocols (HTTP, WebSocket, MQTT, etc.) and may contain information such as the payload, the timestamp, the topic, the sender, and the receiver of the message.

Events are abstracted by event types which play two important roles:

- They provide a simple language that allows trace expressions to be defined independently of the underlying event domains. In fact, the trace expression language is stratified in two different layers. The basic layer models an event domain and consists of a simple language of terms corresponding to event types. The top layer defines several operators, whose semantics is independent from the notion of event domain, for building trace expressions on top of event types.
- They enhance the expressive power of trace expressions, by allowing the definition of particular sets of events which are useful for monitoring purposes. The semantics of event types is defined by the generic function *match* which specifies how events match event types. The rewriting rules of the labeled transition system are parametric in the function *match* which is simply assumed to be given. No assumptions are needed on the language used for defining *match*. In practice, we have found convenient to define *match* as a Prolog predicate (see Section 4).

An event type ϑ is a term which can contain free variables. The semantics of ϑ is determined by *match* as follows: we say that $match(e,\vartheta)$ *succeeds* iff there exists a substitution σ s.t. $match(e,\vartheta) = \sigma$, otherwise we say that $match(e,\vartheta)$ *fails*. σ is a finite partial map whose domain $dom(\sigma)$ must coincide with the set of variables in ϑ. Its codomain is a universe of values $\mathcal{V}$ which depends on the considered event domain.

In the following, we provide some examples of event types associated with the event domain we have implemented for Node.js and that is further specified in Sections 3 and 4. One type of event that is useful to trace in Node.js (but also in other programming languages) consists in calls to and returns from functions. Correspondingly, an event domain may include the following two basic event types:

- $\mathit{fun_pre}(\mathit{name},\mathit{args})$ matches all events corresponding to a call to a function with name *name*, and list of arguments *args*. For instance, the event type $\mathit{fun_pre}($`'fs.writeSync'`,`[9,'hello']`$)$ matches all calls to the function `fs.writeSync`, with arguments `9` and `'hello'`, respectively.

$$\text{(main)}\frac{\tau \overset{e}{\rightarrowtail} \tau';\emptyset}{\tau \xrightarrow{e} \tau'} \qquad \text{(prefix)}\frac{}{\vartheta:\tau \overset{e}{\rightarrowtail} \tau;\sigma}\ \sigma{=}match(e,\vartheta) \qquad \text{(or-l)}\frac{\tau_1 \overset{e}{\rightarrowtail} \tau_1';\sigma}{\tau_1 \vee \tau_2 \overset{e}{\rightarrowtail} \tau_1';\sigma} \qquad \text{(or-r)}\frac{\tau_2 \overset{e}{\rightarrowtail} \tau_2';\sigma}{\tau_1 \vee \tau_2 \overset{e}{\rightarrowtail} \tau_2';\sigma}$$

$$\text{(and)}\frac{\tau_1 \overset{e}{\rightarrowtail} \tau_1';\sigma_1 \quad \tau_2 \overset{e}{\rightarrowtail} \tau_2';\sigma_2}{\tau_1 \wedge \tau_2 \overset{e}{\rightarrowtail} \tau_1' \wedge \tau_2';\sigma}\ \sigma{=}\sigma_1{\cup}\sigma_2 \qquad \text{(shuffle-l)}\frac{\tau_1 \overset{e}{\rightarrowtail} \tau_1';\sigma}{\tau_1 \mid \tau_2 \overset{e}{\rightarrowtail} \tau_1' \mid \tau_2;\sigma} \qquad \text{(shuffle-r)}\frac{\tau_2 \overset{e}{\rightarrowtail} \tau_2';\sigma}{\tau_1 \mid \tau_2 \overset{e}{\rightarrowtail} \tau_1 \mid \tau_2';\sigma}$$

$$\text{(cat-l)}\frac{\tau_1 \overset{e}{\rightarrowtail} \tau_1';\sigma}{\tau_1 \cdot \tau_2 \overset{e}{\rightarrowtail} \tau_1' \cdot \tau_2;\sigma} \qquad \text{(cat-r)}\frac{\tau_2 \overset{e}{\rightarrowtail} \tau_2';\sigma}{\tau_1 \cdot \tau_2 \overset{e}{\rightarrowtail} \tau_2';\sigma}\ \epsilon(\tau_1) \qquad \text{(var-t)}\frac{\tau \overset{e}{\rightarrowtail} \tau';\sigma}{<\mathsf{x};\tau> \overset{e}{\rightarrowtail} \sigma\tau';\sigma_{\setminus \mathsf{x}}}\ \mathsf{x}{\in}dom(\sigma)$$

$$\text{(var-f)}\frac{\tau \overset{e}{\rightarrowtail} \tau';\sigma}{<\mathsf{x};\tau> \overset{e}{\rightarrowtail} <\mathsf{x};\tau'>;\sigma}\ \mathsf{x}{\notin}dom(\sigma) \qquad (\epsilon\text{-empty})\frac{}{\epsilon(\epsilon)} \qquad (\epsilon\text{-var})\frac{\epsilon(\tau)}{\epsilon(<\mathsf{x};\tau>)} \qquad (\epsilon\text{-or-l})\frac{\epsilon(\tau_1)}{\epsilon(\tau_1 \vee \tau_2)}$$

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

Figure 1: Transition system for parametric trace expressions

Analogously, $fun_pre($`'fs.writeSync'`$,\mathsf{args})$ matches all calls to the function `fs.writeSync`, with any list of arguments. In this case, a successful match will produce the appropriate substitution associating the actual list of arguments with the variable args.

- $fun_post(name,args,ret)$ matches all events corresponding to a return from a function with name *name*, list of arguments *args*, and returned value *ret*. For instance, the event type $fun_post($`'fs.openSync'`,`['tmp.txt','w']`,`9`$)$ matches all returns from the function `fs.openSync`, with arguments `'tmp.txt'` and `'w'`, and returned value `9`, respectively.

A parametric trace expression τ is a regular term[4] built on top of the following operators[5]:

- ϵ (empty trace)
- $\vartheta:\tau$ (*prefix*)
- $\tau_1 \cdot \tau_2$ (*concatenation*)
- $\tau_1 \wedge \tau_2$ (*intersection*)
- $\tau_1 \vee \tau_2$ (*union*)
- $\tau_1 \mid \tau_2$ (*shuffle*, a.k.a. *interleaving*)
- $<\mathsf{x};\tau>$ (*binder*)

The trace expression $\sigma\tau$ obtained from τ by substituting all free occurrences of $\mathsf{x} \in dom(\sigma)$ in τ with $\sigma(\mathsf{x})$, is coinductively defined as follows:

$$\sigma(\vartheta:\tau) = (\sigma\vartheta):(\sigma\tau) \quad \sigma(\tau_1\ op\ \tau_2) = (\sigma\tau_1)\ op\ (\sigma\tau_2) \text{ for } op \in \{\vee,\wedge,|,\cdot\} \quad \sigma(<\mathsf{x};\tau>) = <\mathsf{x};\sigma_{\setminus \mathsf{x}}\tau>$$

The labeled transition system for parametric trace expressions can be found in Figure 1.

We denote with $\emptyset$ the substitution with the empty domain. The equality $\sigma = \sigma_1 \cup \sigma_2$ holds iff $dom(\sigma) = dom(\sigma_1) \cup dom(\sigma_2)$, and for all $\mathsf{x} \in dom(\sigma)$, $\sigma(\mathsf{x}) = \sigma_1(\mathsf{x})$ if $\mathsf{x} \in dom(\sigma_1)$, and $\sigma(\mathsf{x}) = \sigma_2(\mathsf{x})$ if $\mathsf{x} \in dom(\sigma_2)$ (hence, σ_1 and σ_2 must coincide on $dom(\sigma_1) \cap dom(\sigma_2)$). We denote with $\sigma_{\setminus \mathsf{x}}$ the substitution where x is removed from its domain: $\sigma_{\setminus \mathsf{x}} = \sigma'$ iff $dom(\sigma') = dom(\sigma) \setminus \{\mathsf{x}\}$ and for all $\mathsf{x} \in dom(\sigma')$, $\sigma'(\mathsf{x}) = \sigma(\mathsf{x})$. The notation $\sigma\vartheta$ denotes the event type obtained from ϑ by substituting all occurrences of $\mathsf{x} \in dom(\sigma)$ in ϑ with $\sigma(\mathsf{x})$.

Rule (main) defines the transition relation $\tau \xrightarrow{e} \tau'$ in terms of the auxiliary relation $\tau \overset{e}{\rightarrowtail} \tau';\sigma$, which returns the substitution σ computed during the transition step. This rule can be applied only if the computed substitution is empty, since trace expressions are not allowed to contain free variables.

[4] Since regular terms can be expressed through a finite number of syntactic equations, no explicit recursion operator is needed.

[5] Infix binary operators associate from left, and are listed in decreasing order of precedence, that is, the first operator has the highest precedence.

In rule (prefix) a transition step is possible only when the current event e matches the event type ϑ, and the computed substitution is returned by the *match* function.

Rules for union and shuffle are straightforward: union corresponds to the choice between τ_1 and τ_2, while shuffle consists of the interleaving of τ_1 and τ_2.

For concatenation the auxiliary judgment $\varepsilon(\tau)$ (defined in the same Figure) is needed for identifying the trace expression that accepts the empty trace ε. More precisely, rule (cat-r) states that $\tau_1 \cdot \tau_2$ accepts a trace $\bar{e}$ if τ_1 accepts the empty trace (side condition $\varepsilon(\tau_1)$), and τ_2 accepts $\bar{e}$.

In rule (and) the side condition requires that the substitutions σ_1 and σ_2 computed for τ_1 and τ_2 must coincide on the intersection of their domains. The final substitution σ is obtained by merging σ_1 and σ_2. For instance, if $\tau = \mathit{fun_post}(\mathsf{name}, \mathsf{args}, 9) : \tau_1 \wedge \mathit{fun_post}(\mathsf{name}, [\texttt{'tmp.txt'}, \texttt{'w'}], \mathsf{ret}) : \tau_2$ and the event e corresponds to the return from the call `fs.openSync('tmp.txt','w')` with value 9, then we have

$$\mathit{fun_post}(\mathsf{name}, \mathsf{args}, 9) : \tau_1 \xrightarrow{e} \tau_1; \{\mathsf{name} \mapsto \texttt{'fs.openSync'}, \mathsf{args} \mapsto [\texttt{'tmp.txt'}, \texttt{'w'}]\}$$
$$\mathit{fun_post}(\mathsf{name}, [\texttt{'tmp.txt'}, \texttt{'w'}], \mathsf{ret}) : \tau_2 \xrightarrow{e} \tau_2; \{\mathsf{name} \mapsto \texttt{'fs.openSync'}, \mathsf{ret} \mapsto 9\}$$

therefore $\tau \xrightarrow{e} \tau_1 \wedge \tau_2; \{\mathsf{name} \mapsto \texttt{'fs.openSync'}, \mathsf{args} \mapsto [\texttt{'tmp.txt'}, \texttt{'w'}], \mathsf{ret} \mapsto 9\}$.

Two rules are required for the $<\mathsf{x}; \tau>$ construct. Rule (var-t) is needed when variable x is contained in the domain of the computed substitution σ. σ is applied to the trace expression τ' in which τ rewrites to, the binder is removed and x is removed from the domain of the computed substitution ($\sigma_{\setminus \mathsf{x}}$). Rule (var-f) is required when variable x is not contained in the domain of the computed substitution σ: the binder is not removed, and the computed substitution coincides with σ.

Rules for the auxiliary predicate $\varepsilon(_)$ are straightforward.

The semantics $[\![\tau]\!]$ of a trace expression τ is defined in terms of the transition relation $\xrightarrow{e}$, and the predicate $\varepsilon(_)$:

- The empty trace ε belongs to $[\![\tau]\!]$ iff $\varepsilon(\tau)$ is derivable.
- If $\bar{e}$ is a non empty, possibly infinite event trace, then $\bar{e}$ belongs to $[\![\tau]\!]$ iff there exists a possibly infinite reduction sequence starting from τ for $\bar{e}$, that is, $\tau = \tau_0 \xrightarrow{e_1} \tau_1 \ldots \tau_{n-1} \xrightarrow{e_n} \tau_n \ldots$, and $\bar{e} = e_1 \ldots e_n \ldots$.

3 Runtime monitoring of Node.js applications

In this section we will provide simple examples of Node.js code to show how asynchronous programming can introduce subtle bugs in Node.js applications, and how some of these bugs can be detected with the help of runtime monitoring. To this aim, we will define trace expressions to ensure the safe use of the asynchronous functions provided by the `fs` module. Such trace expressions are based on an event domain able to deal with calls to and returns from asynchronous functions, and their associated callbacks.

We start by showing a simple Node.js example which manipulates files through the `fs` module in a synchronous way:

```
const fs=require('fs')
var fd=fs.openSync('tmp.txt','w')
fs.writeSync(fd,'Hello world!\n')
fs.closeSync(fd)
```

While for Node.js applications the use of asynchronous functions is more customary, for simplicity, in this first example we use the synchronous version of the open, write and close functions for manipulating files. In this case, the expected correct pattern for writing data on a file is the standard one: open the file, write data on it, possibly several times, then close it. This behavior can be defined by the following trace expression T:

$$T = \varepsilon \vee open : W \qquad W = write : W \vee close : \varepsilon$$

For simplicity, we have abstracted away several details, and the event types in the trace expression refer to the corresponding operations performed on a single file, hence, there is a unique implicit file descriptor, and the specification is not parametric. To make it parametric, a binder needs to be used to introduce the variable fd which is instantiated at runtime with the file descriptor returned by the open operation:

$$PT = \varepsilon \vee <\mathsf{fd}\,; open(\mathsf{fd}) : (PW \mid PT)> \qquad PW = write(\mathsf{fd}) : PW \vee close(\mathsf{fd}) : \varepsilon$$

The event type $open(\mathsf{fd})$ captures all events corresponding to a return from `fs.openSync` with value fd. In other words, by using the event types introduced in the previous section, the set of events matched by $open(\mathsf{fd})$ coincides with the set of events matched by $fun_post($`'fs.openSync'`$, [fname$ `,` `'w'`$], \mathsf{fd})$ for all strings *fname*. The other event types $write(\mathsf{fd})$ and $close(\mathsf{fd})$ are defined analogously.

The main trace expression PT is recursively defined and the shuffle operator is needed to correctly taking into account the possibility that several files can be opened and manipulated simultaneously. Shuffle expresses the fact that operations on different file descriptors can be safely interleaved.

To give an example of reduction, let us assume that the following two subsequent events are captured (or, in other words, the following events are received by the monitor): `fs.openSync(`*fname₁*`,'w')` returns file descriptor 9, `fs.openSync(`*fname₂*`,'w')` returns file descriptor 10. Then, the trace expression PT defined above successfully reduces with two transition steps into the following trace expression PT':

$$PT' = PW_1 \mid PW_2 \mid PT \quad PW_1 = write(9) : PW_1 \vee close(9) : \varepsilon \quad PW_2 = write(10) : PW_2 \vee close(10) : \varepsilon$$

In other words, the program is correct with respect to the given specification.

The previous code example does not exploit the advantages offered by Node.js in terms of performances, since synchronous operations are used. Let us now consider a more complex asynchronous version:

```
const fs=require('fs')
fs.open('tmp.txt','w',(err,fd)=>{
    if(!err)
        fs.write(fd,'Hello world!\n',()=>fs.close(fd,()=>{}))
})
```

In comparison to the synchronous version, this code is definitely less readable, and chances to introduce bugs are higher. In this case the code is correct, and exhibits three callbacks defined by arrow functions (lambda expressions in the JavaScript jargon) at three different nesting levels: the callback passed to `open` which is called as soon as the file has been opened, with an error object[6] and the file descriptor associated with the newly opened file passed as arguments. The callback passed to `write` which, for simplicity, ignores its arguments and calls `close`, and, at the deepest level, the callback passed to `close`, which does not perform any operation.

[6]For sake of simplicity, we have removed all code that would be necessary for correctly handling errors.

If we observe the events matching the *fun_post* event type while running the code fragment above, we get the following sequence: return from `fs.open`, return from its associated callback, return from `fs.write`, return from its associated callback, return from `fs.close`, and, finally, return from its associated callback. This is, indeed, the correct sequence of expected events: a callback associated with an asynchronous operation on a file descriptor needs to be executed **before** a subsequent asynchronous operation is performed on the same file descriptor. For instance, if a monitor detects a return from `fs.close(`*fd*`)` before a return from the callback of `fs.write(`*fd*`,`*data*`)`, then it has to signal an anomaly, since the monitored program is trying to close a file before a write operation on it is completed.

A common source of bugs in Node.js consists in using asynchronous functions as they were synchronous. This may be reasonable in situations where a controlled number of operations can be safely interleaved, as happens, for instance, if a program has to send a bunch of HTTP requests to different servers, or even to the same server, providing that they can be safely handled in any order. But there are many other situations where extreme care is needed when calling asynchronous functions to avoid subtle bugs. Unfortunately, this is not always explicitly stated in the Node.js API reference documentation. There are, however, cases where the documentation is clear about this issue: for instance, one can find the following notice regarding the specification of `fs.write`: *"Note that it is unsafe to use `fs.write` multiple times on the same file without waiting for the callback"*.

Let us consider, for instance, the following unsafe example[7]:

```
const fs=require('fs')
const limit=2**24
fs.open('tmp.txt','w',(err,fd)=>{
    if(!err)
        for(let i=0;i<limit;i++)
            fs.write(fd,i+'\n',()=>{})
    fs.close(fd,()=>{})
})
```

The behavior of such a program is non deterministic, and the expected correct order of the integers written on the file is not ensured. Even when the programmer is deliberately fine with this non deterministic behavior, this program will exhibit serious heap memory problems if the constant `limit` holds values which are large enough: either the program crashes[8] after some time with a fatal error due to JavaScript heap out of memory, or successfully terminates, but it exhibits very poor performance.

To guarantee that functions are called only after some callbacks are completed, we have to refine the event domain presented in the previous section to be able to track callbacks associated with function calls. To this aim, coupling of calls to asynchronous functions and their corresponding callbacks is achieved by emitting a unique identifier associated with both. Accordingly, the event types *fun_pre* and *fun_post* are extended: $fun_pre(name, id, args)$ (and, similarly, event type *fun_post*) matches a call to a function **which is not a callback**, having name *name*, **identifier** *id*, and argument list *args*. Furthermore, we introduce the new event types *cb_pre* and *cb_post* corresponding to a *call to* and a *return from*, respectively, a callback: $cb_pre(name, id, args)$ (and, similarly, event type *cb_post*) matches a call to a **callback**, with name *name*, identifier *id*, and argument list *args*. Matching a callback with its associated asynchronous function call is made possible by the fact that both calls carry the same identifier.

With this more expressive event domain we can now provide a trace expression *AT* identifying the correct pattern for using functions `open`, `write`, and `close` of module `fs`. Not only the sequence `open`,

[7] The `**` notation stands for exponentiation.

[8] We have experienced this behavior for `limit=`2^{24} by running the code above with Node.js v7.2.1 on an Intel(R) Core(TM) i7-4510U CPU at 2.00GHz with a 16GB RAM, getting the error after several minutes.

write, and close must be respected, but also asynchronous operations on the same file descriptor must be invoked after the callback associated with the previous operation has been called:

$$AT = \varepsilon \vee <\mathsf{id}\,; open(\mathsf{id}) : (CB \,|\, AT)> \qquad CB = <\mathsf{fd}\,; cb(\mathsf{id},\mathsf{fd}) : AW>$$
$$AW = <\mathsf{id}_2\,; write(\mathsf{id}_2,\mathsf{fd}) : cb(\mathsf{id}_2) : AW> \vee <\mathsf{id}_3\,; close(\mathsf{id}_3,\mathsf{fd}) : cb(\mathsf{id}_3) : \varepsilon>$$

In comparison to the trace expression PT for the synchronous case, the event types *open*, *write*, and *close* have been extended to include the identifiers of the corresponding call. Furthermore, function calls (event type *fun_pre*) rather than returns (event type *fun_post*) are tracked, because the asynchronous version of open does not return any file descriptor: that value is retrieved as an argument passed to the associated callback.

The event type *cb* (defined in terms of *cb_pre*) corresponds to calls to callbacks. It must depend on the corresponding identifier id, and, optionally, the arguments passed to the callback. In this example, the fd argument passed to the open callback is considered, whereas for the callbacks associated with write and close arguments are ignored.

In AT two different binders id and fd are required: the former captures the identifier associated with a call to open to be able to match its subsequent callback, while the latter captures the file descriptor on which the callback of open is called.

Two more binders id_2 and id_3 are required to match the callbacks associated with calls to write and close. For clarity, we use three different identifiers id, id_2 and id_3, but we could have equivalently employed the same name for all three cases, thanks to the scoping rules of the binder construct.

Similarly to what happens for the PT trace expression in the synchronous case, also here recursion and shuffle allow interleaving of several events: different calls to open, each uniquely identified by id, and calls to write/close with different file descriptors.

Finally, we consider an example of a rewriting reduction in case of error. Let us suppose that a file is correctly opened and then two subsequent writes are performed without waiting for any callback to be executed. The reduction below accept the trace up-to the first write operation; after that, the second one would be rejected:

$$
\begin{array}{lr}
AT \xrightarrow{e_1} <\mathsf{fd}\,; cb(42,\mathsf{fd}) : AW> \,|\, AT & \{\mathsf{id} \mapsto 42\} = match(e_1, open(\mathsf{id})) \\
\quad \xrightarrow{e_2} <\mathsf{id}_2\,; write(\mathsf{id}_2,9) : cb(\mathsf{id}_2) : AW> \vee <\mathsf{id}_3\,; close(\mathsf{id}_3,9) : cb(\mathsf{id}_3) : \varepsilon> & \{\mathsf{fd} \mapsto 9\} = match(e_2, cb(42,\mathsf{fd})) \\
\quad \xrightarrow{e_3} (cb(43) : AW) \vee <\mathsf{id}_3\,; close(\mathsf{id}_3,9) : cb(\mathsf{id}_3) : \varepsilon> & \{\mathsf{id}_2 \mapsto 43\} = match(e_3, write(\mathsf{id}_2,9))
\end{array}
$$

4 Implementation

The implementation of the runtime verification system for Node.js can be divided in two main parts. On one hand, we (statically) instrument the source code of the program that needs to be verified, adding a piece of code that is able to capture all the relevant events for the domain in use. On the other hand, we have a Prolog server implementing the operational semantics of trace expressions and offering a simple REST interface. The monitor and the server communicate through HTTP requests and responses, effectively implementing a runtime verification system (see Figure 2).

We use the standard JSON format to exchange data: in our event domain (Node.js functions and callbacks) the instrumented program sends all the information regarding the invoked function, and the monitoring server replies with an error, if any, thus the bidirectional channel.

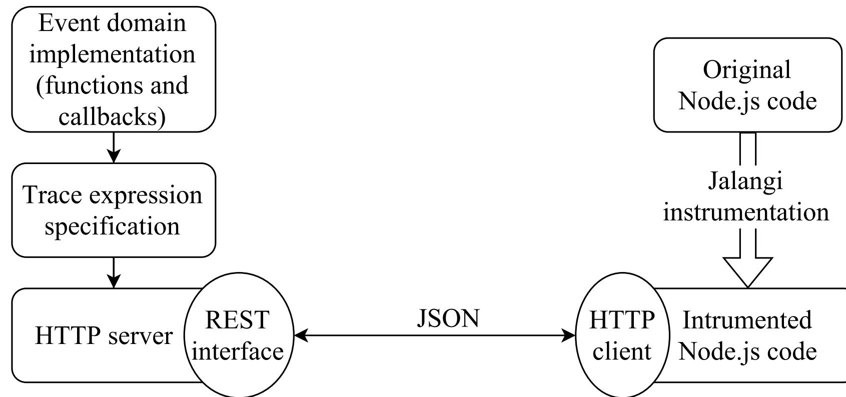

Figure 2: High-level view of the implementation.

4.1 Event domain implementation and source code instrumentation with Jalangi

Jalangi [19][9] is a framework for the implementation of dynamic analyses in JavaScript. The tool is based on code instrumentation: given a JavaScript program, operations are preceded and followed by calls to Jalangi callbacks, which can be overwritten to implement different analyses. Instrumented operations include (but are not limited to) accesses to object properties, read and write instructions on variables, loop and conditional statements, and most importantly for us, function and method invocations.

Jalangi offers four callbacks devoted to function invocation analyses:

- `invokeFunPre` is executed on the call site *before* the function call;
- `invokeFun` is executed on the call site *after* the function call;
- `functionEnter` is executed at the *beginning* of the function body;
- `functionExit` is executed when a *return* statement is evaluated.

Let us consider, for instance, the simple Jalangi analysis logging function invocations for Node.js shown in Listing 1.

In the analysis `iid` is the instruction identifier statically associated with any call in the source code, `f` is the function object that has to be invoked, `base` and `dis` are the receiver object, and the other arguments have the obvious meaning.

It is worth noting that most of Jalangi callbacks return the received data to the instrumented code, even if the example above does not exploit this feature. This allows more complex analysis to actually *change* the behavior of the source code by modifying these values.

Let us now consider a simple Node.js program writing on a file:

```
const fs=require('fs')
function foo() {
  fs.writeFile('file.txt', 'Hello, world!', function callback(err) {
    if (err) return console.error(err)
    console.log('The file was saved!')  })
}
foo()
```

[9]We refer to Jalangi2 (`https://github.com/Samsung/jalangi2`).

```
invokeFunPre = function (iid, f, base, args) {
  console.log('going to invoke %s...', f.name)
  return {f: f, base: base, args: args, skip: false}
}
invokeFun = function (iid, f, base, args, result) {
  console.log('after %s...', f.name)
  return {result: result}
}
functionEnter = function (iid, f, dis, args) {
  console.log('beginning of %s...', f.name)
}
functionExit = function (iid, returnVal, wrappedExceptionVal) {
  console.log('returning...')
  return {returnVal: returnVal, wrappedExceptionVal: wrappedExceptionVal,
   isBacktrack: false}
}
```

Listing 1: Jalangi callbacks for function invocations logging. For the sake of brevity, here we ignore other metadata available to the callbacks as additional arguments.

The output of the Jalangi analysis (Listing 1) when applied to this program would be the following[10]:

```
going to invoke foo...
beginning of foo...
going to invoke writeFile...
after writeFile...
returning...
after foo...
beginning of callback...
returning...
```

The log shows that not all function calls are treated in the same way. The output makes sense if we recall that Jalangi instruments the given program, not the library code. As a result, we have three possible scenarios. For functions both defined in, and called from the program, all the four Jalangi callbacks are executed. Calls to library functions only get the "outer" instrumentation, but we have no information about what is happening inside the function itself. Viceversa, when an asynchronous library function is completed and a callback defined in the program is executed, only the "inner" instrumentation is available.

This approach has advantages, especially regarding Node.js: not only it gives a flexible way to implement a precise analysis, but it also gives the opportunity to track both (asynchronous) library functions and the registered callbacks. Unfortunately, this also leads to an inconsistency between different kind of functions, as shown in the example above, and some synchronization mechanism is needed between the different instrumentation levels to avoid inconsistencies in the analysis output.

The single-thread nature of Node.js makes this task easier since we can safely assume that only one (instrumented) function at a time will be executed. Thanks to this we can simply use a stack data structure to record which Jalangi callback has been invoked for each function call in the original program, matching `invokeFunPre/invokeFun` with `functionEnter/functionExit` when needed. This allowed us to build a more abstract and coherent layer over Jalangi, generating the events *fun_pre* and *fun_post* (see previous section) *exactly once* for *every* function in the original program (before and after it, respectively).

[10] We made a few simplifying assumptions here: only logs related to functions `foo` and `fs.writeFile` are shown, and for the moment we assume every function has a name, which is often not the case in JavaScript.

(Anonymous) Callbacks In the last example some simplifying assumptions were made: all functions were named, and asynchronous callbacks were treated in the same way as other functions. This is both unrealistic and not so useful for analyzing Node.js applications. Not only callbacks are usually anonymous functions, but we also want to know exactly when the callback of a specific asynchronous function call gets executed. Consider the following example:

```
const fs=require('fs')
const cb=function() { /* do something */ }
fs.writeFile('hey.txt', 'Hey there!', cb)
fs.writeFile('wow.txt', 'Cool!', cb)
fs.createWriteStream('enough.txt').write('really', cb)
```

The callback above is anonymous and it is also used twice with the same asynchronous operation, and once with a different one. In order to correctly match the asynchronous function calls with their callbacks, our instrumentation creates a unique identifier for each call and substitutes the callback argument with a wrapper where the identifier is stored. This way, when the callback is actually executed, we can retrieve the information of the original asynchronous call.

At this point we have all the information we need to implement the event domain we used for runtime verification in Section 3:

$$fun_pre(name,id,args) \qquad cb_pre(name,id,args)$$
$$fun_post(name,id,args) \qquad cb_post(name,id,args)$$

4.2 Implementing the monitoring server with SWI-Prolog

The use of logic programming, and SWI-Prolog [21] in particular, for the implementation of the monitoring server offers several advantages, as it allows both the trace expression semantics and the specification of program behavior to be encoded in a natural way.

Our monitoring server implementation is modular: we have developed an HTTP server receiving requests and parsing JSON strings. Such a server is parameterized with respect to the event domain in use and to the trace expressions encoding the specification that needs to be verified.

How to implement a new trace expression We will now show how it is possible to implement a trace expression as a SWI-Prolog module to be loaded by the server, built on top of the function and callbacks event domain.

Since trace expressions are expressed as (recursive) syntactic equations, a crucial feature of SWI-Prolog is its library support for coinduction [20] and regular terms [13].

For instance, let us consider again the parametric trace expression for monitoring asynchronous write operations on multiple files with Node.js:

$$AT = \varepsilon \vee <\mathsf{id}\,;open(\mathsf{id}):(CB\,|\,AT)> \qquad CB = <\mathsf{fd}\,;cb(\mathsf{id},\mathsf{fd}):AW>$$
$$AW = <\mathsf{id}_2\,;write(\mathsf{id}_2,\mathsf{fd}):cb(\mathsf{id}_2):AW> \vee <\mathsf{id}_3\,;close(\mathsf{id}_3,\mathsf{fd}):cb(\mathsf{id}_3):\varepsilon>$$

This can be easily encoded in a SWI-Prolog predicate `spec` as follows:

```
spec(AT) :-
 AT = eps \/ var(ID, open(ID):(CB | AT)),
 CB = var(FD, cb(ID,FD):AW),
 AW = var(ID2, write(ID2,FD):cb(ID2):AW) \/ var(ID3, close(ID3,FD):cb(ID3):eps).
```

Note that (parametric) event types can be easily encoded in functional terms (like `open(FD)`). In this case, trace expression variables correspond to Prolog logical ones (recall that SWI-Prolog variable names start with an uppercase letter).

The next step is to define a *match* function for the event types in use:

```
match(E, open(ID))       :- match(E, func_pre('fs.open', ID, _)).
match(E, write(ID, FD)) :- match(E, func_pre('fs.write', ID, [FD|_])).
match(E, close(ID, FD)) :- match(E, func_pre('fs.close', ID, [FD|_])).
match(E, cb(ID))         :- match(E, cb_pre(_, ID, _)).
match(E, cb(ID, FD))     :- match(E, cb_pre(_, ID, [_,FD|_])).
```

New event types can be defined on top of the event domain for functions and callbacks we implemented. Recall that, for *fun_pre* and *cb_pre*, the first argument encodes the name of the function being called, the second one is the unique identifier of the call (and of the registered callbacks, if any), and finally the third one is the list of arguments. In SWI-Prolog the underscore is used as a placeholder variable (always fresh) for terms we want to ignore. The syntax $[x_1,\dots,x_n|l]$ denotes the list starting with elements $x_1,\dots,x_n$ followed by the list l.

Finally, the last step consists in exporting the two predicates `spec` and `match` so that they can be used by the server:

```
:- module(spec,[spec/1, match/2]).
```

With the directive above we created a module named `spec` exporting the two predicates together with their arity. The module will then import the event domain implementation in order to get the *match* function implementation for *fun_pre*, *fun_post*, *cb_pre* and *cb_post*.

Implementing the event domain The event domain implementation works at lower level on the parsed JSON object. For instance, the following is the implementation of the event type *fun_pre*:

```
match(json(O), func_pre(Name, Id, Args)) :-
  member(event='func_pre', O), member(name=Name, O),
  member(id=Id, O), member(args=Args, O).
```

Other event domains could be implemented in a similar way.

Operational semantics Another advantage of Prolog is that we can almost directly translate the transition rules for the semantics of trace expressions into clauses:

$$\text{(and)}\frac{\tau_1 \xrightarrow{e} \tau_3;\sigma_1 \quad \tau_2 \xrightarrow{e} \tau_4;\sigma_2}{\tau_1 \wedge \tau_2 \xrightarrow{e} \tau_3 \wedge \tau_4;\sigma}\ \sigma{=}\sigma_1{\cup}\sigma_2$$

```
next(T1/\T2, E, T3/\T4, S) :-
  next(T1, E, T3, S1),
  next(T2, E, T4, S2),
  merge(S1, S2, S).
```

The merge predicate produces the union of the given substitutions ensuring they are coherent on the intersection of their domains.

HTTP server The last piece of the implementation is the HTTP server. Thanks to the library support of SWI-Prolog and to the conciseness of the language, the HTTP server can be implemented in a few lines of code (for space reasons we do not include the directives importing standard library modules for HTTP and JSON):

```
:- use_module(trace_expressions).
:- use_module(spec).
:- http_handler(/,manage_request ,[]).

server(Port) :- http_server(http_dispatch ,[port(localhost:Port),workers(1)]).

manage_request(Request) :-
  http_read_json(Request ,E),
  nb_getval(state ,T1),
  (next(T1 ,E ,T2)->nb_setval(state ,T2),reply_json(json([error=(@false)]))
                   ;reply_json(json([error=(@true)]))).
exception(undefined_global_variable ,state ,retry) :- spec(T),nb_setval(state ,T).
```

Modules `trace_expressions` and `spec` implement the transition system and the specification, respectively, while `http_handler` tells the system that `manage_request` should be used to handle requests.

Once an HTTP request is received, first the JSON payload is parsed, then the current state of the server is retrieved, i.e., the current trace expression. If the incoming event is accepted, then the state is updated with the trace expression yielded by performing a transition step, otherwise an error is detected (`p->p1;p2` is the SWI-Prolog syntax for conditionals). The server always replies with a JSON object containing a single boolean-valued field `error`. The last line of code deals with the initialization of the server, loading the initial trace expression.

Finally, the server can be run by loading the program above and executing the goal clause `server(80).` (or any other port).

4.3 Monitoring Node-RED

In this section we show how the simple REST interface offered by the SWI-Prolog monitor described in the previous subsection can be easily used for monitoring Node-RED flows.

For simplicity, we consider a simple “ping-pong” protocol specified by the following trace expression:

$$PP = <\mathsf{val}_1\,; ping(\mathsf{val}_1, 0) : T>$$
$$T = <\mathsf{val}_2\,; pong(\mathsf{val}_2, \mathsf{val}_1) : <\mathsf{val}_1\,; ping(\mathsf{val}_1, \mathsf{val}_2) : T>>.$$

The two employed event types *ping*, and *pong*, both depend on pairs of integer values, and are based on a simple event domain where events are sent messages with two attributes: the type of message (either ping or pong), and its payload (an integer value). Event type $ping(i_1, i_2)$ matches all ping messages sent with payload i_1 strictly greater than integer i_2, and $pong(i_1, i_2)$ has an analogous definition. Hence, the trace expression *PP* defines a protocol consisting of an infinite sequence of alternating ping and pong messages where the first one must be a ping message with payload strictly larger than 0, while the payload of all other messages must be strictly larger than the payload of the immediately preceding message.

We have implemented such a protocol with the Node-RED flow depicted in Figure 3, by exploiting the WebSocket technology. All nodes used in the flow are directly offered by the standard library, and have required very simple configuration settings.

There are two WebSocket nodes, one is an input and the other is an output node, both configured as clients connected[11] to a simple echo demo server.

[11] Available at `wss://echo.websocket.org`.

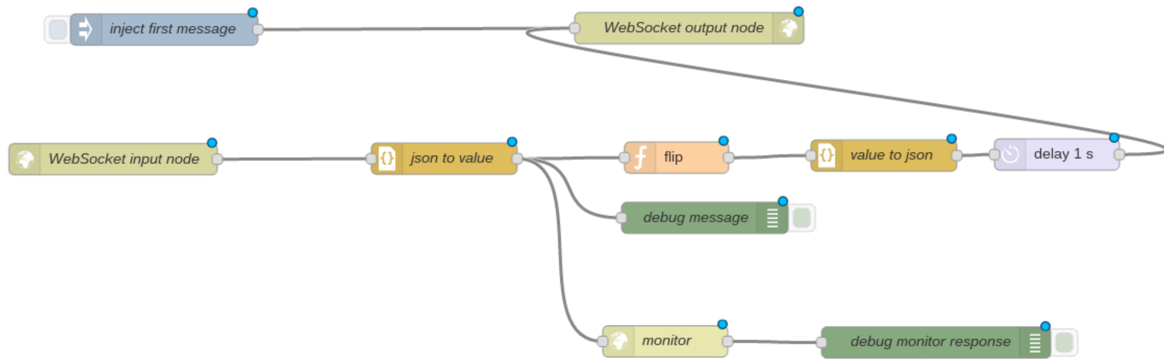

Figure 3: A simple Node-RED flow implementing a "ping-pong" protocol based on WebSocket

The input node receives JSON messages sent from the WebSocket server and send it to the nodes directly connected to it. In this case, there is a single connected node performing the standard conversion from a JSON string into a corresponding JavaScript value. The output WebSocket node sends to the echo server the JSON messages arriving at it from the delay node directly connected to it. The delay node has been inserted simply for limiting the net traffic.

The "flip" node is a function node containing simple JavaScript code for transforming the message received at the input, before sending it to the output. It flips ping into pong messages, and the other way round, and increases the payload of the message arrived at the input by a random non negative integer, before sending the transformed message to the output. The increase of the payload is intentionally allowed to be 0 to simulate errors.

The message transformed by the "flip" node is output to a node which converts it back to the JSON format, before it flows to the output WebSocket node through the delay node. Finally, an injection node is responsible for starting the flow with a ping message with payload 1.

Besides the trace expression defined at the beginning of this subsection, monitoring of the flow is simply achieved by adding[12] a monitor node, which receives as input the messages that flow throw the WebSocket input node. Such a node is a standard HTTP request node configured to send to the SWI-Prolog server a post request with the message received at the input of the node, and to output the corresponding response as a parsed JSON value.

5 Conclusion and future work

This work is a first step towards the use of parametric trace expressions for runtime monitoring of Node.js applications and Node-RED flows to help detect bugs. We have developed a prototype implementation based on Jalangi and SWI-Prolog which offers a simple REST interface for monitoring IoT systems. The tool supports an event domain useful for checking the correct use of asynchronous functions and their associated callbacks in Node.js applications. Even if developers may not always have a deep knowledge of the used libraries, which would be required in order to write down a correct specification, ideally the verification system can be deployed together with the library and used by the programmer in a transparent

[12]The flow exhibits also two debugging nodes for logging purposes.

way.

Still a considerable amount of work is needed to assess the approach and prove its effectiveness. We are planning to experiment the tool with real Node.js applications and to systematically inspect the documentation of basic modules, such us `fs`, `http` and `express`, on which most Node.js applications rely on, to identify patterns of correct use of the exported functions that can be expressed with trace expressions, as shown in Section 3. Similarly, we intend to experiment our tool with Node-RED by considering actual IoT applications.

This study will be essential for identifying scalability issues, and provide possible solutions. To this aim, integration of runtime monitoring with formal verification and testing may prove useful. Runtime monitoring has been used in conjunction with testing techniques for instance by Artho et al. [8], who propose a framework able to combine automated test case generation, based on systematically exploring the input domain of the program, with runtime verification, where execution traces are monitored and verified against properties expressed in temporal logic. Deductive software verification [1] is an interesting technique which seems to offer advantages when employed together with runtime monitoring. Regarding static analysis, in the context of parametric verification a dependent type system based on session types has been proposed [22].

Finally, this work is mainly concerned with logging and error detection, but runtime verification techniques can go further including error recovery procedures, see for instance "runtime reflection" pattern [9] and the monitor-oriented programming (MOP) methodology [11]. Though Jalangi allows arbitrary code to be executed inside its callbacks, such a possibility would require either a monitor-aware program or some ad-hoc solutions for the system under test. We leave these considerations for future work.

References

[1] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P.H. Schmitt & M. Ulbrich (2016): *Deductive Software Verification - The KeY Book - From Theory to Practice*. LNCS, Springer, doi:10.1007/978-3-319-49812-6.

[2] D. Ancona, V. Bono, M. Bravetti, J. Campos, G. Castagna, P. M. Deniélou, S. J. Gay, N. Gesbert, E. Giachino, R. Hu, E. B. Johnsen, F. Martins, V. Mascardi, F. Montesi, R. Neykova, N. Ng, L. Padovani, V.T. Vasconcelos & N. Yoshida (2016): *Behavioral Types in Programming Languages*. *Foundations and Trends in Programming Languages* 3(2-3), pp. 95–230, doi:10.1561/2500000031.

[3] D. Ancona, D. Briola, A. Ferrando & V. Mascardi (2015): *Global Protocols as First Class Entities for Self-Adaptive Agents*. In: *AAMAS 2015*, pp. 1019–1029. Available at http://dl.acm.org/citation.cfm?id=2772879.2773282.

[4] D. Ancona, G. Delzanno, L. Franceschini, M. Leotta, E. Prampolini, M. Ribaudo & F. Ricca (2017): *An Abstract Machine for Asynchronous Programs with Closures and Priority Queues*. In: *Proceedings of the 11th International Workshop on Reachability Problems*, pp. 59–74, doi:10.1007/978-3-319-67089-8_5.

[5] D. Ancona, S. Drossopoulou & V. Mascardi (2012): *Automatic Generation of Self-Monitoring MASs from Multiparty Global Session Types in Jason*. In: *DALT 2012*, *LNAI* 7784, Springer, pp. 76–95, doi:10.1007/978-3-642-37890-4_5.

[6] D. Ancona, A. Ferrando & V. Mascardi (2016): *Comparing Trace Expressions and Linear Temporal Logic for Runtime Verification*. In: *Theory and Practice of Formal Methods*, Springer Verlag, pp. 47–64, doi:10.1007/978-3-319-30734-3_6.

[7] D. Ancona, A. Ferrando & V. Mascardi (2017): *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*, pp. 1457–1459. Available at http://dl.acm.org/citation.cfm?id=3091328.

[8] C. Artho, H. Barringer, A. Goldberg, K. Havelund, S. Khurshid, M. Lowry, C. Pasareanu, G. Roşu, K. Sen, W. Visser & R. Washington (2005): *Combining test case generation and runtime verification*. Theoretical Computer Science 336(23), pp. 209 – 234, doi:10.1016/j.tcs.2004.11.007.

[9] A. Bauer, M. Leucker & C. Schallhart (2006): *Model-based runtime analysis of distributed reactive systems*. In: Australian Software Engineering Conference (ASWEC'06), pp. 10–, doi:10.1109/ASWEC.2006.36.

[10] D. Briola, V. Mascardi & D. Ancona (2014): *Distributed Runtime Verification of JADE Multiagent Systems*. In: IDC, Studies in Computational Intelligence, Springer, pp. 81–91, doi:10.1007/978-3-319-10422-5_10.

[11] F. Chen & G. Roşu (2007): *Mop: An Efficient and Generic Runtime Verification Framework*. In: Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications, OOPSLA '07, ACM, pp. 569–588, doi:10.1145/1297027.1297069.

[12] C. Colombo, G. J. Pace & G. Schneider (2009): *LARVA — Safer Monitoring of Real-Time Java Programs (Tool Paper)*. In: 2009 Seventh IEEE International Conference on Software Engineering and Formal Methods, pp. 33–37, doi:10.1109/SEFM.2009.13.

[13] B. Courcelle (1983): *Fundamental properties of infinite trees*. Theoretical Computer Science 25, pp. 95–169, doi:10.1016/0304-3975(83)90059-2.

[14] S. Hallé & R. Villemaire (2010): *Runtime Verification for the Web - A Tutorial Introduction to Interface Contracts in Web Applications*. In: RV 2010, pp. 106–121, doi:10.1007/978-3-642-16612-9_10.

[15] S. Hall, T. Bultan, G. Hughes, M. Alkhalaf & R. Villemaire (2010): *Runtime Verification of Web Service Interface Contracts*. Computer 43(3), pp. 59–66, doi:10.1109/MC.2010.76.

[16] M. Leotta, F. Ricca, D. Clerissi, D. Ancona, G. Delzanno, M. Ribaudo & L. Franceschini (2017): *Towards an Acceptance Testing Approach for Internet of Things Systems*. In: EnWoT 2017. To appear in Springer's Lecture Notes in Computer Science.

[17] M. Leucker & C. Schallhart (2009): *A brief account of runtime verification*. The Journal of Logic and Algebraic Programming 78(5), pp. 293–303, doi:10.1016/j.jlap.2008.08.004.

[18] Q. Luo, Y. Zhang, C. Lee, D. Jin, P. O. Meredith, T. F. Serbanuta & G. Rosu (2014): *RV-Monitor: Efficient Parametric Runtime Verification with Simultaneous Properties*. In: RV 2014, 8734, Springer, pp. 285–300, doi:10.1007/978-3-319-11164-3_24.

[19] K. Sen, S. Kalasapur, T. Brutch & S. Gibbs (2013): *Jalangi: A Selective Record-replay and Dynamic Analysis Framework for JavaScript*. In: ESEC/FSE 2013, pp. 488–498, doi:10.1145/2491411.2491447.

[20] L. Simon, A. Mallya, A. Bansal & G. Gupta (2006): *Coinductive Logic Programming*. In: ICLP 2006, pp. 330–344, doi:10.1007/11799573_25.

[21] J. Wielemaker, T. Schrijvers, M. Triska & T. Lager (2012): *SWI-Prolog*. Theory and Practice of Logic Programming 12(1-2), pp. 67–96, doi:10.1017/S1471068411000494.

[22] N. Yoshida, P.M. Deniélou, A. Bejleri & R. Hu (2010): *Parameterised Multiparty Session Types*, pp. 128–145. Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1007/978-3-642-12032-9_10.