An Abstract Machine for Asynchronous Programs with Closures and Priority Queues

Davide Ancona, Giorgio Delzanno, Luca Franceschini, Maurizio Leotta, Enrico Prampolini, Marina Ribaudo, Filippo Ricca

Abstract:

We present the operational semantics of an abstract machine that models computations of event-based asynchronous programs inspired to the Node.js server-side system, a convenient platform for developing Internet of Things applications. The goal of the formal description of Node.js internals is twofold: (1) integrating the existing documentation with a more rigorous semantics and (2) validating widely used programming and transformation patterns by means of mathematical tools like transition systems. Our operational semantics is parametric in the transition system of the host scripting language to mimic the infrastructure of the V8 virtual machine where Javascript code is executed on top of the event-based engine provided by the C++ libuv concurrency library. In this work we focus our attention on priority callback queues, nested callbacks, and closures; these are widely used Node.js programming features which, however, may render programs difficult to understand, manipulate, and validate.

Digital Object Identifier (DOI):

https://doi.org/10.1007/978-3-319-67089-8_5

An Abstract Machine for Asynchronous Programs with Closures and Priority Queues

Davide Ancona, Giorgio Delzanno, Luca Franceschini, Maurizio Leotta, Enrico Prampolini, Marina Ribaudo, and Filippo Ricca

Dip. di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi (DIBRIS), Università di Genova, Italy

Abstract. We present the operational semantics of an abstract machine that models computations of event-based asynchronous programs inspired to the Node.js server-side system, a convenient platform for developing Internet of Things applications. The goal of the formal description of Node.js internals is twofold: (1) integrating the existing documentation with a more rigorous semantics and (2) validating widely used programming and transformation patterns by means of mathematical tools like transition systems. Our operational semantics is parametric in the transition system of the host scripting language to mimic the infrastructure of the V8 virtual machine where Javascript code is executed on top of the event-based engine provided by the C++ libuv concurrency library. In this work we focus our attention on priority callback queues, nested callbacks, and closures; these are widely used Node.js programming features which, however, may render programs difficult to understand, manipulate, and validate.

1 Introduction

Asynchronous programming is getting more and more popular thanks to emergent server-side platforms like Node.js and operating systems for applications that require a constant interaction with the user interface; this programming paradigm reduces the need of controlling concurrency using synchronization primitives like lock and monitors. As an example, the execution model of Node.js is based on a single threaded loop used to poll events with different priorities and to serialize the execution of pending tasks by means of callbacks. Accordingly, I/O operations are performed through calls to asynchronous functions where callbacks are passed to specify how the computation continues once the corresponding I/O operations completed asynchronously.

Taking inspiration from previous work [1,2,5,6,4,7,8,10], in this paper we focus our attention on a formal semantics of asynchronous programs with the following features: Priority callback queues; Nested callback definitions to model anonymous callbacks; Closures used to propagate the caller scope to the postponed callback invocation. The formal semantics of the resulting system is given in a parametric form. Namely, we give a presentation that is modular in the transition system of the host programming language. We use callbacks as a bridge

between the two layers as in implementation of scripting languages like Node.js running on virtual machines like Chrome V8. In our framework a callback is a procedure whose execution is associated with a given event. When the event is triggered by the program or by an external agent, the corresponding callback is added to a queue of pending tasks. In our model the execution of callbacks is controlled by an event-loop. After the complete execution of a callback, the event loop polls the queue and selects the next callback to execute.

This kind of behavior is typically supported via concurrent executions of I/O bound operations on a pool of worker threads. To get closer to the Node.js execution model, we model the behavior of the event loop using different phases. More precisely, we provide continuations as a means to define pending tasks with highest priority (they are executed at the end of a callback) as for the `process.nextTick`[1] operation in Node.js. Furthermore, we provide postponed callbacks as a means to postpone a callback after the poll phase of standard callbacks (internal, I/O, and networking events). This mechanism is similar to the `setImmediate` operation provided in Node.js. As in Node.js, nested continuations (a continuation that invokes a continuation) or the enqueuing of tasks during the poll phase are potential sources of starvation for the event loop. In actual implementations non termination in the poll phase is avoided by imposing a hard limit on the number of callbacks to be executed in each tick. All pending postponed callbacks are executed when the poll phase is quiescent.

Examples To illustrate all above mentioned concepts, let us consider some examples taken from the standard `fs` module supporting file system operations in Node.js.

```
var fs = require(‘fs’);
fs.readFile(‘abc.txt’,function(err,data){err||console.log(data);});
console.log(‘ok’);
```

Function `readFile` is asynchronous; once the read operation has completed, its continuation is defined by an anonymous callback with two parameters `err` and `data` holding, respectively, an optional error object and the data read from the file (in case no error occurred `err` contains `null`). If the read operation completes successfully, then the callback will print the read data on the standard output, but only after string `ok`. Let us now consider an example that uses the `events` module in Node.js for emitting events and registering associated callbacks.

```
var EventEmitter = require(‘events’);
var Emitter = new EventEmitter();
var msg = function msg() { console.log(‘ok’); }
Emitter.on(‘evt1’, msg);
Emitter.emit(‘evt1’);
while (true);
```

On line 4 the callback `msg` is registered and associated with events of type `’evt1’`, then on the subsequent line an event of type `msg` is emitted; since method `emit`

[1] Despite of the name, this is the current semantics of Node.js

exhibits a synchronous behaviour, the associated callback is immediately executed and the message `ok` is printed before the program enters an infinite loop where no other callback will ever be executed. We now modify the program above as follows.

```
eventEmitter.on('evt1', function () { setImmediate(msg); });
eventEmitter.emit('evt1');
while (true);
```

Also in this case, after the event has been emitted, the associated callback is synchronously executed, but then `setImmediate` is used to postpone the execution of function `msg` to the next loop iteration, and therefore the program enters immediately the infinite loop without printing `ok`. System functions like `setImmediate` are used to interleave the main thread and callbacks. While there exists a rigorous semantics for Javascript, see e.g.[9], the Node.js on-line documentation [12] has not a formal counterpart and contains several ambiguities as can be read in [13], and in several discussions on the meaning of operations like `setImmediate`, `nextTick`, etc. see e.g. [14,11,15].

The combination of asynchronous programming with nested callbacks and closures may lead to programs with a quite intricate semantics; let us consider, for instance, the following fragment.

```
function test(){
   var d = 5;
   var foo = function(){ d = 10; }
   process.nextTick(foo);
   setImmediate(() => { console.log(d) })
}
test();
```

Function `test` defines a local variable `d`, which, however, is global to the inner function `foo` which is passed as callback to `process.nextTick`; technically, `foo` is called a closure, since it depends on the global variable `d`, which is updated by `foo` itself. Function `process.nextTick` postpones the execution of `foo` to the next loop iteration, however `foo` has higher priority over the callback passed to `setImmediate` on the following line. After the call to `test` is executed, the variable `d` local to the call is still allocated in the heap since it is referenced by the closure `foo`; when `foo` is called, the value of `d` is updated to `10`, therefore the execution of `console.log(d)` will print `10` as result.

In the paper we propose a formal model for describing computations and to clarify the semantics of this kind of programs.

Plan of the paper In Section 2 we present the formal definitions of the operational semantics of a scripting language and of the abstract machine that captures the event-driven behavior of our scripts. Both components are inspired to Node.js. Section 3 illustrates how to apply the operational semantics to reason about asynchronous programs. In Section 4 we define a specification language for reasoning about computations of the resulting combined framework (abstract

machine and semantics of scripting language). Finally, in Section 5 we address related work and future research directions.

2 Abstract Machine for Asynchronous Programs

In this section we define the formal semantics of the abstract machine. We will first try to use a compositional method with respect to the semantics of the host language. We will instantiate the language with a simplified scripting language.

Preliminaries In the rest of the paper we will use the following notation. $A^* = \{v_1 \dots v_n | v_i \in A, i : 1, \dots, n,\ n \geq 0\}$ denotes the set of words with elements in A. We use $w_1 \cdot w_2$ to denote concatenation of two lists and ϵ to denote the empty word. $A^{\otimes} = \{\{v_1 \dots v_n\} | v_i \in A,\ i : 1, \dots, n,\ n \geq 0\}$ denotes the set of multisets of elements in A. We use $m_1 \oplus m_2$ to denote multiset union of m_1 and m_2. We also use $a \oplus m$ to denote the addition of element $a \in A$ to m. $A^n = \{\boldsymbol{v} = \langle v_1, \dots, v_n \rangle | v_i \in A,\ i : 1, \dots, n\ n \geq 1\}$ denotes set of tuples with elements in A. $[A \to B]$ denotes the set of maps from A to B. We use $[\boldsymbol{x}/\boldsymbol{a}]$ to denote the sequence of substitutions or maps $[x_1/a_1, \dots, x_n/a_n]$ for $i : 1, \dots, n$, $n \geq 1$. We use $t[s/x]$ to denote the term obtained from t by substituting every free occurrence of x with s. Furthermore, $m[v/x]$ denotes the maps m' defined as $m'(x) = v$ and $m'(y) = m(y)$ for every $y \neq x$. To manipulate callbacks we will use lambda-terms. A variable x, a value v and the constant *any* are lambda-terms. For a variable x and a lambda term t, $\lambda x.t$ is a lambda-term. If t and t' are lambda-terms, then $t(t')$ is a lambda-term. A variable x is free in t if there are occurrences of x in t that are not in the scope of a binder λx. Lambda terms are usually considered modulo renaming of bounded variable, i.e., all free occurrences under the scope of the same binder can be renamed without changing the abstract structure of the term. If x does not occur free in t', the application $(\lambda x.t)(t')$ reduces to the term $t[t'/x]$ obtained by substituting all free occurrences of x in t with the term t'.

2.1 Host Language

We assume here that programs are defined starting from *sequential* programs in a host language L with basic constructs and (recursive) procedures. Let F be a denumerable set of function names. Let us consider a denumerable set Val of primitive values (pure names in our example) and a denumerable set Var of variables. Expressions are either values or variables. We will also use the special term *any* to denote a non-deterministically selected value and in some example consider natural numbers and expression with standard semantics. To simplify the presentation, we will represent programs as words over the alphabet of program instructions I with variables in Var. An (anonymous) callback definition is a lambda-term $\lambda \boldsymbol{x}.s$, where $\boldsymbol{x} \in Var^k$ is the set of formal parameters, and $s \in I^*$. Now let A be a finite set of names of asynchronous operations. $Events$ is a finite set of (internal and external) event labels s.t. $Events = Events_i \cup Events_e$ and $Events_i \cap Events_e = \emptyset$. Var may contain variable and function names in F. In

order to define environments we will extend Val in order to contain closures, i.e., pairs in $Env \times Callback$ where $Callback$ is the set of lambda terms that denotes callback (function) definitions.

Furthermore, we introduce a denumerable set Loc of locations that we use to denote memory references so as to obtain a semantics with stateful closures. The association between locations and values will be defined via a global heap memory H that will be part of the system configuration. The heap memory H is a list of maps $[l_1/v_1, \ldots, l_k/v_k]$ s.t. l_i is a location and v_i is a value (primitive value or closure). We will use $Heap$ to denote the set of possible heaps.

The set of environments Env consists of list of substitutions $[x_1/l_1, \ldots, x_n/l_n]$ s.t. $x_i \in Var$ and $l_i \in Val$ for $i : 1, \ldots, n$. Given $\ell = [x_1/l_1, \ldots, x_n/l_n]$, $\ell(x) = l$ if there exists i s.t. $x_i = x$, $\ell_i = l$, and $x_j \neq x$ for $j > i$. In other words to evaluate x in ℓ we search the first occurrence of x from right to left and return its value. Given an environment ℓ and the heap memory H, we use ℓ_H to denote the function defined as $\ell_H(v) = H(\ell(v))$ for $v \in Var$.

In this way we can use an environment as a stack and represent therein nested scopes of variables. $Listener$ is the finite set maps $Events \to (Env \times F^*)^*$. A listener maps an event to a sequence of pairs each one consisting of an environment (the current environment of the caller) and a list of callback names (defined in the corresponding environment). $Call_F$ is the set of callback calls $\{f(\boldsymbol{v}) | f \in F, \boldsymbol{v} \in Val^k, k \geq 0\}$. $Call_A$ is the set of asynchronous calls $\{call(a, cb) | a \in A, cb \in F\}$ where A is a set of labels. Finally, a frame (a record of the call stack of the main program) is a pair $\langle \ell, u \rangle$ s.t. $\ell \in Env$ and $u \in I^*$.

The host language provides a denumerable set of global variables Var_G and an operation $store(x, e)$ to write the evaluation of expression e in the global variable x. We assume here that store operations on undefined variables add the variable to the environment (i.e. global environment can only be extended). Furthermore, we provide a special operation *obs* to label specific control points with the current value of an expression. The label is made observable by labeling the transition relation with it. The proposed instance of the host language allows us to design an assertional language to specify properties of computations in the abstract machine.

A program expression has either the structure $let\ x = e\ in\ B$ where x is a local variables and e an expression denoting a primitive value (not a function), or $let\ f_1 = \lambda \boldsymbol{y}_1.P_1, \ldots, f_k = \lambda \boldsymbol{y}_k.P_k\ in\ B$ where $P_1, \ldots, P_k$ are program expressions (they may contain let declarations). In both cases B is a finite sequence of instructions built on top of the above mentioned instruction set. We consider then the following types of instructions.

- $obs(e)$ is used to observe a certain event (a value)
- $store(x, e)$ is used to store a value (the evaluation of e) in the global or local variable x. We use the expression *any* to denote a value non deterministically selected from the set of values.
- $f(\boldsymbol{e})$ is used to synchronously invoke a callback f with the vector of parameters $\boldsymbol{e}$. Actual parameters are global or local variables.

We assume that all necessary procedure definitions are declared in the program (we will introduce an example language with *let* declarations) so that their names are always defined in the current local environment. We now define the set of configurations C_L of the host language. C_L consists of tuples of the form $\langle G, H, S\rangle$, where $G \in Env$, H is the global heap, and $S \in Frame^*$. In other words S has the form $\langle \ell_1, S_1\rangle \dots \langle \ell_n, S_n\rangle$ for $i : 1, \dots, n$. A word of frames will be interpreted as the stack of procedure calls. In a pair $\langle \ell, w\rangle$ ℓ is the local environment and w is the corresponding program to be executed.

$$\frac{\ell' = \ell[x/l],\ l_H(e) = v,\ H' = H[l/v],\ l \notin dom(H)}{\langle G, H, \langle \ell, let\ x = e\ in\ B\rangle \cdot S\rangle \to_L \langle G, H', \langle \ell', B\rangle \cdot S\rangle}\ s1v$$

$$\frac{\begin{array}{c}\ell' = \ell[f_1/l_1, \dots, f_k/l_k],\ H' = H[l_1/\langle \ell, \lambda \boldsymbol{x}_1.P_1\rangle, \dots, l_k/\langle \ell, \lambda \boldsymbol{x}_k.P_k\rangle] \\ l_i \notin dom(H),\ l_i \neq l_j,\ for\ i, j : 1, \dots, k,\ i \neq j\end{array}}{\langle G, H, \langle \ell, let\ f_1 = \lambda \boldsymbol{x}_1.P_1, \dots, f_k = \lambda \boldsymbol{x}_k.P_k\ in\ B\rangle \cdot S\rangle \to_L \langle G, H', \langle \ell', B\rangle \cdot S\rangle}\ s1f$$

$$\frac{}{\langle G, H, \langle \ell, obs(e) \cdot B\rangle \cdot S\rangle \to_L^{\widehat{\ell_H(e)}} \langle G, H, \langle \ell, B\rangle \cdot S\rangle}\ s2$$

$$\frac{x \notin dom(\ell) \quad G \cdot \ell_H(e) = w \neq \lambda \boldsymbol{y}.e}{\langle G, H, \langle \ell, store(x, e) \cdot B\rangle \cdot S\rangle \to_L \langle G[x/w], H, \langle \ell, B\rangle \cdot S\rangle}\ s3g$$

$$\frac{x \in dom(\ell) \quad \ell_H(e) = w \neq \lambda \boldsymbol{y}.e \quad \ell(x) = l}{\langle G, H, \langle \ell, store(x, e) \cdot B\rangle \cdot S\rangle \to_L \langle G, H[l/w], \langle \ell, B\rangle \cdot S\rangle}\ s3l$$

$$\frac{\begin{array}{c}\ell_H(f) = \langle \ell', \lambda \boldsymbol{y}.u\rangle,\ G \cdot (\ell_H) \cdot (\ell'_H)(\boldsymbol{v}) = \boldsymbol{v'},\ H' = H[\boldsymbol{l}/\boldsymbol{v'}],\ \ell'' = \ell[\boldsymbol{y}/\boldsymbol{l}], \\ for\ \boldsymbol{l} = l_1, \dots, l_k,\ l_i \notin dom(H),\ l_i \neq l_j,\ for\ i, j : 1, \dots, k,\ i \neq j\end{array}}{\langle G, H, \langle \ell, f(\boldsymbol{v}) \cdot B\rangle \cdot S\rangle \to_L \langle G, H', \langle \ell'', u\rangle \cdot \langle \ell, B\rangle \cdot S\rangle}\ s4$$

$$\frac{}{\langle G, H, \langle \ell, \epsilon\rangle \cdot S\rangle \to_L \langle G, H, S\rangle}\ s5$$

Fig. 1. Operational semantics of the host language

2.2 Operational Semantics

We assume that the operational semantics of programs in L is defined via a transition system $\langle C_L, \to_L\rangle$, where $\to_L \subseteq (C_L \times C_L)$ defines small step operational semantics of generic statements of programs in L. We will use λ-terms to represent callbacks in the local environment during program evaluation. In the semantics of our language, lambda terms are used as values for function names. Indeed, a local environment ℓ is a map that associates function names to locations, and H associates locations to lambda expressions. By using location for both variables and function names we obtain a more general semantics open to extensions in which variables can contain functions (as in Javascript) that can

be dynamically updates. The transition system is obtained as the minimal set satisfying the rule schemes defined in Fig. 1. Rule $s1v$ models the semantics of the *let* expression for variables with primitive values (we assume here that e is not a function nor a function call). Its effect is to update the local environment with a new substitution between the variable name and a fresh location, and the heap with an association between the new location and the value of the expression. Rule $s1f$ models the semantics of the *let* expression. Its effect is to update the local environment with new substitutions between function names and locations, and the heap with associations between locations of pairs that represent the current environment and the lambda term that represents the body of the callback definition. Rule $s2$ models the semantics of instructions. This rule captures the effect of observing an event by exporting the label to the meta-level (i.e. as a label of the transition step). We assume here that if an expression a is not defined in ℓ_H (and it is not a function name) than it is simply viewed as a constant, i.e., $\widehat{\ell_H}(e) = e$ if $\ell_H(e)$ is not defined. Rule $s3$ captures the effect of a $store(x, e)$ operation. We assume that *store* is defined only if x is a global variable and e evaluates to a value that is not a function. Furthermore, $\ell(any) \in Val$ (non deterministically selected). Rule $s4$ models synchronous calls. In this rule we first evaluate f in the current local environment to retrieve its definition. We then evaluate the parameters in the concatenation of global and local environments. We push a new frame onto the call stack containing a copy of the current environment (so as to transport the scope information from the current frame to the new one) concatenated with the evaluation of the parameters and the body of the function definition. Finally, rule $s5$ models the return from a call by eliminating the frame on top of the stack when its body is empty. In our host language, we can define a stronger rule by observing that local environments are copied from old to new frames and that local environments are not used as state by other frames. In other words, we can reason modulo the following equivalence between stack expressions: $(S_1 \cdot \langle \ell, \epsilon \rangle \cdot S_2) \equiv (S_1 \cdot S_2)$.

2.3 Abstract Machine

In this section we define the formal semantics of an abstract machine that can handle programs as those specified in the previous section extended with built-in instructions for associating event-handlers (callbacks) to event names, and to control the priority queues via special enqueues built-in primitives. More precisely, programs are defined by enriching the language L with the following control instructions:

- $reg(e, u)$: registers callbacks in the word (list) $u \in F^*$ for event e, we use a list since the callbacks must be processed in order.
- $unreg(e, P)$: unregisters all callbacks in the set $P \in \mathcal{P}(F)$ for event e.
- $call(op, cb)$: invokes an asynchronous operation op and registers the callback cb to be executed upon its termination. We assume here that the operation generates a vector of input values that are passed, upon termination of op, to the callback cb.

– $nexttick(f, \boldsymbol{v})$: enqueues the call to f with parameters $\boldsymbol{v}$ in the nextTick queue.
– $setimmediate(f, \boldsymbol{v})$: postpones the call to function f with parameters $\boldsymbol{v}$ to the next tick of the event loop.
– $trigger(e, \boldsymbol{v})$: generates event $e \in Events_i$ (pushing callbacks in the poll queue) with actual parameters $\boldsymbol{v}$.

We now define the operational semantics of programs. We first define the set of configurations C_L of the host language. C_L consists of tuples of the form $\langle G, H, S\rangle$ where $G \in Env$, H is the heap memory, and $S \in Frame^*$. In other words S has the form $\langle \ell_1, S_1\rangle \dots \langle \ell_n, S_n\rangle$ for $i : 1, \dots, n$. A word of frames will be interpreted as the stack of procedure calls. In a pair $\langle \ell, w\rangle$ ℓ is the local environment and w is the corresponding program to be executed. We assume that the operational semantics of programs in L are defined via a transition system $\langle C_L, \rightarrow_L\rangle$, where $\rightarrow_L \subseteq (C_L \times C_L)$ defines small step operational semantics of generic statements of programs in L.

A configuration is a tuple $\langle G, H, E, S, C, Q, P, R\rangle$, where $G \in Env$, $H \in Heap$, $E \in Listener$, $S \in Frame^*$, $C, Q, P \in (Env \times Call_F)^*$, and $R \in (Env \times Call_A)^{\otimes}$. C, Q, P and R are sequences of pairs consisting of a local environment and of a function invocation. C is the (nexttick) queue of pending callback invocations generated by *nexttick*, Q is the (poll) queue of pending callback invocations generated by *trigger* and by external events. P is the (setimmediate) queue of pending callback invocations generated by *setimmediate*. Local environments are used to evaluate variables defined in the body of a callback at the moment of registration, synchronous or asynchronous invocation.

We associate to every L-program Π enriched with control instructions a transition system $T_\Pi = \langle Conf, \rightarrow\rangle$ in which $Conf$ is the set of configurations, and $\rightarrow$ is a relation in $Conf \times Conf$. Furthermore, we will use labeled versions of the transition relations, namely, $\rightarrow^\alpha$ and $\rightarrow^\alpha_L$, in order to keep track of observations generated by the evaluation of programs instructions. Labels are either values, variable or function names. For simplicity, we will use $\rightarrow$ [resp. $\rightarrow_L$] to denote $\rightarrow^\epsilon$ [resp. $\rightarrow^\epsilon_L$].

The initial configuration is the tuple

$$\gamma_0 = \langle \emptyset, \emptyset, \emptyset, \langle \epsilon, P\rangle, \epsilon, \epsilon, \epsilon, \emptyset\rangle$$

where we use $\emptyset$ to denote an empty map or multiset, ϵ to denote empty queues and local environments (both treated as words). In the rest of the paper we will use $\perp$ to denote the empty call stack (it helps in reading configurations). In order to evaluate expressions we need to combine G and a local environment ℓ. We use $G \cdot \ell$ to indicate the concatenations of the substitutions contained in G and ℓ. As for environments, to evaluate x we inspect the list of substitutions in $G \cdot \ell$ from right to left (a local variable can hide a global one with the same name). In the rest of the section we assume that procedure names occurring in a rule are always defined in the corresponding local environment. The transition system is obtained as the minimal set satisfying the rule schemes defined in Fig. 2. Rule $r1$ is used to embed the semantics of L into the semantics of the

$$\frac{\langle G, H, S\rangle \rightarrow_L^{\alpha} \langle G', H', S'\rangle}{\langle G, H, E, S, C, Q, P, R\rangle \rightarrow^{\alpha} \langle G', H', E, S', C, Q, P, R\rangle}\ r1$$

$$\frac{E' = E[evt/(E(evt) \cdot \langle \ell, u\rangle)]}{\langle G, H, E, \langle \ell, reg(evt, u) \cdot w\rangle \cdot S, C, Q, P, R\rangle \rightarrow \langle G, H, E', \langle \ell, w\rangle \cdot S, C, Q, P, R\rangle}\ r2$$

$$\frac{E' = E[evt/(E(evt) \ominus u)]}{\langle G, H, E, \langle \ell, unreg(evt, u) \cdot w\rangle \cdot S, C, Q, P, R\rangle \rightarrow \langle G, H, E', \langle \ell, w\rangle \cdot S, C, Q, P, R\rangle}\ r3$$

$$\frac{\begin{array}{c} evt \in Events_i \quad E(evt) = \langle \ell_1, u_1\rangle \ldots \langle \ell_m, u_m\rangle \quad u_i = p_1^i \cdot \ldots \cdot p_{k_i}^i \ \ for\ i : 1, \ldots, m \\ r = \langle \ell_1, p_1^1(\boldsymbol{v})\rangle \cdot \ldots \cdot \langle \ell_1, p_{k_1}^1(\boldsymbol{v})\rangle \ldots \langle \ell_m, p_1^m(\boldsymbol{v})\rangle \cdot \ldots \cdot \langle \ell_m, p_{k_m}^m(\boldsymbol{v})\rangle \quad \boldsymbol{v} \in Val^k \end{array}}{\langle G, H, E, \langle \ell, trigger(evt, \boldsymbol{v}) \cdot w\rangle \cdot S, C, Q, P, R\rangle \rightarrow \langle G, H, E, \langle \ell, w\rangle \cdot S, C, Q \cdot r, P, R\rangle}\ r4$$

$$\frac{R' = R \oplus \{\langle \ell, call(a, cb)\rangle\}}{\langle G, H, E, \langle \ell, call(a, cb) \cdot w\rangle \cdot S, C, Q, P, R\rangle \rightarrow \langle G, H, E, \langle \ell, w\rangle \cdot S, C, Q, P, R'\rangle}\ r5$$

$$\frac{u = \langle \ell, cb(\boldsymbol{v})\rangle \quad \boldsymbol{v} \in Val^k \quad R' = R \setminus \{\langle \ell, call(a, cb)\rangle\}}{\langle G, H, E, S, C, Q, P, R\rangle \rightarrow \langle G, H, E, S, C, Q \cdot u, P, R'\rangle}\ r6$$

$$\frac{\begin{array}{c} evt \in Events_e \quad E(evt) = \langle \ell_1, u_1\rangle \ldots \langle \ell_m, u_m\rangle \quad u_i = p_1^i \cdot \ldots \cdot p_{k_i}^i \ \ for\ i : 1, \ldots, m \\ r = \langle \ell_1, p_1^1(\boldsymbol{v})\rangle \cdot \ldots \cdot \langle \ell_1, p_{k_1}^1(\boldsymbol{v})\rangle \ldots \langle \ell_m, p_1^m(\boldsymbol{v})\rangle \cdot \ldots \cdot \langle \ell_m, p_{k_m}^m(\boldsymbol{v})\rangle \quad \boldsymbol{v} \in Val^k \end{array}}{\langle G, H, E, S, C, Q, P, R\rangle \rightarrow \langle G, H, E, S, C, Q \cdot r, P, R\rangle}\ r7$$

$$\frac{G \cdot \ell_H(\boldsymbol{v}) = \boldsymbol{v'}}{\langle G, H, E, \langle \ell, nexttick(f, \boldsymbol{v}) \cdot w\rangle \cdot S, C, Q, P, R\rangle \rightarrow \langle G, H, E, \langle \ell, w\rangle \cdot S, C \cdot \langle \ell, f(\boldsymbol{v'})\rangle, Q, P, R\rangle}\ r8$$

$$\frac{G \cdot \ell_H(\boldsymbol{v}) = \boldsymbol{v'}}{\langle G, H, E, \langle \ell, setimmediate(f, \boldsymbol{v}) \cdot w\rangle \cdot S, C, Q, P, R\rangle \rightarrow \langle G, H, E, \langle \ell, w\rangle \cdot S, C, Q, P \cdot \langle \ell, f(\boldsymbol{v'})\rangle, R\rangle}\ r9$$

$$\frac{\begin{array}{c} \ell_H(p) = \langle \ell', \lambda \boldsymbol{y}.s\rangle,\ G \cdot (\ell_H) \cdot (\ell'_H)(\boldsymbol{v}) = \boldsymbol{v'},\ H' = H[\boldsymbol{l}/\boldsymbol{v'}],\ \ell'' = \ell[\boldsymbol{y}/\boldsymbol{l}], \\ for\ \boldsymbol{l} = l_1, \ldots, l_k,\ l_i \notin dom(H),\ l_i \neq l_j,\ for\ i, j : 1, \ldots, k,\ i \neq j \end{array}}{\langle G, H, E, \bot, \langle \ell, p(\boldsymbol{v})\rangle \cdot C, Q, P, R\rangle \rightarrow \langle G, H', E, \langle \ell', s\rangle, C, Q, P, R\rangle}\ r10$$

$$\frac{\begin{array}{c} \ell_H(f) = \langle \ell', \lambda \boldsymbol{y}.s\rangle,\ G \cdot (\ell_H) \cdot (\ell'_H)(\boldsymbol{v}) = \boldsymbol{v'},\ H' = H[\boldsymbol{l}/\boldsymbol{v'}],\ \ell'' = \ell[\boldsymbol{y}/\boldsymbol{l}], \\ for\ \boldsymbol{l} = l_1, \ldots, l_k,\ l_i \notin dom(H),\ l_i \neq l_j,\ for\ i, j : 1, \ldots, k,\ i \neq j \end{array}}{\langle G, H, E, \bot, \epsilon, p(\boldsymbol{v}) \cdot Q, P, R\rangle \rightarrow \langle G, H', E, \langle \ell', s\rangle, \epsilon, Q, P, R\rangle}\ r11$$

$$\frac{}{\langle G, H, E, \bot, \epsilon, \epsilon, P, R\rangle \rightarrow \langle G, H, E, \bot, \epsilon, P, \epsilon, R\rangle}\ r12$$

Fig. 2. Operational semantics of the abstract machine

abstract machine. Rule $r2$ associates the current environment and the ordered list of callbacks u to the event evt. Everytime evt is triggered, the callbacks in u will be added to the queue of pending tasks together with the environment in the same order as they occur in u. Rule $r3$ unregisters all callbacks in P for event evt. We use $\ominus$ to denote this operation. Rule $r4$ assigns a semantics to the *trigger* instruction. It corresponds to the combination of emit and `setImmediate` discussed in the introduction. The rationale behind this definition is that when evt is triggered, all registered callbacks are added to the pending queue. The current local environment is stored together with the callback invocation. The formal parameters are instantiated with the actual parameters defined in the *trigger* statement. The environment can be used to evaluate variables occurring

in the body of the callback definition. The actual parameters $\boldsymbol{v}$ are evaluated using the composition of the global and local environment.

We now consider asynchronous calls of built-in libraries to be executed in a thread pool. Rule $r5$ adds the call to a pool of pending tasks submitted to the pool thread. We do not model the internal behavior of asynchronous calls. The only information maintained in R is a pointer to the callback cb that has to be executed upon termination of the thread execution. The pool R is used to keep track of the operations that have been submitted to the thread pool. Since the termination order of these calls is not known a priori and, at least in principle, the calls might be processed in parallel by different threads, we abstract from the order and use a multiset for modeling the pool. According to this idea, rule $r6$ models the termination of a thread and the invocation of the corresponding callback cb. We assume here that cb expects k parameters. The callback is invoked with k non-deterministically generated values (they model the result returned by the asynchronous call). We remark that the constant a is used as a label and has no specific semantics (it helps in the examples).

In order to handle the response to external events (e.g. connections, etc) we use the non-deterministic rule $r7$. We assume here that the data generated by the operation are non-deterministically selected from the set of values and associated with the formal parameters of the callback functions $p_1, \ldots, p_k$. Every callback invocation is stored together with the corresponding local environment.

Rule $r8$ deals with nextTick callbacks. Invocation of such an operation is defined as follows. The callback invocation is stored together with the current local environment. Rule $r9$ is used to deal with setImmediate invocations. The callback invocation is stored together with the current local environment. In $r8$ and $r9$ the actual parameters are evaluated using the composition of the global and local environment.

Rule $r10$ and $r11$ define the selection of pending tasks. The selection phase is defined according with the following priority order: nextTick, poll, setImmediate. nextTick callbacks are selected every time the call stack becomes empty (at the end of a callback execution). Poll callbacks are selected only when the call stack is empty and there are no nextTick callbacks to execute. The local environment is initialized with the stored environment ℓ and the map that associates parameters to actual values. SetImmediate callbacks are selected only when both the call stack and the nextTick queue are empty. We assume that the definition of p is available in the local environment stored with the callback (the global environment does not contain function definitions). The local environment is initialized with the stored environment ℓ and the map that associates parameters to actual values. Finally, in rule r12 all (pending) setImmediate callbacks are executed before passing to the next tick of the event loop.

Given a program P with initial state γ_0 and transition relation $\rightarrow$, we use $\rightarrow^*$ to denote the reflexive and transitive closure of $\rightarrow$. The sequence of labels generated during the unfolding of the transition relation $\rightarrow$ gives rise to possibly infinite words in $Labels^*$ that we will use to observe the behavior of a given instance of the host language when executed in the abstract machine. Starting

from an initial state γ_0, a program can give rise to several different computations obtained by considering every possible reordering of asynchronous operations. This feature, in combination with the callback mechanism that can delay the execution of a function, makes our programs a non trivial computational model even in the simple case of Boolean programs, i.e., programs in which all data are abstracted into a finite set of possible values.

We will restrict our attention to infinite executions under fairness conditions to ensure the termination of asynchronous callbacks after finitely many steps. Indeed, asynchronous callbacks are typically built-in operations that terminate with an error if something goes wrong in their execution. Similarly, we might restrict our attention to infinite executions in which external events, for which there are registered callbacks, occur infinitely often.

3 Formal Reasoning

In this section we will consider some example of formal reasoning via the transition systems introduced in the previous sections. For the sake of simplicity, in all examples but the last one on closures with state, we will omit the heap component and consider only environments mapping variables to values. We will use the complete semantics when considering side effects on closures.

Let us first consider the program defined as follows.

$$P = let\ f = (let\ (cb = \lambda x.\ obs(x))\ in\ call(read, cb) \cdot f)\ in\ f()$$

A possible computation, in which we apply the reduction $\langle \ell, f\rangle \cdot \langle \ell, \epsilon\rangle \equiv \langle \ell, f\rangle$, is given below.

$$\begin{array}{l}
\rho_1 = \langle \emptyset, \emptyset, \emptyset, \langle \epsilon, P\rangle, \epsilon, \epsilon, \epsilon, \emptyset\rangle \rightarrow \\
\rho_2 = \langle \emptyset, \emptyset, \emptyset, \langle \ell, f\rangle, \epsilon, \epsilon, \epsilon, \emptyset\rangle\ s.t.\ \ell\ defines\ f \rightarrow \\
\rho_3 = \langle \emptyset, \emptyset, \emptyset, \langle \ell, f\rangle, \epsilon, \epsilon, \epsilon, \emptyset\rangle \rightarrow \\
\rho_4 = \langle \emptyset, \emptyset, \emptyset, \langle \ell, let\ cb = \lambda x.obs(x)\ in\ call(read, cb) \cdot f\rangle, \epsilon, \epsilon, \epsilon, \emptyset\rangle \rightarrow \\
\rho_5 = \langle \emptyset, \emptyset, \emptyset, \langle \ell', call(read, cb) \cdot f\rangle, \epsilon, \epsilon, \epsilon, \emptyset\rangle\ s.t.\ \ell'\ defines\ cb\ \rightarrow \\
\rho_6 = \langle \emptyset, \emptyset, \emptyset, \langle \ell', f\rangle, \epsilon, \epsilon, \epsilon, \{\langle \ell', call(read, cb)\rangle\}\rangle
\end{array}$$

In the configuration ρ_6 we can either assume that the asynchronous call is already terminated or continue with the execution of the current callback. In the former case we have the following continuation.

$$\rho_6 \rightarrow \rho_7 = \langle \emptyset, \emptyset, \emptyset, \langle \ell', f\rangle, \epsilon, \epsilon, \langle \ell', cb(d)\rangle, \emptyset\rangle\ for\ d \in Val$$

In the latter case we will add a new frame to the call stack with the body of f. We now observe that, in both cases, the callback stack will never get empty again. Therefore, the callback cb, defined in the local environment ℓ', will never be selected for execution even under additional fairness conditions. As a consequence, the transition system will never generate observations during any of its infinitely many possible computations (the termination of the asynchronous call can happen anytime). This formally explains why in systems like Node.js everytime the

main application has a recursive structure (e.g. to model an infinite loop) in order to make it responsive to external events, it is necessary to encapsulate the recursive call into invocations of primitives like `setImmediate` and `nextTick`. To illustrate this idea, let us consider the following modified program.

$$P_1 = let\ f = (let\ cb = \lambda x.\ obs(x)\ in\ call(read, cb) \cdot setimmediate(f))\ in\ f()$$

We obtain the following behavior.

$$\begin{array}{l}\rho_1 = \langle \emptyset, \emptyset, \emptyset, \langle \epsilon, P_1 \rangle, \epsilon, \epsilon, \epsilon, \emptyset \rangle \rightarrow \\ \rho_2 = \langle \emptyset, \emptyset, \emptyset, \langle \ell, f \rangle, \epsilon, \epsilon, \epsilon, \emptyset \rangle s.t.\ \ell\ defines\ f \rightarrow^* \\ \rho_3 = \langle \emptyset, \emptyset, \emptyset, \langle \ell', setimmediate(f) \rangle, \epsilon, \epsilon, \epsilon, \{\langle \ell', call(read, cb) \rangle\} \rangle\ s.t.\ \ell'\ def.\ f, cb\end{array}$$

Again we have a bifurcation here. For instance, let us assume that *read* terminates. We obtain then the following computation.

$$\begin{array}{l}\rho_3 \rightarrow \rho_4 = \langle \emptyset, \emptyset, \emptyset, \langle \ell, setimmediate(f) \rangle, \epsilon, cb(d), \epsilon, \emptyset \rangle \rightarrow^* \\ \rho_5 = \langle \emptyset, \emptyset, \emptyset, \bot, \epsilon, \langle \ell', cb(d) \rangle, \langle \ell', f \rangle, \emptyset \rangle \rightarrow \\ \rho_6 = \langle \emptyset, \emptyset, \emptyset, \langle \ell', cb(d) \rangle, \epsilon, \langle \ell', f \rangle, \emptyset \rangle \rightarrow \\ \rho_7 = \langle \emptyset, \emptyset, \emptyset, \langle \ell'[x/d], obs(x) \rangle, \epsilon, \langle \ell', f \rangle, \emptyset \rangle \rightarrow^d \\ \rho_8 = \langle \emptyset, \emptyset, \emptyset, \langle \ell'[x/d], \epsilon \rangle, \epsilon, \langle \ell', f \rangle, \emptyset \rangle \ldots\end{array}$$

It is interesting to observe here that callbacks are evaluated in the environment of the caller. For instance, in this example function *cb* and *f* are both defined in ℓ', the local environment used to initialize the frame associated to the invocation $cb(d)$. In other words, under fairness conditions on the termination of asynchronous callbacks, the execution of program P_1 generates infinite traces labeled with an arbitrary sequence of values that correspond to successful termination of read operations.

To understand the difference between `nextTick` and `setImmediate`, let us first consider the nextTick operation.

$$\begin{array}{l}NT = let\ \ f_1 = (\lambda d_1.obs(a)),\ f_2 = (\lambda d_2.obs(b)),\ f_3 = (\lambda z.obs(c)) \\ \quad in\ nexttick(f_1) \cdot call(b, f_2) \cdot call(c, f_3)\end{array}$$

We then apply our operational semantics to study its behavior.

$$\begin{array}{l}\rho_1 = \langle \emptyset, \emptyset, \emptyset, \langle \epsilon, W \rangle, \epsilon, \epsilon, \epsilon, \emptyset \rangle \rightarrow \\ \rho_2 = \langle \emptyset, \emptyset, \emptyset, \langle \ell, nexttick(f_1) \cdot call(b, f_2) \cdot call(c, f_3) \rangle, \epsilon, \epsilon, \epsilon, \emptyset \rangle \\ s.t.\ \ell\ cont.\ all\ def. \rightarrow \\ \rho_3 = \langle \emptyset, \emptyset, \emptyset, \langle \ell, call(b, f_2) \cdot call(c, f_3) \rangle, \langle \ell, f_1 \rangle, \epsilon, \epsilon, \emptyset \rangle \rightarrow \\ \rho_4 = \langle \emptyset, \emptyset, \emptyset, \langle \ell, call(b, f_2) \rangle, \langle \ell, f \rangle, \epsilon, \epsilon, \{call(c, f_3)\} \rangle\end{array}$$

We now have possible bifurcations due delays in the termination of *c*. We could for instance reach one of the following two configurations:

$$\begin{array}{l}\rho_5 = \langle \emptyset, \emptyset, \emptyset, \langle \ell, call(c, f_3) \rangle, \langle \ell, f_1 \rangle, call(b, f_2), \epsilon, \emptyset \rangle \\ \rho'_5 = \langle \emptyset, \emptyset, \emptyset, \bot, \langle \ell, f_1 \rangle, \epsilon, \epsilon, \{call(b, f_2), call(c, f_3)\} \rangle\end{array}$$

From the latter we can obtain

$$\rho'_6 = \langle \emptyset, \emptyset, \emptyset, \bot, \langle \ell, f_1 \rangle, call(b, f_2) \cdot call(c, f_3), \epsilon, \emptyset \rangle$$
$$\rho''_6 = \langle \emptyset, \emptyset, \emptyset, \bot, \langle \ell, f_1 \rangle, call(b, f_3) \cdot call(c, f_2), \epsilon, \emptyset \rangle$$

In all cases f_1 will be executed before f_2 and f_3 because of the priority order used to inspect the queue of pending calls when the call stack becomes empty, i.e., neither f_2 nor f_3 can overtake f_1.

Now let us consider the same program with *setimmediate* replacing *nexttick*.

$$NT = let \;\; f_1 = (\lambda d_1.obs(a)),\;\; f_2 = (\lambda d_2.obs(b)),\;\; f_3 = (\lambda z.obs(c))$$
$$in \; setimmediate(f_1) \cdot call(b, f_2) \cdot call(c, f_3)$$

We then apply our operational semantics to study its behavior.

$$\rho_1 = \langle \emptyset, \emptyset, \emptyset, \langle \epsilon, W \rangle, \epsilon, \epsilon, \epsilon, \emptyset \rangle \rightarrow$$
$$\rho_2 = \langle \emptyset, \emptyset, \emptyset, \langle \ell, setimmediate(f_1) \cdot call(b, f_2) \cdot call(c, f_3) \rangle, \epsilon, \epsilon, \epsilon, \emptyset \rangle$$
$$s.t.\ \ell \; cont.\; all \; def. \rightarrow$$
$$\rho_3 = \langle \emptyset, \emptyset, \emptyset, \langle \ell, call(b, f_2) \cdot call(c, f_3) \rangle, \epsilon, \epsilon, \langle \ell, f_1 \rangle, \emptyset \rangle \rightarrow$$
$$\rho_4 = \langle \emptyset, \emptyset, \emptyset, \langle \ell, call(b, f_2) \rangle, \epsilon, \epsilon, \langle \ell, f_1 \rangle, \{call(c, f_3)\} \rangle$$

We now have possible bifurcations due delays in the termination of c. We could for instance reach one of the following two configurations:

$$\rho_5 = \langle \emptyset, \emptyset, \emptyset, \langle \ell, call(c, f_3) \rangle, \epsilon, call(b, f_2), \langle \ell, f_1 \rangle, \emptyset \rangle$$
$$\rho'_5 = \langle \emptyset, \emptyset, \emptyset, \bot, \epsilon, \epsilon, \langle \ell, f_1 \rangle, \{\langle \ell, call(b, f_2) \rangle, \langle \ell, call(c, f_3) \rangle\} \rangle$$

From the latter we can obtain one of the following configurations:

$$\rho_6 = \langle \emptyset, \emptyset, \emptyset, \langle \ell, f_1 \rangle, \epsilon, \epsilon, \{\langle \ell, call(b, f_2) \rangle \cdot \langle \ell, call(c, f_3) \rangle\} \rangle$$
$$\rho_7 = \langle \emptyset, \emptyset, \emptyset, \bot, \langle \ell, call(b, f_2) \rangle \cdot \langle \ell, call(c, f_3) \rangle, \langle \ell, f_1 \rangle, \emptyset \rangle$$
$$\rho_8 = \langle \emptyset, \emptyset, \emptyset, \bot, \langle \ell, call(c, f_3) \rangle \cdot \langle \ell, call(b, f_2) \rangle, \langle \ell, f_1 \rangle, \emptyset \rangle$$

4 Property Specification Language

The formalization of the operational semantics for an abstract machine for event-loop based programs has several possible applications. First of all, it gives a formal meaning to complex computational models underlying widely used systems like Node.js. The informal documentation of operations like nextTick, setImmediate, etc that we found in the dev site [12] is ambiguous and difficult to parse. A transition system like the one presented in this paper gives a precise mathematical meaning to each operation in terms of evolution of configurations. Based on this, we can use the operational semantics as a formal tool to reason about computations in the abstract machines. For this purpose, we introduce an extension of regular expressions in order to handle values from an infinite domain, i.e., represent labels associated to values. A finite state automata is a tuple $A = \langle Q, \Sigma, \delta, q_0 \rangle$ in which Q is a finite set of states, $\delta : Q \times \Sigma \rightarrow Q$ is the transition relation and q_0 is an initial state. A computation is a (possibly

infinite) sequence $q_0a_0q_1a_1q_2\ldots$ s.t. $q_{i+1} = \delta(q_i, a_i)$ for $i \geq 0$. The observation of a computation $q_0a_0q_1a_1q_2\ldots$ is the possibly infinite word $a_0a_1a_2\ldots$ of labels occurring along the computation.

For an alphabet Σ, let $\widehat{\Sigma} = \{\widehat{a}|a \in \Sigma\}$. An extended automata EA is an automata $A = \langle Q, \Sigma \cup \widehat{\Sigma}, \delta, q_0\rangle$. We use $\widehat{x}$ as a sort of variable. The effect of $\widehat{x}$ is to associate a value v, non deterministically chosen, to x and to instantiate x with v in every successive transition labeled with x until the next $\widehat{x}$ label is encountered and so on. If there are no occurrences of $\widehat{a}$ then the label is interpreted as a. In other words during the unfolding of the transition relation of an extended automata for each label we maintain its current state. The initial state of label a is a itself. However everytime we encounter $\widehat{a}$ its state is updated with some value taken from D. More precisely, the configuration of an extended automata is a tuple $\langle q, \rho\rangle$ where $\rho : \Sigma \to D$. The initial state is defined as $\langle q_0, id_\Sigma\rangle$, where $id_\Sigma(a) = a$ for each $a \in \Sigma$. Given a configuration $\langle q, \rho\rangle$, $a \in \Sigma$ and $\delta(q, a) = q'$, we have that $\langle q, \rho\rangle \to^a \langle q', \rho\rangle$. Given a configuration $\langle q, \rho\rangle$, $a \in \Sigma$ and $\delta(q, \widehat{a}) = q'$, we have that $\langle q, \rho\rangle \to^v \langle q', \rho'\rangle$ where $\rho'(a) = v$ for some $v \in D$ and $\rho'(b) = \rho(b)$ for $b \neq a, b \in \Sigma$. A computation is a sequence $\langle q_0, \rho_0\rangle\alpha_0\langle q_1, \rho_1\rangle\alpha_1\ldots$ such that $\langle q_i, \rho_i\rangle \to_i^\alpha \langle q_{i+1}, \rho_{i+1}\rangle$ for $i \geq 0$. The sequence of labels $\alpha_0\alpha_1\ldots$ is called here observation. This simple extension of finite automata allows us to represent observations of value non-deterministically generated during a computation and passed from one callback to another. By definition, we consider only a finite number of variables in order to keep the model as simplest as possible. Finally, we say that a program P conforms to specification given as an extended automata A iff for every computation c of program P in the abstract machine that generates a sequence of labels σ, there exists a computation in the extended automata with observation σ.

To validate examples extracted from Node.js code, we have written a meta-interpreter in Prolog that can explore all possible bounded executions of the proposed model and catch unexpected execution orders with respect to specifications given in a sublanguage of the automata based language proposed in this section. The interpreter [16] exploits the search mechanism of the Prolog runtime system in order to analyze sets of executions of a given input program. We remark that, although Node.js are at first sight sequential programs, their computation is often highly non- deterministic due to the heavy use of asynchronous operations and internal and external events.

5 Conclusions and Related Work

We have presented a first attempt of formalizing the operational semantics of the Node.js event-loop asynchronous computation model including some of the more intricate elements (priority callback queues, nested callbacks, closures) of such a programming system. Following the underlying structured of the event-based loop (inspired to the V8 engine), we have formulated the semantics in terms of an abstract machine operating on a parametric transition system describing the semantics of the host scripting language. We believe that formal specification

and verification of this kind of systems will be more and more important in order to improve the development process of Internet of Things applications, their reliability, and in order to provide non ambiguous documentations of low level details of primitives like those described in this paper. More work has still to be done concerning automation of the verification task e.g. by exploiting approximated algorithms and abstraction for both procedural and functional scripting languages.

There exist several works on formal models of asynchronous programs. In [10] the authors provide verification algorithms for asynchronous systems modeled as pushdown systems with external memory. The external memory is defined as a multiset of pending procedure calls. Theoretical results on recognizability of Parikh images of context-free language are used to obtain an algorithmic characterization of the reachable set of the resulting model. The algorithms have been extended to other types of external memory in [2]. Algorithms for liveness properties are studied in [7] and for real-time extensions are given in [5]. A complexity analysis of decidable fragments is given in [6]. In [8] the authors consider a general model of event-based systems in which task are maintained in FIFO queues. The focus of their analysis again is providing algorithmic techniques for different types of restrictions of the model via reductions to Petri Nets, PDS, and Lossy channel systems. In [3] the authors define a model for asynchronous programs with task buffers in which events and buffers are dynamically created. Decidable fragments are obtained via reductions to Data nets. Differently from the above mentioned work, the goal of the present paper is not that of isolating decidable fragments. We are interested instead in giving a precise semantics to the interplay between asynchronous architecture like Node.js and scripting languages executed on top of them see e.g. more empirical works like [1,4]. In this sense we think that, more than restrictions, our framework needs further extensions in order to capture for instance objects and dynamic memory allocation as done in formal semantics of languages like Javascript [9]. Our validation approach is based on enumeration techniques and partial search similar to tools used for concurrent systems.

References

1. S. Alimadadi, S. Sequeira, A. Mesbah, and K. Pattabiraman. Understanding javascript event-based interactions with clematis. *ACM Trans. Softw. Eng. Methodol.*, 25(2):12:1–12:38, 2016.
2. R. Chadha and M. Viswanathan. Decidability results for well-structured transition systems with auxiliary storage. In *CONCUR 2007*, pages 136–150, 2007.
3. M. Emmi, P. Ganty, R. Majumdar, and F. Rosa-Velardo. Analysis of asynchronous programs with event-based synchronization. In *ESOP 2015*, pages 535–559, 2015.
4. K. Gallaba, A. Mesbah, and I. Beschastnikh. Don't call us, we'll call you: Characterizing callbacks in javascript. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2015, Beijing, China, October 22-23, 2015*, pages 247–256, 2015.
5. P. Ganty and R. Majumdar. Analyzing real-time event-driven programs. In *FORMATS 2009*, pages 164–178, 2009.

6. P. Ganty and R. Majumdar. Algorithmic verification of asynchronous programs. *ACM Trans. Program. Lang. Syst.*, 34(1):6:1–6:48, 2012.
7. P. Ganty, R. Majumdar, and A. Rybalchenko. Verifying liveness for asynchronous programs. In *POPL 2009*, pages 102–113, 2009.
8. G. Geeraerts, A. Heußner, and J.-F. Raskin. On the verification of concurrent, asynchronous programs with waiting queues. *ACM Trans. Embedded Comput. Syst.*, 14(3):58:1–58:26, 2015.
9. D. Park, A. Stefanescu, and G. Rosu. KJS: a complete formal semantics of javascript. In *PLDI 2015*, pages 346–356, 2015.
10. K. Sen and M. Viswanathan. Model checking multithreaded programs with asynchronous atomic methods. In *CAV 2006*, pages 300–314, 2006.
11. `https://howtonode.org/understanding-process-next-tick`.
12. `https://nodejs.org/en/docs/`.
13. `https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/`.
14. `http://stackoverflow.com/questions/15349733/setimmediate-vs-nexttick`.
15. `https://www.quora.com/`.
16. `http://www.disi.unige.it/person/DelzannoG/NODE/`.