An Interface Theory for the Internet of Things

Marten Lohstroh* and Edward A. Lee*
{marten,eal}@eecs.berkeley.edu

EECS Department
University of California
Berkeley, CA 94720, USA

Abstract. This paper uses interface automata to develop an interface theory for a component architecture for Internet of Things (IoT) applications. Specifically, it examines an architecture for IoT applications where so-called "accessors" provide an actor-oriented proxy for devices ("things") and services. Following the principles of actor models, an accessor reacts to input stimuli and produces outputs that can stimulate reactions in other accessors or actors. The paper focuses on a specialized form of actor models where inputs and outputs to accessors and actors are time-stamped events, enabling timing-sensitive IoT applications. The interaction between accessors and actors via time-stamped events forms a "horizontal contract," formalized in this paper as an interface automaton. The interaction between an accessor and the thing or service for which it is a proxy is a "vertical contract," also formalized as an interface automaton. Following common practice in network programming, our vertical contract uses an asynchronous atomic callback (AAC) pattern. The formal composition of these interface automata allows us to reason about the combination of a timed actor model and the AAC pattern, enabling careful evaluation of design choices for IoT systems.

1 Introduction

Two major fields of research in engineering, one centered around cyber-physical systems (CPS) and another around computer networks, now focus their attention on what is believed to be the next big thing after the rise of the Internet, the **Internet of Things** (IoT). The vision embodied by this term appeals to the imagination of many—our environment and virtually anything in it will turn "smart" by having otherwise ordinary things be furnished with sensors, actuators, and networking capability, so that we can patch these things together and have them be orchestrated by sophisticated feedback and control mechanisms. As Wegner argued in [23], *interaction* opens up limitless possibilities for things to harness their environment and compensate for a lack of self-sufficient cleverness. Sensors aside, a connection to the Internet alone allows a thing to tap into

---

an exceedingly rich environment—unleashing a real potential for making things smarter. To exploit this potential, however, a precise and well-defined coordination between a vast and heterogeneous collection of interfaces, protocols, and components is required.

1.1 Accessors

In [10], **accessors** are proposed to take on the challenge of coordinating interaction between networked resources across different domains without imposing standardized over-the-wire protocols or middleware. Accessors provide a formal framework based on **actors** [8] that leverages **platform-based design** [20] as a methodology to deal with the heterogeneity that characterizes the IoT. Accessors are essentially proxies for things and services, endowing them with an actor interface. This interface consists of a set of input and output ports through which the accessor may receive and send tokens, along with a set of action functions that are triggered when inputs arrive or other relevant events occur. An **actor abstract semantics** [13] provides ways to compose accessors with disciplined and understandable concurrency models, while accessors abstract the mechanisms by which they provide access to sensor data, control actuators, communicate to devices, or outsource computation. Accessors run on a **host** that, according to some **model of computation** (MoC), coordinates communication with other actors or accessors. More formally, an accessor interfaces two different MoCs. On the outside, the accessor is coordinated by some actor-oriented MoC, while on the inside, an interpreter governs the execution of a script that defines its key functionality.

The overarching goal of accessors is to lift existing functionality implemented using a heterogeneous collection of scripting languages and network protocols into a library of reusable components that are amenable to composition on a unifying platform for the development of IoT applications. The focus of this work thus moves away from protocol-specific APIs and language-specific design patterns and centers the discussion around the composition semantics of accessors.

1.2 Code Mobility and Trust

An accessor provides access to a thing or service that is not necessarily local to the host. The host is a microcontroller, mobile device, or server, whereas a thing is typically a separate piece of hardware, not necessarily proximate to the host, and a service is possibly cloud based, accessed over the net. The accessor itself is software that runs on the host, serving as a local **proxy** for the thing or service.

A well established precedent for such proxies is found in the Web, where a website serves HTML5 and JavaScript that executes in a browser. The script is a local proxy for a remote service. The script is **mobile code**, supplied by a website, downloaded, and executed on a host (in a browser). It is essential that browsers be able to execute largely **untrusted** code, carefully regulating its access to local resources such as the host file system. Although the security

model is not perfect, after two or so decades of experience, the Web community has accumulated a great deal of experience with such untrusted code, and we can reliably access important services, such as banking, through such proxies.

JavaScript proves to be a well-suited language for such proxies for several reasons. One key reason is that the core JavaScript language includes no I/O mechanisms. These must be provided by the host in the form of a **context** in which the JavaScript code runs. In a browser, for example, the context provides functions to manipulate a document and to control how a document is rendered in the browser window. It also provides functions for soliciting input from a human user and for accessing remote resources through the network. It does not provide functions for accessing the local file system or executing command-line programs on the host. It took many years, but today most of the capabilities provided by the browser context are standard across browsers, so most JavaScript programs will work in a similar way in different browsers.

Accessors require a similar hosting mechanism. A host downloads possibly untrusted code and executes it locally. The host, therefore, functions like a browser, but instead of interfacing humans to network services, it interfaces physical things to each other (and to network services). For example, an accessor for a thing may provide output data that is massaged in some computation to determine an action to be performed by some actuator. To be very specific, an accessor for your front door lock may provide a notification that the door has been opened, which could then trigger another accessor to turn on a light.

For accessors, the emphasis is not on rendering information for humans nor on soliciting input from humans. Hence, the context provided by an accessor host will not have the same facilities that a browser provides. Nevertheless, there are strong commonalities. The accessor code is provided by a third party that often cannot be completely trusted. Authentication, encryption, sandboxing, and networked interactions are all just as relevant to accessors as to browsers. Hence, leveraging the decades of experience with browsers is well justified. For this reason, we focus on JavaScript as the accessor specification language.

1.3   Concurrency

Because accessors are local proxies for things and services that are not necessarily local, concurrency becomes important. Physical things are intrinsically concurrent, in that any two physical devices act and react at the same time. They also act and react concurrently with any software that may be interacting with them. And networked services, of course, are also intrinsically concurrent. The concurrency model used by accessors therefore becomes a central feature.

JavaScript has an event-based concurrency model, and it typically interacts with its environment asynchronously. For example, when accessing a web resource, instead of blocking to wait for a response from the server, when the script queries the server, it provides a **callback function** or **handler** to be invoked when the response arrives. A key feature of JavaScript is that every function invocation is **atomic** with respect to every other function invocation. Hence, unlike interrupt-driven I/O or threads, a callback function does not get
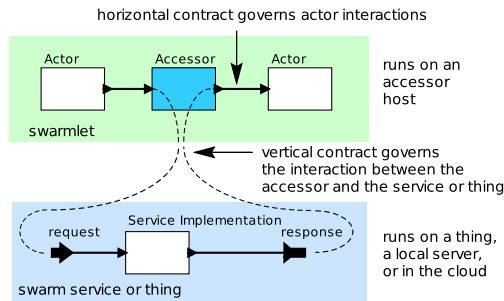
Fig. 1. Accessor in a actor network of actors.

invoked at arbitrary points during the execution of the main program. A function executes to completion before any other function can begin executing. We call this pattern of concurrency **asynchronous atomic callback** (AAC).

The AAC pattern is used extensively in web programming, both on the server side (as in Node.js (http://nodejs.org) and Vert.x (http://vertx.io)) and on the client side, in browsers. It has also been used in some other (non-web) applications such parallel computing (e.g. Active Messages [22]) and embedded systems (e.g. TinyOS [15]).

The AAC pattern dramatically mitigates the difficulty of concurrent programming [11], but at considerable cost. First, it becomes essential to write code carefully to consist only of quick, small function invocations. Second, it accentuates the chaos of asynchrony, where achieving coordinated action can become challenging. The latter problem is particularly important for IoT, where coordinated physical actions are often needed.

Because of these limitations, several efforts are under way to mix AAC with other concurrency models. ECMAScript 6, a recent version of JavaScript, enriches AAC with a cooperative multitasking model, which allows a function to suspend execution at well-defined points, allowing other functions to be invoked while it waits for some event. The Vert.x framework enriches AAC with so-called "verticles" (think "particles"), which can execute in parallel while preserving rigorous atomicity. Verticles can interact with one another through a publish-and-subscribe concurrency model or through shared but immutable data structures. But these are not the only concurrency models that could be usefully combined with AAC. Click [9], for example, mixes push and pull interactions in very interesting ways to create very efficient network routers. Ptides [24] leverages synchronized clocks on a network to create coordinated real-time behavior. Spanner [3] leverages synchronized clocks in a similar way, but for distributed databases rather than distributed real-time systems. Calvin [18] uses a dataflow concurrency model for IoT interactions.

In this paper, we advocate separating the AAC style of concurrency, which an accessor uses to interact with a thing or service, and other styles of concurrency (publish-and-subscribe, push-pull, timed events, dataflow, etc.), which accessors

use to interact with one another. Following Nuzzo et al. [17] and Benveniste et al. [2], we formalize the first style as a **vertical contract** and the second as a **horizontal contract**. As illustrated in Figure 1, the vertical contract defines the interface between the accessor and the thing or service that it is providing access to. The horizontal contract defines the interface between the accessor and the context in which it executes, which can include other actors and accessors. In fact, the very concept of accessors hinges on this separation of concerns.

This separation of concerns is a generalization of the classical separation between computation and coordination that was promoted by Gelernter and Carriero [7] in the 1990s. In the era of the Cloud, ubiquitous computing, and swarms of smart things, a clear-cut division between computation and coordination seems no longer attainable, yet an organization in terms of horizontal and vertical contracts can still facilitate portability and support for heterogeneity.

In this paper, we focus on vertical contracts based on AAC and horizontal contracts based on **discrete events** (DE), by which we mean timed events like those used in Ptides [24] and Spanner [3]. In Figure 1, a DE **director** would govern the interaction between the accessor and the actors (realizing the horizontal contract), while the accessor internally interacts using AAC with a thing or service (the vertical contract).

In DE, every input to or output from an accessor has a **time stamp**, and the host ensures that events are processed in time-stamp order. DE is more deterministic than publish-and-subscribe (because of the use of time stamps), and unlike dataflow, provides a semantic notion of time, which is important for the "things" in IoT. This paper uses the formal idea of behavioral interfaces [4] to provide rigor to these contracts. The formalism reveals subtleties in the interplay between AAC and a timed discrete-event concurrency model.

1.4 Outline

The remainder of this paper is organized as follows. Section 2 gives background material covering actors, models of computation, interface automata, behavioral types, and timing and causality. We then introduce a formal model in Section 3 and apply it to combining AAC with DE. We draw conclusions in Section 4.

2 Background

2.1 Actors

The term "actor" was introduced by Hewitt to describe the concept of autonomous reasoning agents [8]. The term evolved through the work of Agha and others to describe a formalized model of concurrency [1]. Agha's actors each have an independent thread of control and communicate via asynchronous message passing. The term "actor" was also used in Dennis's dataflow models [6] of discrete atomic computations that react to the availability of inputs by producing outputs sent to other actors.

In this paper, the term "actor" embraces a larger family of models of concurrency. They are often more constrained than general message passing and do not necessarily conform with a dataflow semantics. Our actors are still conceptually concurrent, but unlike Agha's actors, they need not have their own thread of control. Unlike Dennis' actors, they need not be triggered by input data. Moreover, although communication is still achieved through some form of message passing, it need not be asynchronous.

Actors are *components* in systems and can be compared to objects, software components in object-oriented design. In prevailing object-oriented languages (such as Java, C++, and C#), the interfaces to objects are primarily **methods**, which are procedures that modify or observe the state of objects. By contrast, the actor interfaces are primarily **ports**, which send and receive data. They do not imply the same sequential transfer of control that procedures do, and hence they are better suited to concurrent models.

In this paper, we will focus on a discrete-event actor model, where inputs and outputs received and sent by actors have time stamps, and actors process these events in time-stamp order. It is useful in IoT applications to bind these time stamps to real time when software has an interaction with the outside world. For example, in Spanner [3], a database query receives a time stamp equal to the value of the local clock at the machine that receives the query. In Ptides [24], a sensor measurement receives a time stamp equal to the value of the local clock of the machine hosting the sensor. By ensuring that events are processed in time-stamp order, it becomes well-defined how a system should react to these external stimuli. For example, in a distributed database, a query for the value of a record and an update to the value of the record are ordered by time stamp, so the correct response to the query is defined by the relative values of the time stamps. If the time stamp of the query is less than or equal to the time stamp of the update, then the correct response is the updated record value. Otherwise, the correct response is the value before the update.

2.2 Behavioral Interfaces

The notion of contracts is much more useful if the contracts have a formal encoding and the composition of components can be checked for compliance with the contracts. Specifically, in our case, the AAC style of concurrency used in the vertical contract manifests as timed events in the DE horizontal contract.

Subtle questions arise from these interactions. For example, in the DE model, an actor **fires** at a (logical) time, and during the firing it can determine what **input events** are present at that time, and for each event that is present, what its value is. Similarly, while firing, an actor can produce **outputs events**. In an AAC model, a callback function is invoked when some condition has been satisfied, for example a reply has arrived from a remote server. In our model, the invocation of such a callback is an **internal event**, in that it is neither a actor input nor a actor output event. But the handling of such an internal event may require observing inputs or producing outputs. Suppose that an accessor (with a DE actor horizontal contract) observes an input in a callback function that was

triggered by an internal event. What should this mean? Suppose that callback is executed asynchronously, nondeterministically interleaving its execution with processing of time-stamped events. What is the semantics of observing an input? Observing an input in DE only has meaning at a logical time. Under what conditions should an input event be present? What is the logical time (the time stamp) of that event?

Similar questions arise if a callback function triggered by an internal event wishes to produce outputs in the DE world. What should the time stamp of those events be? If an output depends on an input event, is the timestamp of that input event then strictly earlier than the timestamp of the output event, or can they be the same? The purpose of this paper is to develop a formal framework for reasoning about such alternatives.

Interface automata (IA), proposed by Henzinger and de Alfaro in [4], offer an attractive approach for defining and composing behavioral interfaces. Interfaces are automata with inputs and outputs, and interaction between interfaces occurs through synchronized **actions**. Output actions are denoted with an exclamation mark, and input actions with a question mark. Internal transitions (also known as τ-transitions or silent steps), which do not involve input or output, are interleaved asynchronously across components. When two IA are composed, an input action in one and an output action in the other are matched by name and become a shared transition, an internal transition in the resulting composition automaton. Note that inputs and outputs in the context of IA have no relation with inputs or output in actor semantics, nor should *actions* be confused with *events* in DE or JavaScript.

Compatibility. Two interfaces A and B are **compatible** if, when they are composed (i.e., $A \otimes B$, which coincides with the composition of I/O automata [16]), there exists *some* environment that satisfies the constraints that the composition automaton imposes. Error states in $A \otimes B$ are those in which one automaton produces an output that the other one does not accept as an input. Since the environment is unable to prevent the automata from reaching these states, the composition of two interface automata prunes away all error states and all states from which error states are reachable. Two interface automata are compatible if the pruned composition, $A||B$, is not empty. A compelling advantage of the pruning is that the resulting composite interface automaton is relatively compact, in contrast to the entire product state space.

Refinement. Interface automata feature a refinement relation that acts contravariantly on input assumptions and output guarantees; i.e., in a refinement, the former can only be relaxed and the latter can only be restricted. This relation is defined as an **alternating simulation** between components. Since we do not use refinement relations here, we will say nothing further about them.

Behavioral Types. Lee and Xiong [14] used interface automata to formulate behavioral type signatures for several directors in Ptolemy II [19]. In their paper, several examples illustrate the interactions between a producer and consumer that exchange tokens, mediated by different directors. Their DE automaton has a key feature that it formally models the constraint that it is illegal for an

actor to get or send tokens (DE events) in between firings. The firings provide the temporal coherence of the DE model, and by constraining consumption of inputs and production of outputs to occur during a firing, the time stamps of those inputs and outputs become unambiguous. We leverage this key feature in this paper.

2.3 Time & Synchrony

In DE, two events can occur simultaneously. Operationally, this means that they have the same time stamp and that an actor that observes these events will see them in the same firing. In AAC, events are invocations of callback functions. These are mutually exclusive; only one event can occur at a time. Hence, if the callback functions observe or produce DE events, we need to reconcile these conflicting properties.

Typical implementations of the AAC pattern have no temporal semantics. Yet time matters for them. The order in which responses come back from a remote web server, for example, matters, so the time of arrival of the responses matters. Programs that interact with things will typically need to exercise some control over timing, for example in order to estimate the trajectory of a moving object based on the order in which events are reported by different sensors. Most JavaScript contexts provides a function `setTimeout(f, t)` which causes a callback function `f` to be invoked after time `t`. But without temporal semantics, the time `t` is an informal notion. There is no assurance, for example, that if `setTimeout(f1, t1)` and `setTimeout(f2, t2)` are called with `t1 < t2`, that `f1` will be invoked before `f2`. If these two callback functions produce timed DE events, then what time stamps should be assigned to those events? A well-designed combination of AAC and DE would bind the timeout times and the DE times, giving a much stronger temporal semantics and more controlled and predictable interaction with things.

Of course, because there is no preemption in JavaScript, the real-time accuracy of the timeouts may vary wildly. The DE model, nevertheless, provides a model of time that is synchronous among all of its components. It is a **logical time**, not a **physical time**. Logical time can be used to guarantee that `f1` will be invoked before `f2` if `t1 < t2`, for example, regardless of when these invocations occur in real time. More interestingly, if `t1 = t2`, the DE logical time model can guarantee that if the two callbacks both produce an output event, then any downstream observer will see these events *simultaneously*. Such guarantees make concurrent programs much more deterministic and understandable.

Moreover, if logical time can be made to closely approximate real time, as is done in Ptides and Spanner, then it can make the interactions of these programs with things much more deterministic and understandable. A simple way to establish a relationship between logical time and physical time is to delay the processing of any time-stamped event until the local real-time clock matches or exceeds the logical time of the time stamp. A more sophisticated mechanism, implemented in Ptides, introduces such delays only where there is an interaction with the physical world.

2.4 Causality & Predictable Timing

Consider an accessor that responds to an input event with time stamp t by issuing a query to an external thing or service that will take some time to respond. Under the DE MoC, the actor fires at logical time t and consumes the input event. Using the AAC pattern, this accessor makes the request to the thing or service and provides a callback function to be invoked later with the response to the query. The fire method returns immediately, allowing the accessor to function like a *pipeline* that can handle a number of requests concurrently. However, because of unpredictable network delays for example, responses may arrive out-of-order. Suppose each response to the query causes the accessor to produce a time-stamped DE event as an output. Should the time stamps of those responses be required to respect the same order of the time stamps that trigger the queries? Should they be required to match those of the input events? Or be offset from those of the input events by some fixed constant? In any of these cases, extra machinery is required to relate the accessor's output to the input that triggers the query. Similar problems have been solved in computer architecture (Tomasulo's algorithm [21]) and distributed systems (PTIDES safe-to-process analysis [24]).

An extreme choice is to require the time stamp of an output to match the time stamp of the input that triggers the query. In this case, the accessor has a logical zero delay, but the physical delay may be substantial. This choice comes at the cost of sacrificing the pipelining capability of the component. Worse, the component may block other components, preventing them from handling events with time stamps t or greater, because of the DE constraint that events always be processed in time-stamp order.

A better choice that provides determinism without sacrificing (as much) concurrency is to require an output to have a time stamp $t + \delta$, for some fixed offset δ, for each input that has time stamp t. If δ is at least as large as the worst-case delay for a response to the query, then no concurrency will be sacrificed.

A third choice is to nondeterministically assign a time stamp to each response, for example giving it as a time stamp the time-stamp of the most recently handled DE event. This choice results in the order of outputs not necessarily matching the inputs that trigger the queries, but it could nevertheless be useful if the time stamps are in fact used to represent physical response times. All three of these choices are available in Ptolemy II [19] using the ThreadedComposite actor [12]. And all three can be used with accessors that combine AAC with DE. How should we choose which one to use? The next section offers the beginnings of a formalism for reasoning about such choices.

3 A Formal Model

Our formalization comprises three interfaces: the DE director, the accessor, and the JavaScript environment that features AACs. The goal is to model each as an interface automaton and to check the compatibility of the composition of all three. If the interfaces are compatible then their composition (denoted by ||) will be non-empty.
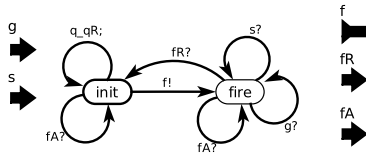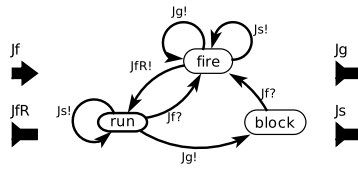
Fig. 2. DE director.

Fig. 3. JavaScript (1).

An interface automaton for the DE director is shown in Figure 2. The automaton has four inputs: g (get), s (send), fR (return from fire), and fA (fire at), and one output: f (fire). This director will fire an actor at a given (logical) time t if either an upstream actor has sent it an input with time stamp t, or the actor has requested to be fired at logical time t. These events are inserted in an **event queue**, sorted by time stamp, and processed by the director when the current (logical) time corresponds to the time stamp of the event. This bookkeeping happens internally, so it is not part of the director's interface. For completeness, however, we added an internal action q_qR in the initial state that represents the director consulting the event queue. In any state of the director, an actor may request a firing at the current (logical) time or some time in the future. Hence, every state accepts an fA? action.

Figure 2 illustrates a key property of interface automata. In state init, the automaton does not accept inputs s and g. The assumption is that the environment will never generate these illegal inputs. Hence, the interface imposes constraints on the environment. de Alfaro and Henzinger [5] distinguish interface theories from component theories in precisely this sense; an interface may impose constraints on its environment, whereas a component exhibits some behavior (not necessarily desired behavior) in *every* environment.

Only after taking the transition to state fire, guarded by action f!, is the director accepting of g and s actions. In other words, it is illegal for an actor to consume inputs or produce outputs when it is not being fired. After observing an fR? action, meaning the actor has concluded its firing, the director returns to its initial state where it can consult its event queue to process new DE events. The composition of the DE director and the accessor formalizes the horizontal contract. The composition between the accessor and the JavaScript environment formalizes the vertical contract. All composed together, we obtain a closed **labeled transition system** (LTS) describing all possible interactions through our interfaces. This LTS is amenable to further analysis. For instance, one could check whether the composition satisfies some LTL property using a model checker such as SPIN (http://spinroot.com). This, however, is outside the scope of this paper.

As to the interface automata for the accessor and the JavaScript environment, we have several options, and we explore two candidate solutions. But first, we list the primitives of the vertical contract. The accessor host provides a `get()` and a `send()` function in the JavaScript context through which, respec-

tively, actor inputs can be read and actor outputs can be sent. We encode these JavaScript functions using actions Jg (JS get) and Js (JS send). In addition, we define an input action Jf (JS fire) to allow the accessor to signal the JavaScript environment that it is currently being fired by the director. Similarly, we define an action JfR (JS return from fire) for the JavaScript environment to notify the accessor that it can now safely end its firing. Finally, the host offers a function `setTimeout()`, represented by the action t. This primitive allows the accessor implementation to schedule itself to be fired at some time in the future.

To achieve compatibility between the DE director and the JavaScript environment through the accessor, we need to prevent Jg! and Js! actions, which may be invoked asynchronously in a callback, from triggering the accessor to emit g! or s! actions before it observes an f? action and after it emits a fR! action. There are multiple solutions to this puzzle that yield useful behavior.

The first option we explore is to have the JavaScript environment block on reading actor inputs when the accessor is not currently being fired by the director. This may occur during a callback that originates from an internal event. Actor outputs produced during an AAC are queued by the accessor and emitted during the next firing. The accessor is responsible for requesting a new firing at the current time upon the occurrence of an AAC.

The second option that we explore is for any AAC to trigger a request for a firing of the actor, and to suspend until that firing occurs.

3.1   Blocking Inputs & Delayed Outputs

The interface automaton that models our JavaScript environment, illustrated in Figure 3, has three states: run, block, and fire. The initial state is run, and in this state it can either emit a Js! action (invoke `send()`), observe a Jf? action (a signal that the accessor is currently fired) or emit a Jg! action (invoke `get()`). When Jg! happens, the automaton transitions to the state block in which the only legal action is Jf?, which enables the transition to fire. In fire, actions Jg! and Js! guard self loops, meaning that they return immediately. Emitting JfR! will let the automaton transition back to run. In summary, Js! actions return immediately, whereas Jg! actions block until the accessor reaches a state that is synchronous with a firing. During fire, Js! and Jg! are handled immediately. Note that shared t transitions are excluded from the interface in order to simplify the example, but their use is described in Section 3.2.

The automaton that models the accessor interface, depicted in Figure 4, is more complex. For each output of the automata in Figures 2 and 3 it has a corresponding input, and for each of their outputs it has a corresponding input. The ports interfacing with the JavaScript environment are grouped at the bottom of the figure, the ports interfacing with the DE director on the sides.

The initial state of the IA in Figure 4 (indicated by a bold outline) is init, from where it can either observe f! and transition to start or observe Jg? or Js? (from a callback invoked in the JavaScript execution environment) and transition to fireAt. From fireAt, an fA! action leads back to init whereas f? enables the transition to start. The intuition here is that observing f? eliminates the need
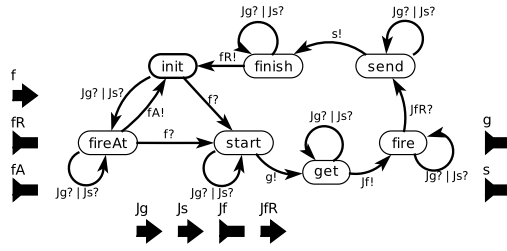
Fig. 4. Accessor (1).

to request a new firing. Note that, due to the asynchrony of the AACs, the automaton has to be accepting Jg? and Js? *in every state*, and because it is not receptive, each state must thus be augmented with a guarded (self-)transition that is guarded by these actions. Ignoring these transitions for a moment, the remainder of the automaton is no more than a simple linear sequence of actions. First it gets new actor inputs (g!) and signals to the JavaScript environment that it is now firing, then it waits until the JavaScript signals JfR!, and finally it sends any queued outputs (s!) and returns from fire (fR!).

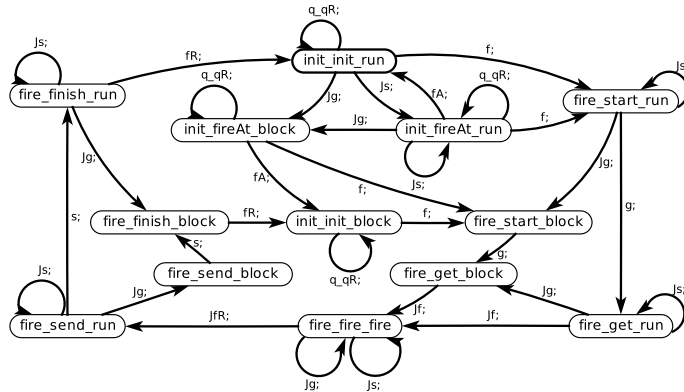**Fig. 5.** DE director || Accessor (1) || JavaScript (1).

The composition of the automata from Figures 2, 3, and 4 is depicted in Figure 5.[1] The automaton is non-empty (and closed), hence the three components are compatible. Notice that product of the state spaces has size 42, and yet the composition automaton only has 13 states. This is because illegal states, where one component outputs an action that is not accepted by another, are pruned away. The LTS in Figure 5 is not per se intended for human analysis, but it does

---

[1] This composition was constructed automatically using software written by Yuhong Xiong over 12 years ago [14].
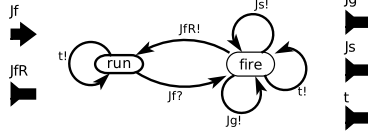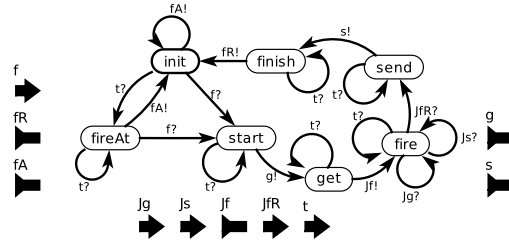
Fig. 6. JavaScript (2).

Fig. 7. Accessor (2).

show quite neatly how the accessor coordinates the interaction between DE and JavaScript. The outer states in the diagram correspond to the steps taken in one iteration in the DE semantics, whereas the inner states deal with AACs by blocking and issuing firing requests to the DE director.

3.2 Deferred AACs

An alternative solution to the one proposed in Section 3.1, is to formulate the vertical contract such that *any* AAC that can possibly invoke `get()` or `send()` will be synchronized, regardless of whether it happens to emit a Jg! or Js! action. The horizontal contract remains the same. Interestingly, this approach results in a much simpler model. To demonstrate this solution, we need a slightly different representation of the accessor and the JavaScript environment. Their interface automata, Figures 7 and 6 respectively, are very similar to the ones in Figures 4 and 3, so we only discuss the differences.

First of all, we include a shared transition t that represents `setTimeout()`. We assume that any internal event will be caught by the host and that it will defer (i.e., suspend, *not block*) the associated AACs until the accessor is fired; a t! will be triggered to request a new firing if needed. This is realized in our implementation using the standard CommonJS `EventEmitter` pattern.

The interface of the JavaScript environment now only has two states: run and fire. In either state it can emit t!, but only in fire can it emit Jg! and Js!. As before, Jg! and Js! are only legal after observing Jf? and before emitting JfR!.

For the same reason as stated in section 3.1, the accessor must accept t? in any state. Only when observed in init will it be followed by an immediate fA! action. The t? actions observed in other states will be cached and processed after completing the firing, upon arriving again in state init.

The composition of the automata from Figures 2, 7, and 6, is depicted in Figure 8. Again, the automaton is non-empty, which shows that the three components are compatible. We still recognize the same general structure of the LTS shown in 5, but the number of states is reduced from 13 to 7.
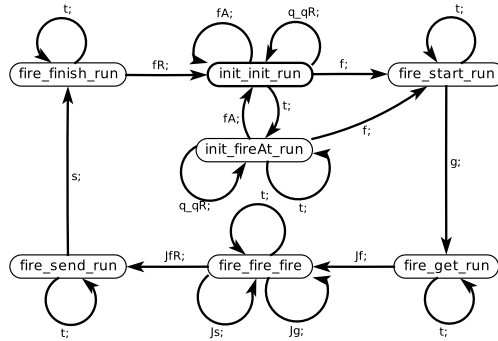
Fig. 8. DE director || Accessor (2) || JavaScript (2).

4 Conclusion

For IoT applications, where networked "things" and services interact with the physical and information worlds, combinations of concurrency models and models of timed behavior are essential. We have developed a formal framework based on interface automata that enables rigorous definition of behavioral interfaces, and we have shown that this framework enables formal analysis of a combination of two popular and useful models of computation, both used (usually separately) for IoT applications. The discrete-event MoC, which models timed concurrent interactions, is formally modeled in this paper as a horizontal contract between peer components (actors). In an IoT application, some of these actors will be "accessors," which provide access to things and services. The interaction between the accessor actors and their thing or service is formally modeled as a vertical contract. Both contracts are represented as interface automata. An automated tool, previously developed, is used to compose these interface automata to validate compatibility of the contracts and to produce a labeled transition system representing the overall system behavior. This LTS can be subjected to further formal analysis, for example using model checking to verify safety conditions.

References

1. G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.
2. A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier, A. Sangiovanni-Vincentelli, W. Damm, T. Henzinger, and K. G. Larsen. Contracts for System Design. Research Report RR-8147, Nov. 2012.
3. J. C. Corbett, et al. Spanner: Google's globally-distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(8), 2013.
4. L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE), ACM*, pages 109–120. Press, 2001.

5. L. deAlfaro and T. A. Henzinger. Interface theories for component-based design. In *First International Workshop on Embedded Software (EMSOFT)*, volume LNCS 2211, pages 148–165. Springer-Verlag, 2001.

6. J. B. Dennis. First version data flow procedure language. Report MAC TM61, MIT Laboratory for Computer Science, 1974.

7. D. Gelernter and N. Carriero. Coordination Languages and their Significance. *Communications of the ACM*, 35(2):97–107, Feb. 1992.

8. C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–363, 1977.

9. E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, 2000.

10. E. Latronico, E. Lee, M. Lohstroh, C. Shaver, A. Wasicek, and M. Weber. A vision of swarmlets. *Internet Computing, IEEE*, PP(99):1–1, 2015.

11. E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

12. E. A. Lee. ThreadedComposite: A mechanism for building concurrent and parallel Ptolemy II models. Technical Report UCB/EECS-2008-151, EECS Department, University of California, Berkeley, December 7 2008.

13. E. A. Lee, S. Neuendorffer, and M. J. Wirthlin. Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers*, 12(3):231–260, 2003.

14. E. A. Lee and Y. Xiong. A behavioral type system and its application in Ptolemy II. *Formal Aspects of Computing*, 16(3):210 – 237, August 2003.

15. P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler. The emergence of networking abstractions and techniques in TinyOS. In *First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2004)*, 2004.

16. N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, pages 137–151, New York, NY, USA, 1987. ACM.

17. P. Nuzzo, A. Sangiovanni-Vincentelli, X. Sun, and A. Puggelli. Methodology for the design of analog integrated interfaces using contracts. *Sensors Journal, IEEE*, 12(12):3329–3345, Dec 2012.

18. J. Perssson. Open source release of IoT app environment Calvin, June 4 2015. Ericsson Research Blog, http://ericsson.com/research-blog/cloud/open-source-calvin/.

19. C. Ptolemaeus, editor. *System Design, Modeling, and Simulation using Ptolemy II.* Ptolemy.org, 2014.

20. A. Sangiovanni-Vincentelli and G. Martin. Platform-based design and software design methodology for embedded systems. *IEEE Design & Test of Computers*, 18(6):23–33, 2001.

21. R. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, Jan 1967.

22. T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: A mechanism for integrated communication and computation. *SIGARCH Comput. Archit. News*, 20(2):256–266, Apr. 1992.

23. P. Wegner. Why interaction is more powerful than algorithms. *Commun. ACM*, 40(5):80–91, May 1997.

24. Y. Zhao, E. A. Lee, and J. Liu. A programming model for time-synchronized distributed real-time systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 259 – 268. IEEE, 2007.