A Vision of Swarmlets

"Swarmlets" are applications and services that leverage networked sensors and actuators with cloud services and mobile devices. The authors offer a way to construct swarmlets by composing "accessors," which are wrappers for sensors, actuators, and services, that export an actor interface. An actor semantics provides ways to compose accessors with disciplined and understandable concurrency models, while abstracting the mechanisms by which the accessor provides sensor data, controls an actuator, or accesses a service. Recognizing a huge variability in the use and requirements of swarmlets, accessors embrace heterogeneity instead of attempting to homogenize.

Elizabeth Latronico, Edward A. Lee, Marten Lohstroh, Chris Shaver, Armin Wasicek, and Matthew Weber
University of California, Berkeley

Rapid growth of networked smart sensors and actuators presents enormous challenges and opportunities. Called the Internet of Things (IoT), Industry 4.0, the Industrial Internet, machine-to-machine (M2M), the Internet of Everything, the Smarter Planet, TSensors (Trillion Sensors), or the Fog (like the cloud, but closer to the ground), the vision is of a technology that deeply connects our physical world with our information world. Many emergent technologies address the diverse concerns of the IoT (see a few listed in Table 1). These mechanisms provide ways to discover and communicate with devices and services. Many leverage established technologies originally developed for ordinary Internet use.

Providing truly universal interoperability to build applications, compose services, and manage data is difficult, however. A key challenge is the enormous diversity of requirements. Power, bandwidth, latency, and assurance requirements can differ dramatically. For example, at one extreme, an energy-harvesting device can't support elaborate protocols and doesn't require much communication bandwidth. At the opposite extreme, real-time visual feedback from a flying drone to its remote controller requires high bandwidth and low latency.

Established Internet technologies fit many cases, but can encounter trouble at the extremes. Many IoT devices today, such as Internet-controllable light bulbs, thermostats, and home security systems, provide an interface using a REST architectural style[1] via the HTTP protocol. However, a tiny resource-constrained device probably can't run an entire TCP/IP stack, and it's unlikely that HTTP can deliver adequate bandwidth or latency for the communication between a drone and its controller. Hence, any solution for the IoT should embrace heterogeneity.

Composing services presents further problems. Suppose you want to connect your light bulbs to your home alarm system? Smartphone apps are the closest we have today to a de facto standard means of accessing IoT devices, yet they're difficult to compose. This problem has created an opportunity for aggregators. On the software side, for example, IFTTT (if this, then that), supports constructing "recipe" rule sets that accept input from Internet-accessible services and issue responses using Internet-accessible services. All-in-one hubs like Revolv, the Thing System (TTS), and SmartThings provide hardware or software that consolidate access to various smart-home devices into a single application. Lacking standards, services and devices operate using proprietary APIs and mechanisms, so at any given time, aggregators support only a bounded set of third-party devices and services. This can slow innovation by hindering new entrants to the market, whose products aren't supported.

The Industrial Internet Consortium (with the Object Management Group and affiliates) and the IEEE Standards Association are leading efforts to promote interoperability standards. In this article, we offer a particular approach to such interoperability that scales well, and enables millions of creative minds to invent new IoT technologies by repurposing existing ones. In keeping with previous work,[2] here we call applications that integrate networked sensors and actuators *swarmlets*, a bow towards Jan Rabaey's phrase "the swarm at the edge of the cloud," which hints at these devices' potential vastness (and ominousness).[3] We want to shift the focus away from standardizing over-the-network communication and APIs, and towards mechanisms by which diverse, proprietary, and secure communication protocols and APIs don't hinder devices' and services' interoperability.

A Scenario

Consider a scenario in which a startup company produces a robot called C4PO that wanders a space, such as a factory floor, that already contains a variety of networked sensors, security devices, and other robots made by other vendors. Those vendors have never heard of the company providing C4PO and offer no support for its interfaces; its vendor has no knowledge of the over-the-network APIs provided by those third-party devices. Upon removing the robot from the box, the user configures it with credentials for accessing the local wireless network, perhaps using a smartphone app and optical communication (as done, for example, by Electric Imp). Once it's on the local network, it sends a multicast discovery packet, in the style of universal plug and

Table 1. Some Internet of Things (IoT) resources.

Resource	Category
Alljoyn www.alljoyn.org	Discovery
Contiki www.contiki-os.org	Operating system
Chromosome www.fortiss.org/en/research/projects/chromosome	Middleware
Constrained Application Protocol (CoAP) http://coap.technology	Connectivity protocol
Krikkit http://eclipse.org/proposals/technology.krikkit	API
Message Queuing Telemetry Transport (MQTT) http://mqtt.org	Connectivity protocol
Node-Red http://nodered.org	Software integration platform
OceanStore http://oceanstore.cs.berkeley.edu	Data storage
OpenRemote www.openremote.org	Software integration platform
Open Sound Control http://opensoundcontrol.org	Networking and communication
OpenWSN http://openwsn.org	Networking and communication for wireless sensor networks (WSN)
POMI http://pomi.stanford.edu	Networking and communication
Sensor Andrew www.ices.cmu.edu/censcir/sensor-andrew	Connectivity protocol
SmartHome http://eclipse.org/smarthome	Software integration platform
sMAP www.cs.berkeley.edu/~stevedh/smap2	Connectivity protocol
The Thing System http://thethingsystem.com	Software integration platform
Thread www.threadgroup.org	Wireless mesh network
Universal plug and play (UPnP) www.upnp.org	Discovery
Extensible Messaging and Presence Protocol (XMPP) http://xmpp.org	Connectivity protocol

play (UPnP), for example. It receives responses providing IPv6 addresses, from which it obtains XML files defining accessors. The robot downloads an accessor and executes it in a manner similar to the way a browser downloads HTML5 with JavaScript and executes it.

C4PO begins by filtering the accessors it discovered to select those providing proximity data. Specifically, it looks for accessors that, when triggered, output a measure of the physical distance from C4PO to the device to which the accessor provides access. C4PO needn't know exactly how the accessor obtained this distance measure. All C4PO needs to know is that the accessor indeed provides a distance measure, and that it's capable of executing the accessor (it's a suitable accessor host). For example, such an accessor might declare that it "requires" the accessor host (C4PO) to provide a Bluetooth API so that the accessor can provide a measure of distance using a Bluetooth low energy (BLE) beacon technology such as iBeacon. C4PO has Bluetooth capability, so it's compatible with the accessor and can execute it. Another accessor might declare that it requires a GPS API, and because C4PO has no GPS capability, it can't execute that accessor. C4PO collects all compatible accessors that it discovers on the local network and executes them.

Some of the accessors then provide output data, and some don't (some devices are out of range). Those that do provide (noisy) distance measures to a device such as a video camera, motion sensor, or door lock. C4PO wishes to construct a local environment map, but noisy distance measures don't provide enough information. However, because C4PO can move, and dead reckoning can estimate its motion, it can combine multiple measurements to convert noisy distance information into relative location estimates. To do this, C4PO leverages another accessor that provides a cloud-based machine-learning service that, behind the scenes, uses particle filtering to estimate location, and an optimization algorithm to recommend motion vectors to the robot.

The machine-learning and optimization algorithms are quite computationally heavy and memory intensive, and are beyond C4PO's energy budget, so offloading them to the cloud makes sense. However, the cloud-based service may aggregate data from multiple robots that happen to be measuring distances to the same devices, as identified by their globally unique identifiers (GUIDs). The possibility for such data aggregation, which can improve accuracy, is a key advantage of using the cloud-based service instead of performing computation locally. This might be even more important than energy savings.

In the end, C4PO forms a map of the objects' relative locations in the environment. It then starts using that map to navigate through the space, detect intruders, or track assets. It can leverage additional accessors provided by devices it discovers, such as video cameras that provide it with images of its own surroundings, or sensors that provide temperature, air quality, and light levels. And it can instantiate accessors to leverage cloud-based services to, for example, perform image classification, face recognition, and natural language understanding. Most importantly, because of the accessors' actor interface, C4PO can compose accessors, building more complex and capable services out of simple ones.

This scenario illustrates that if we endow devices with the ability to download and execute accessors in a manner similar to how browsers today download and execute HTML5 with JavaScript, then we can change the IoT infrastructure from one dominated by closed proprietary systems that manifest themselves as platforms, the interoperability of which is negotiated between stakeholders, to one that embraces heterogeneity and encourages innovative new technologies to emerge from the composition of independently designed components. This way, vendors no longer need to support all other vendors' over-the-network APIs, a strategy that discourages innovation and new entrants, because at best each new entrant is incompletely supported by already deployed systems. Moreover, the cost of interoperating with all existing deployed systems is prohibitive, precluding small, energy-efficient devices, and increasing barriers to entry into the market.

An Actor Model

The underlying principle behind accessors is a design pattern called *actors*, whereby concurrent components send each other messages via ports. Actors are a concurrent model of computation (MoC) originally proposed by Carl Hewitt.[4] The actor's interface — input ports that receive messages and output ports that send messages — contrasts with a more typical imperative API, which defines which procedures we invoke. Actors are intrinsically concurrent, consistent with many views of dataflow, process networks, interaction, and rendezvous-based concurrency.

The Ptolemy Project showed that we can unify these various approaches to actors under actor abstract semantics.[5] The Ptolemy II open source software framework[6] demonstrates that we can combine these various approaches usefully in heterogeneous designs.

In the following, we describe a way to construct swarmlets by composing accessors, which are actors that wrap sensors, actuators, and services. A swarmlet host instantiates an accessor, treating it as a local actor. It sends inputs to the accessor's ports, invokes one or more of a small set of action methods, and receives outputs via the accessor's output ports. The swarmlet host needn't necessarily trust the accessor, because the host executes the accessor in a sandbox to constrain potentially malicious code, similar to how a browser executes JavaScript on a webpage.

The mechanism by which the accessor interacts with a sensor, actuator, or service can be any of those in Table 1, but it can also be something entirely customized and proprietary. In effect, we shift the standardization problem from trying to find commonality among the varied protocols in Table 1 to using an interfacing schema (actors) that already demonstrated its ability to handle many of the required concurrent interaction patterns. Moreover, this architecture makes it much easier to compose separately developed accessors to create new accessors and services. Actor compositions have well-studied and desirable formal properties, which helps yield predictable behavior under composition. The architecture thus facilitates creative composition of multivendor IoT devices and services.

Accessors

An accessor is a component class that a swarmlet instantiates to access a device or service. Figure 1 shows the pattern. An accessor has *inputs*, by which the swarmlet makes requests, and *outputs*, by which the service issues responses. The responses needn't be synchronous with the requests. They can be callbacks or "push" notifications. An accessor needn't even have any inputs; instead, it can produce outputs spontaneously.

Examples of Accessors

To help make the idea concrete, consider the simple accessor sketched in Figure 2. This defines a class that extends a base class called JSAccessor, provided by the TerraSwarm Research Center (www.terraswarm.org). This base class provides a JavaScript environment in which the accessor executes.

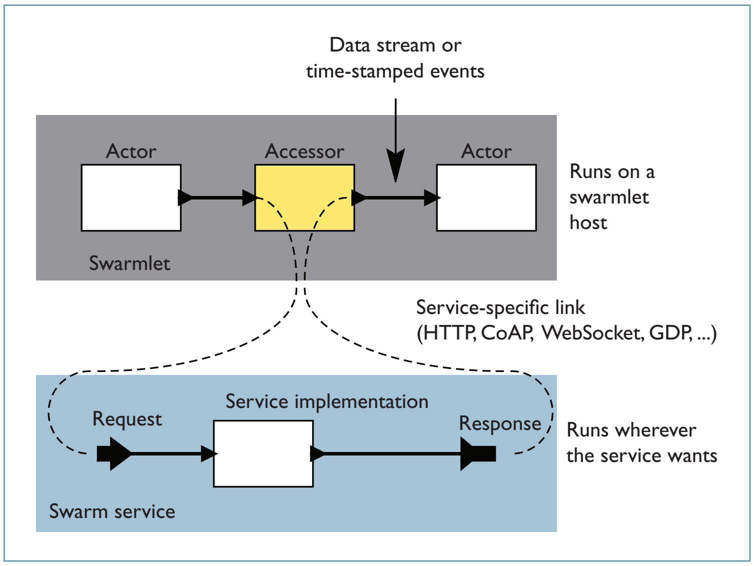

Figure 1. Design pattern of accessors. An accessor has inputs, by which the swarmlet makes requests, and outputs, by which the service issues responses.

The accessor in Figure 2 has several inputs, some of which specify a bulb and bridge (a gateway providing Internet access to the bulb), and some of which specify commands to the bulb. C4PO might use such an accessor to illuminate its surroundings to make another accessor, providing images from a video camera, more effective. A swarmlet host (which could be C4PO, a computer, a handheld device, an embedded device, or a virtual machine) could download the accessor specification (the XML text in Figure 2) from an accessor library. The host must trust the locally implemented base class, `JSAccessor`, but it needn't trust the accessor itself. The base class will sandbox the script's execution.

The script inside the `<script> ... </script>` element defines three functions, one of which is shown. The `fire` function reads commands from the accessor inputs, constructs a JSON data structure, the layout of which is specific to the Philips Hue, and sends it to the gateway using an HTTP PUT. Although using HTTP PUT is standard in RESTful interfaces, the command's syntax isn't, and using accessors, we needn't standardize this syntax.

The swarmlet host *instantiates* the accessor, provides it with inputs, and *invokes* it whenever it chooses. In this case, "invoking" the accessor means executing the specified `fire` function in a JavaScript context that provides a way to read

```
<?xml version="1.0" encoding="utf-8"?>
<class name="Hue" extends="org.terraswarm.JSAccessor">
 <requires name="http" .../>
 <input name="bridgeIPAddress" ... />
 <input name="lightID" ... />
 <input name="on" ... />
 <input name="brightness" ... />
 ...
 <output name="success" .../>
 <output name="error" .../>
 ...
 <documentation type="text/html"> ... </documentation>

 <script type="text/javascript">
  // <![CDATA[
function initialize() {...}
function fire() {
  var command = '{"on":false,';
  if (get(on)) {
   command = '{"on":true,';
  }
  command = command
     + '"bri":' + get(brightness) + ','
     + '"hue":' + get(hue) + ','
     + '"sat":' + get(saturation) + ','
     + '"transitiontime":' + get(transitiontime)
     + '}';
  var response
   = httpRequest(url, "PUT", null, command, timeout);
  if (...response is an error...) {
            send(response[0].error.description, error);
  }
}
function wrapup() {...}
  // ]]>
 </script>
</class>
```

Figure 2. Sketch of an accessor definition file for a Philips Hue lightbulb.

to the accessor inputs (the `get` function in Figure 2), a way to send outputs (the `send` function), and other facilities such as the `httpRequest` function. Using such a function is part of the accessor interface given by the `requires` entry. A swarmlet host must provide this function to be compatible with this accessor.

In this example, the particular underlying RESTful service doesn't require authentication, and neither the request nor the response is encrypted. Yet an accessor-server pair might require both authentication and encryption. The mechanism by which the accessor communicates with a server is up to the service provider and only factors into the interface between the accessor and the swarmlet host through "required" capabilities, which might be very low-level (network access using UDP, Bluetooth radio, or HTTP access to the Web). This separation of concerns is central to the concept of accessors.

Many interesting variants of this example exist. An accessor could provide outputs spontaneously whenever the data satisfy some criterion — for example, whenever a sensor value moves by a specified delta. Or it could provide historical data over some time range. For example, the accessor could time stamp the data provided by an accessor or commands to it, to leverage synchronized clocks. At the very least, accessors could provide database accesses, time series data, data analysis, actuation and control, and interactive services.

Implementation

A swarmlet host instantiates and executes an accessor. A simple swarmlet will consist of only one such host, but a more interesting one might have many such hosts that interact through swarmlets (functioning as a distributed swarmlet).

The example in Figure 2 deliberately uses industry-standard notations and mechanisms for untrusted code (XML and JavaScript). The base class, `JSAccessor`, which provides the context for parsing XML and executing JavaScript, is installed on the swarmlet host, and is presumably trusted. It can provide, for example, a sandbox environment for executing the JavaScript, as is done in modern browsers.

The simplest implementation of a swarmlet host that could invoke the accessor in Figure 2 is an ordinary browser, provided via a webpage, with a JavaScript implementation of the swarmlet host (we prototyped such a host). It instantiates accessors and constructs a webpage where their inputs are rendered as entry boxes in HTML form. A “fire” button provided on the HTML page fires the accessor, upon which any output displays on the HTML page using standard asynchronous JavaScript and XML (AJAX) mechanisms.

We also prototyped a swarmlet host in Node.js (also JavaScript, but independent of a browser), and in Ptolemy II,[6] which is written in Java. Ptolemy II provides a graphical editor for composing accessors and operating on their I/O data. Figure 3 shows a whimsical example of a swarmlet realized in Ptolemy II. This swarmlet uses an instance of another StockTick accessor to read a set of stock values from a cloud-based service (top A1 icon labeled “StockTick”). It then averages the prices, and if the average is greater than the previous average, it sets the color of a Hue light bulb to green using an instance of the accessor in Figure 3 (see the bottom A1 icon labeled “Hue”).

A key issue is the accessors’ semantics of composition. In Figure 3, the composition semantics is a particularly simple variant of dataflow known as synchronous dataflow (SDF), executed periodically within some specified period. A more intelligent swarmlet would use a time-based MoC such as discrete events (DE) to activate the light only when markets are open, and use a third accessor to activate the light only when someone

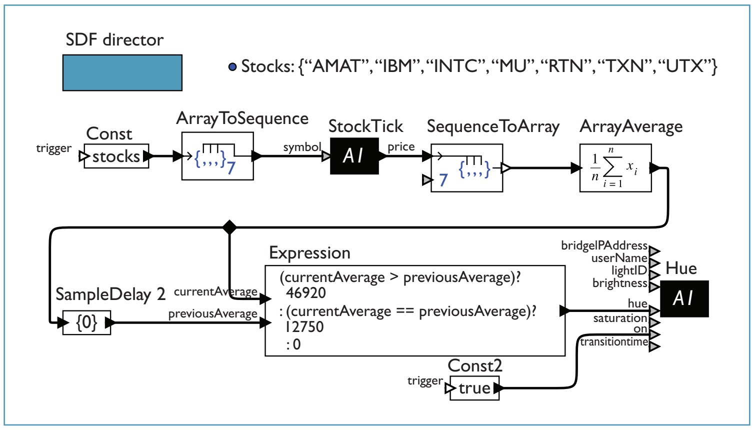

Figure 3. Composition of accessors in Ptolemy II. Here, a swarmlet uses a StockTick accessor to read a set of stock values from a cloud-based service. It then averages the prices, and if the average is greater than the previous average, it sets the color of a Hue light bulb to green. (SDF = synchronous dataflow.)

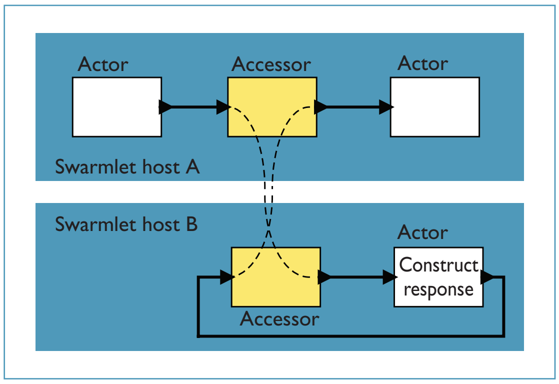

Figure 4. Peer-to-peer accessors, where a swarmlet on one host provides a service to a swarmlet on another host. Each swarmlet interacts with the other through an accessor, which functions as a local proxy.

is in the room. Note that three distinct vendors could provide the occupancy sensor, the light bulb, and the stock price service and use three distinct protocols for access to their respective devices or services. But they’re easily composed, and the swarmlet designer chooses the composition semantics according to his or her needs.

The server side for the accessor in Figure 2 is a gateway box on a network for which an accessor instance must have an IP address. Alternatively, the server side could itself be implemented as an accessor, as Figure 4 shows. Here, swarmlet host *B* provides the “service” for the accessor on swarmlet host *A*. An input on *A* is an output on *B*, and vice versa. When the accessor on *A* receives an input, it sends a request to *B* (by whatever mechanism the accessor defines — for example, an HTTP GET). The accessor on *B* produces this request as an output. The actor on *B* constructs a response to this request and sends it back to its accessor as an input. The accessor then sends the response back to *A* (by whatever mechanism — for example, a response to the HTTP GET), whose accessor produces that response as an output. This peer-to-peer style of accessor interaction suggests that we can construct swarmlets as large networks of interacting accessors, and that gateways needn't be vendor-locked, an untenable situation that will lead to a large proliferation of gateways. The peer-to-peer nature of this architecture contrasts starkly with the centralized composition mechanism provided by IFTTT, for example.

Accessors' and Hosts' Compatibility

To be useful, a swarmlet host must be able to use and compose a variety of accessors. However, accessors can get quite sophisticated, some swarmlet hosts are constrained devices, and the particular security implications of running swarmlets could require different gradations of trust. Hence, a given swarmlet host might not be able to instantiate all accessors. A well-defined interface ensures that a swarmlet host can check compatibility statically.

Accessor Interfaces

An accessor interface defines the inputs, outputs, and required capabilities (such as network access). Such an interface enables search, discovery, and service composition. Figure 2 shows the interface in XML, and tools checking for compatibility need only check the XML. In particular, the swarmlet host needn't check the body of the `<script>` element, because at runtime if the accessor attempts to invoke capabilities that the accessor hasn't declared "required," the swarmlet host can reject the execution.

Here's where standardization is valuable. We envision a basic standard that's extensible through "requires" tags and references to ontologies for sensors, actuators, and data. Researchers must address many issues in the standardization process, including the type system used for I/O and version management.

Component Accessors

A *component accessor* has an interface and a script, as Figure 2 shows. The script defines one or more functions invoked by the swarmlet host. For example, in Figure 2, the script defines a `fire` function. In this case, "standardization" is provided through defining the accessor base class, which is `JSAccessor` in Figure 2. That base class supports JavaScript scripts, and invokes specific procedures defined in the scripts, like `initialize`, `fire`, and `wrapup` in Figure 2. A different base class might use a different scripting language or a different set of functions that it invokes.

We envision other base classes, such as `PythonAccessor`, but in our experiments, we focused on JavaScript because of its well-established use in browsers and its relatively well-understood security risks. We also envision component accessors whose scripts require specialized extensions in the swarmlet host — for example, to access local hardware resources. Such accessors might require that the swarmlet host "install" the extension, where "installation" is an explicit expression of trust. An installation process allows for a careful assessment of the origin and function of an extension being installed, much like installing software on personal computers today.

A component accessor is executable on any swarmlet host that includes an implementation of the accessor's base class and specified required extensions. The base class effectively defines a standard for a family of accessors. It also significantly reduces security risks by binding particular functions and variables usable by the script, as done today in browsers.

Composite Accessors

A *composite accessor* is an accessor composed of other accessors on a single swarmlet host. The composition connects outputs of one accessor to inputs of another and defines an execution policy for the component accessors. Specifically, when a swarmlet host invokes the composite accessor, how should it invoke the component accessors? Whatever mechanism it uses, the composition must itself define an accessor. One possibility is for the swarmlet host to implement a MoC that governs the execution and communication between accessors. The swarmlet host realized in Ptolemy II is already capable of this, using Ptolemy directors.[6] A second possibility is for the accessor to script the accessors' composition in some scripting language (such as JavaScript), where the language's capabilities are sufficiently restricted to ensure safe execution. In this case, the accessor itself defines the MoC that binds the component accessors.

Possibilities for MoCs that we can implement through either of these mechanisms (directors installed on the hosts or scripts provided by the accessors) include at least the following:

- imperative programming, giving a sequence of actions;
- pipe-and-filter (for example, a dataflow model);
- process networks (deterministic Kahn networks, message passing interface, or Scala-style nondeterministic message passing);
- publish-subscribe, of which there are many variants;
- event notification (like the callbacks in Node.js);
- discrete events (with locally defined time stamps, or with globally defined time stamps relying on clock synchronization); and
- synchronous-reactive models and their asynchronously executed variants.

Note that for untrusted accessors, it might be important to choose a less expressive MoC, where a composition can be statically analyzed for memory usage and deadlock, for example. See Len Sassaman and his colleagues' work for a discussion of why less expressive mechanisms improve security.[7]

MoCs

Composability is essential to the concept of accessors. But what should be the semantics of composition? The interface we chose for accessors is deliberately designed to enable disciplined concurrent composition. Specifically, the accessor interface we define conforms to the actor abstract semantics,[5] which generalizes the classical notion of actors[4] to embrace a richer set of concurrency models. In particular, such interfaces have been shown to be compatible with dataflow, process networks (PN), publish-and-subscribe, DE systems, synchronous-reactive (SR) systems, and rendezvous-based models. We implemented all of these MoCs using the actor abstract semantics in the open source Ptolemy II system.[6]

The same accessor definition can be useful in multiple MoCs, a concept that we call *domain polymorphism*.[6] The notion of "invoking" an accessor is a key part of the actor abstract semantics. The script given in Figure 2 defines an *action method*, `fire`. In dataflow, we invoke this method when there's sufficient input data for the accessor. In discrete events, a firing occurs when one or more inputs of the accessor have an event with an earliest time stamp. In synchronous-reactive models, `fire` is invoked at ticks of a conceptual global clock. In process networks, actors fire asynchronously, each in a separate thread of control.

The actor abstract semantics defines a few other action methods. The swarmlet invokes the `initialize` method when it starts executing (or if it gets reinitialized during execution). The swarmlet **host** invokes the `wrapup` method when it stops executing due to either normal termination or an unhandled exception. In addition, we need a mechanism for an accessor to request a future firing. Our `JSAccessor` base class provides a timeout function in the JavaScript context for this purpose.

Using a timed concurrent model of computation such as discrete events to run accessors offers some interesting possibilities for composition of accessors. It's common when using a callback style of concurrency, such as in Node.js, to use timeouts to manage concurrency.[8] However, timeouts are difficult to control, because they have weak semantics in JavaScript. If you want two actions to be performed later in a specific order, setting a longer timeout for the second action doesn't ensure that it executes later. Moreover, there's no notion of simultaneity or causality, so it's difficult to ensure that we execute two actions in time to simultaneously provide inputs to a third action. We might achieve these effects with large differences in timeouts, but there are no guarantees. A timed MoC like DE, however, provides deterministic timed concurrency.[6] Even more interesting is to leverage network clock synchronization to get distributed services with deterministic composition semantics.

The key is that the swarmlet, not the accessors, determines the composition of accessors, what concurrency model to use (if any), and whether to use a timed model. The accessor's provider, which might be an IoT device vendor or a third party, needn't have any idea that the device will be used in a distributed real-time system that exploits a sophisticated coordination mechanism. And there's no need to standardize that coordination mechanism, because many swarmlets won't need it.

This article offers a starting point for developing a rich mechanism. Because of the ability to download and compose accessors, create new accessors, and construct swarmlets out of peer-to-peer communicating accessors, the code mobility usage is more sophisticated than conventional client-side code evaluation in a browser; Web applications don't have a general mechanism for composition.

This raises some interesting and challenging research questions, many of which we left untouched. For instance, how should we publish, register, search for, and discover accessors? Users might need to sign some accessors to establish trust. To what extent can we use static analysis to ensure some measure of safety regarding accessor scripts? What constraints should be imposed on the context in which we evaluate scripts? What services should we provide for encryption and authentication? Most accessors require interaction with a persistent process or service. How should the accessor handle discovery of that process or service? What should the life cycle of those persistent processes look like? How do we deploy a swarmlet in the cloud? Could a swarmlet migrate from one host to another? How should we manage versioning of accessors? A great deal of work remains.

Acknowledgments

This work was supported in part by the TerraSwarm Research Center, one of six centers administered by the STARnet phase of the Focus Center Research Program (FCRP), a Semiconductor Research Corporation program sponsored by *Microelectronics Advanced Research Corporation (*MARCO) and the US Defense Advanced Research Projects Agency (DARPA).

References

1. R.T. Fielding and R.N. Taylor, "Principled Design of the Modern Web Architecture," *ACM Trans. Internet Technology*, vol. 2, no. 2, 2002, pp. 115–150.
2. E.A. Lee et al., "The Swarm at the Edge of the Cloud," *IEEE Design & Test*, vol. 31, no. 3, 2014, pp. 8–20.
3. J.M. Rabaey, "The Swarm at the Edge of the Cloud: A New Perspective on Wireless," *Proc. Symp. VLSI Circuits*, 2011. pp. 6–8; http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5985988.
4. C. Hewitt, "Viewing Control Structures as Patterns of Passing Messages," *J. Artificial Intelligence*, vol. 8, no. 3, 1977.
5. E.A. Lee, S. Neuendorffer, and M.J. Wirthlin, "Actor-Oriented Design of Embedded Hardware and Software Systems," *J. Circuits, Systems, and Computers*, vol. 12, no. 3, 2003; doi:10.1142/S0218126603000751.
6. C. Ptolemaeus, ed., *System Design, Modeling, and Simulation Using Ptolemy II,* Ptolemy.org, 2014; http://ptolemy.eecs.berkeley.edu/books/Systems.
7. L. Sassaman et al., "Security Applications of Formal Language Theory," *IEEE Systems J.*, vol. 7, no. 3, 2013, pp. 489–500.
8. S. Pasquali, *Mastering Node.js.*, Packt Publishing, 2013.

Elizabeth Latronico is an associate project scientist in the Electrical Engineering and Computer Science (EECS) Department at the University of California, Berkeley. She focuses on predictable composition for the Internet of Things. Latronico has a PhD in electrical and computer engineering from Carnegie Mellon University. Contact her at beth@berkeley.edu.

Edward A. Lee is the Robert S. Pepper Distinguished Professor in the EECS Department at the University of California, Berkeley, and director of the nine-university TerraSwarm Research Center. He works on cyber-physical systems and models of computation. Lee has a PhD from the University of California, Berkeley. He's an IEEE Fellow. Contact him at eal@eecs.berkeley.edu.

Marten Lohstroh is a PhD student in the Computer Science Department at the University of California, Berkeley. He studies models of computation, programming languages, and systems design. Lohstroh has an MS in grid computing from the University of Amsterdam. Contact him at marten@eecs.berkeley.edu.

Chris Shaver is a graduate student in the Computer Science Department at the University of California, Berkeley. He studies the formal semantics of models of computation, focusing on composition and heterogeneity. Shaver has a BS in engineering from the Cooper Union for the Advancement of Science and Art. Contact him at shaver@eecs.berkeley.edu.

Armin Wasicek is a postdoctoral researcher at the University of California, Berkeley. He researches cyber-physical systems, safety and security, and model-based design. Armin has a PhD in computer engineering from the Vienna University of Technology and is a Marie Curie Fellow. Contact him at arminw@eecs.berkeley.edu.

Matthew Weber is a PhD student in the Computer Science Department at the University of California, Berkeley. His research interests include cyber-physical systems, indoor localization, and the semantics of context in the Internet of Things. Weber has a BS in computer engineering and computer science from the University of Virginia. Contact him at matt.weber@eecs.berkeley.edu.