Meadow: A Software System for Cooperating Devices

1 Introduction

Programming multiple, autonomous devices to accomplish a well-defined task is not easy. Consider programming n devices to perform a task which requires interaction of these devices. Assuming that they depend on *message passing* for interaction, which is the norm in distributed computing, guaranteeing that the system works correctly under any order of interactions is not trivial. Though huge amount of research efforts have been devoted to beahvior of concurrent systems, most efforts have beeen focused on *specification* and verification of systems, rather than "programming" such systems [2, 3, 4, 5].

As more programmable devices are connected, coordinating them to perform tasks is becoming common activities. Many new technologies, such as Internet of Things, robot programming, industrial automation, and smart cars, require interaction of autonomous components to accomplish a task.

The Meadow system allows a developer to program the devices from a higher viewpoint, just like a conductor orchestrate a group of musicians to play a single piece of music. Devices are considered as "programmable objects" which has some predefined native capabilities. Using **the Meadowview language**, users can define a workflow over a set of devices. The defined workflow can be executed using **the Meadow runtime**, which is installed on each of the participating devices.

1.1 Devices

In the Meadow system, a **device** is a programmable entity, which offers three types of services to other devices – *functions*, *properties*, and *events*. A **native function** is a predefined (or built-in) function provided by a device. For example, an LED banner can display a message on the screen. The Meadow system uses the notion of "method invocation" or "function call" as an abstraction through which one devices gets the native function service from another device (as in RPC). A **property** is a value stored on the device which can be read or written by itself and other devices. Through properties, devices can perform demand-driven communication (a.k.a. pull dataflow model) An **event** of a device represents a change of state which can be of interest to outside. Any outside object which is interested in the event can add a callback process to the event. Using events, devices can perform data-driven communication (a.k.a. push dataflow model).[1]

[1]Note that event-based programming is inevitable in distributed systems, where a single process does not have a global knowledge about when a certain event will happen. Consider a traditional single-threaded program – for simplicity, assume the program is a straight-line code. You can follow the thread of control from the first statement of the `main()` function until you hit the last statement of this function. A single statement in this program, you have the knowledge when it will happen – "after the previous statement is executed". However, in distributed system, sometimes we want to execute a statement S_0 in process P_0 right after statement S_1 in process P_1 has finished its execution. How could this be done when P_0 and P_1 are totally independent? The most obvious solution is to schedule the execution of S_0 on an event e which will be fired by P_1 upon its completion of S_1.

*The notion of **devices** is very similar to that of **classes** in a (class-based) object-oriented programming language, such as C++.* Just like a C++ class represents a set of objects which has the same set of fields and member functions, a device represents a set of device instances which has the same set of properties, events, and native functions. Differences include that devices support"events" which facilitate *asynchronous programming* through event-based, reactive programming model of the Meadowview language.[2]

Lifetimes of devices and device instances Another difference is the lifetimes of classes and objects are different than those of devices and device instances. For example, let a C++ program that defines a class `MyClass` be given. Then, this class begins to exist when object file of the program which uses this class is executed. An object of the `MyClass` class is created when the runtime generates an internal structure for the object, at the time when "`new MyClass`" is to be executed. Also, it is destroyed as this object is destroyed by the C++ semantics.

Device is more like a database scheme, which is created and registered into a global registry or a name directory. Unless deregistered, the device persists. While objects are abstract entities which can be created and destroyed by the runtime, a device instance is an "handle" to a concrete, physical entity. So, its creation and deletion is more like a registration and deregistration to a registry or a name directory.

While objects are created by the runtime, the existence of device instances is possible inside the runtime by the device instances themselves by *sign-on* and *sign-off* procedures.

Processes The Meadow system allows to describe collective behavior of the involved devices using *processes*. A **process** consists of a set of events, to which it is reactive, and a piece of code. The code is executed when the given events occur. The code can perform computation using the services from the involved devices.

Workflows A **workflow** is a set of processes, prarameterized by *devices*. A workflow can be instantiated with actual device instances.

For further details about the concepts discussed in this section, see the companion document, "Meadowview: A Language for Device Orchestration".

2 Example Setup: Devices

The rest of the document will present a walk-through of main concepts of the Meadow system through an example. For this, we make a few assumptions.

- We assume a common middleware for device communication, such as AllJoyn, is installed in each device. Also, native functionality of the device (e.g. switch channel of a TV) is programmatically registered as AllJoyn methods, signals, and properties.

  In addition to this, the common service interfaces which is defined in AllJoyn service frameworks (e.g. `org.alljoyn.About`, `org.alljoyn.Onboarding`, etc.) are already implemented.

[2] The event-based programming model is arguably the most scalable approach to build large, complicated, distributed systems. The Verilog hardware description language [1] is based on event-based programming model and has been used to build highly-complex chips with billions of gates. One characteristics of distributed systems is that we are don't have a knowledge about when "something will happen", since we don't have a full control/knowledge over all computational activities. In many cases, only way to add some behavior is through adding callbacks to events and let these events–reaction chains do the job.

Note that even if a industry consensus on a single common framework is not possible, we can always create a common wrapper framework, say using AllJoy, which is located above proprietary frameworks such as Apple Home Kit or Google Nest framework.

- *Manufacturers do not know about the Meadow system yet.*

2.1 Device #1: An iPhone app

Assume that the 24 hour fitness club developed an iPhone app using the AllJoyn framework. While this app is being executed, this iPhone (i.e. app) is discoverable from within the AllJoyn network and can provide services to other AllJoyn network. We assume that it supports a very simple service – a `userid` property. By this, we mean that other AllJoyn application can query the value of `userid` in a programmatic way when this app joins an AllJoyn bus.

Making the app known to the Meadow system Now, to make this app programmable in the Meadow system, we need to do a few more things

- register the app as the Meadow device
- register each device instance name
- install the Meadow VM (runtime) on top of the AllJoyn framework

First, to register this app as a device, interface of the device must be defined using the **Meadowview Interface Definition Language (IDL)**.[3]

```
device com.24.app.iphone {
  // property name must match the ``AllJoyn'' property name
  string userid;
  string username;
};
```

This textual definition needs to be compiled and be registered to the **Meadow name server** (for simplicity, we assume there is a central name server. however, to avoid bottleneck, we may elect to implement a decentralized server.) by either the app developer or somebody who wants to create a Meadow workflow. Let's assume that above definition is stored in a file `24app.dev`. Then, the following command will register a globally-unique name to the global server and bind it to the device definition.

```
# meadow-register 24app.dev
```

Second, we need to assign a device instance name. Each installation of the App is unique and we need to tell the Meadow system that

- this app is an instance of `com.24.app.iphone`, and
- the name of this device instance is `abc`, and
- a password (or a public-key) is created for this device instance and registered to the Meadow system.

[3]Typically, an IDL description is mandatory in binding statically compiled languages since it allows to generate stubs against which the callers can be compiled. However, in dynamically-typed languages, these are not mandatory. Meadowview is a dynamically-typed language but we still optionally allow MIDL descriptions to facilitate early error checking instead of runtime error.

The fully-qualified name of the device instance will be `com.24.app.iphone[abc]` in the Meadow system.

Now, when this device signs in, the Meadow system will ask the device to identify itself. The device will tell the system that "I am `com.24.app.iphone[abc]`", which is a globally-unique name, along with a password[4]. When device instance name and password is correct, this device instance is signed on and now this device instance is discoverable within the Meadow network.

2.2 Device #2: Treadmill

The treadmills installed at the 24-hour fitness club in San Jose has the following device definition.

```
device com.24.sj.treadmill {
  string instid;            // instance id
  int status;               // 0: stopped, 1: running

  void start();             // start the treadmill
  void stop();              // stop the treadmill
  void resetStepCount();    // resets the step count
  int getStepCount();       // number of steps since last reset

  event startButtonPressed;
  event stopButtonPressed;
};
```

Also, we assume there are 50 treadmills in the club and each of them are given unique names – e.g. `com.24.sj.treadmill[1F.01]`, `com.24.sj.treadmill[1F.02]`, etc.

2.3 Device #3: TV monitor on Treadmill

Also, for each treadmill, a third-party TV with larger screen was installed. The device definition is given below.

```
device com.24.sj.tv.samsung {
  string id;                // instance id
  int status;               // 0: on, 1: off
  int channel;              // current channel

  void turnOn();
  void turnOff();
  void switchChannel(int ch);
  void printMessage(string msg);

  event turnedOn;
  event turnedOff;
  event channelSwitched {
    int oldChannel;
    int newChannel;
  };
```

[4]Or encryption of a system-provided text using the private-key so that the System can authentication of the device instance.

```
};
```

A total of 50 TVs were installed and their names were assigned to match those of treadmills – e.g. `com.24.sj.tv.samsung[1F.01]`.

2.4 Device #4: Database server

Also, a database server will be registered as a Meadow device. There will be 2 servers, each named as `com.24.sj.dbserver[0]` and `com.24.sj.dbserver[1]`, respectively.

```
device com.24.sj.dbserver {
  void log(time t, string userid, string message);
  void saveStepCount(time t, string userid, int count);
};
```

3 Default interfaces

There are some built-in functions, properties, and signals that are supported by Meadow devices.

```
device Device {
  list<Device> getPairedDevices();
  list<Device> getPairedDevicesByType(string type);

  event devicePairedEvent {
    string devid;
  }
}
```

4 Example #1: Recording Step Counts

Consider the following scenario.

> *Whenever a person, who installed 24-hour fitness app on iPhone, steps onto a treadmill located in 24hour fitness San Jose site, store the step count in the DB. Also, print a welcome message on the treadmill screen.*

Maybe 24hour fitness wants to sell a new service, which will record all customer exercise history, which can be browsed through Web. For this, they can sell a small 10-dollar "SecureID" device, which will contain the "ID" of the owner. They want to program their treadmills that when a user with this ID steps on, it will pair with the device and send the treadmill usage to the DB server.

4.1 Static workflow

One way to program the flow in the Meadowview language is given below.

```
workflow StepCountFlow(com.24.app.iphone app,
                       com.24.sj.treadmill tms[],
                       com.24.sj.tv.samsung tvs[]);
{
  // get handle of any DB server instance
  Device dbserver = getDeviceType("com.24.sj.dbserver");
```

```
  // "foreach" may be similar to Verilog generate statement;
  // not sure if foreach construct is essential but could be
  // useful for configuring multiple devices
  foreach tm in tms {

    // whenever the treadmill is paired with a device,
    // the following process will wake up; "dev" will be
    // bound to a Device object which was paired with the
    // the treadmill
    process @(tm.devicePaired(dev)) {
      if (dev.getDeviceType() == "com.24.app.iphone") {
        string appowner = dev.username;
        tvdev = tm.getPairedDeviceByType("com.24.tv.samsung");
        if (tvdev)
          tvdev.printMessage("Welcome: " + appowner);
      }
    }

    // the following process wakes up when the start button
    // of the treadmill is pressed;
    process @(tm.startButtonPressed()) {
      if (tm.isPaired()) {
        list<Device> pdevs = tm.getPairedDevices();
        foreach pdev in pdevs {
          int pdevid = pdev.getID();
          tm.resetStepCount();
          @(tm.stopButtonPressed());
          int steps = tm.stopStepCount();
          dbserver.addStepCount(dev.getID(24HOUR), steps);
        }
      }
    }
  }
}
```

4.2 Dynamic discovery

```
workflow StepCountFlow(com.24.app.iphone app,
                       com.24.sj.treadmill tms[],
                       com.24.sj.tv.samsung tvs[]);

  list<Device> tms = getDevicesByName("24fitness.sanjose.treadmill*");
  Device dbserver = getDeviceByType("com.unitedhealth.dbserver");

  // actually "foreach" may be similar to Verilog generate statement;
  // I'm not sure if foreach construct is essential but could be useful for
  // configuring multiple devices
  foreach tm in tms {
```

```
    // whenever someone with a device steps on a treadmill, the following
    // process will wake up; "dev" will be bound to a Device object which
    // represents a device that the user owns
    process @(tm.deviceFound(dev)) {
      if (dev.getDeviceType() == "com.myid") {
        tm.pairDevice(dev);
        String devowner = dev.getOwnerName();
        tm.printMsg("Welcome: " + devowner);
      }
    }

    process @(tm.startButtonPressed()) {
      if (tm.isPaired()) {
        Device dev = tm.getPairedDevice();
        int devid = dev.getID();
        tm.resetStepCount();
        @(tm.stopButtonPressed());
        int steps = tm.stopStepCount();
        dbserver.addStepCount(dev.getID(24HOUR), steps);
      }
    }
  }
endflow
```

The Meadow system compiles the above description into "native" code which can be executed by Meadow runtime (VM). And then, the code is deployed to relevant devices. (**NOTE**: This somewhat reminds of the use model of Java VM, though Java does not have the notion of automatic synthesis of choreography of components. Might need to prepare what differentiates Meadow VM from Java VM.)

5 Example #2: Streaming Producer and Consumer

Next example and an interaction between the iPhone and the TV attached to each treadmill.

> *Whenever a person, who installed 24-hour fitness app on iPhone, steps onto a treadmill located in 24hour fitness San Jose site, store the step count in the DB. Also, print a welcome message on the treadmill screen.*

6 Components of the System

6.1 Infrastructure

The Meadow system takes advantages of available infrastructures for device communication. One such infrastructure is AllJoyn framework developed by Qualcomm. This framework allows devices to talk to each other through "method invocation". In particular, under this framework, each device exports its services which can be used in the form of a **method call** (pull service) or **signal notification** (push service).

Another alternative is ZeroMQ message system. In ZeroMQ, it does not have any notion of function calls but allows to deliver a free-form messages to be sent from one device to another.

Application can impose meaning on these messages by defining its own format for service request/reply (i.e. ZeroMQ allows to build customized application-level communication protocol easily).

Basically, infrastructural software allows:

- to name/discover devices
- to export service of a device to outside world
- to get service from other device
- to define events
- to publish events
- to subscribe events
- to add callbacks to the subscribed events

These frameworks typically support multiple protocols: low-level physical protocols such as Bluetooth, ZigBee, WiFi, ethernet, etc. and high-level application protocol such as HTTP, CoAP, etc. Also, they support multiple platforms: Linux, Windows, iOS, MacOS, Andriod, small embedded operating systems, etc.

These infrastructures typically exist in two forms: software library (.so or .a) and daemon. To use these infrastructure, developers could either link with the library or use IPC/socket to communicate with the daemon. Both AllJoyn and ZeroMQ support multiple languages through language bindings. For example, they provide Java class library allows Java programs to the call the software library. These Java class library could use JNI (Java Native Interface) to call the native library in .a or .so.

ASSUMPTION For discussions below, we will assume that AllJoyn is used as our infrastructure. (*However, we can relatively easily change our tool to target different infrastructures – i.e. runtime can be configured to target multiple infrastructures.*)

One major assumption is that all services (e.g. turn on LED, print message on a OLED screen, etc) provided by a device are already developed and registered as an AllJoyn method or AllJoyn signal. Device manufacturer sells a device which has a set of exported services. That is, if we want to define workflow over some devices using Meadow, we expect that all services provided by these devices are exported as AllJoyn methods (also, these devices must internally have AllJoyn daemon inside). In addition to this AllJoyn, they also run Meadow runtime on top of AllJoyn. **THIS COULD BE THE MOST DIFFICULT ISSUE TO OVERCOME. How can we bell the cat – i.e. have manufacturers install Meadow runtime?**

When two AllJoyn-compatible devices, say A and B, are given, we can use AllJoyn API to develop a code which make A and B interact with each other by manually deploying the code. The Meadowview system tries to avoid this hard-coding. We want to control the interaction of devices from a higher-level than than going down to AllJoyn-level.

6.2 The Meadow VM (or Runtime)

Each device runs a Meadow runtime (e.g. in the form of daemon). The runtime runs on top of the infrastructure and is responsible for the following:

- Maintains the set of processes (Verilog always-process like): some are **built-in processes**, some are **user-defined processes**.
  - A process is basically "if event X happens, do Y." The event X is either a "method call request either internal or coming from other device" or "signal which is generated internally or by other device".

  - The "action Y" might need to be limited to very simple actions. For example,
    * **method call**, which is available locally or remotely
    * **signal generation**: generate signal with some payload
    * **simple branching**: IF (simple-cond) THEN Y' ELSE Y"

- Maintains the set of events (i.e. method call request, property get/set request, signals in AllJoyn terminology[5]) that this runtime subscribes or publishes: an event is either a **built-in event**, a **manufacturer's event**, or **Meadow's event**.
  - **built-in event**: some events are very basic events provided by AllJoyn
  - **manufacturer's event**: The method call request, signals defined and registered by the manufacturer
  - **Meadow's event**: Meadow will create internal events for instrumentation purpose.
- Waits for the lower-level framework (e.g. AllJoyn Bus or ZeroMQ message queue) on Meadowview events.
  - Though the device reacts to all three different types of events , at Meadow runtime level, only Meadowview events are relevant. Built-in events and manufacturer's events should have been already registered to infrastructure (infrastructure does not know anything about Meadow runtime) and should be handled directly by the infrastructure.
  - The behavior of the runtime is quite similar to the event loop executed by Verilog simulator.

6.3 Two languages

- **Meadowview: A workflow definition language**: Users define workflows over some devices using a language, to **orchestrate** or **compose device services**.
  - This is like seeing the problem from a conductor viewpoint.
- **Native language for VM** Given that workflow is defined as an orchestration of device services, now we have to derive (or **synthesize**) what each device needs to do to accomplish the job described by the workflow.
  - This is like creating a choreography of devices – define which needs to be done from a viewpoint of "each" device. A device involved in the workflow doesn't need to know the entire picture of the workflow. All it needs to know is "under this circumstance, do this or do that, etc."
  - So, from a Meadowview workflow definition, we need to derive VM code for each devices involved. This code will be have the form of "a set of Verilog process-like code fragments". However, the syntax of such a process must be limited to very simple constructs for efficient VM implementation.

    For example, We may not even need loops since we could restrict any real computation to be provided by the manufacturer-provided method implementation.

    This VM language is a native language for the Meadow system, which is directly executable by the Meadow runtime (VM). See the next section for generated VM process examples.

[5]Essentially, method call or property get/set calls is essentially an "event with a payload" that is sent from other device to this device. A signal an "event (with our without a payload)" that is sent from this device to other device. So, all three different services can be represented using a single notion of "events"

6.4 Synthesis Tool

Given a workflow definition written in Meadowview, this needs to be synthesized to VM processes for execution. There may be multiple ways to synthesize a single workflow definition (just like there are multiple ways to synthesize an Verilog design using AND/OR gates) and synthesizing efficient code will be a big challenge.

The following is one possible example of how the workflow definition given in Section 1 can be synthesized in to VM code. (**NOTE**: The VM language is not yet defined. So, there are plenty of holes, missing pieces, etc.)

```
// signal definition; internal signal generated to represent
// that function call reply has received from the function call
// to getDeviceType of Device dev
//
// synthesis tool must define/register these internal signals
// to infrastructure (e.g. AllJoyn)
signal signal000 {
  string devtype;    // payload of the signal000 (return value of
                     // getDeviceType)
};
process @(deviceFound(dev)) {
  // funccall is a primitive which calls a function and,
  // upon termination of function call, will generate a signal
  // putting the return value of the function call as a
  // payload of the signal
  funccall(dev,            // target device
           getDeviceType,  // function name
           NULL,           // args
           signal000);     // signal that will be generated when
                           // function returns
}

process @(signal000) {
  // process body can have exactly one statement;
  // either a function call or a IF-statement or IF-ELSE
  // statement
  if (signal000.devtype == "com.myid") {
    funccall(self, pairDevice, dev, signal001);
  }
}

signal signal002 {
  string ownername;
};
process @(signal001) {
  funccall(dev, getOwnerName, NULL, signal002);
}

signal signal002 { bool boolvalue; }
process @(signal002) {
```

```
  // how do we support string concatenation? should we support this?
  funccall(self, printMsg, "Welcome: " + signal002.ownername, signal003);
}

process @(startButtonPressed) {
  funccall(self, isPaired, NULL, signal003);
}

process @(signal003) {
  if (signal003.boolvalue) {
    funccall(self, getPairedDevice, NULL, signal004);
  }
}

...
```

As seen above, synthesis tool might need to break down the original workflow definition into tiny small processes. Each process can have one "event" to which the process sensitized and up to one condition checking at the beginning and one action. An action can be either a function call or a signal generation. A called function can be either local or remote.

6.5 Deployment of VM processes

Let's say that a developer employed by 24hour fitness has created a workflow and used a Meadowview compiler to get the synthesized VM code. Then, the problem is how to deploy this code to the treadmills in the San Jose site. For this, several things might be needed. First, two built-in VM processes inside each Meadow VM runtime may be needed:

- **int loadProcess(Process proc)**: a built-in Meadow VM process, which loads the given process into the runtime
- **unloadProcess(int procid)**: a built-in Meadow VM process, which unloads the process with the given id

Also, a loader utility, which loads a VM process(es) to a device may be needed. This loader utility can be triggered either by the person or dynamically by a tool.

6.6 Meadow VM runtime

A VM runtime maintains a list of processes. Whenever some event happens, it picks a process in the list and executes it. Essentially, runtime executes an infinite event loop like that of Verilog S/W simulator.

References

[1] IEEE Std 1364-2001. IEEE Standard for Verilog Hardware Description Language, 2001.

[2] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[3] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer Verlag, 1991.

[4] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[5] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of The IEEE*, 77(4):541–580, April 1989.