Meadow VM: Virtual Machine for The Meadow System

July 28, 2014

1. Introduction

The Meadow VM is a virtual machine with is capable of executing **processes** reactive to events.

2. Basic Concepts

2.1. Events. An *event* represents a class of occurrences such as "something has happened", "state of an object has changes", etc. Each event has a **name**, which is globally unique in the entire Meadow system.

2.2. Event types. Every event has a type. An *event type* is either a scalar type, a structure type, or an array type. An event type specifies the values which instances of the corresponding event can have. Each event type has a **name** which is globally unique in the Meadow system.[1]

2.3. Event instances. While an event means a class of occurrences, an *event instance* is a specific occurrence of the event. In this document, the terms, *event instance*, *event object*, and *event occurrence* will be used interchangeably.

2.4. Method invocation and signals. The Meadow VM doesn't support the notion of method invocation and signals. Instead, we simulate these notions using the notion of events. The next two sections will elaborate this.[2]

2.5. Events vs method invocations. A method invocation in a distributed system consists two separate events: 1) an request event generated by the caller and is delivered to the callee, and 2) a reply event generated by the callee and is delivered to the caller. The first event carries arguments as its payload and the second event carries (optional) return value as it payload.

2.6. Events vs signals. The notion of signals is very close to that of events. The only difference is that the payload carried by signals are limited while events support payload with complex types.

2.7. Events and event instances. In the programming language parlance, one can consider that an event is a *variable* while an event instance is the *value of the variable* at a specific time. Like variables, an event can have a type, called an *event type*.

One important difference between variables and events is that a single variable has a single value at some specific time but an event can have multiple event instances alive at a specific time.

[1]Which event types to support is a topic to discuss. Either we can support static types (at the time of event definition, a type must be associated) or we can support dynamic types (or latent types, where event instances carry its type rather than events).

[2]The notions of **method invocation** and **signals** can be best explained by the Unix system call and Unix signals. When a user program wants to get a service from a kernel, the user program actively calls the system call (PULL). Signals are another way for user programs to get a service from system call. Now, we provide function to be invoked by the kernel when some event happens (PUSH).

One big different between PULL and PUSH that, in the former, the servicee knows exactly when it needs service. In the latter,

2.8. Processes. A *process* is a computational entity which reacts to events. A process has a event[3] to which it is sensitive. When the given event occurs, the task is executed. A task can only perform finite number of primitive operations.

2.9. Primitive operations. A *primitive operation* is an operation that can be natively executed by the Meadow VM. Only a small set of operations are allowed: **variable definition**, **branching**, **assignment**, **foreign function call**, and **event generation**. Each of these operations will be discussed in detail in a later section.

2.10. Variables. A *variable* in the typical sense is also supported in the Meadow VM. The scope of a variable is the VM where the variable is defined. That is, a variable defined in a task can be accessed from any task in the same VM. Also, a VM cannot access a variable defined in a different VM.

Note that a *variable* is a way to model the notion of **state**. By limiting the scope of a variable to a single VM, we effectively remove the globally-maintained state. Supporting the notion of global state may not be *scalable*.

3. Namespaces

4. Processes

4.1. Process specification. A process specification is a code which describes the behavior of a reactive process. A process specification consists of two parts: the **event** and the **task body**.

4.2. Event list. The event list is a list of names of events which will wake up the process.

4.3. Process body. The process body is a code which will be executed when any event in the event list gets triggered – that is, any event instance of an event in the event list is generated.

4.4. Example process specification. The following is a simple process which wakes up whenever an event instance of `order_event` is generated, it will trigger new events depending on the value of the event instance.

```
process handle_order_event(order_event order) {
  if (order.quantity > 10)
    -> large_order_event(order_event.quantity);
  else
    -> small_order_event(order_event.quantity);
}
```

4.5. Event list specification. There are three forms of event list allowed: a single event (e.g. `(event1 a)`), disjunction of events (e.g. `(event1 a OR event2 b)`), or conjunction of events (e.g. `(event1 a OR event2 b)`). More than two events in the event list are not allowed – such cases can be recoded using only two events in the list.

4.6. Process body specification. The body of a process can be one of the following statements: `IF-THEN-ELSE`, `FOR-LOOP`, `TIMING`, `EVENT-TRIGGER`, `REMOTE-FUNC-CALL`.

4.7. Expressions.

4.8. Statements.

[3]Note that a process is only sensitive to one event. Note that this definition can support more complicated situations. First, let a process wake up when both event *A* and event *B* occurs. Then, we can create four processes and two events. always @(A) halfEnabledA = 1; always @(B) halfEnabledB = 1; always @(B) if (halfEnabledA) AB; always @(A) if (halfEnabledB) AB;

5. Examples of processes

5.1. Event filters. An *event filter* is a special type of processes whose input events and output events are the same. It is used to filter out some event instances which are not interesting or useful. For example, the following process filter out all order events whose quantity is 0. Event filters can be used in event window definition.

```
filter postive_quantity(order_event o) {
  if (o.quantity > 0) {
    -> o;
  }
}
windows w0(order_event o) = {
  filter(positive_quantity);
  #5(events); // max 5 events
  @5(events); // max 5 seconds
};
```

5.2. Event aggregators. One can join two different events into one.

```
window agg(order_event, cancel_event) = {
  -> (order_event, cancel_event)
};
```

5.3. Event windows. An **event window** is a special type of processes which takes events as its input and produces events as its output. Event windows are defined based on the number of events which can be retained inside the window and/or the time period.

```
window order_event_default_window(order_event) {
  // size-1 window with infinite interval; which is
  // created implicity by default
  #1(events);
}

windows w0(order_event) = {
  #5(events); // max 5 events
  @5(events); // max 5 seconds
};
```

6. Usages of Meadowview EPL

6.1. Building State Machines using Processes.

```
process StateMachine(InputEvent e) {
  int state;

  switch (state) {
  case 0:
    if (e.value == 1)
      state = 1;
      -> OutputEvent(e.value + 1);
    else
      state = 2;
      -> OutputEvent(e.value + 2);
    break;
  case 1:
  }
}
```

6.2. Describing workflow.

```
  process Process1(LaunchProcess1 e) {
    if (e.value == 2) {
      e.process1_finished = true;
      -> LaunchProcess2(e);
    }
    else if (e.value == 3) {
      e.process1_finished = true;
      -> LaunchProcess3(e);
    }
  }

  process Process2(LaunchProcess2 e) {
    -> Process4(e);
  }

  process Process3(LaunchProcess3 e) {
    -> Process4(e);
  }

  process Process4(LaunchProcess4 e) {
    // finish the workflow
    -> FlowFinished(e);
  }
}
```

7. Using Event Cloud

7.1. Users of event cloud. By defining events and processes, we have built event highways. Now, the question is how we can use this. There are two types of users of events and processes: *event producers* and *event consumers*.

7.2. Event producers. After an event is defined, say `order_event`, one can generate its event instances using event producer API in different languages. The following example shows such an example in C++.

```
mv::Runtime *rt = mv::Runtime::getInstance();
mv::Event *ev = rt->lookup("order_event");
mv::EventIntanceArgs args;
args.addArgument("symbol", "AMD");
args.addArgument("price", 4.5);
args.addArgument("quantity", 32);
rt->createEventInsance(ev, args);
```

7.3. Event consumers. Also, event consumers can register its callback to the event.

```
mv::Runtime *rt = mv::Runtime::getInstance();
mv::Event *ev = rt->lookup("order_event");
// make it not copy-constructrable;
// should be heap-alloc'd
mv::EventListener *listener = rt->createListener();
rt->addListener(ev, listener);
```

8. Event Types

8.1. Type definitions.

9. Events

10. Processes

10.1. Processes sensitive to a single event. Simpler processes are sensitive to a single event, as shown below.

```
  process classify_order(order_event order) {
    if (order_event.quantity > 10) {
      -> midsize_order_event(order_event.quantity);
    }
    else if (order_event.quantity > 100) {
      -> large_order_event(order_event.quantity);
    }
    else {
      -> small_order_event(order_event.quantity);
    }
  }
}
```

10.2. Processes sensitive to multiple events. A process can be sensitive to multiple events. In the example below, when at least one of the three events are triggered, the process will be executed.

```
  process p1(alices_value_changed e0,
             bobs_value_changed e1) {
    if (e0.value  e1.value == e3.value) {
      -> value_sum_match_event(e0.value + e1.value);
    }
  }
```

10.3. Processes sensitive to event windows. Processes can be sensitive to event windows.

```
  process p0(w0) {
    if (valid_order(w0.order)) {
      -> ack_event(create_order_id());
    }
  }
```

11. Event Producers

11.1. Event producer API. Meadow library provides event producer APIs in many different programming languages.

12. Event Consumers

12.1. Direct procedure call. Inside the process, users can call a C or C++ function directly.

```
  process @(ack_event) {
    // DPI call in C
    dpi_db_store(ack_table, ack_event.order_id,
                 ack_event.date);
  }
```

13. Language Constructs

13.1. Namespace.

13.2. Device (a.k.a. Agents). A device is any entity which can either produce or consume events.

13.2.1. *Example.* The following code defines a device which can produce an event named `button_pressed`.

```
producer button {
  event {
    button_pressed;
  }
  action {
  }
}
```

13.3. Event instances. An event instance can be created using an event constructor.

```
order_event("cjeong", {"AMD", 3.80, 100}, 0)
```

- Exchange format can be XML, etc.

13.4. Event patterns.

- An **event pattern** denotes a finite state machine, which generates outputs from inputs.
- Does this require stateful implementation?

```
pattern large_order(order_event e1) {
  e1.order.quantity > 100;
}

pattern chip_company_order(order_event e) {
  e.reqexp("AMD | INTC") == true;
}

pattern followed_by(order_event e1,
                    cancel_event e2) {
   (e1.order_id == e2.order_id) and
   (e1.date < e2.date);
};
```

13.5. Event generation.

- **API**: some external application calls API function call for which language binding exists
- **internal**

  ```
  always @('5s) begin // time tick
    -> timegen(5sec_passed);
  ```

14. Implementation

- Oblivious computation
  - particular computation need not be executed at particular core, processor, machine
  - code can be migrated very easily
  - for this, we must not pass large data between sites
  - need a uniform and distributed way to lookup things
  - we may need a distributed memory, cache, or lookup/symbol table which produces a consistent and efficient way of storing commonly accessed data

14.1. Runtime.

- pipelines
- routers
- all stages that consumes ev will wake up
- routing plan (workflow) is generated for eve, where the plan is a topological order of the work, where work is (processor, code) e.g.

  ```
  (pattern-detector, large_order_detect, e1)
  ```

14.2. Compiler.

- A Hadoop-like work-distribution system will compile the code and generate the code
- generate code
  - process: seq of instructions that executes the code in proc body
  - pattern: FSM in some format
  - window: some primitives which can be executed to generate the event window

14.3. Stages.

- kinds of stages
  - pattern detector
  - window generator
  - processor executor
  - router
- components
  - code lookup table
  - event lookup table
  - event processor (which machine is responsible for event) lookup table
- each stage retrieves the arguments, and executes and puts the output result

14.4. Event entry.

- find the given event and insert

```
Event ev = lookup(Event_id);
insertEvent(ev);
```

- put the event to the system
  - event ev is serialized (w/ known data layout) and put into address X
- (optional) rout the event to the relevant server

14.5. Triggering events: Generating event instances.

- calls API function: `insertEvent(event_id, data)`
- sends a data to a TCP/IP port

14.6. Event windows.

- Define some low-level primitive, universal set of operators and then define high-level clauses (retain, slide, etc.) using these universal set.

Contents

Bibliography

[1] IEEE Std 1364-2005. IEEE Standard for Verilog Hardware Description Language, 2005.

[2] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, 1986.

[3] O. Etzion and P. Niblett. *Event Processing in Action*. Manning Publications Co., 2011.

[4] C. Hewitt. Viewing control structures as patterns of passing messages. AI Memo 410, AI Laboratory, MIT, 1976.

[5] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the KEPLER system. *Concurrency and Computation: Practice & Experience*, 18(10):1039–1065, 2005.

[6] J. McKeller. Twisted. In A. Brown and G. Wilson, editors, *The Architecture of Open Source Applications*, volume II: Structure, Scale, and Few More Fearless Hacks, chapter 21. lulu.com, 2008.

[7] J. Saltzer and M. F. Kaashoek. *Principles of Computer System Design: An Introduction*. Morgan Kaufmann Publishers, 2009.

[8] J. Vasseur and A. Dunkels. *Interconnecting Smart Objects with IP: The Next Internet*. Morgan Kaufmann Publishers, 2010.