A Framework for Executable UML Models

Joe Mooney
General Dynamics
Scottsdale, Arizona
Email: Joe.Mooney@asu.edu

Hessam Sarjoughian
ACIMS
Department of Computer Science and Engineering
Arizona State University
Email: Sarjoughian@asu.edu

Keywords: Discrete event simulation, DEVS, Executable UML, State Machine.

Abstract

One approach to support the creation of executable UML models is to utilize an existing DEVS simulation environment. The Discrete Event System Specification (DEVS) formalism excels at modeling complex discrete event systems. An approach to specifying DEVS-compliant models is presented via Unified Modeling Language (UML) state machines. Resultant UML models are executable within DEVS simulation frameworks such as DEVSJAVA. Constructing DEVS-compliant UML models enables early simulation and verification of a design. This paper outlines how the specifics of simulation can be naturally expressed in UML models without significant burden to the UML practitioner. Simulatable models are an excellent precursor and companion to the current models normally developed during design and implementation and may result in significant cost and time savings.

1. INTRODUCTION

1.1. Motivation

Modeling via a combination of DEVS [1] and UML [2] provides a structured approach for the creation of a UML model that can be simulated under an executable DEVS framework. Although, such models cannot be simply handed off to developers for implementation, instead these executable models promote early understanding of a system and allow for formal verification of important aspects of a system which would otherwise have been difficult using UML alone. Additionally, simulatable models have reuse and extensibility potential throughout the software development lifecycle. Simulation is a more attractive proposition when the creation of executable UML models requires only a basic understanding of simulation for a UML practitioner and these simulation constructs can be included into a model with little overhead.

1.2. Why Simulate?

El Sheik et al. [3] present thirteen incentives for employing simulation. Simulation allows for experimentation, time compression and expansion, replaying events to discover why they occurred, animation and visualization of a system, and training. However, objections to simulation of software systems have been raised including the cost of testing twice, once in the simulated version and then again in the real system, and also a belief that with a modern iterative development approach simulation is no longer necessary since the real system can be evolved incrementally thereby nullifying many of the incentives for simulation.

We recommend creating a simulation model using UML wherein most of the aspects of the model particular to simulation are modeled separately wherever possible. Since the UML becomes the common modeling language for both the simulation model and the real software system, the perception of simulation as a disjoint and unnecessary exercise can be reduced. Furthermore, the perceived value of the simulation models will likely be enhanced since these models are the easiest to create and execute due to their relative simplicity. In terms of iteratively developing models, a modeler can begin with a simulatable model and as requirements are solidified evolve a separate model driving the specification of the system for the developers. In so doing we can diminish the resistance to employing simulation during the conceptualization and development of a system. It should be noted that it is possible to evolve a simulatable model into greater and greater levels of precision through decomposition of the model whilst still remaining a simulatable model.

Once we progress into the prototype and production phases, the models become qualitatively different in nature and this progression should not be seen as a simple one-to-one mapping between models and neither should it been seen as sequential phases, rather the development and evolution of the simulation models can continue in parallel through all phases of the software development lifecycle. Beyond the architectural phases, where the simulatable models become mature, the models continue to serve as important tools to verify, experiment, and instruct.

Whenever changes are under consideration, or when various what-if scenarios need examination, additional experimentation using simulatable models may be employed.

1.3. Why DEVS?

According to Zeigler et al [1] "DEVS is the unique form of representation that underlies any system with discrete event behavior". UML is inherently a discrete modeling language [4]. It is therefore natural to consider what forms a DEVS-compliant UML model may take. Models expressed using the Discrete Event System Specification (DEVS) represent a class of systems theoretic models that permit parallel event-based behavior to be expressed concisely and in a manner that lend themselves to formal verification [1]. Although many different simulation formalisms have been advanced over the years, the DEVS formalism has emerged as the preferred formalism due to the fact that other formalisms have been proven to have an equivalent DEVS representation [1]. In particular, a differential equation system specification (DESS) can be simulated by a discrete time system specification (DTSS) through the selection of a sufficiently small constant time interval. A DTSS model, in turn, can be simulated by a DEVS model by constraining the time advance to a constant time. As such, simulations based on DEVS are more general in nature than other approaches such as continuous simulation [6]. DEVS is appealing since it operates at a high level of abstraction yet can yield critical information during an architectural phase that might otherwise not come to light until much later.

Further, it has been shown that DEVS models are particularly suited to the expression of many design patterns and allow an architect to employ patterns usefully at an architectural and modeling stage [7]. It is important to recognize DEVS models solve a general class of problems, but are by no means suitable for all types of problems.

2. MODELING APPROACH

Various approaches to modeling DEVS in UML already exist [5,13,14,11,15]. The focus of this paper is to remedy issues relating to time and synchronization of message delivery such that we have an approach that will enable executable models using existing DEVS simulation frameworks. The approach outlined in this paper summarizes existing thesis work [16].

2.1. Model Architecture

In DEVS, any component that contains other components is called a *coupled model*; non-container components are called *atomic models*. All behavior is derived from atomic models. Generally in UML, a state machine is used to model an atomic model. A coupled model can also be modeled using a state machine. Thus, from a UML perspective one way to think of a DEVS-compliant model is a hierarchy of communicating state machines. Each non-leaf sub-tree represents a *coupled model*. Each leaf node represents an *atomic model*. In our approach messages bound for a peer node must travel through the parent node representing the coupled model and then down to the peer node. This way the state machine for each node is only aware of a generic parent state machine. Data messages always originate from atomic models.

2.2. Ports

In DEVS components (atomic and coupled models) have input and output ports. Output ports from one component can be connected to the inputs of another peer component or to the output ports of a containing component. Likewise, input ports of a containing component connect to the input ports of immediate sub-components or directly to their own output ports.

In UML, a Structured Class is a rough analogue to a DEVS coupled model though it has the capability to have its own responsibilities beyond being a simple container and its ports are bi-directional. We recommend using a UML Composite Structured Diagram to represent a DEVS coupled model, with the restrictions that ports must be named and unidirectional, and there can be no connections from a part back to itself. Corresponding to each port we introduce an event signal type – by convention the name of the port matches the name of the event signal type. In UML, connectors need not attach to components (more correctly *parts)* via ports; this is not an option in DEVS and hence not an option in DEVS-compliant UML. If ports are specified to provide or require an interface, there should only be one such interface specified in DEVS-compliant UML.

For state machines, we map DEVS input ports to events and output ports to event signal generation. If desired, atomic and coupled models could join a system dynamically at runtime. A registration process maps ports to event signal types. In the event output ports remain unconnected after registration they are connected to a null output port for the coupled model, meaning that their outputs are discarded.

2.3. Time in UML

Time is central to DEVS models but in UML 2.0, the handling of time, especially as required for simulation, has significant shortcoming. The UML Profile for Schedulability, Performance and Time Specification [8] seeks to address these shortcomings but is unnecessary for our needs since the profile introduces more complexity than required to achieve a UML representation of a DEVS model. Instead, a simple protocol of time-related events is introduced to resolve these issues. UML does have specific time-related constructs such as *after*, but in terms of simulation the use of UML *after* is problematic since synchronized behavior among models cannot be guaranteed because the time cost of simulation is left unaccounted. Within our modeling approach, time passes only as accounted for by the special event *evSleep(n)*. The act of setting state variables, performing transitions, generating output etc. all occurs in *zero time*. This simulation-specific overhead cannot be reliably accounted for via the UML *after* function.

2.4. Simulating Time

When we communicate between models we need to ensure that, where timing is relevant, the passage of time witnessed by both models is the same. Although a global clock is not defined in UML our protocol of event signaling provides the timing coordination necessary for simulation. Time is counted via event ticks, which are simulations of real time. The elapsed real (wall clock) time between each tick may be of varying duration. The outermost coupled model, the *coordinator*, issues a *evTick(n,sleepExpired)* to each of its contained models and awaits an acknowledgement. These *evTick* events are passed along recursively to all *active* sub-models. Depending on the desired simulation speed, the delay between each tick is adjusted to run faster or slower than wall clock time. Since the execution of the simulation itself, such as sending and receiving messages, takes some time, this time is subtracted from the amount of time to sleep between clock ticks. Thus, by specifying a simulation time of zero, thereby indicating that we should not sleep between ticks, the simulation speed is dictated by the speed of the computer and its resources. We can get close to our intended real time in our simulation if we pause between ticks for the amount of time remaining after the simulation control logic has completed. Obviously, if the amount of overhead involved in simulating is longer that the amount of time we intend each tick to represent then we need a faster computer for our simulation but rarely do we need simulated time to match real time. The beauty of DEVS is that the simulated time unit can be shortened or lengthened to accommodate whatever level of granularity we choose to model.

2.5. Sequence of Events

In DEVS, the outputs from (or events generated by) an atomic model are generated in the *output function* which is invoked *immediately prior* to the *internal transition function* and *never* in direct response to an external event. This is the primary contractual obligation of a designer creating UML2.0 state charts compatible with DEVS. Whilst this may appear counterintuitive at first, it is natural from a simulation perspective – outputs only occur after some (perhaps zero) amount of processing time. Maintaining this restriction keeps the model specification consistent and reduces complexity for large systems. In our approach the output event signals are generated as part of a transition triggered on the internal transition event, *evTick*, which is generated in response to an earlier *evSleep(n)* event.

2.6. Simulating Processing

Our executable models don't actually perform any real work instead we simulate the amount of time the real system would spend on a task by sleeping through the generation of an *evSleep(n)* event where *n* is the number of units of time after which an *evTick* signal will be triggered. This is analogous to the UML *after* event but *after* is not suitable for use in state machines in DEVS-compliant UML since all events must be globally coordinated due to timing considerations.

If a signal should be generated after some amount of time, then instead of using the *after* keyword we generate an event signal to the containing coupled model requesting to be woken after that period of time. In DEVS, it is possible for the subsequent time expiration event *evTick* and an input event to occur simultaneously. We can set a precedence for which event is to be handled first. In DEVS this is called the *confluent* function.

2.7. Modeling Simultaneous Events

There is a thorny issue of handling multiple simultaneous events. If we perceive the inputs to an atomic model as events, then we are confronted with the restriction that multiple simultaneous events cannot be expressed in a UML state machine unless they occur in orthogonal regions [2]. This restriction may appear reasonable where events are processed in close to zero time, but from a DEVS perspective it represents a fundamental hurdle in the UML specification for reactive behavior. DEVS supports multiple events being processed at a given point in time. DEVS also supports time events (e.g. after 10 seconds) occurring simultaneously with other events. Practically speaking, whether two events are truly simultaneous is debatable, but from a modeling perspective it is nonetheless possible, reasonable, and practical to say two events happen say simultaneously precisely 10 seconds from now.

We are left with the challenge of how we react to such simultaneous events. Since simultaneous events are only partially supported in UML2.0, we must compensate for this in our modeling approach. For example, if events e_1 and e_2 are simultaneous, in DEVS, we can model handling these events and then ignoring an event e_3 that occurs during the processing of e_1 and e_2. In UML2.0 this is less straightforward. In UML events are handled one at a time.

As an aside, one may argue that the likelihood of accepting multiple simultaneous events and then rejecting subsequent events does not have many practical applications and simultaneity is only a function of the accuracy of the clock: if the clock were at a much finer grain, simultaneous events may not be simultaneous at all. Simultaneous are therefore events are those that occur within a given window of time and from a simulation perspective these event are unordered – they should be presented together.

In order to simulate the simultaneous arrival of multiple messages we wrap the atomic model with another model that is responsible for bagging all the individual messages destined for the atomic model at the same time. To complicate matters, DEVS allows a model to react to a message and send an output message without time passing, hence multiple bags of input messages may be delivered separately to model at the same time (the clock is stopped during the delivery phase). Each such delivery occurs during a different simulation cycle. The reason we use a message *bag* and not a *list* is to represent the fact that the messages arriving during one clock period have no order since they arrive "at the same time" even if, during simulation, one event appears to precede another (remember the clock is stopped so any ordering during this time must be invisible to the observer – any order must be made explicitly in the model itself).

2.8. Simulation Cycles

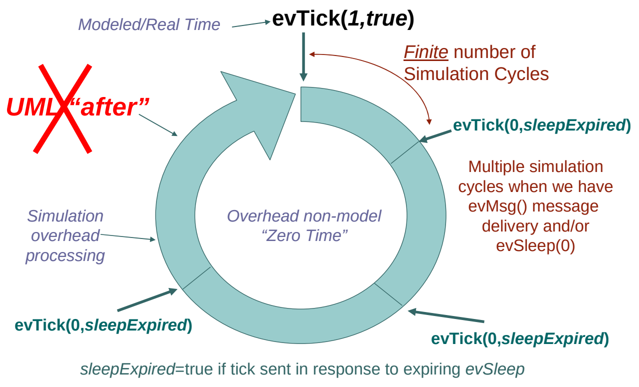

Modeled/Real Time
evTick(1,true)
Finite number of Simulation Cycles
UML "after"
evTick(0,sleepExpired)
Multiple simulation cycles when we have evMsg() message delivery and/or evSleep(0)
Simulation overhead processing
Overhead non-model "Zero Time"
evTick(0,sleepExpired)
evTick(0,sleepExpired)
sleepExpired=true if tick sent in response to expiring evSleep

Figure 1. Simulation Cycle

A clock cycle may have multiple simulation cycles. During the first simulation cycle, a model receives an *evTick(n,sleepExpired)* event with a parameter indicating the amount of time that has elapsed since the last *evTick* message received. A coupled model will only receive an *evTick* message in the event that it has a *timeNext* of zero. Since atomic models may generate outputs in response to an *evTick* signal those messages must be delivered during this clock cycle. However, it is preferred that these messages be delivered as a bag of messages and not delivered individually. To facilitate delivering bags of messages, a coupled model marks as active any model to which it sends an *evMsg* message. Atomic model simulators do not pass messages directly to atomic models upon receipt but rather wait for an *evTick(n=0)* message to arrive. An *evTick(n=0)* will never be the first *evTick* message in the clock cycle since there will never be undelivered messages from a previous clock cycle. Thus, the first *evTick* message in a clock cycle will always have a non-zero amount of time elapsed. An *evTick(n=0)* may also be triggered by the generation of an *evSleep(0)* message by an atomic model.

3. MODELING BEHAVIOR

As in UML where is no formal action language specification, there is no formal action language syntax defined as part of DEVS. Often a DEVS specification involves informal pseudo-code but for executable UML this is not an option. Within the DEVJAVA [12] framework Java is the language of choice. There are reservations to using a procedural language such as Java, for example Stephen Mellor [9] objects to using Java or a similar programming language since modelers are likely to develop specifications that compromise the intended level of abstraction with non-domain specific constructs such as pointers and arrays. Whilst these objections are justified, a pragmatic approach suggests that Java employed in the specification of a DEVS model is not necessarily a poor choice so long as the modeler exercises good choices with respect to its application. Further since simulation models are disjoint from the models used as specification for developers, such code is less likely to leach unchecked into the production code.

4. MODEL COMPONENTS

To recap, each system is composed of a hierarchy of models. The leaf nodes are atomic models, each with an associated atomic model simulator. An atomic model resides in a coupled model which may in turn reside in another coupled model. The outermost coupled model is the *coordinator*. Also, a special type of atomic model called an *experiment* may be specified to drive test execution.

4.1. Atomic Model Simulator

The *atomic model simulator* acts as the interface to the atomic model. This separate state machine handles the arrival and bagging of messages. We also account for the *confluent function* in this model. Also, since UML event signals must be sent to an object, and since we want state charts for atomic models to be reusable components all events generated by an atomic model are sent via the *atomic model simulator*.

4.2. Simulation State Machines

During simulation the coupled models and atomic model simulators control message flow and timing. Both share a similar state machine. The signal *evTick* comes from the containing model and is relayed to any *active* model and to any model with an expiring *timeNext*. The *evAck* signal is generated by a model when it has received an *evAck* from each sub-model to which it sent an *evTick* signal.

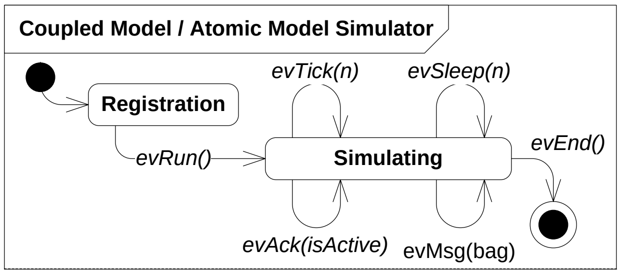


Figure 2. Simplified State Machine for Coupled Models and Atomic Model Simulators

The *evAck* signal has one parameter, *isActive*, which indicates whether any sub-model is active. Note, an active model is any model for which the corresponding atomic model simulator has a non-empty message bag. In this way, the coordinator knows whether there are any active models in the system, and if so, whether another simulation cycle is necessary. An atomic model must generate an *evAck* in response to an *evTick* event after it has generated any external output messages – that is, at the end of its output function

5. EVENTS

Events are either application or control events. Application events are those used for passing the application messages between models during execution and should be derived from the *evMsg* event type. Control events are those used to control the execution of the simulation itself, such as controlling time (*evTick)* and performing registration of models with their respective containers. Since most of the classes used in the simulation are state machines, they receive messages such as the declaration of the input and output ports of the contained models within their event loops via event signals from the atomic or coupled model instances that they contain. The important events are now presented. There are other control events involved in registration and test setup that are not presented.

5.1. evMsg

All application data messages should derive from this event signal type. This event signal as it arrives at its destination atomic model holds an unordered bag of messages sent to the model during that clock cycle.

When a coupled model receives an event it is forwarded to the models that have registered to receive it. If the coupled model has an output port mapped to the event it is sent to the containing coupled model.

If the event is simply passed through directly from an input port to an output port and since these port names must be different the event is signaled using the new output port name.

cmUponEvMsg(Event ev)

```
foreach Model m in models {
    if ( m.isInputEvent(ev) ) {
      m.signal(ev);ackExp++;
      activeClients[m]=true;
      isActive=true;
    }
 }
 if ( container != null ){
   if ( isOutputEvent(ev) )
     container.signal(ev);
   if ( isPassThruEvent(ev) )
     signalPassThru(ev);
}
```

Figure 3. Transition action for *evMsg* in Coupled Model

amsUponEvMsg(Event ev)

```
if ( isOutputEvent(ev) )
  container.signal(mapEvent(ev));
else
   messageBag.concat(ev.messageBag);
```

Figure 4. Atomic Model Simulator *evMsg* transition action

The *evMsg* signal is generated by an atomic model as part of its output function. The output function is the logic performed upon receipt of an *evTick* signal and before the *evAck* signal is generated. The output function is the only time during which external *evMsg* messages may be generated. The *evMsg* type is itself an abstract message type. The atomic model must send a concrete sub-class of this message type. For a coupled model, when an *evMsg* message is received, it is relayed to any sub-models that have the corresponding concrete message type as an input. The *evMsg* is also relayed to the containing model if the coupled model has itself the corresponding concrete message type as an output

5.2. evSleep(*n)*

evSleep(n) signal is a request to be sent a *evTick* after expiration of *n* units of time. This is generally used to simulate the amount of time take to perform processing.

cmUponEvSleep(Event ev)

```
Model m=models[ev.sender]
m.timeNext=ev.n; // n is ticks to sleep
m.timeElapsed=0;
int newTimeNext = getMinTimeNext();
if ( this.timeNext != newTimeNext ){
   this.timeNext=newTimeNext;
   if ( container != null ) // not coordinator
     container.signal(new EvSleep(newTimeNext));
}
```

Figure 5. Transition action for evSleep in Coupled Model

The *evSleep(n)* signal is initially generated by the atomic model and relayed through the atomic model simulator to the coupled model. The coupled model stores the time contained in the *evSleep(n)* event as the *timeNext* for the model. As part of the event signal, the sender is also identified. A coupled model also contains a *timeNext* for itself representing the earliest *timeNext* of all its sub-models. This is recalculated each time an evSleep event arrives since such an event may change the overall *timeNext* for the model. If the new *timeNext* is different than the coupled model's current *timeNext*, then it communicates this new *timeNext* to its own container.

```
amsUponEvSleep(Event ev)
-------------------------
container.signal(ev);
```

Figure 6. Atomic Model Simulator evSleep transition

Also note that if the atomic model wishes to indicate a *passive* state where *evSleep(n)* has a value of infinity, it can pass a value of -1 as the duration of the *evSleep(n).* In this way, the atomic model simulator will remain dormant until it receives another external event

5.3. evTick(*n,sleepExpired)*

evTick(n,sleepExpired) is an event that represents the passage of *n* units of time. Since we must not employ UML time constructs such as *after* we must model time explicitly through our own events. Each *evTick(n,sleepExpired)* event has an *n* parameter that specifies the amount of time that has elapsed since the last tick. Within the same clock cycle, there may be a need for the message bag to be relayed to the atomic model multiple times (each time with a new bag). The fact of its being in the same clock cycle should be transparent to the atomic model simulator with a new tick event having a value of zero for *n*.

When an external event is received the *timeNext == 0* condition will evaluate to false since *timeNext* will be less than zero. The *confluent* function will not be entered. One thing that has not been addressed thus far is passing the elapsed time since the last internal or external transition to the atomic model itself. This is easily accommodated by adding this time as an argument to the messages sent to the atomic model from the atomic model simulator.

5.4. Processing Ticks in Coupled Models

The algorithm for accepting *evTick* events in a coupled model simulator is as follows:

```
cmUponEvTick(Event ev)
-------------------------
ackCount=0;ackExp=0;isActive=false;
foreach Model m in models
   if ( m.timeNext >= 0 ){
     m.timeNext -= ev.n;  // n: number of ticks
     m.timeElapsed += ev.n;
     if ( m.timeNext == 0 ){
        m.signal( new EvTick(m.timeElapsed,true) );
        ackExp++;
        if ( activeModels.contains(m) )
          activeModels.delete(m)
        m.timeNext= -1;
     }
   }
}
foreach Model m in activeModels
       activeModels.delete(m);            // tick not due
       m.signal(new EvTick(0,false) ); // to expired
       ackExp++;                           // timeNext
}
```

Figure 7. Transition action for evTick in Coupled Model

For each model contained within the coupled model, a *timeNext* is maintained containing the amount of time remaining until the next scheduled *evTick.* Also, a *timeElapsed* is maintained containing the amount of time elapsed since the last *evTick* was sent to that model. If there is no scheduled *evTick*, then the time remaining will be a negative number. Atomic models send an *evSleep(-1)* to indicate a passive state where the sleep value should be considered as infinity. A coupled model is considered active if any of its sub-models are active. Once a coupled model sends an *evTick* to a sub-model, that sub-model is no longer considered active.

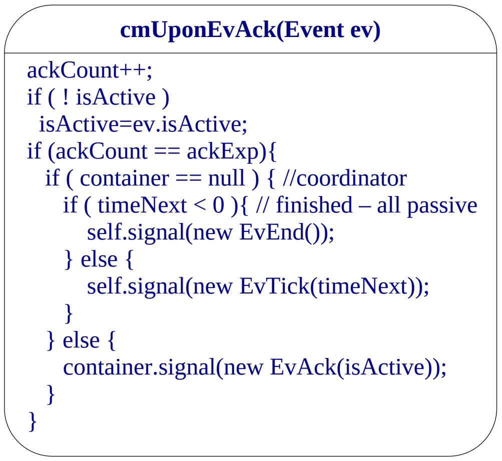
```
amsUponEvTick(Event ev)
-------------------------
if ( messageBag.isEmpty() ) {
  model.signal(ev);
} else { // n will always be zero
  model.signal(new EvMsg(messageBag));
  if ( e.isSleepExpired ){
    container.signal(new EvTick(0));
  }
}
```

Figure 8. Atomic Model Simulator *evTick* transition

When signaled to an atomic model *evTick* corresponds to invocation of the DEVS *internal transition function* – this signals the expiration of time intended to simulate the performance of a task as indicated by the preceding *evSleep* signal generated by the atomic model. This signal is issued by the atomic model simulator and sent to the atomic model when the atomic model simulator receives an *evTick* event.

5.5. evAck(*isActive)*

The *evAck* signal is generated by the atomic model when it receives an *evTick* signal and after it has completed generating any external *evMsg* messages.



```
cmUponEvAck(Event ev)

ackCount++;
if ( ! isActive )
  isActive=ev.isActive;
if (ackCount == ackExp){
   if ( container == null ) { //coordinator
      if ( timeNext < 0 ){ // finished – all passive
         self.signal(new EvEnd());
      } else {
         self.signal(new EvTick(timeNext));
      }
   } else {
      container.signal(new EvAck(isActive));
   }
}
```

Figure 9. Transition action for evAck in Coupled Model

The *evAck* message has one parameter, called *isActive,* which is always set to false in the case of an atomic model. For coupled models, the *evAck* signal is generated upon receipt of the final *evAck* from each of its sub-models and the *isActive* parameter is set to true depending upon whether there are any active sub-models.

5.6. Confluent Events

By default, any atomic models that have an expiring *evSleep*, and thus an imminent internal transition via an *evTick* event, receive that expiration notification simultaneously (during the first simulation cycle of the clock cycle). Since outputs are delivered as part of the next simulation cycle, the notion of a *confluent function* becomes a non-issue, except where a model issues an *evSleep(0)* during a simulation cycle, and that model is also sent messages during that same simulation cycle by another model. In such a case, there are confluent internal and external events. *Confluent events* are simultaneous events that occur during the same simulation cycle. Events occurring at the same time but occurring during *different* simulation cycles during the same clock cycle are not considered confluent events.

The precedence for confluent events is handled in the atomic model simulator. If external *evMsg* events are to be given precedence over internal *evTick* events then upon a *evTick(n,sleepExpired)* event where *sleepExpired* is true, an *evSleep(0)* should be issued and upon the next *evTick(0)* received any messages can be passed to the atomic model and then the *evTick* can be passed to the atomic model. If desired you can repeated signal *evSleep(0)* until there are no outstanding *evMsg* messages incoming for the atomic model and only then the *evTick()* message can be delivered to the atomic model.

6. UML MODELING RULES

Under DEVS-compliant UML, an instance of a class may participate in a model, if and only if, it is contained within an instance of *simulatable object* or is itself an instance of a *simulatable object.*

A *simulatable object* is one that has its behavior defined via a *compliant state machine*. This implies that all communication is essentially asynchronous insofar as replies to messages require an internal transition before a response is available. However, since *evSleep(n)* can be specified with zero time delay, the distinction is moot.

A *compliant state machine* has the following characteristics:

- Each event signaled must correspond to an output port name defined for the model.
- Each event received must correspond to an input port name defined for the model.
- Input and output port names must be disjoint.
- All time-dependent behavior is expressed via *evSleep(n)* signals and *evTick* events. There shall be no *after* or other UML time-related references in the state machine specification.
- Each atomic model state machine has an associated *atomic model simulator.*
- The state machine must only send external signals to *atomic model simulator*. A state machine may send signals to itself.
- The state machine must only receive external signals from *atomic model simulator*. A state machine may receive signals from itself.
- All signals sent to the *atomic model simulator* must occur upon receipt of an *evTick* event and before the *evTick* transition completes – this is the *output phase*. Such activity is defined in the action part of the specification of the transition triggered by the *evTick* event.
- An *evAck* signal must be generated and sent to the atomic model simulator upon completion of processing of an external event. All processing from receipt of the external event through the generation of the *evAck* signal must be atomic – it must run to completion.
- An *evSleep(n)* event signal is generated and sent to the *atomic model simulator* whenever the state machine wishes to simulate the amount of time required to complete some hypothetical processing, transmission time, or other such delay. This event is sent to the associated *atomic simulator* object. The *atomic simulator* will send an *evTick* event back to the state machine upon expiration of this time. The state machine remains dormant (*quiescent*) during this period or until it receives another external event (i.e. other than an *evTick* event).

- If an *evSleep(n)* event signal is generated, it *must immediately precede* the *evAck* signal generation.
- Upon receipt of an external event, an atomic model may issue a new *evSleep(n)* signal which supersedes any previously generated *evSleep(n)* signal event. Otherwise, any existing *evSleep(n)* will remain in effect. A duration of *-1* in an *evSleep(n)* signal indicates infinity or *passive* state. As such no *evTick* event will be issued to the atomic model and the state machine will be dormant until the next external event.
- All processing time involved in handling events during simulation is performed in *zero time* unless explicitly accounted for via *evSleep(n)* signals.
- All messages (event signals) are transmitted and received in *zero time* – the simulation clock is stopped. Likewise, all logic performed in the state machine is performed in *zero time*. Any requirement to model this time must be explicitly accounted for in the model via *evSleep(n)* signals.
- At any point there should only be one state in which simulated processing is ongoing as expressed via an *evSleep(n)* signal generation. This signal expresses the fact that there is processing that takes a certain amount of time, possibly zero, before it completes. Such states are, however, be *interruptible*. Note, in UML it is possible to have states in orthogonal regions within a state machine that may be performing actions simultaneously – for DEVS-compliant UML, the restriction is that only one state responds to an *evTick* event at any time.

7. CONCLUSIONS

Those unfamiliar with DEVS and more comfortable with the UML now have a convenient and relatively straightforward modeling approach by which their UML models can be executed and verified at an early stage of design. This approach has a wide generality in terms of the types of systems and problems to which it can be applied. The models produced are component-based, and given the closure property of DEVS, any component can be replaced by a different component with a greater degree of decomposition with the resulting system having an equivalent behavior. This modeling approach fits neatly within the modern iterative approach used to develop software systems. Modeling in this manner helps broaden and deepen the appreciation and application of simulation as a discipline within the field of software architecture.

Reference List or References

[1] Zeigler, B. P., H. Praehofer, and T. G. Kim. 2000. Theory of modeling and simulation: Integrating discrete event and continuous complex dynamic systems. 2nd ed. San Diego: Academic Press.

[2] OMG. 2007. *UML Profile for Schedulability, Performance, and Time*. http://www.omg.org.

[3] El Sheik, A., A. Al Ajeeli, and E. Abu-Taieh. 2008. *Simulation and modeling: Current technologies and applications*. New York: IGI Publishing.

[4] Douglass, B. P. 2004. *Real-time UML: Developing efficient objects for embedded systems*. 3rd ed. Boston: Addison-Wesley Longman Publishing Co., Inc.

[5] Hong, S.-Y. and T. G. Kim. 2004. Embedding UML Subset into Object-oriented DEVS Modeling Process, Proceedings of the Summer Computer Simulation Conference, San Jose, CA, July

[6] Kofman, E., M. Lapadula, and E. Pagliero. 2003. PowerDEVS: A DEVS–based environment for hybrid system modeling and simulation. *TR-LSD0306*.

[7] Ferayorni, A., and H. S. Sarjoughian. 2007. Domain driven modeling for simulation of software architectures. *Proceedings of the Summer Computer Simulation Conference*. San Diego, CA.

[8] OMG. 2005. *UML 2.0 Superstructure Specification*. http://www.omg.org/cgi-bin/doc?formal/2005-01-02

[9] Mellor, S. J., and M. J. Balcer. 2002. *Executable UML - A foundation for model-driven architecture*. Boston: Addison-Wesley Longman Publishing Co., Inc.

[10] OMG. 2003. *MDA Guide*. http://www.omg.org/cgi-bin/apps/doc?omg/03-06-01.pdf.

[11] Nikolaidou M., V. Dalakas, G. Kapos, L. Mitsi, and D. Anagnostopoulos. 2007. A UML2.0 profile for DEVS: Providing code generation capabilities for simulation. *Proceedings of Software Engineering and Data Engineering*, Las Vegas, NV.

[12] ACIMS. 2007. *DEVSJAVA*. http://www.acims.arizona.edu.

[13] Huang, D., and H. S. Sarjoughian. 2004. Software and simulation modeling for real-time software-intensive systems. *Proceedings of the 8th IEEE International Symposium on Distributed Simulation and Real-Time Applications*. Washington, DC.

[15]Risco-Martin, J. L., S. Mittal, B. P. Zeigler, and J. de la Cruz. 2007. From UML state charts to DEVS state machines using XML. *Proceedings of the IEEE/ACM International Conference on Model-Driven Engineering Languages and Systems*. Nashville, TN.

[14] Zinoviev, D. 2005. Mapping DEVS models onto UML models. *Proceedings of the DEVS Integrative M&S Symposium*, San Diego, CA.

[15] Schulz, S., T. C. Ewing, and J. W. Rozenblit. 2000. Discrete event system specification (DEVS) and statemate statecharts equivalence for embedded systems modeling, *Proceedings of the 7th IEEE International Conference and Workshop on the Engineering of Computer Based Systems*. Edinburgh.

[16] Mooney J. 2008. DEVS/UML - A Framework for Simulatable UML Models [M.S. thesis], Department of Computer Science and Engineering, Arizona State University, Tempe, AZ.