A Middleware Based Standard for DEVS Simulator Interoperability

James Nutaro
Arizona Center for Integrative Modeling and Simulation
Electrical & Computer Engr. Dept.
1230 E. Speedway Blvd.
University of Arizona
Tucson, AZ 85721-0104
(520)626-4846
nutaro@ece.arizona.edu

Keywords:
DEVS, simulation interoperability, simulation middleware

ABSTRACT: *The growing number of DEVS-based simulation tools has resulted in a need for standards to promote tool interoperability. This paper presents the skeleton of an interoperability standard for DEVS-based simulations. The aim of the standard is to provide services for coordinating the execution of simulation components, where simulation components encapsulate particular implementations of the DEVS abstract simulators. The standard allows individual components to be realized using event stepped and risk-free optimistic algorithms. In addition to time managed coordination of DEVS simulators, the proposed services can be implemented on top of existing real-time distributed simulation protocols.*

1. Introduction

This paper proposes a middleware based approach to interoperability between DEVS-based simulation tools (see [Zeigler 2000] for an extensive introduction to DEVS and its abstract simulator concepts). A small collection of services, implemented as middleware, can allow DEVS models that have been implemented with different simulation tools to be composed into a coherent whole. This can benefit large scale modeling efforts, where different model components are more easily realized in different simulation environments. It also removes one barrier to reuse when candidate components have been implemented in diverse environments.

The proposed approach is similar to the High Level Architecture (HLA) [IEEE 2000]. However, the aim of the proposed standard is to allow DEVS simulation engines to interoperate across operating system, programming language, and design boundaries. This distinguishes it from the HLA, which does not directly support interoperability between constructive, DEVS-based simulations (see, e.g., [Sarjoughian 2000], [Lake 2000], and [Zeigler 1999]). The services discussed in this paper could be implemented as part of a standalone middleware system, or they could be attached to an existing standard (e.g., the HLA) in order to provide a more feature rich simulation interoperability tool.

The need for a capability to interconnect various DEVS-based simulation tools is indicated by current industry trends in simulation software development, and the proliferation of DEVS-based simulation tools (see, e.g., the list of tools at www.sce.carleton.ca/faculty/wainer/standard/tools.htm). Consider, for instance, two of the largest users of component-based simulation technology; the Department of Defense (DoD) and the computer game industry. Within the DoD modeling and simulation community, the use of component-based simulation technology is driven by a desire to reuse existing simulations in new applications. One of the technical barriers to effective model reuse in the DoD domain is not addressed by this, or any other, standard. Namely, that models described using different world views do not always interact as intended. However, even when modeling paradigms are compatible, it is not uncommon for components to be developed using different languages and on different computing platforms. In the absence of component-based simulation tools, integration requires that the code be re-implemented in an appropriate language, ported to a new environment, or both. This results in duplicated functionality (i.e., there are now two copies of essentially the same model) with an associated rise in maintenance costs, introduction of new errors in the ported code, and a corresponding loss of confidence in the model.

In the computer gaming industry, reuse of legacy software is not the primary cause of multi-language and multi-

platform simulation development. Instead, proliferation is the result of the diverse teams and short deadlines that are intrinsic to commercial game development (see, e.g., the discussion in [Phelps 2004]). For instance, it is not unusually for a computer game to have a "physics engine" implemented in C or C++, while its "AI engine" is implemented in Lisp, Prolog, or some other symbol processing language. This is done because integrating two different programming languages is less demanding than developing, for instance, complex artificial intelligence functions in C.

Large scale modeling and simulation efforts encounter both of these issues, even when the modeling and simulation problem is formulated entirely within the DEVS framework. For example, suppose that we want to model a bat that is hunting for insects. The model has three primary pieces. These are the bat, the insects, and the acoustic phenomena that allows the bat to find the insects. The bat is likely to be modeled as an intelligent agent that can generate and react to sounds. It is natural to implement the bat and insect models as intelligent agents. This could be done using, for instance, the IDEVS simulation environment which was designed expressly for this purpose [El-Osery 2000]. The acoustic effects could be simulated as propagating waves in a large 3D cell space. This model might be implemented in adevs, which has been used to represent continuous processes as discrete event systems [Nutaro 2003]. It is also possible that one or both of the models already exist in their respective simulation environments.

In the absence of a component-based simulation framework, it would be necessary to select a single simulation environment in which to develop the entire model. However, since the modeling concepts that underlay both environments are identical, it is only implementation specific items that prevent (technical) interoperability. If the proposed standard were in place, it would allow the development of each model component (i.e., bat, insect, and acoustics) using a suitable simulation engine. The disparate simulators could then be integrated into a coherent whole that operates as expected.

2. The Simulation Protocol

The protocol is described as a set of middleware services. It is assumed that these services are implemented as a middleware system with support for several popular programming languages (e.g., Java, C, C++, Python) and operating systems (e.g., Windows, Linux, Macintosh). The middleware implementation might support distributed simulation in order to fulfill the objective of cross platform interoperability. However, distributed simulation is not the primary goal of the standard.

It also assumed that the target simulation engine exports an interface that allows it to be driven externally. The essential requirements are that the simulation can be asked for its time of next event and output events, and that it can be told when to compute the model's state transition function. An example of such an interface specification is described in [Park 2001]. Table 1 describes the simulator interface that is exported by the adevs and DEVSJAVA simulation engines (these simulators are available from www.ece.arizona.edu/~nutaro and www.acims.arizona.edu/software/SOFTWARE.shtml respectively). Both of these examples are based on [Park 2001]. Note that the proposed standard does not dictate the form of the exported interface. It only assumes some such interface exists, and that it can be used as the basis for integrating the simulation engine with the middleware.

Adevs methods	***Method description***
inject(const PortValue& x) inject(const adevs_bag<PortValue>& x)	Inject a single event, or a bag of events, that will be applied at the next call to deltFunc().
ADEVS_TIME_TYPE nextTN()	Returns the time of next event.
const adevs_bag<PortValue>& computeInputOutput()	Returns the output that will be produced by the simulator at timeNext(), assuming there are no inputs in the interim.
void deltFunc (ADEVS_TIME_TYPE t)	Compute that state of the model at time t. The argument must be less than or equal timeNext().
DEVSJAVA methods	
simInject(double e, MessageInterface m) simInject(double e, String portName, entity value)	Inject a single event or a bag of events and compute the next system state after an elapsed time e.
Double nextTNDouble() double nextTN() double getTN()	Returns the time of next event.
wrapDeltfunc(double t) wrapDeltfunc(double t, MessageInterface x)	Compute the system state at time t using an optional bag of events that should be applied at that time.

Table 1. Description of the interfaces exported by the adevs and DEVSJAVA simulation engines.

Similar to the HLA, the protocol is described as a set of services that can be used by simulation integrators to create working simulation systems. While this suggests a need for services that cover component management, interest management, and other aspects of large scale and distributed software development, these issues are beyond the scope of this proposal. The proposed standard could, however, be attached to a standard, such as the HLA, to provide a more feature rich DEVS simulation interoperability tool.

The standard contains only four fundamental services. These services are

```
startSimCycle(lvt: time): time
endSimCycle()
generateEvents(events: bag)
receiveEvents(): bag
```

The startSimCycle() service begins a simulation cycle by computing the global virtual time using the local virtual time provided as the lvt argument. The startSimCycle() service returns the global virtual time after all components have made the startSimCycle() call. In effect, startSimCycle() acts as a reduction operation that computes the global time of next event (see, e.g., [Fujimoto 2000]).

The generateEvents() service is used to distribute output events that are valid at the global virtual time. All of the output produced by the component at the current global virtual time are provided to the generateEvents() service. The generateEvents() service is invoked once per simulation cycle. An empty bag is used to indicate that the component has no output at the current time.

The receiveEvents() service is used to gather all of the input events that are to be applied to the component in the current simulation cycle. The resulting bag of events is comprised of events provided by components via the generateEvents() service. An empty bag is used to indicate that no input is available for the component in the current simulation cycle.

The endSimCycle() service is used to indicate that a component is ready to proceed to the next simulation cycle. This service allows to middleware to do any special processing that is required to advance the simulation to the next cycle.

This set of services is sufficient to allow any kind of DEVS simulation engine that does not require the release of optimistically produced output or that relies on a the existence of a positive lookahead value (see [Nutaro 2003a] for examples of these and other types of DEVS simulation algorithms). For a typical, event stepped implementation of the DEVS abstract simulator, the integration of the simulation engine with the above standard might be realized as shown below. The psuedo-code assumes that there is a simulator object, denoted sim, that exports an interface similar to those shown in Table 1.

```
do
    gvt := startSimCycle(sim.nextTN())
    if (gvt = sim.nextTN() and gvt < ∞)
        generateEvents(sim.getOutput())
    end
    inputs := receiveEvents()
    if (inputs is not empty or (gvt = sim.nextTN() and
    gvt < ∞)
        sim.computeNextState(gvt,inputs)
    end
    endSimCycle()
while(gvt < ∞)
```

The standard can also support risk-free optimistic simulation. One possible implementation of a risk-free optimistic simulation protocol is shown below. The state saving and rollback functions of the simulator are assumed to be built into the computeNextState() method. The simulator is further assumed to have a collectGarbage (gvt) method that cleans up all save states with time tags less than the gvt argument.

```
do
    while (sim.getOutput() is empty and
    sim.nextTN() < ∞)
        sim.computeNextState(sim.nextTN())
    end
    gvt := startSimCycle(sim.nextTN())
    if (gvt = sim.nextTN())
        generateEvents(sim.getOutput())
    else
        generateEvents(empty event bag)
    end
    inputs := receiveEvents()
    if (inputs is not empty)
        sim.computeNextState(gvt,inputs)
    end
    sim.collectGarbage(gvt)
    endSimCycle()
while (gvt < ∞)
```

This algorithm could be pushed down into the simulator's exported interface, thereby allowing the simulator/middleware interface code to appear as in the event stepped case. Doing so would require that the computeNextState() method move forward until the next output is found, and that the timeNext() method return the event time of this next output. The computeNextState() method would continue to be responsible for state staving and rollbacks. Garbage collection could be performed at the beginning of each call to computeNextState(), using the supplied time argument as the global virtual time.

The protocol excludes the use of optimistic algorithms and conservative (i.e., lookahead-based) algorithms. Optimistic algorithms are excluded because the standard does not include a service for canceling events provided

to the generateEvents() service. Conservative algorithms are not supported because there is no provision for providing a lookahead value to the global virtual time procedure (i.e., the startSimCycle() service).

The exclusion of optimistic algorithms can be justified by their complexity and relative rarity. The goal of the standard is to promote interoperability of existing DEVS simulation engines, and not to enable high performance computing. The vast majority of these existing simulation tools are event-stepped implementations. A handful of risk-free optimistic simulators have been presented in the literature. Fully optimistic DEVS simulators, while they have been constructed, do not see significant use in practice.

The exclusion of lookahead based-algorithms can be justified by the fact that large DEVS models do not, in general, have non-zero lookahead. Primarily, this is because the time advance function is allowed to take a value of zero. Even when individual components of a coupled model have strictly non-zero time advance functions, the corresponding time advance of the coupled model can be arbitrarily close to zero. A zero time advance implies that an input can result in an immediate output and so the model has a zero lookahead.

Support for risk-free optimistic simulation can be justified by two arguments. First, components with large computational demands will require the use of parallel algorithms and computers. The use of conservative algorithms is, in general, prohibited by the absence of lookahead in DEVS models. Event stepped parallel algorithms, which compute only simultaneous events in parallel, can be readily accommodated. However, risk-free optimistic simulation algorithms often scale more effectively with machine size (see, e.g., [Nutaro 2001]), thereby making them more useful for very large scale problems.

Secondly, risk-free optimistic algorithms can be readily accommodated within the proposed framework. The only outstanding issue how to handle instantaneous and simultaneous events. Since the simulation problem is restricted to DEVS models, this problem has a relatively simple solution which is expounded upon in the next section.

3. Managing Zero Time and Simultaneous Events

If risk-free algorithms are going to be accommodated by the standard, it is necessary to have a means by which simultaneous events and sequences of zero time events can be properly ordered. What constitutes a proper ordering, in the context of DEVS simulations, is described in detail in [Nutaro 2003a]. Less formally, it is required that inputs always precede outputs. This is not a problem of time ordering, since it is possible to have (finite) sequences of zero time events occur within a simulation cycle. Rather, the issue is how to embed information about causal sequences of events into the simulation time stamping scheme.

A simple solution to this problem is to include a second field field in the simulation time stamp for an event. When a sequence of zero time events occurs, the second field is used to order events according to the simulation cycle in which they took place. Simultaneous events are those events which have equal simulation times (i.e., their first fields match) and equal values in the second field. An algorithm that implements a suitable, two field, simulation clock is described in [Nutaro 2003a] and [Rönngren 1999]. The time stamps generated by this algorithm are used as the definition of time within the context of the standard. In this way, the standard is able to accommodate risk-free optimistic simulations of DEVS models.

The algorithm assigns time stamps to events in the following way. A time stamp is a pair (t,c) where t is a model derived time-stamp (i.e., the time associated with an event) and c is an integer counter. The simulator maintains a time of last event (tL,cL) whose initial value is (0,0). When a model executes an event at model time t, the simulator compares that t to tL. If t = tL, then the simulation time of next event becomes (t,cL+1). Note that the model event time is still t. If tL < t, then the simulation time of next event becomes (t,0). The time of last event is then set to be the new time. There are two rules for comparing time stamps. These are

$(t_1,c_1) < (t_2,c_2)$ if $t_1 < t_2$ or $t_1 = t_2$ and $c_1 < c_2$, and
$(t_1,c_1) = (t_2,c_2)$ if $t_1 = t_2$ and $c_1 = c_2$.

Where the standard requires that time be strictly increasing, the corresponding time order of events is is given by the < relation defined above.

4. Real-time Simulation

The preceding discussion assumes that final simulation system is mean to be used constructively. It is frequently the case that a DEVS simulator will be called on to form a component in a real-time system (e.g., for training or test and evaluation purposes). The proposed services can be readily adapted for this type of use by linking the global virtual time to a real-time clock. A real-time implementation of the standard might appear as follows.

```
when an event is received from the network
    place the events on the input queue
    signal that inputs are available

when startSimCycle(time) called
    wait until the real time clock reaches time or input
        becomes available
    return the current time
```

```
when generateEvents(events) called
    send all of the events to the network

when receiveEvents() called
    return all the events in the input queue and
    empty the input queue

when endSimCycle() called
    do any necessary cleanup
```

The basic standard could be extended to include a timing mode switch that could be set to indicate real time or logical time (constructive) execution, as needed. The real time mode could be implemented as layer above some other underlying network simulation protocol (e.g., the HLA). This would enable DEVS simulation engines that adhere to the proposed standard to be used in existing real-time, distributed simulations.

5. Conclusions

The proposed standard is small, but sufficient to achieve its primary goal. A small standard has two benefits. First, it permits the rapid development of middleware to support the standard. Second, it requires less effort to integrate the standard into existing systems. While the idea of attaching the proposed standard to a larger, more feature rich middleware standard is attractive, this should be done in such a way that it does not prevent the core services from standing on their own.

The proposed standard is deficient in at least three areas. First, it does not specify basic services for describing connections between components (e.g., as is done with the HLA publish/subscribe services). The specification of component connectivity services needs to be compatible with any existing standards that the proposed standard will be attached to. The extent to which this can be done while maintaining a complete set of core services remains to be seen.

A second deficiency is that the standard does specify how failures (e.g., in the network for distributed implementations, or via violations of the service interface specification by components) are to be handled. A carefully considered and standardized failure detection and response mechanism is essential if systems employing the standard are going to be robust.

Lastly, there are pragmatic issues that need to be addressed. These include API specifications (i.e., "language bindings") for programming languages, message format standards for cross platform interoperability, and run-time interoperability between different middleware implementations.

These issues can be best resolved in the context of prototype middleware implementations and small scale, but practical, applications. The standard can evolve as prototypes are built, used, and revised. This evolutionary standards development process should be facilitated by regular discussion between prototype developers, simulation engine designers, and simulation users.

6. References

[IEEE 2000] IEEE Std. 1516-2000. "Standard for modeling and simulation (M&S) high level architecture (HLA) – framework and rules". IEEE, 2000.

[El-Osery 2002] A. El-Osery, J. Burge, M. Jamshidi, A. Saha, M. Fathi, M. Akbarzadeh. "V-Lab – A Distributed Simulation and Modeling Environment for Robotic Agents – SLA-Based Learning Controllers." IEEE Transactions on Systems, Man, and Cybernetics – Part B, Vol. 32, No. 6, pp. 791-803, 2002.

[Fujimoto 2000] Fujimoto, R.M. *Parallel and Distributed Simulation Systems.* John Wiley and Sons, Inc. 2000.

[Lake 2000] T. Lake, B.P. Zeigler, H.S. Sarjoughian, J. Nutaro. "DEVS Simulation and HLA Lookahead". Spring Simulation Interoperability Workshop, 2000.

[Nutaro 2001] J. Nutaro, H. Sarjoughian. "Speedup of a Sparse System Simulation". 15th Workshop on Parallel and Ditributed Simulation, 2001.

[Nutaro 2003] J. Nutaro, B.P. Zeigler, R. Jammalamadaka, S. Akerkar. "Discrete Event Solution of Gas Dynamics within the DEVS Framework". International Conference on Computational Science, pp. 319-328, 2003.

[Nutaro 2003a] Nutaro, .J.J., "Parallel Discrete Event Simulation with Application to Continuous Systems", Dissertation, University of Arizona, Department of Electrical and Computer Engineering, 2003.

[Park 2001] S. Park, B.P. Zeigler, H.S. Sarjoughian. "Interface for Scalable DEVS and distributed container object specifications". IEEE Conference on Systems, Man, and Cybernetics, pp. 3075-3080, Vol. 5, 2001.

[Phelps 2004] A.M. Phelps. "Fun and Games with Multi-Languaged Development". ACM Queue, Vol. 1, No. 10, 2004.

[Rönngren 1999] Rönngren, R., M. Liljenstam. "On Event Ordering in Parallel Discrete Event Simulation". Thirteenth Workshop on Parallel and Distributed Simulation, pp 38-45, 1999.

[Sarjoughian 2000] H.S. Sarjoughian, B.P. Zeigler. "DEVS and HLA: Complimentary Paragimgs for M&S?". Transactions of the Society for Computer Simulation, Vol. 17, No. 4, pp. 187-197, 2000.

[Zeigler 1999] B.P. Zeigler. “Implementation of the DEVS Formalism over the HLA/RTI: Problems and Solutions”. Spring Simulation Interoperability Workshop, 1999.

[Zeigler 2000] Zeigler, Bernard P., Herbert Praehofer, Tag Gon Kim. *Theory of Modeling and Simulation: Second Edition*. San Diego: Academic Press, 2000.