Implementation of the DEVS Formalism over the HLA/RTI: Problems and Solutions

Bernard P. Zeigler
George Ball
Hyup Cho
J.S. Lee
Hessam Sarjoughian

AI and Simulation Group
Department of Electrical and Computer Engineering
University of Arizona,
Tucson, Arizona
{zeigler,sarjoughian,hyup,jslee}@ece.arizona.edu
gball@srnr.arizona.edu

Keywords:
DEVS formalism, HLA, layered architecture, Runtime Infrastructure

ABSTRACT: This paper describes the design and development of DEVS/HLA, an HLA-compliant modeling and simulation environment that supports high level model building using the DEVS (Discrete Event System Specification) methodology, a sound formal M&S framework. After a brief review of the DEVS formalism, we discuss the implementation of the formalism in C++ as a layer above the Runtime Infrastructure (RTI) released by the Defense Modeling Simulation Organization (DMSO) to implement the HLA standard. This leads to a consideration of alternatives for implementing the DEVS protocol which allow for efficient event-based simulation, maximally exploit HLA primitives, and support the full range of DEVS expressive power, which has been shown to subsume all discrete event system behaviors. Direct mapping of the DEVS protocol incurs a mismatch with the HLA specification. We present an alternative solution involving off-loading some of the time-management on a coordinator federate.

1. Introduction

DEVS/HLA is an HLA-compliant modeling and simulation environment formed by mapping the DEVS-C++ system [1] to the C++ version of the DMSO RTI. While HLA supports interoperability at the simulation level, DEVS/HLA supports modeling level features inherited from DEVS – as a generic dynamic systems formalism with a well defined concept of modularity and coupling of components. DEVS/HLA support for hierarchical, modular construction, discrete event approximation of continuous systems and repository reuse are discussed in other papers in this proceedings [2-4]. Design and implementation of the DEVS/HLA environment and its support of predictive filtering is discussed in detail elsewhere [5]. This paper focuses on implementation issues that arose in implementing the DEVS simulation protocol which allows for efficient event-based simulation, maximally exploit HLA primitives, and supports the full range of DEVS expressive power, which has been shown to subsume all discrete event system behaviors [6]. Direct mapping of the DEVS protocol does not match up with the HLA specification. We discuss such a direct mapping and show where the problems arise in such an approach. Then present an alternative solution involving off-loading some of the time-management on a coordinator federate.

2. Overview of HLA and DEVS/HLA

The operational form of the HLA is a Run Time Infrastructure (RTI) consisting of a centralized executive and node ambassadors (software equivalents of network...
interface cards) that support communication among simulations, called federates. DMSO (Defense Modeling and Simulation Office) has developed an RTI in C++ (currently version 1.3) for use in the public domain. HLA supports a number of features including establishing, joining and quitting federations, time management and inter-federate communication. DEVS/HLA federates communicate through the standard DEVS interfaces of the DEVS models they contain. In this manner, DEVS/HLA can supply an HLA-compliant layer in which to formulate DEVS models and modeling formalisms. The key concept of is that of modularity in which component models are coupled together through distinguishable input/output interfaces. This allows messages to be sent from one federate to another using the underlying HLA message transmission facilities called “interactions”. In addition to communication through interactions, HLA also supports attribute updating through objects declared for this purpose. DEVS/HLA facilitates such attribute updating by supporting quantization-based filtering [3]. For this, the modeler declares HLA objects and attributes desired for updating states of DEVS federates. By attaching quantizer objects supplied by DEVS/HLA to such attributes, the appropriate publish/subscribe mechanisms are automatically set-up. The resulting environment for high level model specification and simulation is depicted in Figure 1. Models developed in a DEVS/C++ can be directly simulated in the DEVS/HLA environment over any TCP/IP, ATM, or other network of hosts executing an HLA C++ RTI. Based on model-supplied information, DEVS/HLA takes care of the declarations and initializations needed to create federations, joining and resigning of federates, communication among federates and time management. This applies, as well to any DEVS-compliant formalism that can be automatically translated into DEVS-C++ code.

Figure 1 DEVS/HLA Supporting High Level Model Specification

3. Brief Review of DEVS Formalism

The structure of a model may be expressed in a mathematical language called a formalism. The DEVS (Discrete Event System Specification) formalism focuses on the changes of variable values and generates time segments that are piecewise constant. In essence the formalism defines how to generate new values for variables and the times the new values should take effect. An important aspect of the formalism is that the time intervals between event occurrences are variable. DEVS-C++ [1] was developed based on DEVS formalism. There are two major classes from which all user-defined models can be developed – atomic and coupled. The atomic class realizes the basic level of the DEVS formalism, while the coupled model embodies DEVS hierarchical model composition constructs [Zeigler, 1998 #174; Zeigler, 1990 #32; Zeigler, 1996 #50]. We provide a brief review of the DEVS formalism for these classes before discussing their implementation in DEVS-C++.

3.1 DEVS Formalism for Basic Models

A DEVS basic model is a structure:

\[ M = \langle X, S, Y, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta \rangle \]

where:

- \( X \) : set of external input events;
- \( S \) : a set of sequential states;
- \( Y \) : a set of outputs;
- \( \delta_{int} : S \rightarrow S \) : internal transition function
- \( \delta_{ext} : Q \times X^b \rightarrow S \) : external transition function
- \( \lambda : S \rightarrow Y^b \) : output function generating external events at the output;
- \( ta : S \rightarrow \text{Real} \) : time advance function;

\( e \) is the elapsed time since last state transition.

To anticipate later discussion, DEVS basic models are implemented as the class atomic models in DEVS-C++.

3.2 DEVS Formalism for Coupled Models

Two major activities involved in coupled models are specifying its component models, and defining the coupling, which creates the desired communication links. A coupled model is specified by a structure:
\[ DN = <X, Y, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}> \]

Where:

- \( X \) : set of external input events;
- \( Y \) : a set of outputs;
- \( D \) is a set of components names;
- for each \( i \) in \( D \), \( M_i \) is a component model
- \( I_i \) is the set of influencees for \( i \)
- for each \( j \) in \( I_i \), \( Z_{i,j} \) is the \( i \)-to-\( j \) output translation function.

DEVS coupled models are implemented as the class `coupled` models in DEVS-C++.

A `coupled` model instance contains the following information:

- the set of components
- the set of input ports through which external events are received
- the set of output ports through which external events are sent
- the coupling specification consisting of:
  - the external input coupling connects the input ports of the coupled to one or more of the input ports of the components
  - the external output coupling connects the output ports of the components to one or more of the output ports of the `coupled` model
  - internal coupling connects output ports of components to input ports of other components

A more complete review of the DEVS formalism is provided in [7].

### 4. Implementation of DEVS Formalism

DEVS is most naturally implemented in computational form in an object-oriented framework. A generic class hierarchy for such implementation is given in Figure 2. The basic class is `entity` from which class `devs` is derived. `Devs` in turn is specialized into classes `atomic` and `coupled`. `Atomic` enables models to be expressed directly in the basic DEVS formalism. `Coupled` supports construction of models by coupling together components. Class `content` is derived from `entity` to carry `ports` and their `values`, where the latter can be any instance of `entity` or its derived classes. Typically a modeler defines such derived classes to structure the information being exchanged among models. The outputs of component models are instances of class `message`, which are containers of `content` instances.

![Figure 2](image-url)

**Figure 2 Basic Class Hierarchy for DEVS/HLA Implementation**

The `devs` class is the basic class to provide methods for the DEVS formalism. Such functions as \( \delta_{ext} \), \( \delta_{int} \), and \( \lambda \) are implemented as virtual methods to be defined by user. The method for time advance \((ta)\) and port information is also provided by this class. The destination for sending output message is determined by coupling methods which manipulate coupling information. An `atomicbase` class realizes the atomic level of the underlying model formalism. It has variables corresponding to each of the parts of this formalism: internal transition function \( (\delta_{int}) \), external transition function \( (\delta_{ext}) \), confluent function \( (\delta_{con}) \), output \( (\lambda) \) function, and time advance function \( (ta) \). These methods are applied to the state of the model. The `atomic` class has additional features for convenient model construction and testing.

The `coupled` class is the major class to support the hierarchical model composition constructs of the DEVS formalism. The `digraph`, `block` and `digraphcell` are specializations which enable specification of `coupled` models in specific ways.

#### 4.1 Parallel DEVS Simulation Protocol

The separation of concerns between modeling and simulation underlying the DEVS methodology is illustrated in Figure 1. Parallel and Distributed Discrete Event Simulation (PDES) protocols make it possible to execute discrete event models in networked computing environments. They take care of the synchronization and data communication needed to correctly executed models. Such protocols usually assume a simplified abstraction of the models they are simulating, based on the concept of logical processors. On the other hand, DEVS is a modeling formalism that supports development of discrete event models. Thus, to execute
DEVS models requires that they be mapped into the simulation protocols that control the underlying computation[8, 9].

There are three approaches to mapping the DEVS formalism into PDES protocols. Two of these approaches are specializations of the more generic protocols, called conservative and optimistic schemes. The third is the direct mapping into a tailored simulation algorithm, called the Parallel DEVS protocol. In the conservative and optimistic schemes simulation is viewed as moved forward by the processing of time-stamped messages. In the third, direct approach, simulators are designed based directly on the DEVS formalism. The Parallel DEVS protocol can be viewed as an extreme form of risk-free optimism (not even local rollback occurs) and does not incur the overheads of conservative and optimistic schemes. Instead of trying to overlap processing of input events with different time-stamps, it seeks to exploit parallelism in the simultaneous occurrence of internal events (hence with the same, or close, time stamps) among many components.

As shown in Figure 3, the Parallel DEVS scheme differs from the LP-based schemes in that there is a coordinator to synchronize the simulation cycle through its steps. The coordinator, \( C \) collects all times of next event from the component simulators. It sends the minimum of these times back to the components, thereby allowing them to determine whether they are imminent, and if so to generate output. More than one components may be imminent and the outputs of all such imminents are sorted and distributed to others according to the coupling rules. The transition functions of the imminent components, as well as all other recipients of inputs, are applied. Which transition is applied, depends on the state and input of a component – imminents with no inputs apply internal transition functions, imminents with inputs apply confluent transition functions, and non-imminent components with input apply external transition functions. The resulting changes in states may cause new values for time advances and these are sent to the coordinator. The cycle continues.

4.2 The DEVS/HLA Environment

The HLA is comprised of three elements: rules for federates and federations, the runtime interface specification, and the object model template (OMT) [10]. HLA rules govern how to achieve proper interaction of simulations in a federation and describe the responsibilities of simulations and of the data distribution mechanism called the runtime infrastructure (RTI). The HLA interface specification provides the definition of interface functions between the runtime infrastructure and federated simulations. The object model template is a format for recording the information required by the HLA Object Model for a federation and its federates.

In the terminology of HLA [10], a federate is one simulation acting as a member of a federation, while the federation is a set of federates intended to work together for a composite simulation. An object handled by the RTI is of interest to more than one federate and can have more than one attribute.

HLA provides two types of communications: attribute updating between an object in one federate and another object in a second federate, and interaction communicating between two federates. An interaction is a non-persistent, time-tagged event generated by one federate and received by others through RTI. Interactions can have parameters similar to the attributes of objects.

4.3 Mapping DEVS to HLA

The strategy underlying the mapping of DEVS to HLA is to exploit the information contained within DEVS models to automate as much as possible of the programming work required for constructing HLA compliant simulations. For example, DEVS coupling specifications are automatically mapped into HLA interactions. Additionally, we seek to minimize any additional declarations to those that are minimally necessary for HLA operation. Thus, while modelers have to declare quantizers, objects and attributes corresponding to which objects and attributes they wish published and subscribed to, they are freed from the task of writing the methods that will mesh with the RTI commands that enable such attributes to be published and subscribed to.

Classes for the interface between the DEVS and HLA are shown in Figure 2. The DevsFedAmb class provides
interface methods between DEVS and HLA that employ the methods of the `FederateAmbassador` class, which provides primitives to access HLA services. The `devsHLA` library contains the `attribute` and the `quantizer` classes shown in Figure 4. These are used for attribute updates, while the `HLAport` class is used for interactions. A quantizer object is associated with each attribute that the modeler would like to publish. A quantizer is a demon that checks for a threshold crossing of the attribute value. Since attribute value changes can only occur in the execution of model transition functions, it is straightforward to activate such demons. Quantizers are used to reduce message update traffic—the trade-offs between accuracy and speed of computation involved are the subject of the research reported elsewhere in this proceedings [3]. The Appendix describes how DEVS outputs ports and messages are encoded and decoded as HLA interactions.

4.4 Implementation of DEVS Simulation Protocol in HLA

HLA provides two orthogonal styles of time dependence that each federate can independently assume in a federation: `time regulating` (the federate sends out time-stamped events (updates, interactions)) or `time constrained` (the federate agrees to work under the requirement to process input events in the time-stamped order—whether conservatively or optimistically). The default setting in DEVS/HLA is for all federates to be time regulating and constrained as is appropriate for a simulation that runs in logical time (uncorrelated to wall clock time).

Now the logical processor paradigm underlies the time management philosophy of HLA in the time regulating/time constrained mode. The overall modular component concept supported by HLA and the logical processor paradigm are largely compatible with the DEVS modular formalism so that implementation is mostly straightforward. However, implementing the DEVS formalism is constrained by the expectations of the logical processor paradigm that deal only with external time stamped events. One approach is to employ the DEVS versions of the PDES algorithms mentioned above which incorporate DEVS internal event concepts explicitly. Another is to implement the direct Parallel DEVS abstract simulator as faithfully as possible within the constraints of the HLA specification. In our initial implementation, we have chosen the latter approach, and moreover, a particular design for the latter approach that has simplicity as its greatest virtue. We will present the design and indicate problems with its implementation.

Figure 4 presents the logic of the direct implementation. Each federate starts by using its time advance function to compute its time-of-next-event, tN, and sending the latter to the RTI in the form of a next EventRequest with cutoff parameter = tN. If there are no events in the interim, the RTI issues a timeAdvGrant when the time specified in the cutoff parameter arrives. Since there are no events in the system until the minimum of the tNs, the RTI moves time forward to that minimum, which becomes the current time for the federation. The imminents (which have the minimum tN) are precisely the components which are sent the timeAdvanceGrant. When an imminent receives a timeAdvGrant, it computes its output function and sends its output messages, in the form of interactions, to the RTI for distribution to the receiving federates. The DEVS protocol then requires that an imminent wait for any interactions to arrive before it issues a new nextEventRequest. We note that the logic presented abstracts from the actual code which must adhere to other restrictions such as concurrent access constraints imposed by the RTI. Three considerations enter into the implementation.

When receive `nextEventRequest`

When receive `timeAdvanceGrant`

When receive `interaction`

When receive `updateVariables`
(granule in Figure 4) from that of the time advance grant. The time granule size is a parameter to be set and determines the equivalence of times for components to be considered imminent\(^2\) [1].

4.5 Handling Cancellation in DEVS with NER

As a consequence of a change in state of a DEVS federate when it receives an external event, its time-of-next-event may now be different necessitating cancellation of the current time advance and issuing of a new one with a different time. The NextEventRequest service is intended to handle this eventuality. However, we need to provide some background to discuss this issue.

In issuing a time advance, a federate undertakes what we will call the time advance promise (P):

By issuing a time advance request to logical time T, a federate promises that it will issue no TSO messages with time stamps less than T.

We quote the HLA draft specification [12] on time management for what we will refer to as the time advance requirement (R):

“The RTI shall grant an advance to logical time T only when it can guarantee that all TSO messages with time stamps less than T (or in some cases less than or equal to T) have been delivered to the federate. This guarantee enables the federate to simulate the behavior of the entities it represents up to logical time T without concern for receiving new events with time stamps less than T.

Note that in some cases, providing this guarantee will require the RTI to wait for a significant period of wall-clock time to elapse before it can grant a time advancement to a time-constrained federate.”

Thus the RTI can guarantee that no TSO messages are pending for a federate with times earlier than T, and hence consider a time advance to T, if no other federates have time advance requests for times that are earlier than T. However, if after requesting a time advance to T, a federate may subsequently receive a TSO message with a time less than T, which may cause it to “change its mind” about its time advance. In this case, the Next Event Request (NER) service helps to deal with the situation. If a federate issues a NER (rather than a time advance request) with a time advance bound of T, the RTI reduces the advance to the time of the earliest message subsequently delivered to the federate, thus enabling it to issue a new NER.

Now let’s consider an example to see how this might work in the DEVS case. Initially, based on their DEVS model time advance values, Federates F1 and F2 issue next event requests with upper bounds t1 and t2, respectively (Figure 5a). Assuming that t1 is earlier than t2, the RTI will provide a time advance grant (TAG) to F1 (Figure 5b). F1 sends an interaction (DEVS external event) to F2 with time stamp t1+g, where g is the granule time. Upon receiving the interaction, F2 applies its external transition function which results in a new time advance for its model. As in Figure 5c, F2 must issue a NER for the associated time, t2'. However, due to the time requirement, R, the RTI cannot consider this request until it receives a NER from F1. This is because in general, a federate such as F1, could send multiple interactions with time arbitrary time stamp, and the RTI cannot know when this federate has completed its output until it issues a NER (or time advance request) with time greater than t1+g. After receiving the NER from F1 (an imminent), the RTI can entertain an NER from F2 (a non-imminent recipient).

We can summarize this requirement (W): an imminent must issue its NER after receiving its TAG and sending its interactions; a non-imminent must wait for the TAG that the RTI sends after all imminents have issued their NERs.

![Figure 5 HLA mismatch with DEVS](image)

However, now what happens in the case of imminents that also receive inputs (from other imminents)? The DEVS protocol requires that an imminent wait for its inputs before applying its (confluent) transition function and getting the time advance for its next NER. However, this contradicts the above requirement, W, that it issue an NER right after sending its outputs. This argument shows...
that the direct approach to mapping the DEVS protocol cannot render a correct, full capability, implementation.

4.6 Alternative Implementation

An alternative implementation of the DEVS protocol was developed that can correctly realize the simulation cycle. This approach employs the mapping discussed in Figure 4 as a lower layer to support a layer in which the coordination approach in Figure 3 is implemented. In other words, if we do not require cancellation of time advances, the implementation just discussed works in a perfectly deterministic fashion. Indeed, time advance, rather than next event, request service is sufficient in this case. Of course, many discrete systems can not then be directly simulated, specifically, any model with conditionally timed behavior [4]. The subset of systems that can be simulated in this manner, called DEVS without cancellation, can be easily characterized as suggested in Figure 6.

To achieve full DEVS capability, i.e., DEVS with cancellation, we implemented the DEVS simulation protocol with an explicit coordinator in the existing DEVS/HLA as illustrated in Figure 7.

Here, a separate federate is allocated to the coordinator of Figure 3. We exploit quantizers to efficiently share the times of next events among DEVS federates and the coordinator – only changes in local or global T\(N\) (minimum of local T\(N\)s) greater than the granule size are sent across the network. Particularly, advances of zero time in DEVS federates are efficiently handled since they incur no T\(N\) updates.

The phases of the DEVS simulation cycle are advanced (monotonically) through RTI time as illustrated in Figure 8. Model logical time (DEVS time) proceeds along a separate axis as shown, advancing each cycle in the phase where the coordinator updates the federates with any global time.

5. Conclusion

We have discussed the design and development of DEVS/HLA, an HLA-compliant modeling and simulation environment that supports high level model building using the DEVS methodology, a sound formal M&S framework. Since the DEVS formalism is known to capture all discrete event dynamic behaviors, it is important that the DEVS/HLA implementation preserve this capability. In this paper, we have shown that direct mapping of the DEVS simulation protocol incurs a mismatch in the HLA specification, unless restricted to the sub-formalism defined as DEVS without cancellation – which allows only unconditionally timed behaviors. To obtain full expressiveness, i.e., DEVS with cancellation, we presented an alternative solution that
efficiently exploits the functionality of the baseline DEVS/HLA (DEVS without cancellation) to implement the necessary time-management and coordination. How the solution extends to real-time DEVS simulation and implications for the design of real time infrastructures are topics for further research.

6. References


BERNARD P. ZEIGLER is Professor or Electrical and Computer Engineering at the University of Arizona, Tucson. He has written several foundational books on modeling and simulation theory and methodology. He is currently leading a DARPA sponsored project on DEVS framework for HLA and predictive contracts. He is a Fellow of the IEEE.

Appendix: Mapping DEVS messages to HLA Interactions

Interaction communication is a message passing protocol through which a sender federate sends a set of messages to a receiver. The top-level DEVS model of a federate gathers all outgoing DEVS messages, translates them into interaction instances, and sends them to the destination federate. The information required to construct the interactions is contained in the HLAport instances that the modeler defines at the top-level. The receiving federate translates the interaction back into a DEVS messages. The latter is then distributed among to the local child components, according to coupling information in the HLAports of the receiver federate.