The DEVS/HLA Distributed Simulation Environment

And Its Support for Predictive Filtering

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

AI and Simulation Group [1]
Department of Electrical and Computer Engineering
University of Arizona, Tucson, Arizona

A Deliverable in Fulfillment of

Advance Simulation Technology Thrust (ASTT)
DARPA Contract N6133997K-0007

September 1998

[1] **URL:** www-ais.ece.arizona.edu; **Email:** zeigler@ece.arizona.edu

ABSTRACT

This report describes the DEVS/HLA distributed simulation environment under development and its support for predictive filtering research. The underlying DEVS and HLA frameworks are reviewed as a basis to discuss their synthesis in the form of a high level modeling and distributed simulation environment. This is the second report in fulfillment of DARPA Contract N6133997K-0007. The latter report established a theoretical foundation for quantization as a basic approach to predictive filtering. In this report we show how the DEVS/HLA can support the research that we are planning to test the theoretical predictions for quantization as an advantageous method of predictive filtering. We also consider how DEVS/HLA will support study of more advanced predictive contract mechanisms. Several appendices are included, one of which discusses some design and implementation issues in the time management simulation protocol employed in the DEVS/HLA with respect to the underlying HLA specification.

We note that the DEVS/HLA environment is intended to support predictive filtering research but that the results of the research, namely predictive filtering algorithms and mechanisms, will be formulated in generic, portable, extensible specifications.

EXECUTIVE SUMMARY

HLA (High Level Architecture) is standard mandated Department of Defense for simulator interoperability and reuse to be adopted by all defense contractors and agencies by 2001. This report describes the development and application domain of DEVS/HLA, an HLA-compliant modeling and simulation environment. DEVS/HLA supports high level model building using the DEVS (Discrete Event System Specification) methodology. It greatly simplifies the underlying programming details that have to be managed to establish and participate in an HLA federation.

After a brief review of the DEVS formalism, we discuss the implementation of the formalism in the C++ language. The choice of C++ was driven by the fact that this is the language used for the first Runtime Infrastructure (RTI) released by the Defense Modeling Simulation Organization (DMSO) to implement the HLA standard. We show how the DEVS implementation is extended to interface with the C++ RTI Version 1.0. We provide and in-depth discussion of the design and implementation issues involved in this interface in an Appendix. . A code example is provided to illustrate the use of DEVS/HLA as a high level-modeling environment. We also consider the co-evolution of DEVS/HLA with HLA as the HLA standard receives new C++ versions and new language implementations, such as Java.

This is the second report in fulfillment of DARPA Contract N6133997K-0007, "DEVS Formalism as a Framework for HLA Predictive Contract Methodology." The first report established a theoretical foundation for quantization as a basic approach to predictive filtering. In this report we show how the DEVS/HLA environment can support the research that we are planning for the project in testing the theoretical predictions for quantization as an advantageous method of predictive filtering. We also consider how DEVS/HLA will support study of more advanced predictive contract mechanisms.

A clarification is in order concerning the methodology employed in this work.

- The DEVS formalism provides a framework in which to express predictive filtering algorithms
- The DEVS/HLA environment is intended to support the implementation and testing of these algorithms and offers an HLA-compliant vehicle for their general use.
- However the predictive algorithms that result from this research will be *formulated in generic, portable, extensible specifications* that enable them to be applied in arbitrary distributed simulation systems, whether HLA or non-HLA compliant.

1 Introduction

HLA (High Level Architecture) is standard mandated Department of Defense for simulator interoperability and reuse to be adopted by all defense contractors and agencies by 2001. 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 interoperation 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 coupling of components, hierarchical, modular construction, support for discrete event approximation of continuous systems and an object-oriented substrate supporting repository reuse [2-4]. DEVS/HLA supports high level model building in DEVS terms, shielding the modeler from the underlying programming details that have to be managed to establish and participate in an HLA federation.

The purpose of this report is to describe the DEVS/HLA environment and its support for predictive filtering. This is the second report in fulfillment of DARPA Contract N6133997K-0007 and assumes familiarity with the first report on the DEVS theory of quantization[5]. The latter report established a theoretical foundation for quantization as a basic approach to predictive filtering. In this report we will show how the DEVS/HLA can support the research that we are planning for the DARPA project in testing the theoretical predictions for quantization as an advantageous method of predictive filtering. We will also consider how DEVS/HLA will support study of more advanced predictive contract mechanisms.

2 Overview of HLA and DEVS/HLA

As illustrated in Figure 1, 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.

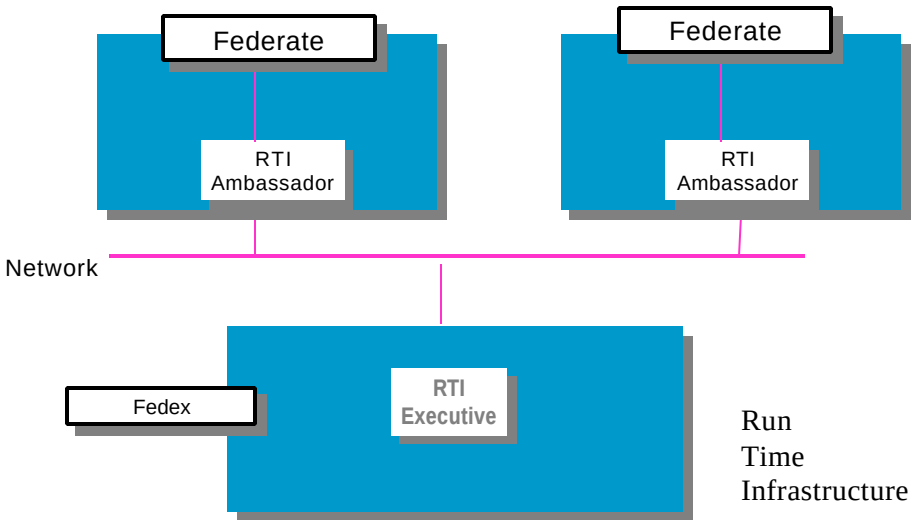


Figure 1 High Level Architecture (HLA)

DEVS/HLA federates communicate through the standard DEVS interfaces of the DEVS models they contain (Figure 2). 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. 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.

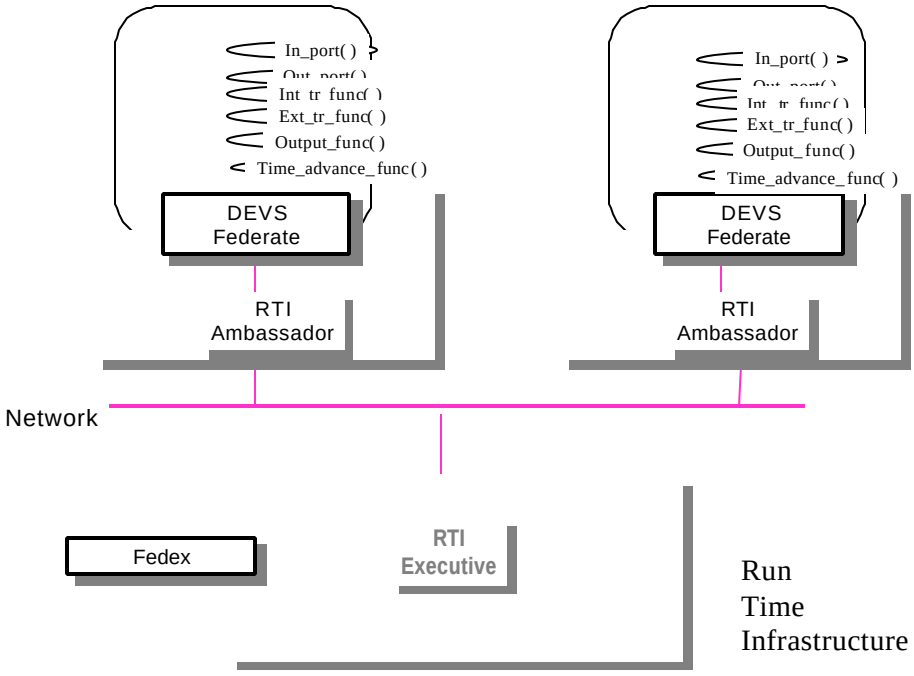


Figure 2 DEVS/HLA

The resulting environment for high level model specification and simulation is depicted in Figure 3. 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.

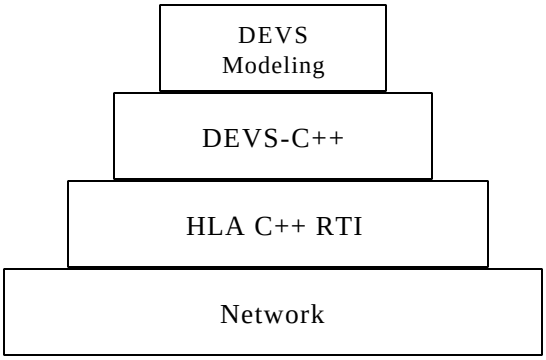


Figure 3 HLA/DEVS Supporting High Level Model Specification

A similar concept holds to enabling the use of DEVS environments to support integrating legacy or other simulation software. As an example, consider the Pleaides [6] system which is a DEVS-compliant simulation environment developed by Lockheed Martin capable of modeling multi-platform spatial interactions. It is being ported to DEVS-C++ and hence further to DEVS/HLA under subcontract to the current contract. Pleaides is being extended so that it incorporates an OPNET model of an inter-satellite communication network as well as an existing warship simulation code. Figure 4 illustrates one approach to such integration of heterogeneous model components – where each is represented by a DEVS federate in DEVS/HLA. The DEVS Bus concept, discussed in the above mentioned report [5], provides a basis for wrapping non-DEVS models within DEVS components.

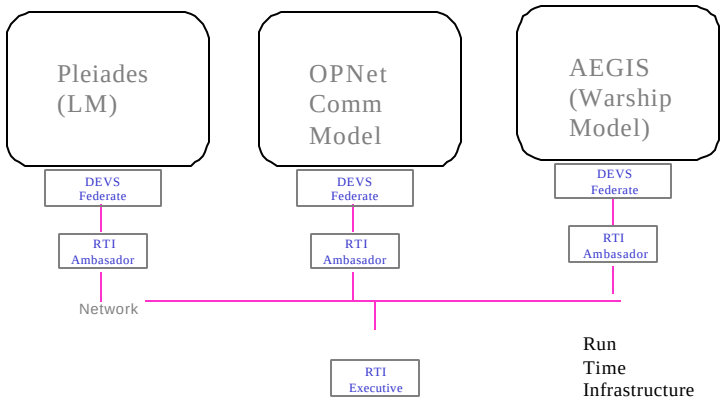


Figure 4 HLA/DEVS Supporting Integration Heterogeneous Components

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 [2-4]. We provide a brief review of the DEVS formalism for these classes before discussing their implementation in DEVS-C++.

3.1.1 DEVS Formalism for Basic Models

A DEVS basic model is a structure:

$M = <X,S,Y, d_{int}, d_{ext}, d_{con}, l, ta>$

where:

- X : set of external input events;
- S : a set of sequential states;
- Y : a set of outputs;
- d_{int}: $S \rightarrow S$: internal transition function
- $d_{ext} : Q \times X^b \rightarrow S$: external transition function
  where $Q = \{ (s,e) \mid s \in S, 0 \le e \le ta(s) \}$
- X^b is a set of bags over elements in X,
  (where $d_{ext}(s,e,\phi) = (s,e)$);
- $d_{con} : S \times X^b \rightarrow S$: confluent transition function;
- $\lambda : S \rightarrow Y^b$: output function generating external events at the output;
- $ta : S \rightarrow Real$: time advance function;

- *e* is the elapsed time since last state transition.

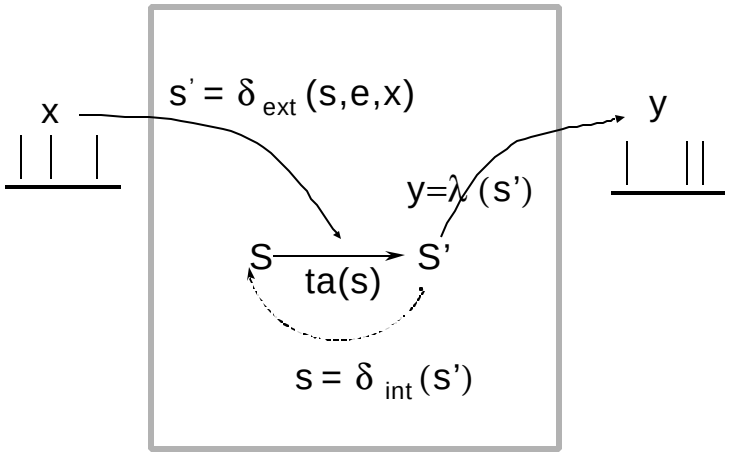


Figure 5. Schematic representation of a basic (*atomic*) model

Figure 5 shows schematic representation of basic models. Events are either externally or internally caused. The state of a model, which persists from one event to the next, is realized by a set of instance variables in the object representation of a DEVS. For example, a queuing system state might include the length of its queue among other instance variables. When an input packet arrives, the length is increased by 1. The external transition function (d_{ext}) processes the input and determines the resulting state depending on the

current state and the elapsed time the model has resided in the state. The residence time in any state is given by the time advance function, *ta*. The internal function (d_{int}) changes the state after this residence time has elapsed without inputs. For example, after a residence time that corresponds to the processing time of the selected customer in the queue, the internal transition function removes the packets that have been completed. This causes the length of the queue to decrease by the number of removed packets. The d_{con} function decides what to do when both external and internal events occur together. One simple approach is to apply the d_{ext} and d_{int} functions in a pre-decided order. The output function (*l)* function produces output based on the current state when triggered just before an internal event.

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

3.1.2 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.

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 Appendix 1.

4 Implementation of DEVS Formalism in DEVS-C++

DEVS is most naturally implemented in computational form in an object-oriented framework. A generic class hierarchy[7] for such implementation is given in Figure 6[2]. The basic class is *entity* from which class

[2] A more complete description of DEVS-C++ implementation is in the DEVS-C++ reference guide available from the web site www-ais.ece.arizona.edu under Software.

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.

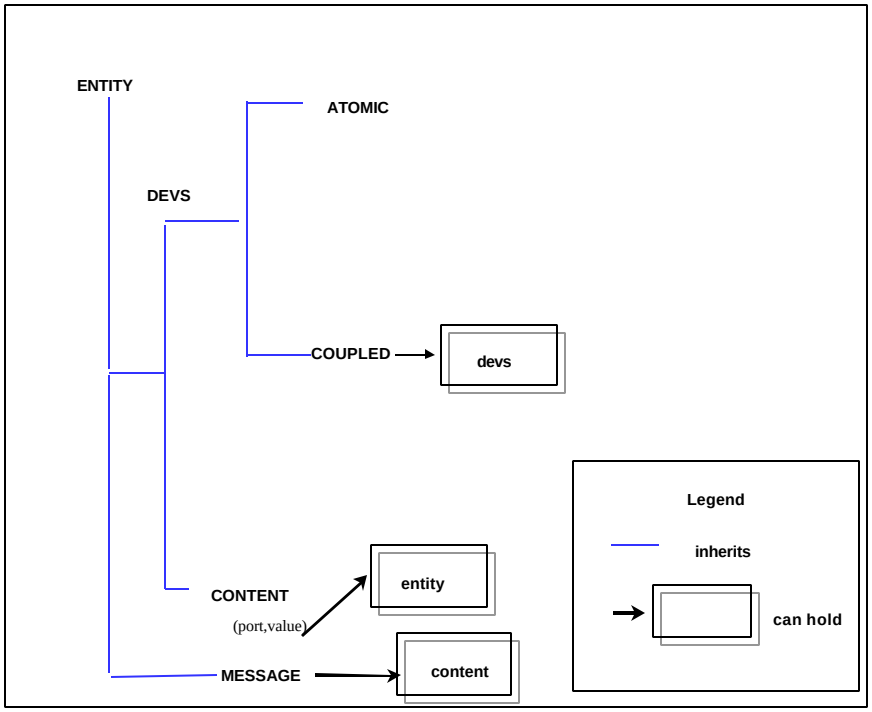
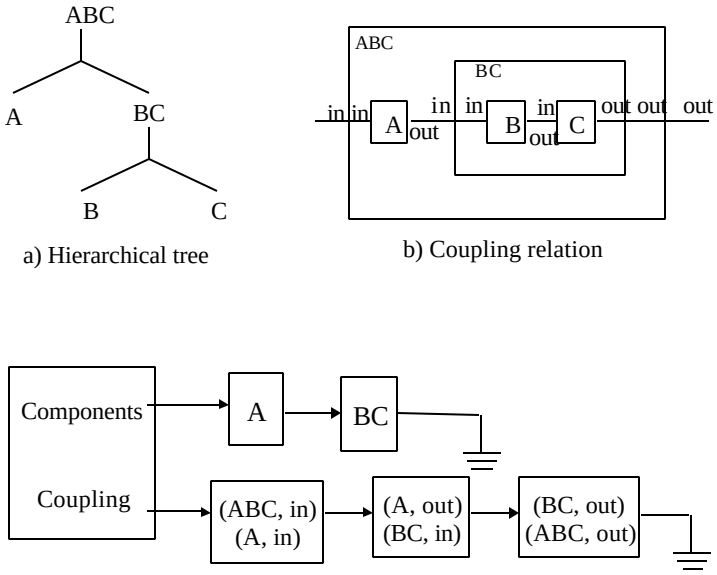


Figure 6 Basic Class Hierarchy for DEVS Implementation

Figure 7 a) depicts a *coupled* model in terms of a hierarchical tree. *Coupled* model ABC has components an *atomic* model A and a *coupled* model BC. In turn, BC's components are *atomic* models, B and C. A coupling specification associated with such a model, which specifies the data paths between components and between the coupled model and these components, is illustrated in Figure 7 b). Figure 7 c) shows how the coupled model information is represented in data structures. *Components* is a container which holds references to atomic or coupled model instances. *Pairs* in the *Coupling* relation holds the coupling information in terms of port-to-port connections.



a) Hierarchical tree

b) Coupling relation

c) Coupled model data structure

Figure 7. Representation of coupled model structure

4.1 Libraries

There are three libraries, *container*, *devs*, and *devsHLA* to provide easy access to the DEVS/HLA environment. The *container* library, based on set theory, supports basic services to manipulate sets of entities. Container, bag, set, relation, and function classes are in the *container* library, including ordered lists such as stacks, queues, and lists. On the other hand, the *devs* library supports several classes of basic models including atomic, cell, coupled, digraph, and block models. Using those predefined models, we can easily construct complex models in a hierarchical way, and build specific(user defined) models which would be expanded versions of basic models. The *devsHLA* library provides interface models or methods to mediate between DEVS and HLA – we'll return to it later.

4.1.1 Container Library

The *Container* class, a generalized form of linked lists based on set theory, provides methods which can store, retrieve and organize interacting objects. A container object is an object containing other objects.

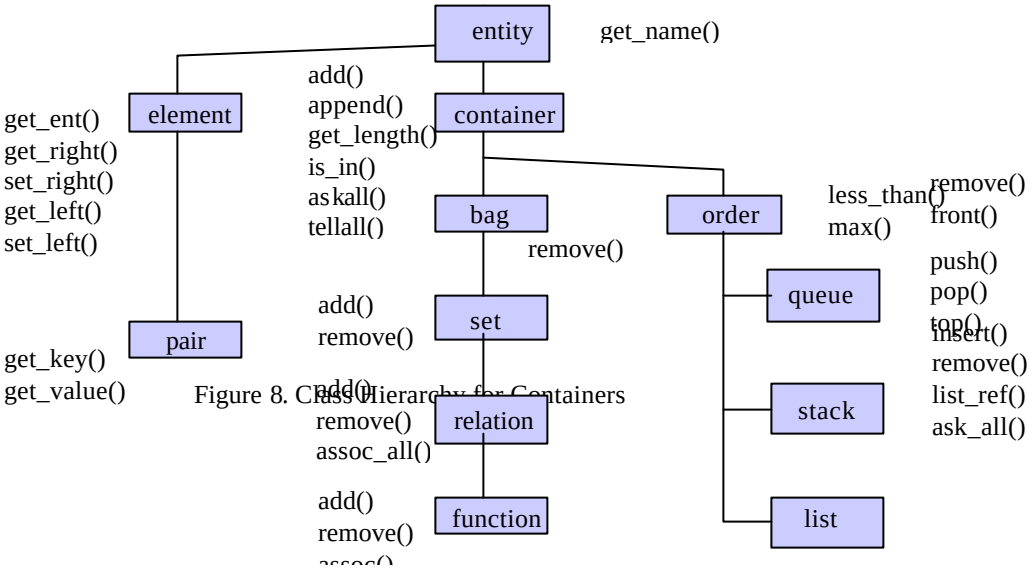


Figure 8. Class Hierarchy for Containers

Figure 8 depicts the *Container* class hierarchy. An *Entity* class will be the base class for all user-defined classes. It provides basically two methods, object name and a test of equality. The *Container* class provides basic services for the derived classes. The *Bag* class counts numbers of object occurrences. Only one occurrence of any object is allowed in the *Set* class. The *Relation* class consists of sets of pairs (key, value). The *Function* class is like the *Relation* except that at most one occurrence of any key is allowed. *Queues*, *stacks*, and *lists* maintain items in FIFO, LIFO, and ordered list respectively.

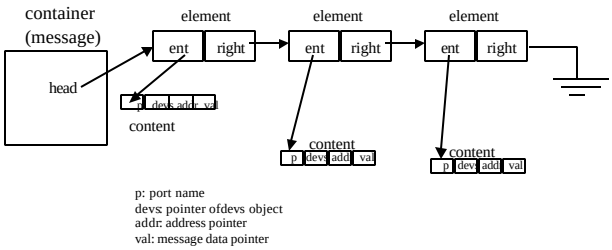


Figure 9. A message (container) structure with 3 contents

Figure 9 shows an example of a container class, specifically a message. Each content is in an element, and the elements are linked in a container. In DEVS-C++, the add method of a container,"add(entity *_ent_)", automatically creates a new element having entity, *ent*, and puts the element into the container. The code for printing contents in the message illustrates how a message is implemented as a linked list.

4.1.2 *Devs* Library

The d*evs* class is the basic class to provide methods for the DEVS formalism. Such functions as d_{ext}, d_{int}, and l 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 *inject* method is useful to send external input events to a model.

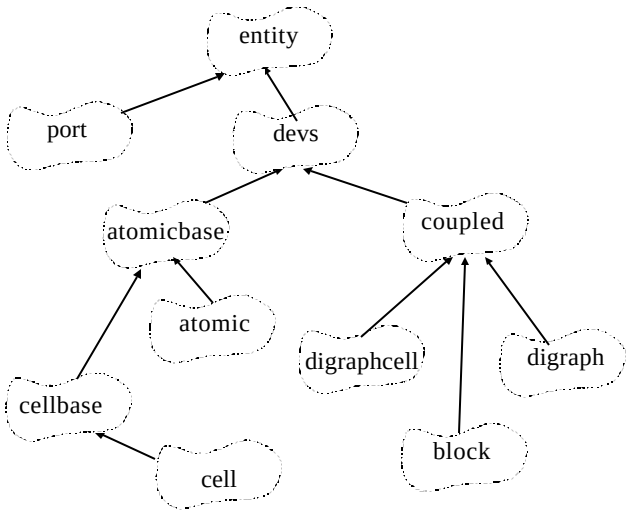


Figure 10. Class Hierarchy for DEVS models in DEVS-C++

As shown in Figure 10, an *atomicbase* has the *sigma* instance variable which holds the time remaining to the next internal event. This is the time-advance value to be produced by the time-advance function. 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 (d_{int}), external transition function (d_{ext}), confluent function (d_{con}), output (out) function, and time advance function (*ta*). These methods are applied to the state of the model. The atomic class has a *phase* variable to describe model's phase.
The *cellbase* and *cell* classes are similar to *atomicbase* and *atomic* respectively. These classes are for cellular models and have address information (which *atomic* models do not). In this way, large numbers of *cellbase* or *cell* models can be manipulated in the same manner. Addresses are used for finding influencees. The *coupled* class is the major class to support the hierarchical model composition constructs of the DEVS formalism. The d*igraph, block* and *digraphcell* are specializations which enable specification of *coupled* models in specific ways.

4.2 DEVS Simulation Engine

DEVS simulation engine consists of three steps: advance time function for next event time, compute for input and outputs, and execute transition functions.

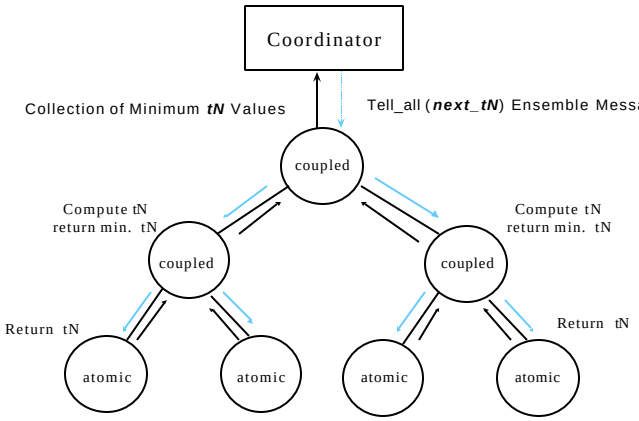


Figure 11. Next Event Time

Figure 11 shows how time advance function works. The highest *coupled* model, coordinator, tell all the components to return next event time, then model components update their *tL* (time of last event), *tN* (time of next event) values. The *coupled* model iterates the simulation cycle until the termination condition is met.

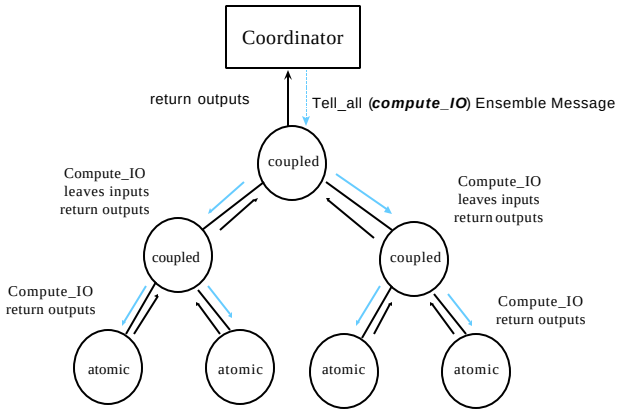


Figure 12. Compute Input and Output

The second step of a simulation cycle as shown in Figure 12 is to identify which components are imminent and to compute their inputs and outputs . Each imminent *atomic* component then generates its output in the Compute Input Output phase. A *coupled* model accumulates all the outputs of its components, and also determines whether each output message of its components is for the internal use (input messages) or outgoing (output messages). After all imminents produce their outputs, the *coupled* model tells components to execute their delta functions.

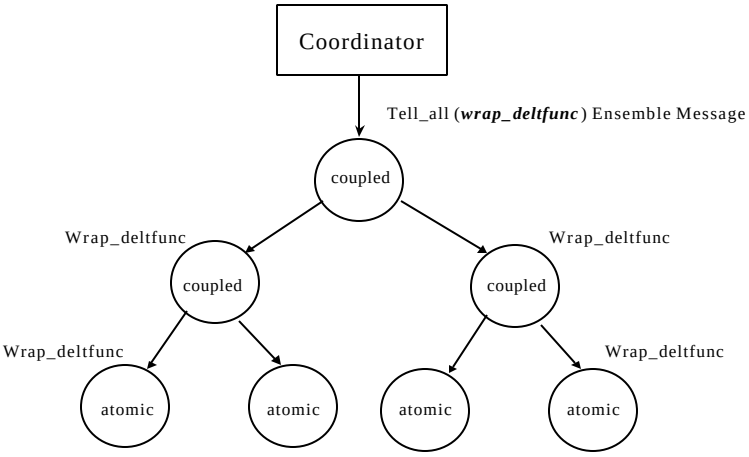


Figure 13. Transition Functions

Figure 13 shows how the external and internal transition functions work. Since all the components work simultaneously, an external input may arrive at a component at the very moment when its internal transition function should be executed. Via the confluent function a user can write tie-breaking rules such as selecting the order of the external transition or the internal transition function. By default, the internal transition function occurs first.

he internal transition function specifies to which next state a component will transit. For an atomic model, the effect is to place the component in a new *phase* and *sigma*, thus scheduling it for a next internal transition. Other state variables may be changed as well. The external transition function (d_{ext}) with inputs

from its influencees also specifies how the system changes state. For an atomic model, the next state is computed on the basis of the present state, the port and value of the input (external event), and the time that has elapsed in the current state.

A more complete consideration of issues in the design and implementation of the DEVS/HLA simulation protocol is provided in Appendix 2.

5 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) [8]. 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 [8]. 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.

Thus, the HLA has two major components: the Object Model Template (OMT) and the Runtime Infrastructure (RTI). Objects and interactions between objects can communicate across different platforms and are clearly visible to the user. They should be described by the user following by the standard format of the OMT. The RTI distributes messages to the appropriate parties and is transparent to the user.

5.1 Mapping DEVS to HLA

The strategy underlying the mapping of DEVS to HLA, illustrated in Figure 14, 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.[3] 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 (see below) 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. Ultimately, our goal is to facilitate a bi-directional transfer of information between the OMT Development Tool (OMDT) that captures OMT information and the DEVS model description. In the DEVS-to-OMDT direction, all DEVS elements are object-oriented and composition and its coupling features can be documented within the OMDT. In the reverse direction, the static elements of a model, described in OMDT, can be mapped into DEVS object-oriented elements. However, the dynamics encodeable in the internal and external transition functions have no parallel in the OMT specification.

[3] Although the computer architecture dependent details of parameter marshalling are not handled by DEVS/HLA.

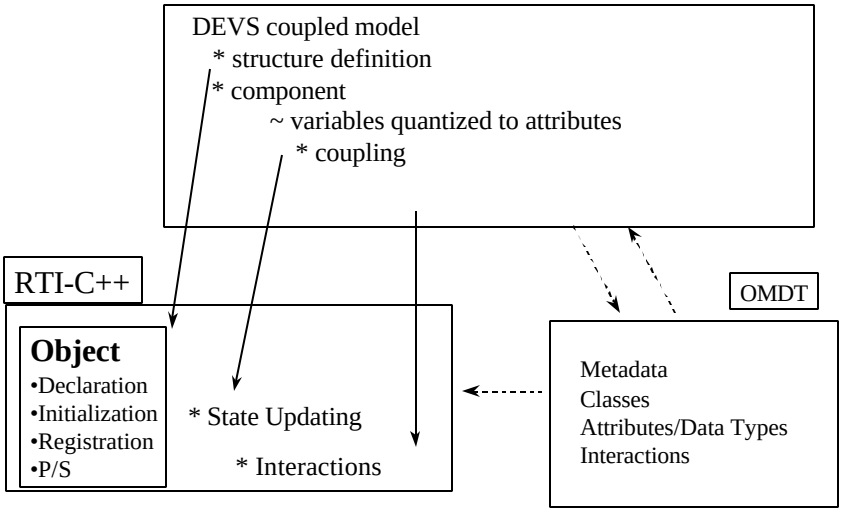


Figure 14 DEVS to HLA mapping strategy

5.1.1 *devsHLA* Library

We return to describe the *devsHLA* library for interfacing between DEVS and HLA in a C++ environment.

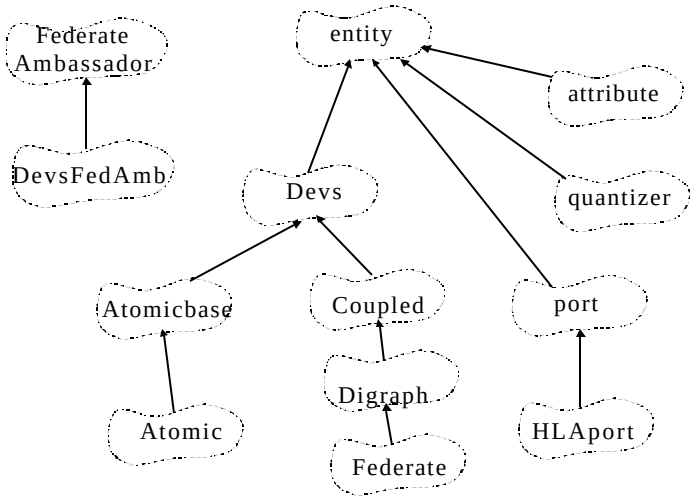


Figure 15. Class Hierarchy in the *devsHLA* library

Classes for the interface between the DEVS and HLA are shown in Figure 15. The *DevsFedAmb* class provides interface methods between DEVS and HLA that employ the methods of the *FederateAmbassador* class, which provides primitives to access HLA facilities. Such primitives include *discoverObject*, *reflectAttributeValues*, and *receiveInteraction* called by RTI. The method *discoverObject* is invoked when a federate receives an attribute update for an object which is not yet discovered. Once the object is found,

the RTI calls the *reflectAttributeValues* method for the rest of the attribute update of the object. For an interaction update, the *receiveInteraction* method is invoked when another federate in the federation executes the *sendInteraction* primitive of RTI. Both the attribute update and interaction update mechanisms will be described in detail.

The *devsHLA* library contains the *attribute* and the *quantizer* classes shown in Figure 15. 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 is used to reduce message update traffic – the trade-offs between accuracy and speed of computation involved are the subject of the research reported in refernce [5],. The *Atomicbase* and *Atomic* classes are extensions of similarly named methods in the *devs* library and provide HLA interface methods for atomic level models. Likewise, the *Coupled* and *Digraph* classes have interface methods for coupled models.

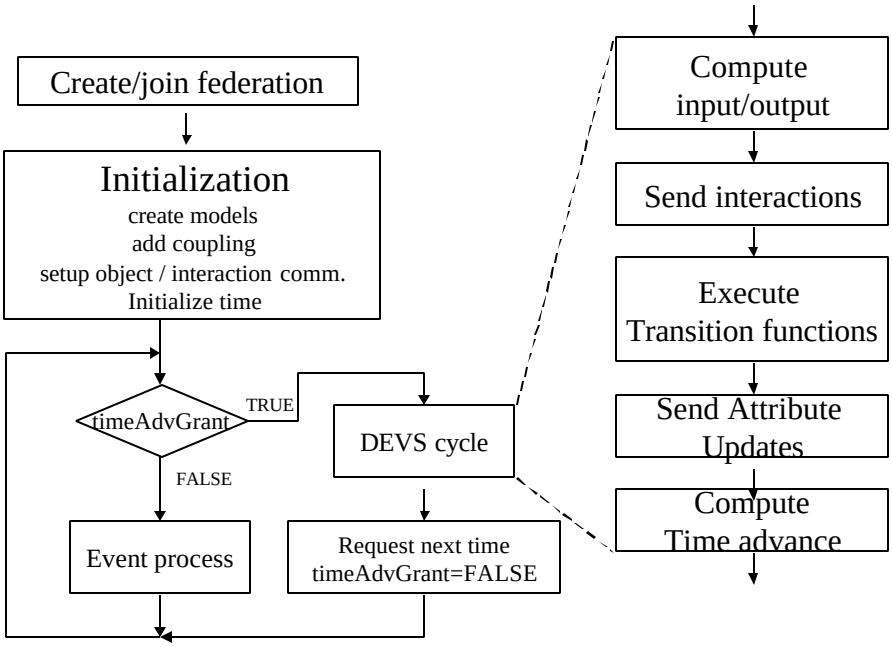


Figure 16. DEVS/HLA Simulation Cycle

Figure 16 provides an overall flow chart of a DEVS/HLA simulation. Most of the work is done by methods in the in the *devsHLA Federate* class based on information resident in DEVS models and in a relatively small increment that must be provided by the user. The first step of a simulation cycle is to set up DEVS/HLA environment by creating and joining a federation. During initialization phase, the top-level model in a federate creates its component models and sets up the appropriate coupling relationships among its components and itself. To communicate across a network, objects defined by the user need to register to the federation for publication of, or subscription to, attribute updates. The last step of the initialization phase is to determine the next-event-time and last-event-times for all components and request a time advance from the RTI for the top-level model.

After initializing sequence, the federate continually checks the TimeAdvGrant variable set by the RTI. While it is FALSE, incoming events generated by other federates for the attribute update or interaction communication are processed as they arrive. When the current time equals the requested time, the RTI sets the TimeAdvGrant to TRUE and the federate then executes the DEVS cycle. In the DEVS cycle, the top-level model identifies the components that are imminent at the current time. Each imminent *atomic*

component then generates its output in the compute-input-output step. The top-level model accumulates the outputs of its imminents and determines whether each such message is for the internal use (input messages) or outgoing (output messages). After all packaging the outgoing outputs into a message, the top-level model sends this message as an interaction communication via the RTI. The components then proceed to execute their transition functions.

The internal transition function (δ_{int}) specifies to which next state a component will transit. For an atomic model, the effect is to place the component in a new *phase* and *sigma*, thus scheduling it for a next internal transition. Other state variables may be changed as well. The external transition function (δ_{ext}) with inputs from its influencees also specifies how the system changes state. For an atomic model, the next state is computed on the basis of the present state, the port and value of the input (external event), and the time that has elapsed in the current state. Since all the components work simultaneously, an external input may arrive at a component at the very moment when its internal transition function should be executed. By employing the confluent function a user can specify how such collision should be handled. For example, one may write tie-breaking rules such as selecting the order of processing the external and internal transition functions. By default, the internal transition function is processed first.

Since the transition functions can change attribute values, a model may need to send attribute values to the model in another federate. Communication between objects in different federates is done by the HLA attribute update reflection mechanism. As we have seen, such messaging contrasts to that between top-level models of each federate which is done by mapping to the HLA interaction communication protocol.

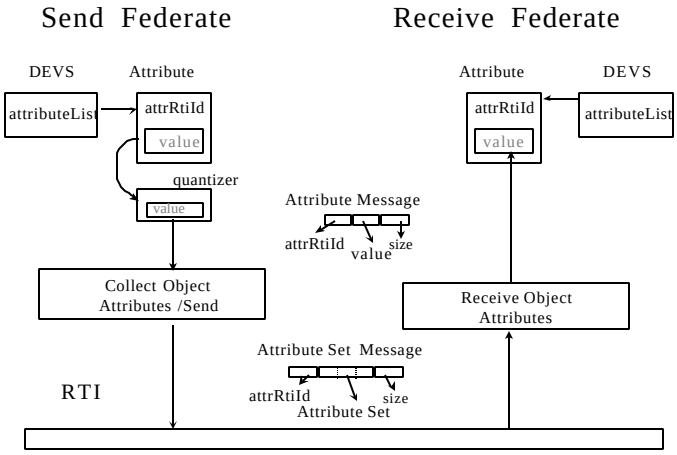


Figure 17. DEVS/HLA interface for the Attribute Update

The attribute update mechanism is employed when an object (sender) in a federate wants to change the specified attribute values of the specified object in another federate (receiver). As shown in Figure 17, the *Devs* class, the base class of DEVS-C++ models, has an *attribute list* (container) which keeps a set of attribute instances with attached quantizers. After every internal or external transition a message is formed of the values of attributes whose quantizers have crossed a threshold. These are sent to other federates and discovered by the *discoverObject* method and then processed by *updateAttributeValues* method. The receiving DEVS model checks the to-be-changed attributes and updates the values of the local copy of each attribute.

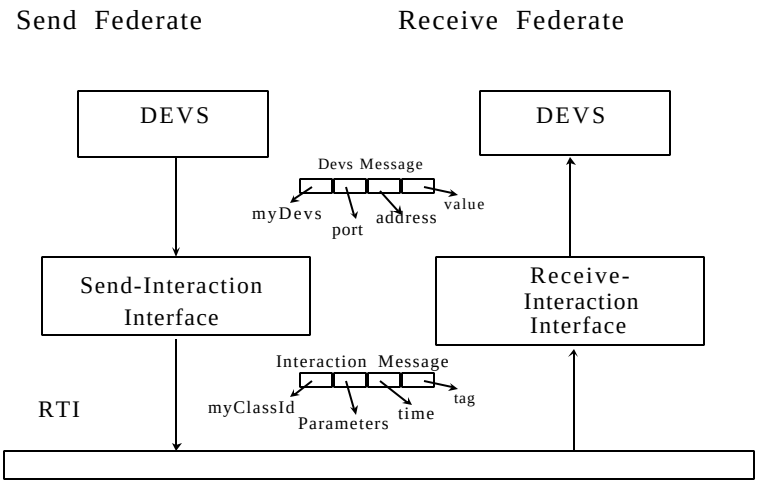


Figure 18. DEVS/HLA Interface for the Interaction Communication

Interaction communication is a message passing protocol through which a sender federate sends a set of messages to a receiver. In Figure 18, 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 interactionb ack 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.

5.1.2 High level modeling paradigm

The high level modeling paradigm supported by DEVS/HLA is an efficient object-oriented design methodology which significantly reduces the complexity of constructing models in a hierarchical modular fashion and implementing them in an HLA compliant simulation. It also improves the maintenance, reusability, and modifiability of models. This paradigm employs data encapsulation mechanism with which the implementation details of the model are kept hidden from the designer. Three steps are required to build a level of the hierarchy between an upper model and its components: create models, add them to the upper model, and define coupling relationship. Figure 19 shows a program example of Figure 7 with respect to how to program and how easy to design.

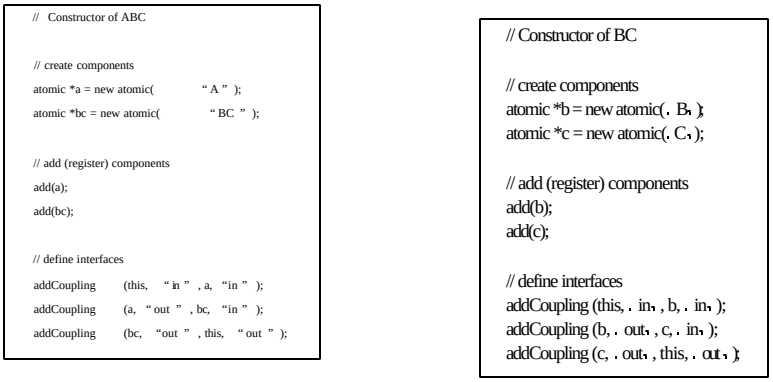


Figure 19. C++ program of Figure 7 using high level modeling paradigm

The *atomic* and *coupled* classes in Figure 19 are taken from the *devs* library, with the *add* and *addCoupling* methods belonging to the *coupled* class. As shown in Figure 20, the underlying network is transparent to a user of DEVS/HLA. The main routine has only two lines: create the top level model *FederateDevs* and run it. *FederateDevs* is a user-defined class, one for each different federate, derived from the *Federate* class in the *devsHLA* library. At the beginning, the constructor of *Federate* class builds the network environment using the underlying HLA primitives.

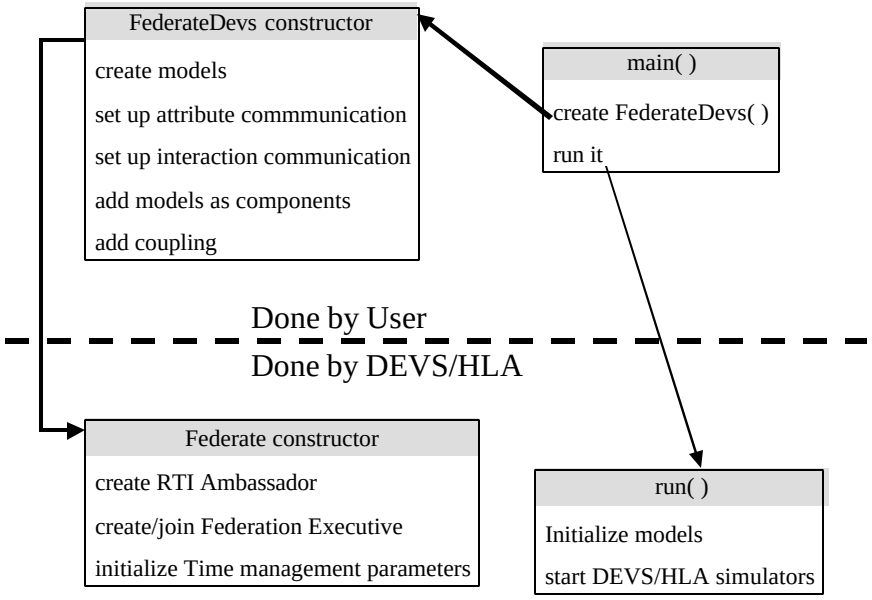


Figure 20. User interface of DEVS/HLA environment

6 Support of Predictive Filtering Studies

DEVS/HLA is being developed for the purpose of studying alternative forms of predictive filtering. It is intended to provide a vehicle for expressing such mechanisms and testing in realistic contexts. One primary form of predicting filtering that has been our initial focus of investigation is that of quantization [5].

6.1.1 Predictive Filtering Background

The basic concept in quantization is illustrated in Figure 21. Rather than represent a continuous curve by points sampled at regular time intervals, the curve is represented by the crossings of an equal spaced set of boundaries, separated by a *quantum* size. In classical Dead Reckoning, quantization is applied to the error between a reduced order model and a high fidelity model of a federate[9]. However, a more fundamental approach is to allow any desired object attribute to be quantized. This has two advantages [5]. First, a wider space of possible algorithms is opened up for investigation, including more direct approaches that do not require federates to keep local models of other federates. Second, it enables us to clearly distinguish the **global** error incurred as a result of predictive filtering from the local error that may be used to, as in Dead Reckoning, to effect a change in local models exported to others. We'll return to this point in a moment.

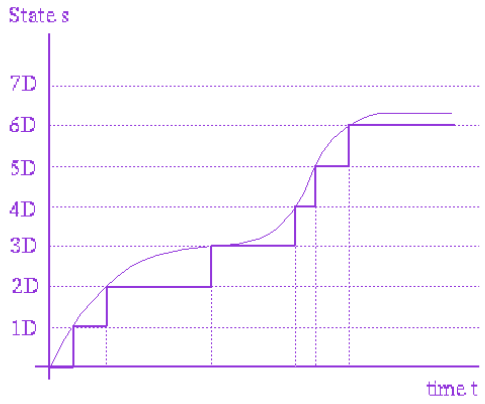


Figure 21 Quantization

The baseline mechanism for quantization, called *non-predictive quantization*, is illustrated in Figure 22. We assume a sender federate is updating a receiver federate on a numerical, real-valued, state variable (dynamically changing attribute), V. In the non-predictive approach, a *quantizer* demon is applied to the sender's output which checks for threshold (boundary) crossings whenever a change in V occurs. Only when such a crossing occurs, is a new value of V sent across to the receiver.[4] We note that in this change-based filtering operation, the frequency of message updates may be substantially reduced, thereby reducing network traffic but potentially incurring error. The cost/benefit analysis between reduced traffic and increased error can be framed in terms of tradeoff curves as we shall soon discuss. Note, for future reference, that while message *traffic* is reduced, the *size* of messages sent is unaffected in this approach. The quantizer demon incurs some computation at the sender federate but this is relatively inexpensive. Such additional computation, does raise another issue – *scalability*, how fast does the additional computation required by a predictive filtering method grow with increasing simulation size. Also, the receiver must be prepared to handle updates arriving asynchronously. If synchronous updating was assumed in its design, this may require redesign – depending on the flexibility of the underlying simulation code, this may, or may not, be easy to achieve. In object oriented designs, with flexible time management, this would not be a major issue. To summarize, the characteristics of non-predictive quantized filtering are:

- Sender federate generates fixed (or variable) time step outputs.
- Quantizer demon is applied to sender output.
- This reduces the number of messages sent (although not their size).
- The quantizer incurs some computation at the sender's federate.
- The sender's model computation is unaffected.

[4] Several possible variations on this theme, concerning for example, whether we send the actual value of V, or a modification depending on the boundary crossing. Such details are beyond the scope of this report.

In sum, this approach is relatively easy to apply and requires minimal restructuring of the federates' state computation processes. However, it can incur loss of accuracy due to the receiver's diminished state updates and this may propagate in a global error due to feedback between sender and receiver as we will discuss. Also, as will be illustrated in an example, the DEVS/HLA environment supports this approach through its development of quantizer objects and their automatic interfacing to the HLA subscribe/publish data distribution service. However, the concept is generic and can be employed in any distributed simulation environment.

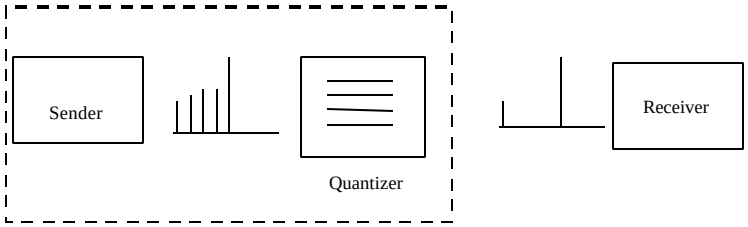


Figure 22 Non-Predictive Quantization

A more efficient form of quantization is *predictive quantization*, as illustrated in Figure 23. Here the sender employs a model to predict the next boundary crossing and time it will occur given its current state. As we will show, such computation can be quite inexpensive depending on the model used. This approach is inherently discrete-event based since the sender waits until the predicted next event (boundary crossing) time before sending its output and effecting its state change. Note that the federate need not be computing state changes during this waiting period, thus gaining computational advantage over non-predictive quantization. Since the next boundary crossing is either one above or one below the last recorded boundary, the sender need not send the full floating point (double word) value to the receiver. Indeed, assume that the receiver keeps track of the last boundary and knows the quantum size. Then only *one bit* of information is required – for example, a +1 indicates adding the quantum to the last boundary, while –1 indicates a similar subtraction. Thus, not only the number of messages but also the message size – in other words, total number of bits transmitted – can be significantly reduced in this approach.

As shown in Figure 23, the stream emitted by the sender must somehow convey the boundary crossing times. Whether this incurs additional network bandwidth depends on the context. In discrete event logical time simulations, messages can be easily time stamped. This usually involves no additional cost, as all messages are time stamped to enable strong synchronization in the accepted distributed simulation protocols. In real time simulations, preservation of the order and time spacing between messages may provide the required information without time stamping. This form of messaging is often assumed in classical DIS (Distributed Interactive Simulation) training exercises. As discussed in Appendix 2, HLA provides the flexibility to choose among logical and real-time time management schemes.

To summarize the characteristics of predictive quantized filtering are:

- The sender employs a model to predict successive boundary crossings.
- It sends a one-bit message at crossings – whether the next higher or next lower boundary has been reached.
- The main advantage over non-predictive quantization is that **both number of messages and their size can be reduced**
- A second advantage, is that if simple predictive models are used, discrete event prediction can also greatly reduce the sender's state transition computation execution time and frequency.
- The messages must convey the time of boundary crossing.
  - In discrete event logical time simulations, messages can be time stamped with usually, no additional cost, as this is the background approach.
  - In real time, preservation of the order and spacing between messages may provide the required information without time stamping.

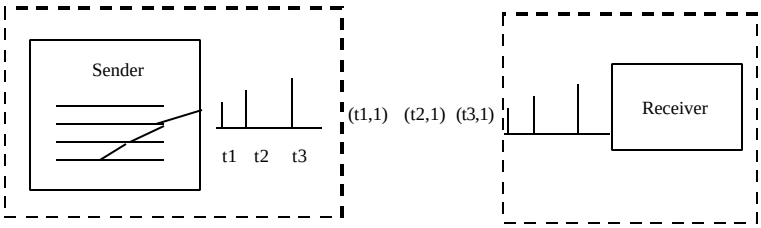


Figure 23 Predictive Quantization

A detailed theoretical and empirical study of the advantages of predictive quantization over non-predictive quantization is provided in [5]. Here we will briefly review some salient elements of this study.

6.1.2 Generic Predictive Quantization Methods

First, we note that the model employed for predictive quantization can be very simple. An approach that is fully generic for differential equation systems is illustrated in Figure 24. An ordinary differential equation system (ODE) consists of a finite number of integrators connected by instantaneous derivative functions to each other and to the external interface. A straightforward mapping of such a network on to a distributed equivalent is as follows:

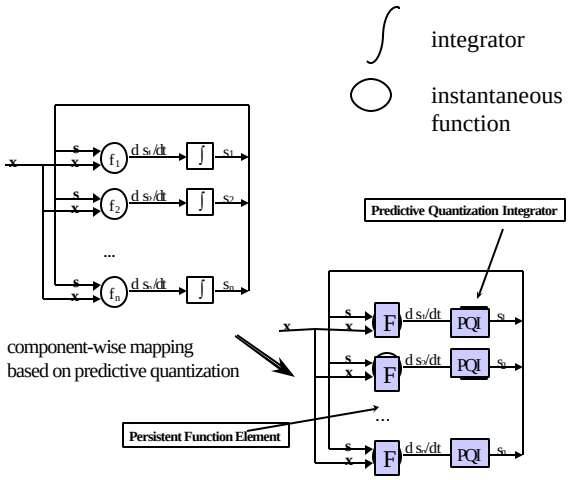


Figure 24 Mapping Differential Equation Systems directly to Distributed Discrete Event Form

Each derivative function is mapped to a *persistent function element* and each integrator is mapped to a predictive quantization equivalent while preserving the interconnection topology. Basically, a persistent function element receives event inputs produces the output of a derivative function after any one of the inputs changes.[5] The *predictive quantization integrator (PCI)* is basically linear extrapolation as illustrated in Figure 25. The time to next boundary crossing is basically the quantum size divided by the input (derivative). The boundary is predicted either to be one up or one down according to the sign of the derivative. We note that when an input event is received, the state is updated using the old input before recalculating the predicted crossing. This provides an important correction for error reduction.

[5] Again, there are variations on this theme.

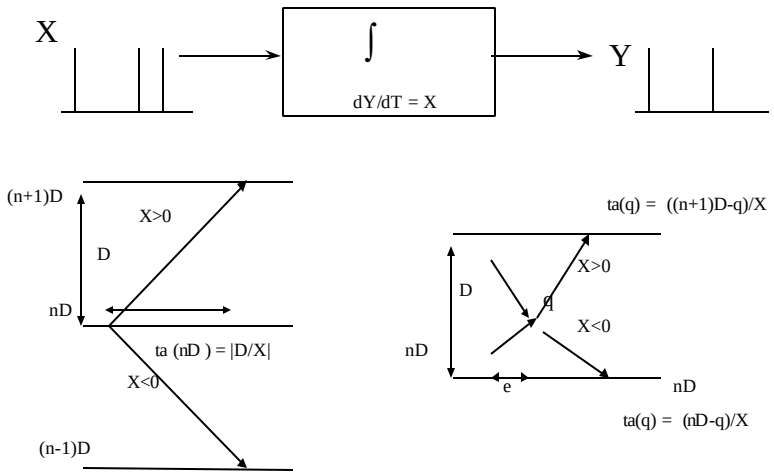


Figure 25 Predictive Quantization Integrator

6.1.3 Application to Dead Reckoning

A second application of interest is to dead reckoning employed in vehicle training simulation exercises. Recall that the conventional approach is employ a reduced order model at each federate which represents the motion of its high fidelity vehicle model and is exported to other federates. Typically, the reduced order model is a 2^{nd} order extrapolator that works on position, velocity and acceleration (p,v,a) updates. These updates are sampled and sent to other federates only when the error, as determined by a federate's comparison with its high fidelity model, exceeds a given threshold.

Two alternatives to the conventional approach that employ predictive quantization can be formulated:

- Direct application to the models involved
- Distance-based predictive quantization.

In the direct application, the predictive quantization approach can be applied to the reduced order model, the high fidelity model and the error comparator (Figure 26). For the 2^{nd} order extrapolator, the time to next crossing and the next boundary can be simultaneously determined by the smallest positive root of an easily formulated algebraic solution with parameters, quantum size, position, velocity, and acceleration. Application to the high fidelity model would employ the generic ODE mapping discussed above. With the quantum size set as the error tolerance, the error comparator would need to determine only whether the +1,-1 inputs from the models are different to trigger a sampling of the (p,v,a) vector and its transmission to other federates.

While this implementation does not affect the number of state updates needing to be exchanged among federates, it can significantly reduce the internal computation within a federate by capitalizing on the discrete event efficiency of the predictive quantization approach as discussed above. A less attractive

alternative – available when the high fidelity model cannot be easily restructured into predictive quantization form -- is to employ non-predictive quantization for the latter.

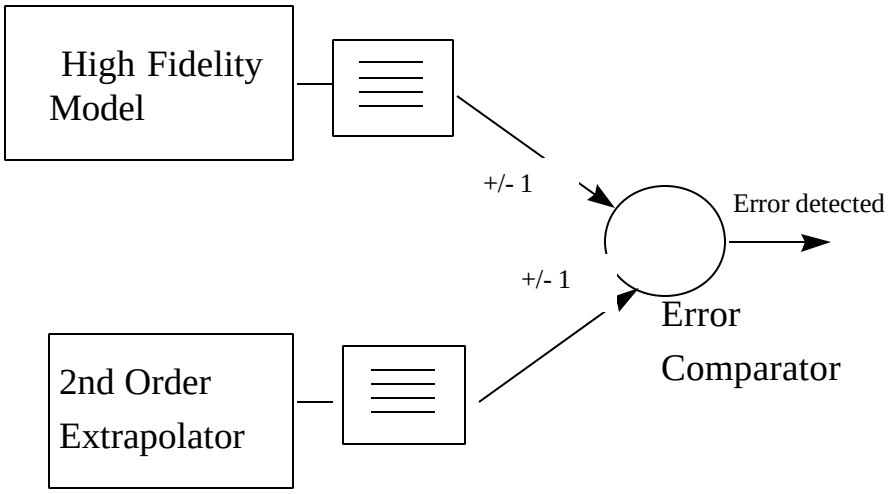


Figure 26 Predictive Quantization formulation of 2nd order extrapolator for Dead Reckoning

We are also developing a concept of dynamic, and in particular, distance-based quantization as a second possible alternative to conventional dead reckoning. As illustrated in Figure 27, in this approach a quantum size manager would allocate quantum sizes for communication between pairs of federates as a function of their distance in "routing" space.[6] Indeed, this scheme can be viewed as a refinement of the routing space mechanism implemented in HLA which control data exchange among federates in all-or-non fashion. The proposed scheme employs variable quantum size to reduce message traffic in more finely-tunable fashion.

This approach has the advantage that each federate need no longer maintain a compliment of reduced order models of other federates, thus greatly reducing the local computation. However, a potential area of concern for scalability is the of the quadratic list scanning required for quantum management. Thus an efficient scheme must be designed to properly test this alternative.

[6] Physical space is standard for many applications such as training but can be generalized to other spaces.

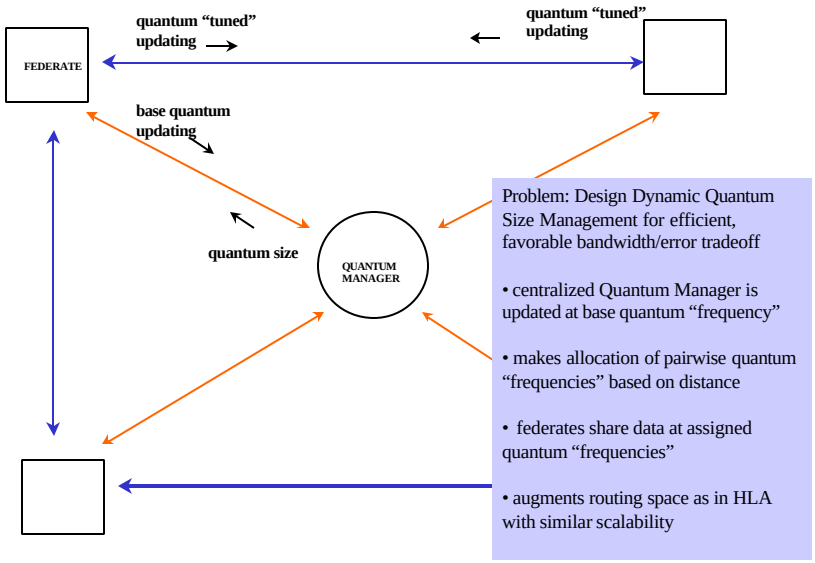


Figure 27 Distance-based Quantization Scheme

The support provided by DEVS/HLA for predictive filtering will include support for investigating and comparing the alternatives just mentioned. Summarizing, these are:

- conventional dead reckoning
- event-based dead reckoning with
  - predictive quantization of 2nd Order Extrapolation.
  - quantization of high fidelity model
- distance-based quantization with quantum size based on separation distance

6.1.4 Scalability issues

As already indicated, the computational cost incurred by a predictive contract method must be taken into account in considering its scalability to large simulations. A view of this issue is presented in Figure 28 which summarizes results obtained so far [5] for the number of messages, total number of bits sent, and number of computations incurred by a sender in the quantization-based methods discussed above. The scalability of the predictive quantization method versus its non-predictive counterpart is apparent. While both have the same message reduction curves as a function of quantum size (a), the predictive mechanism enjoys the same reduction curve in number of computations due to its discrete event nature. Also, shown is reduction in total number of bits sent due to the predictive mechanisms reduced message size.

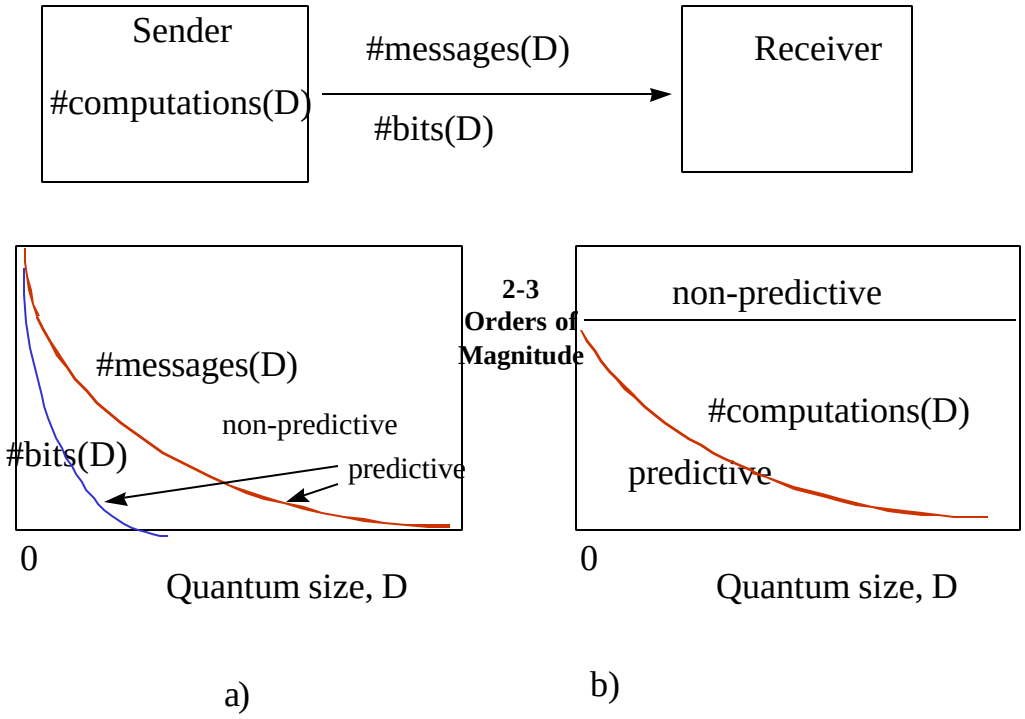


Figure 28 Scalability of Predictive Quantization

To summarize, quantization, especially in predictive form, appears to have excellent scalability properties. For the latter, the incurred computation cost can actually be reduced with increasing quantum size. Computational complexity enters the picture again when we go on to consider quantum size management with the distance-based alternative to conventional dead reckoning. The DEVS/HLA environment will provide the platform for investigating this issue.

6.1.5 Error/Message Reduction Tradeoff

As mentioned earlier, with regard to error, we make the distinction between local and global error. In the Dead Reckoning paradigm, local error is the federate-measured difference between its high fidelity and its reduced order model. However, maintaining this error below threshold cannot guarantee control on global error. The latter is the deviation from true global state that would prevail in the absence of predictive filtering. This is so since remote federates are making judgments based on the their local reduced order representations before they are updated at error threshold crossings. During such periods, errors may be introduced in their responses, propagated globally and fed back for further accumulation.

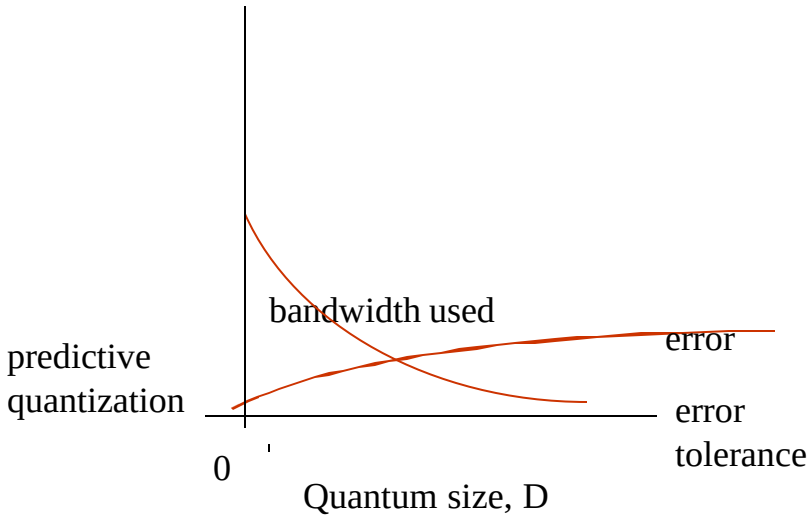


Figure 29 Error Characterization

The theory we have developed takes into account the global error due to predictive filtering. We have formulated and verified the theory behind this predictive quantization realization of arbitrary[7] ordinary differential equations. Figure 29 portrays a view of the tradeoff between performance and error. When plotted against increasing quantum size, bandwidth utilization can be expected to decrease, while (global) error is expected to increase. Given an error tolerance there is a maximum quantum size that will result in errors below or equal to the threshold. At this quantum size, there is a corresponding bandwidth utilization that is incurred representing the best that can be achieved within the given error tolerance. The tradeoff curves are favorable if the error curve rises slowly while the performance curve drops sharply. The most fundamental pair of curves that exemplify this issue has the error increasing proportionately to the quantum size while the bandwidth utilization falls inversely proportional to the quantum size. [8] Indeed, we have obtained very promising empirical results for tradeoff between message bandwidth utilization (number of bits transmitted) versus error incurred [5].

In the next section, we provide an example of how non-predictive quantization is realized in DEVS/HLA

6.1.6 *Pursuer-Evader* Model: An Example

[7] with the same well-posed conditions required for standard numerical integration methods.
[8] Although with hind sight this should have been obvious. However, our data led us to this conclusion.

Figure 30 depicts a *Pursuer-Evader* example to illustrate the high level modeling paradigm and hierarchical model design as well as the use of predictive quantization. FederatePursWQuant (standing for federate having a pusuer with a quantizer) is the highest DEVS model in the Pursuer federate. It has a component called *PursWQuant*, which in turn contains a *RedTank* model whose states will be reflected to the Evader federate. Thus, RedTank includes a shared object, RedTKObj, derived from the objectClass with name "RedTank" which will be shared among federates. All shared objects, with their attributes, must be registered using the setUp method defined in DEVS/HLA for publishing or subscribing to the RTI. An *HLAport* class named *fire* is defined to link two federates via the interaction communication. The Federate model, *FederatePursWQuant* shown below, declares all HLA related object classes and/or interaction classes in this layer which are the shared classes among several federates.

Sample code for a Federate

```
// constructor of FederatePursWQuant
// create the highest component
PursWQuant *PWQ
        =  new PursWQuant("PursWQuant");
// create objectclasses and register them to RTI
objectClass  *RedTKObj
        = new  objectClass("RedTank");
RedTKObj->addAttribute(new
            attribute("position",0,NULL);
setUp(RedTKObj);
// setup HLAport
HLAport *hp = new HLAport("fire");
addOutport(hp);  setUp(hp);
// coupling
addCoupling(PWQ, "fireout", this, hp);
```

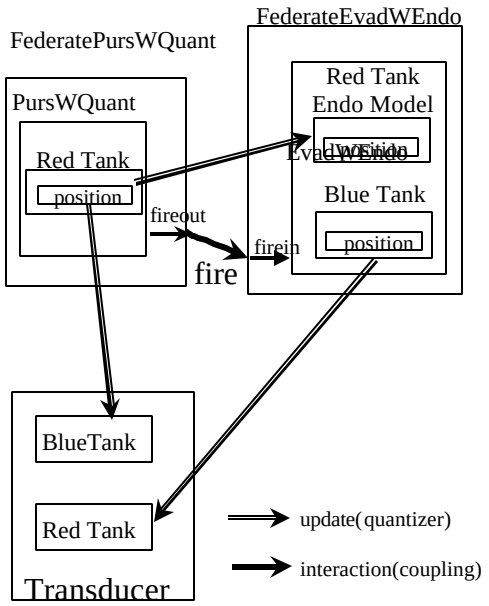
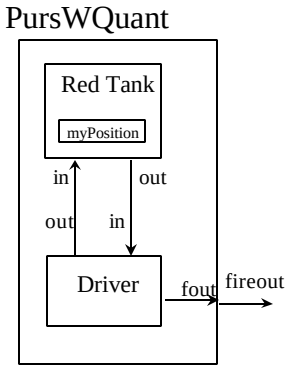


Figure 30. Pursuer-Evader model




```
// Constructor of PursWQuant
// create components
    RedTank *redtank
                = new  RedTank ("RedTank");
    Driver *drver = new  Driver ("Driver");

// add components
    add (redtank);
    add (drver);
// define interfaces (couplings)
    addCoupling (redtank, "out",driver,"in");
    addCoupling (drver, "out",redtank,"in");
    addCoupling (drver, "fout",this,"fireout");
```

```
// constructor of RedTank
// create quantizers and attributes
  quantizer *qt = new quantizer(name,initValue, quantum);
  attribute *at = new attribute("position", &myPosition, qt);
  attributeList->add(at);
```

Figure 31. Pursuer Federate with DEVS/HLA code

Figure 31 shows more of the internals of the *PursWQuant* coupled model. It has two components: the *RedTank* just discussed and a *Driver* atomic model (representing the person driving the tank). Note that the procedure for building models that will ultimately be deployed within Federates includes the same three steps described in Section 4. Here, a *PursWQuant* model creates *RedTank* and *Driver*, adds them as its components, and defines their coupling relationships. In addition, any attributes that may be reflected to other federates are declared and outfitted with quantizers. For example, a quantizer and an attribute are associated with the position of the red tank in Figure 31. Later, when a model is included within a federate, the attributes to be published are associated with the objects declared at the highest level. For example, the position attribute is added to the RedTKObj in Figure 30.

6.1.7 Summary: Support of Predictive Contract Mechanisms

As illustrated, the DEVS/HLA environment currently supports non-predictive quantization. The goal in the development of DEVS/HLA is to provide a test-bed to study generic architectures for predictive contract mechanisms. Our approach in further development will be to employ the DEVSJAVA environment to design and test the logic of the predictive contract mechanisms described above. These classes will be incorporated in the Prototype Toolkit, that is a deliverable of this contract. Since the mechanisms are formalized in DEVS, they can be readily transferred over to the DEVS/HLA-C++ distributed environment for full scale testing. However, an alternative is now possible since DMSO for JAVA has developed an API. This would enable direct execution of DEVS models in DEVS/HLA in JAVA, provided that the interface developed between DEVS-C++ and the HLA C++ RTI is ported to its equivalent between DEVSJAVA and the new HLA RTI API.

7 Conclusion

The theory of predictive filtering we have developed takes into account the global error due to error propagation. We have verified the theory when applied to predictive quantization realization of arbitrary ordinary differential equations. We have obtained very promising results for tradeoff between message bandwidth utilization (number of bits transmitted) versus error incurred The theoretical and empirical results so far indicate that predictive quantization can be very scalable due to reduced local computation demands as well as having extremely favorable message reduction/error tradeoffs.

We are planning to demonstrate the utility of alternatives to conventional dead reckoning including event-based versions using predictive quantization and distance-based quantization. Our hypotheses are that the event-based and distance-based quantization mechanisms will afford distinct advantages in regard to scalability, error propagation, and their tradeoff with message traffic reduction. The platform to support such demonstration studies is the DEVS/HLA distributed simulation environment.

DEVS/HLA has been shown to be an efficient, HLA-compliant distributed simulation environment based on the DEVS formalism and object-oriented technology. It allows building of models in a hierarchical and modular fashion and is especially geared toward the objectives of demonstrating the utility of predictive contract mechanisms.

Currently, the DEVS/HLA environment supports non-predictive quantization though an object oriented family of classes. This facility enables modelers to conveniently attach quantizer objects to model variables and have the required publish/subscribe declaration be handled automatically. The environment will be extended to support the simulation study of advanced forms of predictive contract mechanisms such as non-predictive quantization, distance-based quantization and other alternatives to conventional dead reckoning.

The environment is able to readily co-evolve with the evolution of HLA and is being extended to support JAVA-based simulation as well. The DEVSJAVA toolkit prototype has been instrumental in verifying theoretical predictions that quantization, particularly predictive quantization, can significantly reduce messaging size and traffic, with favorable tradeoff with global error.

Current work is focusing on replicating these results in the DEVS/HLA distributed simulation environment. We expect the results to confirm: the efficiency and scalability benefits of event-based over conventional dead reckoning. We also expect very favorable bandwidth utilization/error tradeoff and scalability of distance-based quantization. As a potential spin-off we think that the distance-based approach will have benefits with respect to After Action Review, a concern expressed with respect to conventional predictive contract management.

In sum, theoretical and empirical results so far indicate that predictive quantization can be very scalable due to reduced local computation demands as well as having extremely favorable message reduction/error tradeoffs.

Once validated in this way, the mechanisms will be usable in other distributed simulation environments, whether DEVS-based or not, whether HLA-compliant or not. Indeed, their formal characterization within DEVS facilitates such a formulation and guarantees that they are not constrained by environment dependent features.

To summarize the methodology employed in this work.

- The DEVS formalism provides a framework in which to express predictive filtering algorithms
- The DEVS/HLA environment is intended to support the implementation and testing of these algorithms and offers an HLA-compliant vehicle for their general use.

- However the predictive algorithms that result from this research will be *formulated in generic, portable, extensible specifications* that enable them to be applied in arbitrary distributed simulation systems, whether HLA or non-HLA compliant.

8 References

1. Zeigler, B.P., *et al*., *The DEVS Environment for High-Performance Modeling and Simulation.* IEEE C S & E, 1997. **4**(3): p. 61-71.
2. Zeigler, B.P., T.G. Kim, and H. Praehofer, *Theory of Modeling and Simulation*. 2 ed. 1998, New York, NY: Academic Press.
3. Zeigler, B.P., *Object-Oriented Simulation with Hierarchical, Modular Models: Intelligent Agents and Endomorphic Systems*. 1990, San Diego,CA: Academic Press.
4. Zeigler, B.P., *et al*., *DEVS Framework for Modelling, Simulation, Analysis, and Design of Hybrid Systems*, in *Hybrid II, Lecture Notes in CS*, P. Antsaklis and A. Nerode, Editors. 1996, Springer-Verlag: Berlin. p. 529-551.
5. Zeigler, B.P., *DEVS Theory of Quantization*, . 1998, DARPA Contract N6133997K-0007: ECE Dept., UA, Tucson, AZ.
6. Hall, S., *Space Manager in Sensorsim*, . 1996, Lockheed Martin Missile & Space.
7. Zeigler, B.P., H. Sarjoughian, and W. Au. *Object-Oriented DEVS*. in *Enabling Technology for Simulation Science, SPIE AeoroSense 97*. 1997. Orlando, FL.
8. Defense, D.o., *High Level Architecture Interface Specification, Version 1.0*, . 1996, Defense Modeling and Simulation Organization,available via http://msis.dmso.mil.
9. Bassiouni, M.A., *et al*., *Performance and Reliability Analysis of Relevance Filtering for Scalable Distributed Interactive Simulation.* ACM Trans. on Model. and Comp. Sim. (TOMACS), 1997. **7**(3): p. 293-331.
10. Zeigler, B.P. and D. Kim. *Design of High Level Modelling / High Performance Simulation Environments*. in *10th Workshop on Parallel and Distributed Simulation*. 1996. Philadelphia.
11. Zeigler, B.P., D. Kim, and H. Praehofer. *DEVS Formalism as a Framework for Advanced Distributed Simulation*. in *First International Workshop on Distributed Interactive Simulation and Real Time Applications(in conjunction with MASCOTS'97 -- International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems)*. 1997. Eilat, Israel: IEEE Press, San Diego, CA.
12. Fujimoto, R.M., *Parallel Discrete Event Simulation.* Communications of the ACM, 1990. **33**(10).
13. Page, E.H., *Zero Lookahead in a Distributed Time-Stepped Simulation*, . 1998, MITRE: McLean, VA.
14. Fujimoto, R.M. *Zero Lookahead and Reeatability in High Level Architecture*. in *Proc. Spring Simulation Interoperability Workshop*. 1997. Orlando, FL.

9 Appendices

9.1 Appendix 1: Informal Review of the DEVS Formalism

We begin with an informal exposition of the DEVS formalism, focusing on the concepts rather than the formal underpinning. More detail is available in many references (e.g., [2, 3] also see URL: www-ais.ece.arizona.edu). A DEVS model is a modular system receiving inputs, changing states, and generating outputs over a time base. Input and output streams are time-indexed series of events. **Error! Reference source not found.** illustrates an input event segment containing inputs x_0 x_1 x_2 arriving at times t_0 t_1 t_2 respectively; and a output event segment with outputs y_0 y_1 departing at times t'_0 t'_1 respectively. There are no constraints on the spacing between events; however, only a finite number of events are allowed in a finite time interval.

Figure 32 Input and Output Event Streams

DEVS models have input and output ports through which all interaction with the external world takes place. By coupling together output ports of one system to input ports of another, outputs are transmitted as inputs and acted upon by the receiving system. Thus, there are two types of DEVS models, *atomic* and *coupled*. An atomic model directly specifies the system's response to events on its input ports, state transitions, and generation of events on its output ports. A coupled model is a composition of DEVS models that presents the same external interfaces as do atomic models. For example, in Figure 33, CM is a coupled model with four components. A coupled model specifies three types of coupling:

- *external input* – from the input ports of the coupled model to the input ports of the components (e.g., from start of CM to start of counter)
- *internal* – from the output ports of components to input ports of other components (e.g., from explosion of bomb to strike of target), and
- *external output* – from the output ports of components to output ports of the coupled model (e.g.. from damage of target to damage of CM.

Although arbitrary fan-out and fan-in of coupling is allowed, no self-loops are permitted. DEVS is closed under coupling, which means that a coupled model can itself be a component within a higher level coupled model, leading to hierarchical, modular model construction.

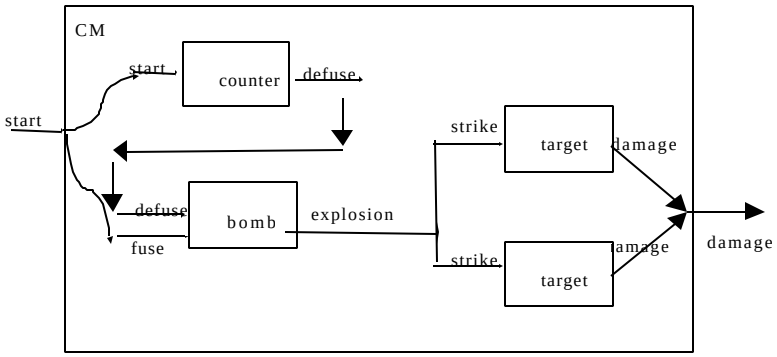


Figure 33. DEVS Coupled Model

To illustrate the DEVS modeling process, consider a scenario in which a bomb can threaten some objects in its vicinity called targets. If the bomb is left undisturbed over a long period it will lose its potency. If its fuse is ignited, it will in a very much shorter time, explode and impact the targets. However, if during this period, it is deactivated, the bomb will not go off but return to its dormant state. Many processes have similar *conditionally timed behavior.* For example, a grain can be dormant until picked up by a bee and transported to pollinate other flowers; but the bee may be killed before arriving at its destination thereby aborting the attempt at plant reproduction. A pre-emptive processor can be leave the job currently being done if interrupted by one of higher priority. A reliable network packet protocol waits for a fixed time for all packets in a message to be acknowledged by the receiver; otherwise it retransmits them (often called a timeout). The modular, state-based concept of DEVS enables it to respond such external inputs based on its current state and the time that has elapsed in that state.

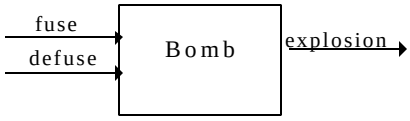


Figure 34 DEVS Atomic Model

To model this type of behavior in DEVS we define a coupled model, as in Figure 33, containing components for the bomb, some targets, and a defuser capable of deactivating the bomb. Each component is specified as a modular atomic model with input and output ports. In particular, the bomb has input ports for fusing (activating) and defusing (deactivating) it and an output port for the explosion event (Figure 34). The behavior over a continuous time line is illustrated in Figure 35. External input events are shown as vertical lines on the X time line – e.g., a fuse event occurs at time = 2 in Figure 35b. Phases (which are states labels such as dormant, dead, fused, etc.) are persistent as shown by constant lines and the transition from one phase to another are shown by jumps in the lines (as from domant to dead in Figure 35a). A DEVS system autonomously determines the *resting time* (also called *time advance*) in a phase as a function of that phase (actually of the state associated with the phase which may contain other variables). For

example, the time to stay in phase dormant is 150 after which time the system transitions to phase dead. The time that has passed in a phase is the elapsed time as depicted on the E access in Figure 35. The change in state occurring when the resting time has elaplsed is called an *internal event*. In a typical uniprocessor simulation program, an internal event is scheduled on an event list.

Due to its modularity, DEVS distinguishes between internal and *external* events. While internal events represent state changes that are self induced, *external input events*, on the other hand, are impressed from outside the model and are inputs to which the system must respond. For example, the arrival of a fuse event at time = 2 in Figure 35b causes the phase to change from dormant to live. *As part of the effect of an external input event, the timing of the next internal event may be changed*. In this case, the time advance associated with phase live is 10, which means that at time = 12, the phase is scheduled to change to explode (whereas before the external event, the dormant-to-dead transition was scheduled at 150). *Output external events* are generated by a system at its internal events. For example, the transition from explode to dead generates the explosion output event in Figure 35b.

Figure 35c illustrates the most critical conditional timing feature of the bomb example – this is the arrival of a defuse external event while the system is in phase live and counting down toward the explosion output. The defuse input causes the system to revert to phase dormant thus aborting the scheduled explosion. Note that from the bomb's point of view, the arrival of the defuse input is unpredictable, it could arrive in time (i.e., before the time elapses in phase live) or after this time has elapsed (including never). The DEVS formalism, which modular concept of external events, requires and allows explicit specification of the dynamics of such conditionally timed events.

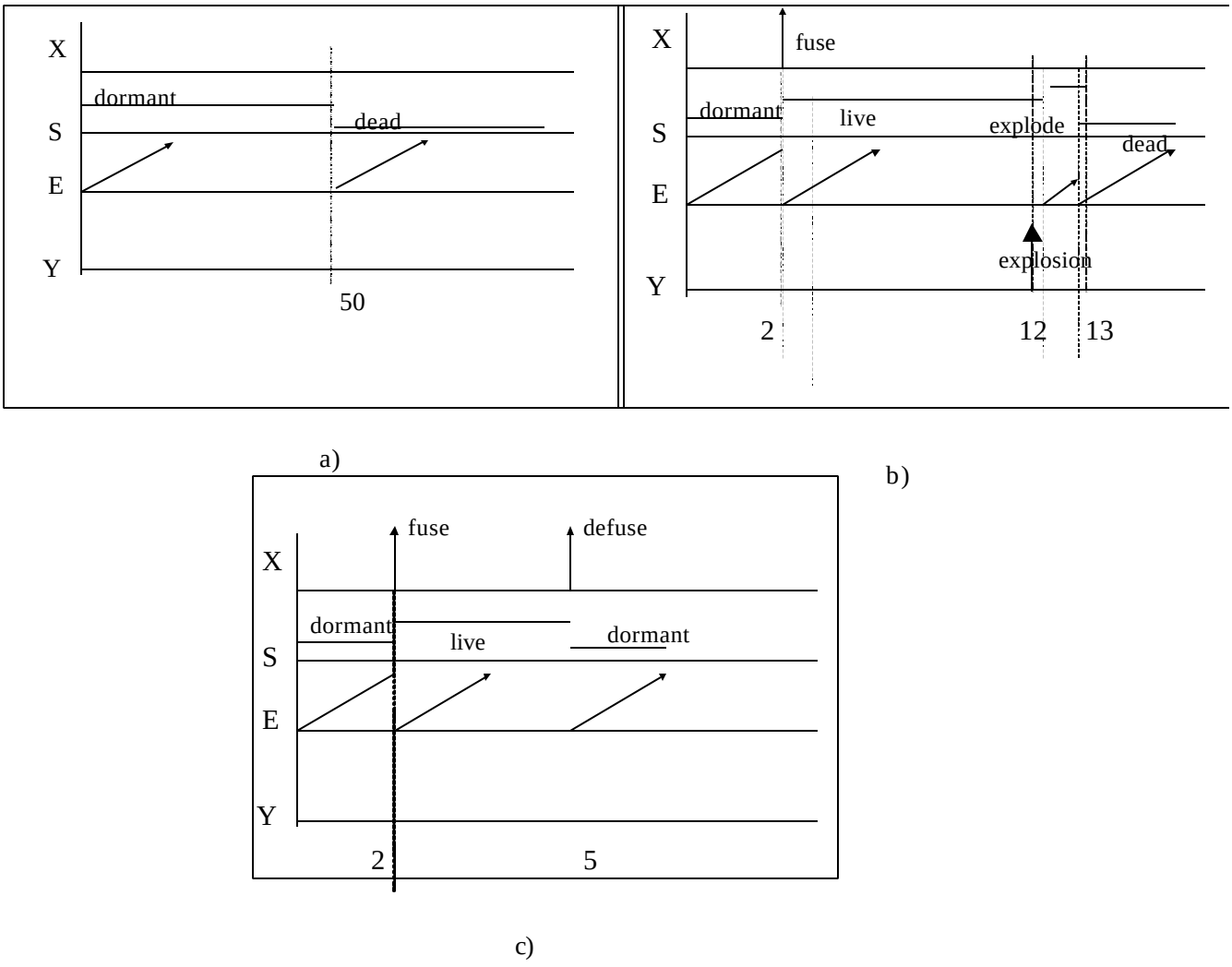


Figure 35 DEVS Temporal Behavior

Figure 36a illustrates a *phase transition diagram* that portrays the dynamics of an atomic model. The two kinds of transitions are shown – *external*, an arrow labeled by an internal port, and *internal* – a dashed arrow labeled with a time advance. For example, the transition from dormant to live is externally induced by the fuse external event. The transition from live to explode is an internal transition as is the transition from explode to dead.

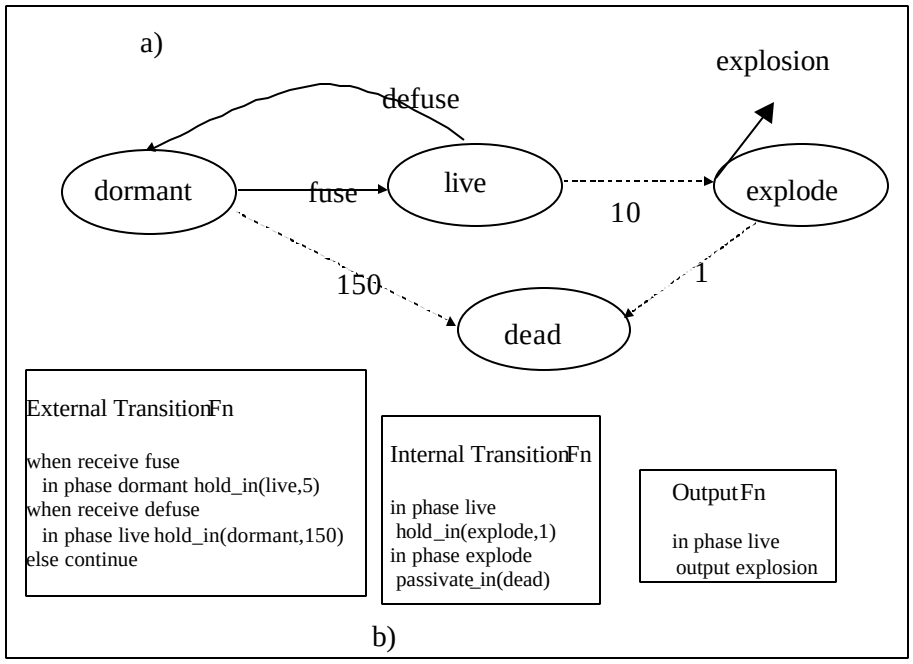


Figure 36 Atomic Model Specification

Such state diagrams are good for giving a graphical abstract representation of the dynamics but they have difficulty conveying their details. The authoritative specification of an atomic model is given in its internal transition, external transition, and output functions as in Figure 36b.

The External Transition Function specifies how the system responds to external events. In general, this response is a change in state conditioned on the nature of the external event (input port of arrival and parameters of the received message), the current state, and the time elapsed since the last event. For example, an arrival on port fuse changes the phase from dormant to live. But the time that the system has been dormant might degrade the power of the explosion. This might be modeled by including a state variable, potency, that would be reset by the external transition function to a decaying exponential function of its original value and the elapsed time in phase dormant.

The Internal Transition Function specifies the change in state that occurs upon an internal event. It is given as a function that maps the state before the event to the state that is to be in effect immediately after it.

The Output Function specifies the output port and value that will be generated just before an internal event. It is given as a function that maps the state before the event to the output space (set of possible outputs). Not all states need to generate outputs. For example, transitioning from live to explode generates the explosion output, while no other transitions generate output events.

Figure 36b does <u>not</u> show another characteristic DEVS feature, *the time advance function*. This assigns to every state the time in which the system will stay in this state before an internal event occurs. Rather then give this function explicitly, we allow a state variable, called *sigma,* to hold its value for the current state, and to have this value set by the external and internal transition functions. The command "hold_in (live, 10)" means set the phase to live and sigma to 10. The command "passivate_in(dead) means to set phase to dead and sigma to infinity. In other words, the system is scheduled to remain in dead forever[9]. It is also possible for a state to have a zero time advance. For example, we might change the time to go from explode to dead (the explosion time) from 1 to 0.

[9] unless an external event changes this), but an event capable doing this is not shown in Figure 36

9.2 Appendix 2: Issues in DEVS/HLA Design and Implementation

In this appendix we present:

- a simplified version of the DEVS simulation protocol that underlies the current HLA implementation
- a brief review of conservative and optimistic schemes for discrete event simulation
- consideration of the alternatives for implementing DEVS simulation – the issue here is that HLA is currently more oriented to the conservative scheme than to the direct DEVS protocol.
- a brief discussion of considerations and issues in current implementation of DEVS simulation in HLA

9.2.1 Parallel DEVS Simulation Protocol

DEVS has a well-defined semantics and associated simulation algorithm, which we present informally here[10]. Each component in a coupled model has two time keeping variables, tL (time of last event) and tN (time of next event). Before starting a simulation run, each component is initialized to its designated initial state and its time keeping variables set: tL= 0, tN = the time advance of the initial state.

We present a somewhat simplified version of a DEVS simulation protocol for a single level coupled model:

1. Set the current global time, t = the minimum of the components' tN's[11]
2. Send t to each component
3. Each component, c then compares t with its tN, if t= tN, this component is said to be imminent and
    - c generates its output (if any) stamped with time t
    - c executes its internal transition function
    - c sets tL = t and tN = tN + time advance of the new state
4. The collected outputs move, as dictated by the coupling specification, to the input ports of other components
5. Each component examines its input ports and:
    - if it receives an input, it applies its external transition function with this input, using the elapsed time, t - tL
    - sets tL = t and tN = tN + time advance of the new state

(if no input was received it does nothing)

1. if not at the end of the run, return to 1.

Figure 37a illustrates this process. Suppose the bomb is in state dormant, the countermeasure has time advance = 15, and the targets are in passive states (with infinite time advance). The bomb is then the only imminent component since it has the smallest tN = 10. Current time then advances to 10, and the bomb transitions to state explode, which having time advance = 1, tN becomes 11. There are no outputs to be sent.

In the next cycle, time is advanced to 11 and the bomb is still the imminent component. It generates the explosion output which the coupling directs to the targets. Since it passivates in dead, the bomb will no longer be imminent in the next cycle. Note that in this case, the countermeasure is too late to save the targets.

---

[10] The algorithm, somewhat simplified, is for a recently introduced version of DEVS called Parallel DEVS. Parallel DEVS removes constraints that originated with the sequential operation of early computers and hindered the exploitation of parallelism, a critical element in more modern computing.

[11] To increase parallelism, a "time granule" may be used here to equivalence next event times that are close enough to each other to be considered as simultaneous. This has a marked effect on performance [1].

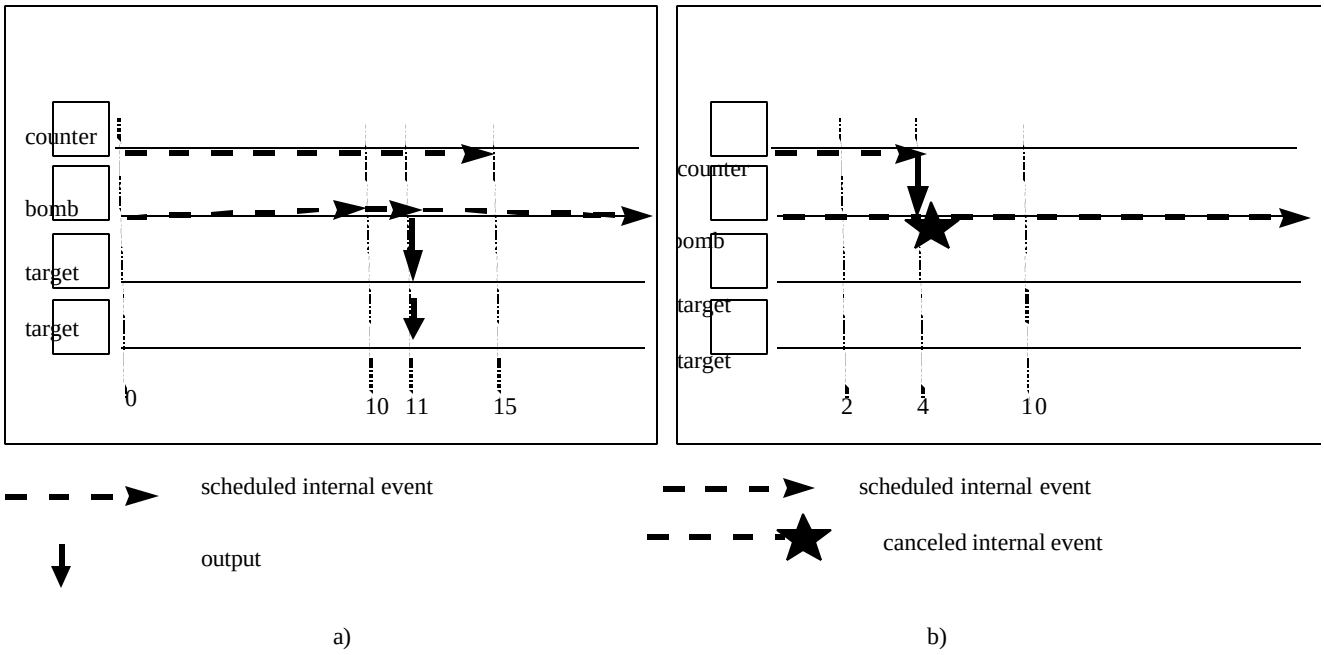


Figure 37 Coupled Model Simulation

Now consider a case in which the countermeasure acts in time as in Figure 37b. Here the countermeasure's initial tN = 4 and being less than the others, it is imminent. Time is advanced to 4 and the countermeasure generates a defuse output which the coupling sends to the bomb's defuse input port. Arrival of this input, causes the bomb's external transition function to execute, which sends it to phase dormant. As a consequence tL becomes 4 and tN becomes 154 (current time + time advance in dormant).

Note that in the last case, the scheduled internal event of the bomb did not materialize since an input interrupted it. This caused the bomb, in effect, to cancel its currently scheduled internal event, reschedule another one for a different time

9.2.2 How DEVS Relates to Parallel and Distributed Simulation Frameworks

The separation of concerns between modeling and simulation underlying the DEVS methodology is illustrated in Figure 38. 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[10, 11].

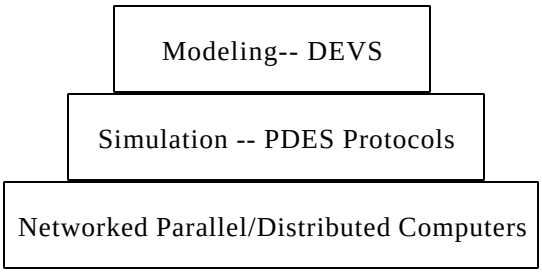


Figure 38 Layering in Modeling and Simulation

There are three approach to mapping the DEVS formalism into PDES protocols illustrated in Table 1. Two of these approaches are specializations of the more generic protocols, called *conservative* and *optimistic* schemes. The third is the direct mapping into a simulation algorithm as already illustrated earlier .

Scheme	**Approach**	**Overhead**	**Pros**	**Cons**
Chandy-Misra Conservative	Process events in strict time stamped order (Causality Preservation) w/ Deadlock Avoidance	Null Messages Lookahead Computation	Relatively Simple to Implement	Does not exploit all parallelism and not intrinsically load balancing
DEVS Version	Propagation Delay for Lookahead			Simultaneous Events are Problematic
Time Warp Optimistic	Permits Causality Violation Detects Violations and Remedies using Rollback	State, Message Saving, Fossil Collection Anti-messages Global Virtual Time Computation	Can exploit feedback free couplings	Complex logic is difficult to implement and verify
Riskfree DEVS Version	Refrain from Output until Safe	Same as Time Warp except no Anti-messages and Rollback is local	Relatively Easy to Implement	Simultaneous Events are Problematic
Parallel DEVS	Riskfree and Strict Causality Adherence	Global Minimum Time Synchronization Simultaneous Output Collection	Easy to Implement Exploits Simultaneous Events Works well for active models with feedback couplings	Does not exploit parallelism in feedback free couplings

Table 1 Comparison of PDES Approaches

In the conservative and optimistic schemes simulation is viewed as moved forward by the processing of time-stamped messages. As depicted in Figure 39, such logical processors have input events queued in order of earliest time-stamp. Two laws govern processing:

- As a result of processing an input event, logical processors are assumed to produce output messages whose time-stamps are not earlier than the input time-stamp (processing can't proceed backwards in time).
- Messages must be processed in order of time-stamps in the queues – schemes differ in how they treat this constraint.

In conservative schemes the time-stamped order constraint is never violated. Optimistic schemes allow temporary violation that must be repaired before the final simulation output is presented. The conservative approach is illustrated in Figure 39, where logical processor LP1 cannot process its next input (a,3) because there is potential for an earlier message from LP2 due to the presence of input (d,1) in its queue. Conservative schemes must somehow arrange for the potential for input events with earlier time stamps to be conveyed to affected processors. This can be done through "lookahead" in which each LP provides a time in the immediate future up to which it promises not to send input events. The minimum of such blackout times at any LP, called the Lower Bound Stamp Time, is the time up to which it can safely process its time-stamped inputs. Thus, simulation proceeds incrementally governed by the lookahead, which is the interval that an LP adds to its current Lower Bound Stamp Time to obtain the blackout time sent to other LPs. In the example, the lookahead for LP2 is 1 and the only way that time can advance is for LP2 to process its input (d,1) which results in the message (d',2) sent to LP1 as shown. Had the lookahead for LP2 been 4 and that for LP1 = 1, then LP1 would have been the first to process its input event, (a,3). Lookahead is difficult to find in representations of reality [12] and large lookahead values are needed to gain advantages over sequential simulation [13].

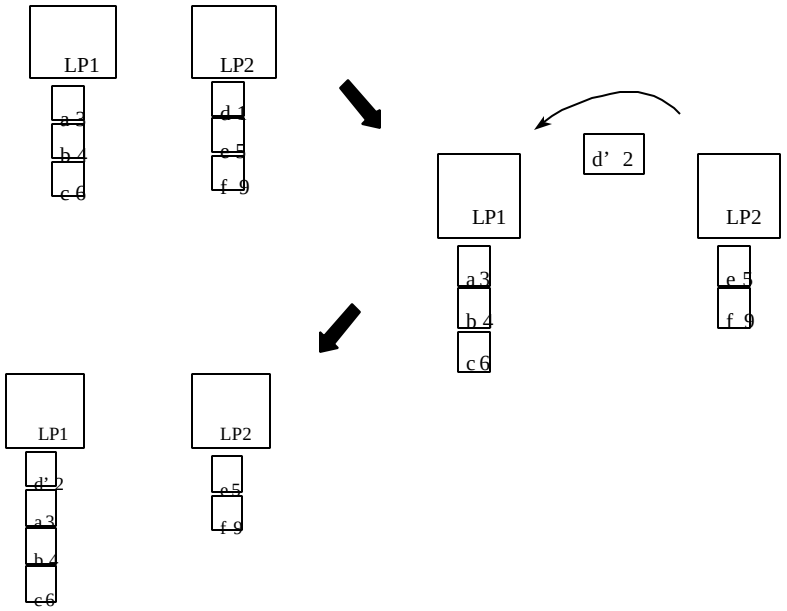


Figure 39 Conservative Scheme

Optimistic schemes allow LPs to march forward in local time and process their input queues as fast as they can. Thus, as in Figure 40, LP1 and LP2 have arrived at the situation shown where LP2 has processed events (d,1) and (e,5) and sent input events (d',5) and (e',6) to LP1. Now, LP1 processes event (a,3) which

causes it send an input (a',3 to LP2 as shown. However, since LP2 has already processed event (e,5), the new input (a',3) is called a straggler since it is out of place in the time-stamped order of processing. To rectify this situation, queues of already processed inputs and their outputs are maintained so that the situation can be restored to what it was just before the arrival of the straggler. This involves sending "anti-messages" such as (e',6) that annihilate the effects of already sent messages and "rolling back" processors' states to those prevailing just before the straggler's detection (this also requires state saving). In the example, note that the processing of (d,1) does not need to be rolled back since it is time-stamped earlier than straggler (a',3). You can see that an extensive apparatus of overhead must be maintained to make this all work offering many opportunities for optimization and investigation by computer scientists. One scheme, called "risk free", limits rollback to only the local processor by forcing LPs to refrain from sending outputs until it is safe to do –while allowing them to proceed unfettered with processing of the input queues.

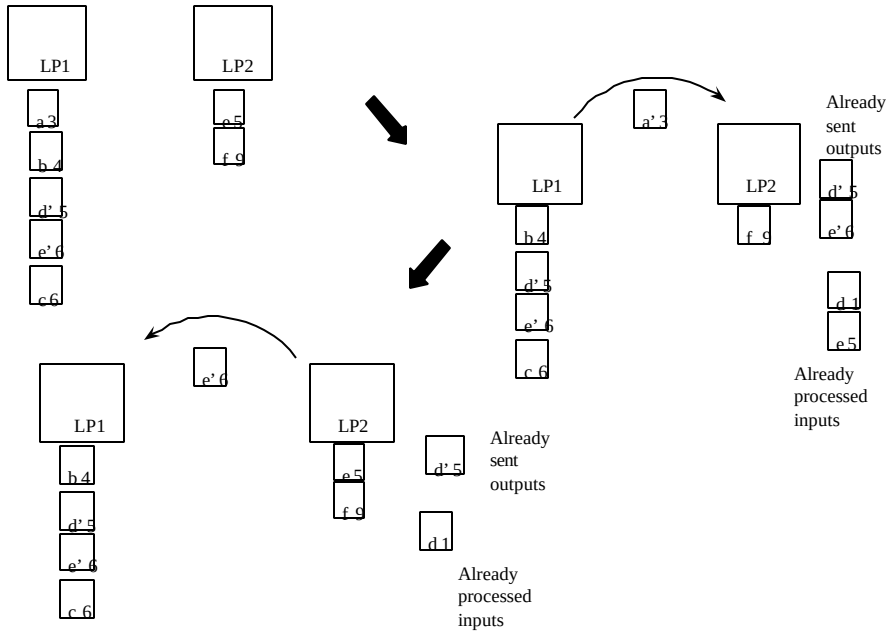


Figure 40 Optimistic Scheme

A DEVS component in a coupled model can map into a logical processor with some adjustments. External input and output events correspond to those handled by LPs. However, internal events are not represented in the LP framework. To include internal events, we can consider them as input events that an LP sends back to itself for processing. However, depending on how it is done, this may add to the traffic load on the underlying communications network. Moreover, while DEVS has specific means of dealing with simultaneous events, most schemes avoid these like the plague. Therefore, as indicated in Table 1, adapting versions of the LP-based schemes to DEVS has been one of the preferred ways to implement DEVS distributed and parallel simulators.

In the third, direct approach discussed in Table 1, simulators are designed based directly on the DEVS formalism. The Parallel DEVS algorithm in Section 9.2.1 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 illustrated in Figure 41, 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 time of next event from the component simulators. It sends the minimum of these

times back to the federates, thereby allowing them to determine whether they are imminent, and if so to generate output and send it to components according to the coupling rules. The transition functions of the imminent components may cause new values for the time advance function, which are sent to the coordinator, and the cycle continues.

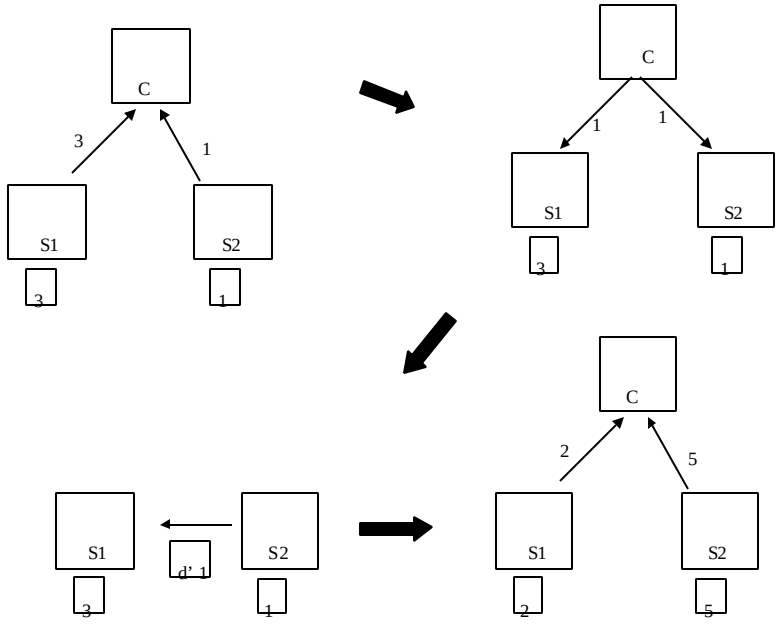


Figure 41 Parallel DEVS Simulator

9.2.3 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, interations)) 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 in Table 1 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 in the RTI version 1.0.

Figure 42 presents the logic of the current 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 paramater = tN. If there are no events in the interim, the RTI issues a timeAdvGrant at 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. It then issues a new nextEventRequest and waits for the arrival of interactions and updates that will arrive from imminent federates in the next cycle.

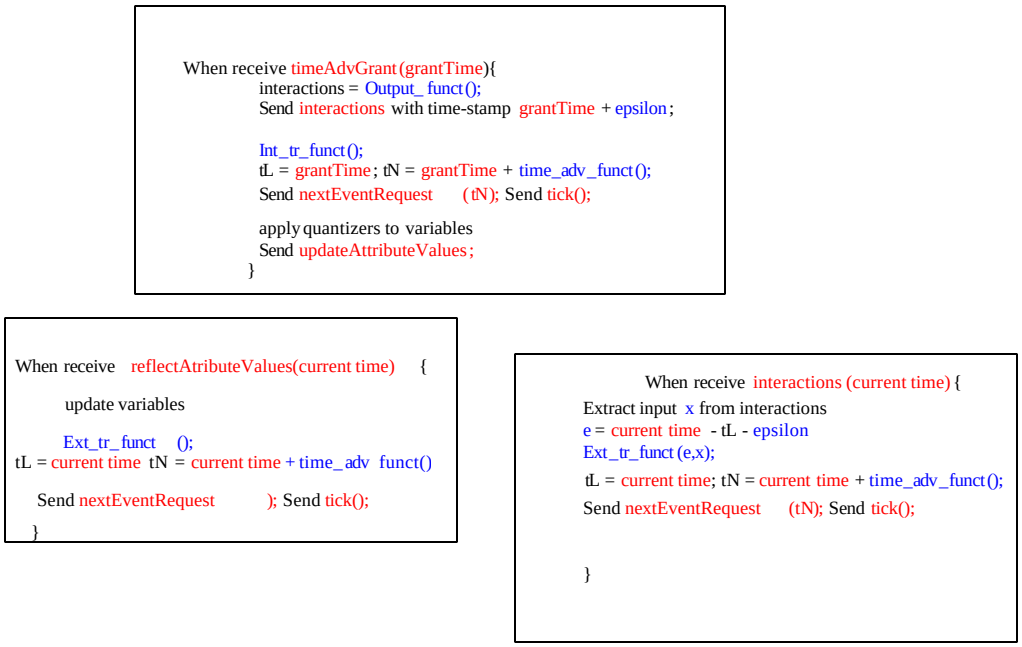


Figure 42 Mapping DEVS Simulation to HLA/RTI

Three considerations enter into the implementation.

- The outputs of DEVS components are time stamped with the current logical time. Recently, HLA has been extended to allow sending interactions with time equal to the current logical time using a concept of "availability" for nextEventRequest[14]. However, we have found that use of this approach slows up execution considerably (at least with the current RTI). Therefore we have adopted an alternative in which the outputs generated by the imminents are embedded in interactions that are stamped slightly in the future of the federate. As illustrated in the bomb example, when the bomb is imminent it sends outputs to the targets when it gets a time advance grant. These are in the form of interactions with the time slightly increased (epsilon in Figure 42) from that of the time advance grant. When a receiving federate receives interactions it subtracts the known artifact to correctly obtain the elapsed time needed by its external transition function.

- The need for rescheduling upon receipt of an interaction in our implementation is illustrated in Figure 43. In the case on the right (b), the bomb receives a defuse input from the countermeasure, causing it to apply its external transition function. As a consequence of a change in state, the time-of-next-event is now different necessitating cancellation of the current nextEventRequest and issuing of a new one with

a different time. The description given by Richard Fujimoto (personal communication) requires that a nextEventRequest is automatically cancelled by an interaction arriving at a time before the cutoff time. However, we are receiving an error indicating an attempt to execute two RTI ambassador commands concurrently. This suggests that the original nextEventRequest is not being cancelled as specified. We are continuing to work with HLA authorities on resolving this issue.[12]

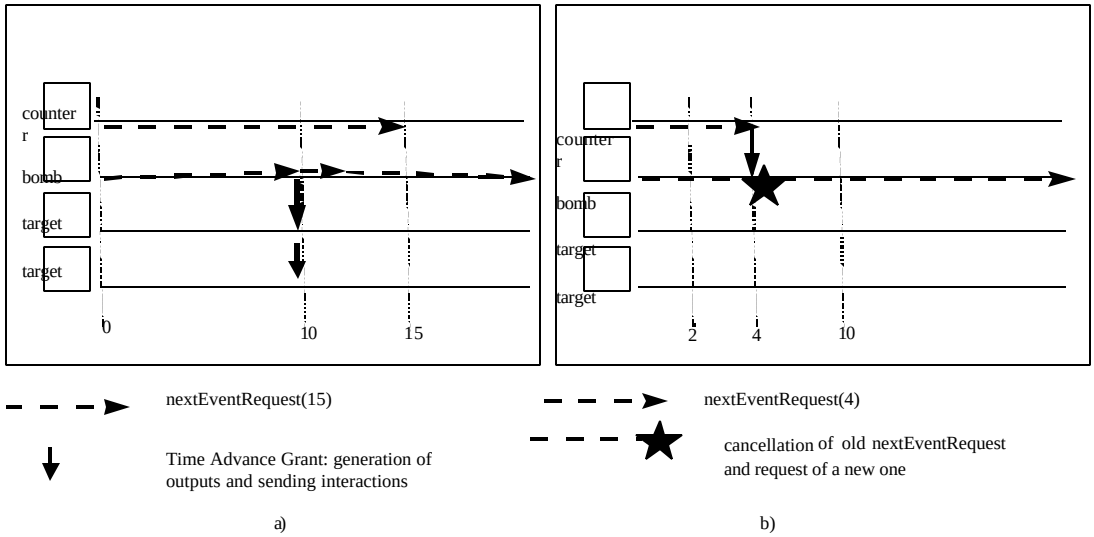


Figure 43 Illustrating the Need for Rescheduling

[12] Another approach is possible in which a method for retracting time advance requests would be added to the HLA specification. This would provide a direct means of achieving the desired functionality. The reason for its exclusion is that in the default RTI protocol, federates "promise" not issue output events during their time advances plus lookahead. Were the time advance to be reduced than this promise might be broken by an earlier than promised output. Indeed, as we have seen, the DEVS formalism allows federates to send outputs to others with later time advances but provides the external transition function for their proper handling (without optimistic processing). Outputs are generated only at internal transitions (hence when a timeAdvGrant occurs) and these are packaged in interactions for the same time (ideally) or slightly in the future. The processing of an interaction causes a new time advance, which is always for a time in the future. Thus an out-of-time-stamp-order interaction will never be generated. Also the RTI need never contend with events stamped with different times since only output events stamped with the current time (or slightly greater) are ever generated as interactions. Thus, allowing a retraction of time advance with a new one for the future can cause no harm when used to implement the DEVS protocol.

The issue can be raised of the effect of allowing retraction on other protocols, including the default scheme employed in the current DMSO RTI. As indicated, this scheme employs both time advance and lookahead to control output event generation and allowing reduction of time advance might enable such promises to be broken. However. it does not follow that this would necessarily be harmful since the parallel DEVS protocol offers a counter-example. Also, even if it were harmful in the some protocols, including the default, there would be no obligation to employ the retraction method in such schemes. Time management modelers would have an option to employ the method with responsibility to use it safely. One could argue that retraction-employing federates might not be able to play with federates using other protocols under these circumstances. However, this is too strong a criterion since HLA is intended to be flexible in accommodating a variety of styles in the same federation. This means that the time management protocol must be decided for each federation anew – a compatible overall protocol must be arrived at that accommodates the individual protocols of the federates (which may have to be modified to do so).

- Since attribute updates are supported in DEVS/HLA but are not a feature of the standard DEVS formalism, the issue of how to handle them arises. In the implementation we have overloaded the external transition function method so that it accepts a signature with zero arguments (the existing method has two arguments). When an update occurs this zero-argument version of the external transition method is called, allowing the modeler to prescribe a response to the arriving update. We are experimenting with alternatives, for example to incorporate the updated attribute and/or the elapsed time, into the signatureOne alternative is to send all updates as interactions, thus requiring no additions to standard DEVS, however, this would not exploit the services provided by the HLA. . Experience will dictate which subset of the alternatives will be most applicable.

9.3 Appendix 3: DEVS/HLA Source Code Example: Pursuer/Evader Federation

```
//==================================

//   mainPursuer.cpp

//==================================


#include <time.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>
#include <RTI.hh>
#include <Digraph.hh>
#include <HLAport.hh>
#include <Federate.hh>
#include "FederatePursWQuant.hh"
#include "PursWQuant.hh"

//#define DEBUG

Federate * myFederate;

int main(int argc, char *argv[])
{
  const char* exeName = argv[0];        // Name of executable process
  int iterationCnt;

  if(argc <= 1)
iterationCnt = 10000000; // default number of iteration
  else
iterationCnt = atoi(argv[1]);


  //-------------------------------------------------------------------------

  // create hightest model and FederatePursWQuant

  //-------------------------------------------------------------------------

  myFederate = new FederatePursWQuant("FederatePursWQuant");

  //-------------------------------------------------------------------------

  // run my Federate
```

```
  //---------------------------------------------------------------------------

  myFederate->run(argc,iterationCnt);
  return 0;
}
```

```
//==================================

//   FederatePursWQuant.hh

//==================================



#ifndef _FederatePursWQuant_HH_
#define _FederatePursWQuant_HH_
#include <objectClass.hh>
#include <Federate.hh>
#include "PursWQuant.hh"

// methods

class FederatePursWQuant: public Federate {

public:

   set *myObjects;

   FederatePursWQuant( char* name );

   ~FederatePursWQuant() ;

};

#endif
```

```
//==================================

//   FederatePursWQuant.cpp

//==================================



#include "FederatePursWQuant.hh"
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>
#include "RedTank.hh"
#include "Driver.hh"
#include "PursWQuant.hh"

//#define DEBUGG

FederatePursWQuant::FederatePursWQuant( char* name ) : Federate(name) {

#ifdef DEBUGG
 cerr<<"FederatePursWQuant [" << getName() << "]: constructor " << endl;
#endif

  myObjects= new set();

  HLAport *hp = new HLAport("fire");
  parameter *px = new parameter("XD");
  parameter *py = new parameter(" YD");
  hp->addParameter(px);
  hp->addParameter(py);
  setUp(hp);


  PursWQuant *myPWQ = new PursWQuant("PursWQuant");
  addCoupling(myPWQ, "positionX", this, hp, px);
  addCoupling(myPWQ, "positionY", this, hp, py);
  add(myPWQ);


  // ---------------------------------------------

  // create Object class(es)

  // ---------------------------------------------
  // gather


  //for each objectClass

  objectClass *RedTankObj = new objectClass("RedTank");     //create an object class

  objectClass *BlueTankObj = new objectClass("BlueTank");     //create an object class
```

```
  //for each attribute

  RedTankObj->addAttribute(new attribute("XDirection",0,NULL));
  RedTankObj->addAttribute(new attribute("YDirection",0,NULL));
  RedTankObj->addAttribute(new attribute("ZDirection",0,NULL));
  setUp(RedTankObj);

  BlueTankObj->addAttribute(new attribute("XDirection",0,NULL));

  BlueTankObj->addAttribute(new attribute("YDirection",0,NULL));

  BlueTankObj->addAttribute(new attribute("ZDirection",0,NULL));

  setUp(BlueTankObj);
  myObjects->add(RedTankObj); // for future use
  myObjects->add(BlueTankObj); // for future use

#ifdef DEBUGG
  cerr<<"FederatePursWQuant [" << getName() << "]: End of constructor " << endl;
#endif
}



FederatePursWQuant::~FederatePursWQuant() {
}
```

```
//=============================================

//   PursWQuant.cpp (Pursuer with Quantizer)

//=============================================



#include "PursWQuant.hh"
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>
#include <mess.h>
#include <HLAport.hh>
#include "RedTank.hh"
#include "Driver.hh"
#include "BlueTankEndo.hh"
//#define DEBUG

// methods

PursWQuant::PursWQuant( char* name ) : Digraph(name) {

#ifdef DEBUG
  cerr<<"PursWQuant [" << getName() << "]: constructor " << endl;
#endif



   // ------------------------------------------------

   // create components

   // ------------------------------------------------



   RedTank *RedTK = new RedTank("RedTank");

   Driver *drv = new Driver("DRIVER");

   BlueTankEndo *BlueTK = new BlueTankEndo("BlueTank");

   // ------------------------------------------------

   // add components

   // ------------------------------------------------

    add(RedTK);
    add(drv);
    add(BlueTK);
    // ------------------------------------------------
```

```
  // add coupling

  // ------------------------------------------------

  addCoupling(drv, "out", RedTK, "in");
  addCoupling(drv, "fireout", this, "fireout");
  addCoupling(RedTK, "out", drv, "mine");
  addCoupling(BlueTK, "out", drv, "other");



  #ifdef DEBUG
     // print statements
     cerr << "["<<getName()<<"] Coupling Information: " << endl;
     getCoupling()->print();
     cerr<<"PursWQuant [" << getName() << "]: End of constructor " << endl;

  #endif
}


PursWQuant::~PursWQuant() {
}
```

```
//=============================================

//   PursWQuant.hh

//=============================================



#ifndef _PursWQuant_HH_
#define _PursWQuant_HH_
#include <Digraph.hh>
// methods

class PursWQuant: public Digraph {

public:
   PursWQuant( char* name );
   ~PursWQuant() ;

};

#endif
```

```
//============================================

//   Driver.cpp

//============================================



#include "Driver.hh"
#include <numEnt.h>
#include <stdlib.h>
#include <iostream.h>
#include <fstream.h>
#include <DimTraj.hh>
//#define DEBUG

// methods

Driver::Driver( char* name ) : Atomic(name) {

#ifdef DEBUG
  cerr<<"[" << getName() << "]: constructor " << endl;
#endif


  phases->add("chase");
  phases->add("retreat");
  phases->add("rest");
  inports->add("mine");
  inports->add("other");


   myX = myY = myZ = 0;
  otherX = otherY = otherZ = 0;
  Xflag = Yflag = Zflag = 0;

  fstream fdelta("delta.in", ios::in);
  fdelta >> deltaTime;

  passivate();

}

Driver::~Driver() {
}

void Driver::deltext(RTI::FederationTime updateTime, message *x) {

#ifdef DEBUG
  cerr<<"[" << getName() << "]: deltext() " << endl;
  x->print();
#endif

   int valX, valY, valZ, otherFlag;
   otherFlag = 0;
```

```
   double shadowSize = 2;
   entity *ent;

   for (int i=0; i< x->getLength();i++) {
if (messageOnPort(x,"mine",i)) {
    ent = x->getValOnPort("mine",i);
          threeDimTraj *threeDT = (threeDimTraj *)ent;
          myX = threeDT->getX();
          myY = threeDT->getY();
          myZ = threeDT->getZ();
          #ifdef DEBUG
          cerr << "myPosition: " << myX <<"   " << myY<<"   " << myZ << endl;
         #endif
}

else if (messageOnPort(x,"other",i)) {
    ent = x->getValOnPort("other",i);
          threeDimTraj *threeDT = (threeDimTraj *)ent;
          otherX = threeDT->getX();
          otherY = threeDT->getY();
          otherZ = threeDT->getZ();
          otherFlag = 1;
          #ifdef DEBUG
          cerr << "otherPosition: "<<otherX<<"  "<< otherY<<"  " <<otherZ<<endl;
          #endif
}

else  {

    cerr << "[" << getName() << "] PANIC: port is not \"mine\" or \"other\"" << endl;
    cerr << endl;

}

      int changeFlag = 0;
      Xflag = Yflag = Zflag = 0;
      if(fabs(myX-otherX) > shadowSize) {
          if(myX>otherX) Xflag = -1;
          else  Xflag = 1;
          changeFlag = 1;
      }

      if(fabs(myY-otherY) > shadowSize) {
          if(myY>otherY) Yflag = -1;
          else  Yflag = 1;
          changeFlag = 1;
     }

      if(fabs(myZ-otherZ) > shadowSize) {
          if(myZ>otherZ) Zflag = -1;
          else  Zflag = 1;
          changeFlag = 1;

      }

      if(changeFlag > 0 && otherFlag > 0) {
```

```
        holdIn("active", deltaTime);
}

   }

#ifdef DEBUG
  cerr<<"End of [" << getName() << "]: deltext() " << endl;
#endif
}

message * Driver::out() {
#ifdef DEBUG
  cerr<<"[" << getName() << "]: out() " << endl;
#endif
   message *m = new message();
   content *con;
   if(Xflag!=0 || Yflag!=0 || Zflag!=0) {
      con = makeContent("out", new threeDimTraj(Xflag,Yflag,Zflag));
      m->add(con);
   }

   // for interaction communication. the value is meaningless
   if(((int)myX)%4 == 0) {
      con = makeContent("fireout", new twoDimTraj(7777));
      m->add(con);
   }

   if(! m->empty())
m->print();

#ifdef DEBUG
  cerr<<"End of [" << getName() << "]: out() " << endl;
#endif
return m;
}

void Driver::deltint() {
#ifdef DEBUG
  cerr<<"[" << getName() << "]: deltint() " << endl;
#endif
   Xflag=Yflag=Zflag=0;       // reset flags
   passivate();

#ifdef DEBUG
  cerr<<"End of [" << getName() << "]: deltint() " << endl;
#endif
}


void  Driver:: deltcon(timetype e,message * x) {
   deltint();
   deltext(e, x);
}
```

```
//=============================================

//   Driver.hh

//=============================================



#ifndef _DRIVER_HH_
#define _DRIVER_HH_

#include <Atomic.hh>



class Driver : public Atomic

{

public:

  // variables

  double myTime, myX, myY, myZ, deltaTime, deltaX, deltaY, deltaZ;
  double otherX, otherY, otherZ;
  double Xflag, Yflag, Zflag;

  // methods

  Driver( char* name );
  ~Driver();


  message * out();
  void deltint();
  void deltext(RTI::FederationTime updateTime, message *m);
  void deltcon(timetype e,message * x);
};



#endif
```

```
//============================================

//   RedTank.cpp

//============================================



#include "RedTank.hh"
#include <port.h>
#include <numEnt.h>
#include <stdlib.h>
#include <fstream.h>
#include <iostream.h>
#include <DimTraj.hh>

//#define DEBUG


// methods

RedTank::RedTank( char* name ) : Atomic(name) {
#ifdef DEBUG
  cerr<<"RedTank [" << getName() << "]: constructor " << endl;
#endif

    setAtomicClassName(name);
    myTime = myX = myY = myZ = 0;
    fstream fdelta("delta.in", ios::in);
    fdelta >> deltaTime;
    fdelta >> deltaX;
    fdelta >> deltaY;
    fdelta >> deltaZ;

    double quantum;
    fstream fqt("quantum.in", ios::in);
    fqt >> quantum;


cerr << "quantum: " << quantum << endl;

    quantizer *qtzer = new quantizer("Time", 0.0, quantum);
    attribute *at = new attribute("Time", &myTime, qtzer);
    attributeList->add(at);
    qtzer = new quantizer("XDirection", 0.0, quantum);
    at = new attribute("XDirection", &myX, qtzer);
    attributeList->add(at);

    qtzer = new quantizer("YDirection", 0.0, quantum);
    at = new attribute("YDirection", &myY, qtzer);
    attributeList->add(at);

    qtzer = new quantizer("ZDirection", 0.0, quantum);
    at = new attribute("ZDirection", &myZ, qtzer);
```

```
   attributeList->add(at);
   passivate();

}




RedTank::~RedTank() {
}


void RedTank::deltext(RTI::FederationTime updateTime, message *x)

{
#ifdef DEBUG
  cerr<<"RedTank [" << getName() << "]: deltext() " << endl;
#endif

   timetype e = updateTime;

   Continue();

   if(phaseIs("passive")) {
     entity *ent;
     for (int i=0; i< x->getLength();i++) {
     if (messageOnPort(x,"in",i)) {
     ent = x->getValOnPort("in",i);
           threeDimTraj *threeDT = (threeDimTraj *)ent;
           if(threeDT->getX() > 0)
              myX += deltaX;
           else if(threeDT->getX() < 0)
              myX -= deltaX;
           if(threeDT->getY() > 0)
              myY += deltaY;
           else if(threeDT->getY() < 0)
              myY -= deltaY;
           if(threeDT->getZ() > 0)
              myZ += deltaZ;
           else if(threeDT->getZ() < 0)
              myZ -= deltaZ;
           holdIn("active", deltaTime);

    #ifdef DEBUG
           cout << "["<<getName()<<"] myX:"<<myX<<", myY:"<<myY<<", myZ:"<<myZ<<endl;
     #endif

  }

     }

   }

   else {
cerr<<"[" << getName() << "] phase: " << phase << endl;
```

```
cerr << "Not accept during ACTIVE phase" << endl;
   }


#ifdef DEBUG
  cerr<<"End of RedTank [" << getName() << "]: deltext() " << endl;
#endif
}



message * RedTank::out() {

#ifdef DEBUG
  cerr<<"RedTank [" << getName() << "]: out() " << endl;
#endif
   message *m = new message();
   content *con;
   con = makeContent("out", new threeDimTraj(myX,myY,myZ));
   m->add(con);
   if(! m->empty())
      m->print();
   cerr << endl;

#ifdef DEBUG
   cerr<<"End of RedTank [" << getName() << "]: out() " << endl;
#endif
   return m;
}

x
void  RedTank::deltint() {
#ifdef DEBUG
  cerr<<"RedTank [" << getName() << "]: deltint() " << endl;
#endif

   passivate();
}




void  RedTank::deltcon(timetype e,message * x) {
   deltint();
  deltext(e,x);
}
```

```
//===========================================

//   RedTank.hh

//===========================================

#ifndef _REDTANk_HH_
#define _REDTANk_HH_
#include <Atomic.hh>

class RedTank : public Atomic
{
public:
  // variables

   double myTime, myX, myY, myZ;
  double deltaTime, deltaX, deltaY, deltaZ;

  // methods

  RedTank( char* name );
  ~RedTank();

  message * out();
  void deltint();
  void deltext(RTI::FederationTime updateTime, message *m);
  void deltcon(timetype e,message * x);
};


#endif
```

```
//=============================================

//   BlueTankEndo.cpp

//=============================================



#include "BlueTankEndo.hh"
#include <port.h>
#include <numEnt.h>
#include <stdlib.h>
#include <fstream.h>
#include <iostream.h>
#include <DimTraj.hh>

//#define DEBUG

// methods
BlueTankEndo::BlueTankEndo( char* name ) : Atomic(name) {
#ifdef DEBUG
  cerr<<"BlueTankEndo [" << getName() << "]: constructor " << endl;
#endif


   setAtomicClassName("BlueTank");
    myX = myY = myZ = 0;
   addInport("stop");

   attribute *atT = new attribute("Time", &myTime, NULL);
   attribute *atX = new attribute("XDirection", &myX, NULL);
   attribute *atY = new attribute("YDirection", &myY, NULL);
   attribute *atZ = new attribute("ZDirection", &myZ, NULL);
   attributeList->add(atT);
   attributeList->add(atX);
   attributeList->add(atY);
   attributeList->add(atZ);



   fstream fin("delta.in", ios::in);
   fin >> PROC_TIME;
   holdIn("active", PROC_TIME);
 #ifdef DEBUG
   cerr<<"BlueTankEndo [" << getName() << "]: End of constructor " << endl;
#endif

}



BlueTankEndo::~BlueTankEndo() {
}
```

```
void BlueTankEndo::deltext(RTI::FederationTime updateTime, message *x) {
#ifdef DEBUG
  cerr<<"[" << getName() << "]: deltext() " << endl;
#endif
}



message * BlueTankEndo::out() {

#ifdef DEBUG
  cerr<<"BlueTankEndo [" << getName() << "]: out() " << endl;
#endif

   message *m = new message();
   content *con;
   if(stopFlag < 1) {
      con = makeContent("out",new threeDimTraj(myX,myY,myZ));
      m->add(con);
   }

   cerr << endl;
   if(! m->empty())
       m->print();
   cerr << endl;



#ifdef DEBUG
  cerr<<"End of BlueTankEndo [" << getName() << "]: out() " << endl;
#endif
   return m;
}




void  BlueTankEndo::deltint() {

#ifdef DEBUG
  cerr<<"BlueTankEndo [" << getName() << "]: deltint() " << endl;
#endif

   if(stopFlag > 0 || myTime >= INFINITY) {
passivate();
   }

   else
      holdIn("active", PROC_TIME);
}
```

```
//=============================================

//   BlueTankEndo.hh

//=============================================



#ifndef _BlueTank_ENDO_HH_
#define _BlueTank_ENDO_HH_
#include <Atomic.hh>

class BlueTankEndo : public Atomic
{
public:

  // variables

  double myTime, myX, myY, myZ;
  int stopFlag;
  double PROC_TIME;

  // methods
  BlueTankEndo( char* name );
  ~BlueTankEndo();

  message * out();
  void deltint();
  void deltext(RTI::FederationTime updateTime, message *m);
};


#endif
```

Source codes for a Evader Federate

```
//============================================

//   Driver.cpp

//============================================

//mainEvader.cpp

#include <time.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>
#include <RTI.hh>
#include <Digraph.hh>
#include <HLAport.hh>
#include "FederateEvadWEndo.hh"
#include "EvadWEndo.hh"
#include <Federate.hh>

//#define DEBUG

Federate * myFederate;

int main(int argc, char *argv[])
{
  const char* exeName = argv[0];        // Name of executable process
  int iterationCnt;

  if(argc <= 1)
iterationCnt = 10000000; // default number of iteration
  else
iterationCnt = atoi(argv[1]);

  //-------------------------------------------------------------------------

  // create FederateEvadWEndo and internal DEVS models

  //-------------------------------------------------------------------------

  myFederate = new FederateEvadWEndo("FederateEvadWEndo");

  //-------------------------------------------------------------------------

  // run my Federate and its internal DEVS models

  //-------------------------------------------------------------------------

  myFederate->run(argc, iterationCnt);
  return 0;
}
```

```
//=============================================

//   FederateEvadWEndo.hh

//=============================================

#ifndef _FederateEvadWEndo_HH_
#define _FederateEvadWEndo_HH_
#include <HLAport.hh>
#include <objectClass.hh>
#include <Federate.hh>
#include "RedTankEndo.hh"
#include "EvadWEndo.hh"

// methods

class FederateEvadWEndo: public Federate {

public:

   // object class
   //objectClass *RedTankObj;

   // DEVS class

   //EvadWEndo *myEWE;
   //HLAport *FirePort;

   set *myObjects;

   FederateEvadWEndo( char* name );
   ~FederateEvadWEndo() ;
};


#endif
```

```
//=============================================

//   FederateEvadWEndo.cpp

//=============================================



#include "FederateEvadWEndo.hh"
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>

#include <mess.h>
#include <RedTankEndo.hh>
#include <perceive.hh>
#include <BlueTank.hh>
#include <EvadWEndo.hh>

//#define DEBUGG

// methods

FederateEvadWEndo::FederateEvadWEndo( char* name ) : Federate(name) {

#ifdef DEBUGG
  cerr << endl;
  cerr<<"FederateEvadWEndo [" << getName() << "]: constructor " << endl;
#endif


   myObjects= new set();

   HLAport *hp = new HLAport("fire");
   addInport(hp);
   setUp(hp);

   EvadWEndo *myEWE = new EvadWEndo("EvadWEndo");
   addCoupling(this, hp, myEWE, "firein"); // data in
   add(myEWE);



   // ---------------------------------------------

   // create Object class(es)

   // ---------------------------------------------



   objectClass *RedTankObj = new objectClass("RedTank");     //create an object class
   RedTankObj->addAttribute(new attribute("XDirection",0,NULL));
```

```
  RedTankObj->addAttribute(new attribute("YDirection",0,NULL));
  setUp(RedTankObj);


  objectClass *BlueTKObj = new objectClass("BlueTank");     //create an object class
  BlueTKObj->addAttribute(new attribute("XDirection",0,NULL));
  BlueTKObj->addAttribute(new attribute("YDirection",0,NULL));
  setUp(BlueTKObj);

  myObjects->add(BlueTKObj); // for future use
  myObjects->add(RedTankObj); // for future use


#ifdef DEBUGG
  cerr<<"FederateEvadWEndo [" << getName() << "]: End of constructor " << endl;
#endif
}


FederateEvadWEndo::~FederateEvadWEndo() {

}
```

```
//=============================================

//   EvadWEndo.hh

//=============================================



#ifndef _EvadWEndo_HH_
#define _EvadWEndo_HH_

#include "HLAport.hh"
#include <Digraph.hh>
#include "Federate.hh"
#include "RedTankEndo.hh"

// methods

class EvadWEndo: public Digraph {

public:

   EvadWEndo( char* name );
   ~EvadWEndo() ;

};

#endif
```

```
//=============================================

//   EvadWEndo.cpp

//=============================================



#include "EvadWEndo.hh"
#include "FederateEvadWEndo.hh"
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>
#include <iostream.h>
#include <mess.h>
#include <Evader.hh>
#include <RedTankEndo.hh>

//#define DEBUGG

// methods

EvadWEndo::EvadWEndo( char* name ) : Digraph(name) {

#ifdef DEBUGG
  cerr<<"EvadWEndo [" << getName() << "]: constructor " << endl;
#endif



   // ---------------------------------------------

   // create components

   // ---------------------------------------------

   addInport("stop");

   RedTankEndo *RedTK = new RedTankEndo("RedTank");
   Evader *evader = new Evader("EVADER"); // need new class

   // ---------------------------------------------

   // add components

   // ---------------------------------------------

    add(RedTK);
  add(evader);

    // ---------------------------------------------

    // add coupling
```

```
    // ---------------------------------------------



    addCoupling(RedTK, "out", evader, "in");
    addCoupling(evader, "stop", RedTK, "stop");

    getInports()->add("firein");
    addCoupling(this, "firein", evader, "fire");

#ifdef DEBUGG
  cerr<<"EvadWEndo [" << getName() << "]: End of constructor " << endl;
#endif

}



EvadWEndo::~EvadWEndo() {
}
```

```
//=============================================

//   Evader.hh

//=============================================



#ifndef _Evader_HH_
#define _Evader_HH_

#include <Digraph.hh>
#include "perceive.hh"
#include "BlueTank.hh"

// methods

class Evader: public Digraph {

public:

   Evader( char* name );
   ~Evader() ;
};

#endif
```

```
//=============================================

//   Evader.cpp

//=============================================

#include "Evader.hh"
//#define DEBUGG

Evader::Evader( char* name ) : Digraph(name) {

#ifdef DEBUGG
  cerr<<"Evader [" << getName() << "]: constructor " << endl;
#endif

    // ---------------------------------------------

    // create components

    // ---------------------------------------------


    perceive *perc = new perceive("PERCEIVE");
x    BlueTank *BlueTK = new BlueTank("BlueTank");


    // ---------------------------------------------

    // add components

    // ---------------------------------------------



    add(perc);
    add(BlueTK);



    // ---------------------------------------------

    // add coupling

    // ---------------------------------------------



    addCoupling(this, "in", perc, "other");
    addCoupling(perc, "out", BlueTK, "in");
    addCoupling(BlueTK, "out", perc, "mine");
    addCoupling(BlueTK, "out", this, "out");
    addCoupling(BlueTK, "stop", this, "stop");

    addInport("fire");
    addCoupling(this, "fire", BlueTK, "fire");
```

```
}


Evader::~Evader() {
}
```

```
//===========================================

//   BlueTank.cpp

//===========================================



#ifndef _BlueTank_HH_
#define _BlueTank_HH_

#include <Atomic.hh>

class BlueTank : public Atomic {

public:

  // variables

  double myTime, myX, myY, myZ, termValue;
  double deltaTime, deltaX, deltaY, deltaZ;

  // methods

  BlueTank( char* name );
  ~BlueTank();


  void Init();
  message * out();
  void deltint();
  void deltext(RTI::FederationTime updateTime, message *m);
};

#endif
```

```
//=============================================

//   BlueTank.cpp

//=============================================



#include "BlueTank.hh"
#include <port.h>
#include <numEnt.h>
#include <stdlib.h>
#include <DimTraj.hh>
#include <fstream.h>
#include <iostream.h>

//#define DEBUG


#define PROC_TIME 1

// methods

BlueTank::BlueTank( char* name ) : Atomic(name) {

#ifdef DEBUG
  cerr<<"[" << getName() << "]: constructor " << endl;
#endif

  setAtomicClassName(name);

  addInport("fire");

  myTime = 0;
  myX = 500;
  myY = 1000;
  myZ = 0;
  termValue = INFINITY;
  fstream fdelta("delta.in", ios::in);

  fdelta >> deltaTime;
  fdelta >> deltaX;
  fdelta >> deltaY;
  fdelta >> deltaZ;

  double quantum;

  fstream fqt("quantum.in", ios::in);
  fqt >> quantum;
  quantizer *qtzer = new quantizer("Time", 0.0, quantum);
  attribute *at = new attribute("Time", &myTime, qtzer);
  attributeList->add(at);


  qtzer = new quantizer("XDirection", 0.0, quantum);
```

```
  at = new attribute("XDirection", &myX, qtzer);
  attributeList->add(at);

  qtzer = new quantizer("YDirection", 0.0, quantum);
  at = new attribute("YDirection", &myY, qtzer);
  attributeList->add(at);

  qtzer = new quantizer("ZDirection", 0.0, quantum);
  at = new attribute("ZDirection", &myZ, qtzer);
  attributeList->add(at);

  passivate();

}


BlueTank::~BlueTank() {
}

void BlueTank::Init() {

}



void BlueTank::deltext(RTI::FederationTime updateTime, message *x) {

#ifdef DEBUG
  cerr<<"[" << getName() << "]: deltext(T,M) " << endl;
#endif



   timetype e = updateTime;
   Continue();
   if(phaseIs("passive")) {
      entity *ent;
       for (int i=0; i< x->getLength();i++) {
     if (messageOnPort(x,"in",i)) {
      ent = x->getValOnPort("in",i);
           threeDimTraj *threeDT = (threeDimTraj *)ent;

            myX += deltaX;
            myY += deltaY;
            holdIn("active", deltaTime);
  }

     else if (messageOnPort(x,"fire",i)) {
      cerr << "  *******  FIRE arrived at [" << getName() << "]  *******  " << endl;
      cerr << endl;
         }
       }

      holdIn("active", PROC_TIME);
   }
   else
```

```
cerr << "Not accept during ACTIVE phase" << endl;
#ifdef DEBUG
  cerr<<"End of [" << getName() << "]: deltext(T,M) " << endl;
#endif


}



message * BlueTank::out() {
#ifdef DEBUG
  cerr<<"[" << getName() << "]: out() " << endl;
#endif

    message *m = new message();
    content *con;

    if(myX >= termValue) {
       con = makeContent("stop", new entity("Finish"));
    }
    else
       con = makeContent("out", new threeDimTraj(myX,myY,myZ));
    m->add(con);
    cerr << endl;
  if(! m->empty())
 m->print();
 #ifdef DEBUG
  cerr<<"End of [" << getName() << "]: out() " << endl;
#endif

    return m;
}

void  BlueTank::deltint() {

#ifdef DEBUG
  cerr<<"[" << getName() << "]: deltint() " << endl;
#endif

       passivate();
}
```

```
//=============================================

//   perceive.hh

//=============================================



#ifndef _PERCEIVE_HH_
#define _PERCEIVE_HH_
#include "Atomic.hh"

class perceive : public Atomic

{

public:

  // variables

  double myX, myY, myZ, deltaX, deltaY, deltaZ;
  double otherX, otherY, otherZ;
  double Xflag, Yflag, Zflag;

  // methods

  perceive( char* name );
  ~perceive();

  message * out();
  void deltint();
  void deltext(RTI::FederationTime updateTime, message *m);
};

#endif
```

```
//=============================================

//   perceive.cpp

//=============================================



#include "perceive.hh"
#include <numEnt.h>
#include <stdlib.h>
#include <fstream.h>
#include <iostream.h>
#include <DimTraj.hh>

//#define DEBUGG

#define DELAY_TIME 1

// methods

perceive::perceive( char* name ) : Atomic(name) {

#ifdef DEBUGG
  cerr<<"[" << getName() << "]: constructor " << endl;
#endif

  addInport("other");
  addInport("mine");

   myX = myY = myZ = 0;
  otherX = otherY = otherZ = 0;
   Xflag = Yflag = Zflag = 0;

  passivate();

#ifdef DEBUGG
  cerr<<"[" << getName() << "]: End of constructor " << endl;
#endif
}


perceive::~perceive() {
 }



void perceive::deltext(RTI::FederationTime updateTime, message *x) {


#ifdef DEBUGG
  cerr<<"[" << getName() << "]: deltext(T,M) " << endl;
#endif
```

```
   int valX, valY, valZ, otherFlag;
   otherFlag = 0;
   double shadowSize = 2;
   entity *ent;
   for (int i=0; i< x->getLength();i++) {
  if (messageOnPort(x,"mine",i)) {
     ent = x->getValOnPort("mine",i);
           threeDimTraj *threeDT = (threeDimTraj *)ent;
           myX = threeDT->getX();
           myY = threeDT->getY();
           myZ = threeDT->getZ();

           #ifdef DEBUG
          cerr << "myPosition: (" << myX <<", " << myY<<", " << myZ << ")"<<endl;
         #endif
}

  else if (messageOnPort(x,"other",i)) {

     ent = x->getValOnPort("other",i);
           threeDimTraj *threeDT = (threeDimTraj *)ent;
           otherX = threeDT->getX();
           otherY = threeDT->getY();
           otherZ = threeDT->getZ();


           otherFlag = 1;
           #ifdef DEBUG
           cerr << "otherPosition: "<<otherX<<"  "<< otherY<<"  " <<otherZ<<endl;
          #endif

}
    }

    int changeFlag = 0;
    Xflag = Yflag = Zflag = 0;

    if(fabs(myX-otherX) > shadowSize) {
       if(myX>otherX) Xflag = 1;
       else if (myX<otherX)  Xflag = -1;
       else Xflag=0;
       changeFlag = 1;
    }

    if(fabs(myY-otherY) > shadowSize) {
       if(myY>otherY) Yflag = 1;
       else if (myY<otherY)  Yflag = -1;
       else       Yflag = 0;
       changeFlag = 1;
    }

    if(fabs(myZ-otherZ) > shadowSize) {
       if(myZ>otherZ) Zflag = 1;
       else if (myZ<otherZ)  Zflag = -1;
       else       Zflag = 0;
```

```
        changeFlag = 1;
    }
    if(changeFlag > 0 && otherFlag > 0) {
        holdIn("active", DELAY_TIME);
    }



#ifdef DEBUGG
  cerr<<"End of [" << getName() << "]: deltext(T,M) " << endl;
#endif
}


message * perceive::out() {


#ifdef DEBUGG
  cerr<<"[" << getName() << "]: out() " << endl;
#endif


   message *m = new message();
   content *con;

   cerr << endl;


   if(Xflag!=0 || Yflag!=0 || Zflag!=0) {
      con = makeContent("out", new threeDimTraj(Xflag,Yflag,Zflag));
      m->add(con);
   }


   Xflag=Yflag=Zflag=0;       // reset flags
   if(m->empty()) {
  delete m;
return NULL;
   }
   else {
m->print();

   }
#ifdef DEBUGG
  cerr<<"End of [" << getName() << "]: out() " << endl;
#endif
  return m;
}




void perceive::deltint() {

#ifdef DEBUGG
```

```
   cerr<<"[" << getName() << "]: deltint() " << endl;
#endif
    passivate();


#ifdef DEBUGG
  cerr<<"End of [" << getName() << "]: deltint() " << endl;
 #endif

}
```

```
//============================================

//   RedTankEndo.hh

//============================================



#ifndef _RedTank_ENDO_HH_
#define _RedTank_ENDO_HH_
#include <Atomic.hh>

class RedTankEndo : public Atomic
{

public:
  // variables

   double myTime, myX, myY, myZ;
  int stopFlag;

  double PROC_TIME;

  // methods

  RedTankEndo( char* name );
  ~RedTankEndo();
  message * out();
  void deltint();
  void deltext(RTI::FederationTime updateTime, message *m);
};


#endif
```

```
//============================================

//   RedTankEndo.cpp

//============================================



#include "RedTankEndo.hh"
#include <port.h>
#include <numEnt.h>
#include <stdlib.h>
#include <fstream.h>
#include <iostream.h>
#include <DimTraj.hh>

//#define DEBUGG


// methods

RedTankEndo::RedTankEndo( char* name ) : Atomic(name) {


#ifdef DEBUGG
  cerr<<"RedTankEndo [" << getName() << "]: constructor " << endl;
#endif

   setAtomicClassName("RedTank");

    myX = myY = myZ = 0;
   addInport("stop");


   attribute *atT = new attribute("Time", &myTime, NULL);
   attribute *atX = new attribute("XDirection", &myX, NULL);
   attribute *atY = new attribute("YDirection", &myY, NULL);
   attribute *atZ = new attribute("ZDirection", &myZ, NULL);
   attributeList->add(atT);
   attributeList->add(atX);
   attributeList->add(atY);
   attributeList->add(atZ);

   fstream fin("delta.in", ios::in);
   fin >> PROC_TIME;
   holdIn("active", PROC_TIME);

#ifdef DEBUGG
   cerr<<"RedTankEndo [" << getName() << "]: End of constructor " << endl;
#endif
}

RedTankEndo::~RedTankEndo() {
}
```

```
void RedTankEndo::deltext(RTI::FederationTime updateTime, message *x) {

#ifdef DEBUGG
  cerr<<"[" << getName() << "]: deltext() " << endl;
#endif

}

message * RedTankEndo::out() {

#ifdef DEBUGG
  cerr<<"RedTankEndo [" << getName() << "]: out() " << endl;
#endif

   message *m = new message();
   content *con;

   if(stopFlag < 1) {
      con = makeContent("out",new threeDimTraj(myX,myY,myZ));
      m->add(con);
   }

   cerr << endl;
   if(! m->empty())
m->print();
   cerr << endl;

#ifdef DEBUGG
  cerr<<"End of RedTankEndo [" << getName() << "]: out() " << endl;
#endif

   return m;

}



void  RedTankEndo::deltint() {

 #ifdef DEBUGG
  cerr<<"RedTankEndo [" << getName() << "]: deltint() " << endl;
#endif


   if(stopFlag > 0 || myTime >= INFINITY) {
passivate();
   }

   else
      holdIn("active", PROC_TIME);
}
```

Source codes for a Viewer Federate

```
//============================================

//   mainView.cpp

//============================================



#include <time.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>
#include <RTI.hh>
#include "FederateView.hh"
 #include <Federate.hh>

//#define DEBUG


Federate * myFederate;

int main(int argc, char *argv[])

{
  const char* exeName = argv[0];         // Name of executable process
  int iterationCnt;

  if(argc <= 1)
iterationCnt = 10000000; // default number of iteration
  else
iterationCnt = atoi(argv[1]);


  //---------------------------------------------------------------------------

  // create FederateEvadWEndo and internal DEVS models

  //---------------------------------------------------------------------------

  myFederate = new FederateView("FederateView");

 //---------------------------------------------------------------------------

  // run my Federate and its internal DEVS models

  //---------------------------------------------------------------------------

  myFederate->run(argc, iterationCnt);

  return 0;
```

}

```
//=============================================

//   FederateView.hh

//=============================================

#ifndef _FederateView_HH_
#define _FederateView_HH_

#include <HLAport.hh>
#include <objectClass.hh>
#include <Federate.hh>

// methods

class FederateView: public Federate {

public:
   set *myObjects;
   FederateView( char* name );
   ~FederateView() ;
};


#endif
```

```
//=============================================

//   FederateView.cpp (Federate for viewer)

//=============================================



#include "FederateView.hh"
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>
#include <mess.h>
#include "Viewer.hh"
#include <objectClass.hh>

//#define DEBUGG


// methods

FederateView::FederateView( char* name ) : Federate(name) {

#ifdef DEBUGG
  cerr << endl;
  cerr<<"FederateView [" << getName() << "]: constructor " << endl;
#endif

   myObjects= new set();

   Viewer *vwer = new Viewer("Viewer");

   add(vwer);

   // ----------------------------------------------

   // create Object class(es)

   // ----------------------------------------------



   objectClass *RedTankObj = new objectClass("RedTank");     //create an object class
   RedTankObj->addAttribute(new attribute("Time",0,NULL));
   RedTankObj->addAttribute(new attribute("XDirection",0,NULL));
   RedTankObj->addAttribute(new attribute("YDirection",0,NULL));
   RedTankObj->addAttribute(new attribute("ZDirection",0,NULL));
   setUp(RedTankObj);

   objectClass *BlueTKObj = new objectClass("BlueTank");     //create an object class
   BlueTKObj->addAttribute(new attribute("Time",0,NULL));
  BlueTKObj->addAttribute(new attribute("XDirection",0,NULL));
   BlueTKObj->addAttribute(new attribute("YDirection",0,NULL));
   BlueTKObj->addAttribute(new attribute("ZDirection",0,NULL));
```

```
  setUp(BlueTKObj);



#ifdef DEBUGG
  cerr<<"FederateView [" << getName() << "]: End of constructor " << endl;
#endif
}



FederateView::~FederateView() {
}
```

```
//=============================================

//   Viewer.hh

//=============================================



#ifndef _Viewer_HH_
#define _Viewer_HH_

#include <HLAport.hh>
#include <objectClass.hh>
#include <Digraph.hh>


// methods

class Viewer: public Digraph {

public:

   Viewer( char* name );
   ~Viewer() ;
};

#endif
```

```
//=============================================

//   Viewer.cpp

//=============================================



#include "Viewer.hh"
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>
#include <mess.h>
#include "RedTank.hh"
#include "BlueTank.hh"

//#define DEBUGG

// methods

Viewer::Viewer( char* name ) : Digraph(name) {


  #ifdef DEBUGG
    cerr << endl;
    cerr<<"Viewer [" << getName() << "]: constructor " << endl;
  #endif

  RedTank *RedTK = new RedTank("RedTank");
  BlueTank *BlueTK = new BlueTank("BlueTank");

  add(RedTK);
  add(BlueTK);

  #ifdef DEBUGG
    cerr<<"Viewer [" << getName() << "]: End of constructor " << endl;
  #endif
}



Viewer::~Viewer() {
}




//=============================================

//   BlueTank.hh
```

```
//===========================================


#ifndef _BlueTank_HH_
 #define _BlueTank_HH_
 #include <Atomic.hh>

class BlueTank : public Atomic
{


public:

   // variables

   double myTime, myX, myY, myZ;
   double termValue;


   // methods

   BlueTank( char* name );
   ~BlueTank();

   void Init();

   message * out();
   void deltint();
   void deltext(RTI::FederationTime updateTime, message *m);
};


#endif
```

```
//===========================================

//   BlueTank.cpp

//===========================================



#include "BlueTank.hh"
#include <port.h>
#include <numEnt.h>
#include <stdlib.h>
#include <DimTraj.hh>


//#define DEBUG

#define PROC_TIME 500


// methods

BlueTank::BlueTank( char* name ) : Atomic(name) {


#ifdef DEBUG
  cerr<<"[" << getName() << "]: constructor " << endl;
#endif

  setAtomicClassName("BlueTank");

  myTime = myX = myY = myZ = 0;
  termValue = INFINITY;

  addInport("fire");

  attribute *at = new attribute("Time", &myTime, NULL);
  attributeList->add(at);

  at = new attribute("XDirection", &myX, NULL);
  attributeList->add(at);
  at = new attribute("YDirection", &myY, NULL);
  attributeList->add(at);

  at = new attribute("ZDirection", &myZ, NULL);
  attributeList->add(at);
  holdIn("active", PROC_TIME);

#ifdef DEBUG
  cerr<<"[" << getName() << "] end of constructor " << endl;
#endif
}

BlueTank::~BlueTank() {
```

```
}




void BlueTank::Init() {
}



void BlueTank::deltext(RTI::FederationTime updateTime, message *m) {
#ifdef DEBUG
  cerr<<"[" << getName() << "]: deltext(T,M) " << endl;
#endif
}



message * BlueTank::out() {


#ifdef DEBUG
  cerr<<"[" << getName() << "]: out() " << endl;
#endif

   message *m = new message();
   content *con;
   con = makeContent("out", new threeDimTraj(myX,myY,myZ));
   m->add(con);

   cerr << endl;
   if(! m->empty())
     m->print();
#ifdef DEBUG
 cerr<<"End of [" << getName() << "]: out() " << endl;
#endif

   return m;
}

void  BlueTank::deltint() {
#ifdef DEBUG
  cerr<<"[" << getName() << "]: deltint() " << endl;
#endif

    passivate();
}
```

```
//=============================================

//   RedTank.hh

//=============================================


#ifndef _REDTANk_HH_
#define _REDTANk_HH_

#include <Atomic.hh>

class RedTank : public Atomic
{


public:

  // variables
  double myTime, myX, myY, myZ;

  // methods

  RedTank( char* name );
x  ~RedTank();


  message * out();
  void deltint();
  void deltext(RTI::FederationTime updateTime, message *m);
};

#endif
```

```
//============================================

//   RedTank.cpp

//============================================



#include "RedTank.hh"
#include <port.h>
#include <numEnt.h>
#include <stdlib.h>
#include <DimTraj.hh>

//#define DEBUG

#define PROC_TIME INFINITY

// methods

RedTank::RedTank( char* name ) : Atomic(name) {

#ifdef DEBUG
  cerr<<"RedTank [" << getName() << "]: constructor " << endl;
#endif

    setAtomicClassName(name);
    myX = myY = myZ = 0;
   attribute *at = new attribute("Time", &myTime, NULL);
   attributeList->add(at);
   at = new attribute("XDirection", &myX, NULL);
   attributeList->add(at);
   at = new attribute("YDirection", &myY, NULL);
   attributeList->add(at);
   at = new attribute("ZDirection", &myZ, NULL);
   attributeList->add(at);
   passivate();
}

RedTank::~RedTank() {
}


void RedTank::deltext(RTI::FederationTime updateTime, message *m)

{
#ifdef DEBUG
  cerr<<"RedTank [" << getName() << "]: deltext() " << endl;
#endif

}



message * RedTank::out() {
```

```
#ifdef DEBUG
  cerr<<"RedTank [" << getName() << "]: out() " << endl;
#endif

   message *m = new message();
   content *con;
   con = makeContent("out", new threeDimTraj(myX,myY,myZ));
   m->add(con);

   if(! m->empty())
      m->print();
   cerr << endl;

#ifdef DEBUG
   cerr<<"End of RedTank [" << getName() << "]: out() " << endl;
#endif

   return m;
}


void  RedTank::deltint() {


#ifdef DEBUG
  cerr<<"RedTank [" << getName() << "]: deltint() " << endl;
#endif

   passivate();

}
```