Integrating Existing DEVS Simulations With The HLA

by

Chunlei Zhang

A thesis submitted to the

Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of

Master of Engineering

In Electrical Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering

Faculty of Engineering

Department of Systems and Computer Engineering

Carleton University

May 2004

Your file Votre référence
ISBN: 0-612-93972-3
Our file Notre référence
ISBN: 0-612-93972-3

PAGINATION ERROR.

TEXT COMPLETE.

ERREUR DE PAGINATION.

LE TEXTE EST COMPLET.

Abstract

Reusing existing simulations can reduce development time and improve reliability. Often, simulation reusability is restricted to one environment due to complex time management and data incompatibility. The objective of this research is to develop a method that integrates standalone discrete event simulations with the distributed High Level Architecture (HLA) simulation framework. The proposed framework is verified by a case study that illustrates the transformation of a standalone DEVS vending machine simulation implemented using CD++ into HLA compliant components that can participate in HLA federations.

This research addresses several limitations found in the current state-of-the-art research in achieving simulation interoperability and reuse. A wrapper methodology is developed to bridge the standalone DEVS tool and the HLA. The approach provides total transparency in that HLA simulation developers can reuse DEVS simulations without having knowledge of specific DEVS support tools, and DEVS model developers can construct HLA compliant simulations without having to learn the HLA.

Acknowledgements

I would like to thank my supervisor, Professor Trevor Pearce for his guidance, assistance, encouragement and suggestion. In particular, this thesis would not have been possible without his attention to detail, enlightening discussions and helping me through the research process.

I would like to thank Professor Wainer who provided the source code of the DEVS CD++ toolkit.

I would like to express my great appreciations to my parents, my wife Lidan and my son Derrick for all the support I have received during the course of this thesis.

Table of Contents

List of Acronyms

COTS	Commercial-off-the-shelf
CPU	Computing Processing Unit
CSPIF	COTS Simulation Package Interoperation Forum
DEVS	Discrete Event System Specification
DoD	Department of Defense
FedExec	The Federation Executive
FOM	Federation Object Model
HLA	High Level Architecture
IPC	Inter-Process Communication
LibRTI	The RTI Library
M&S	Modelling and Simulation
OMT	Object Model Template
RTI	Run Time Infrastructure
RtiExec	The RTI Executive
SLX	Simulation Language with eXtensibility
SOM	Simulation Object Model
TCP/IP	Transmission Control Protocol/Internet Protocol

List of Figures

List of Tables

Chapter 1 Introduction

Modelling and Simulation (M&S) [1] plays an important role in modern system development. It helps design process in studying the dynamic evolution of a system, verifying design correctness and providing low cost test runs before expensive hardware and manufacturing take place. This minimizes risks and provides a more cost effective engineering process. Distributed Modelling and Simulation [2] in particular has been inspired towards ambitious development, goals by the continually expanding capabilities of communication, networking, hardware, software and related engineering technologies. As a result, the traditional view of a simulation as a standalone program executing on a single processor is now being challenged [2, 24]. The reusability, interoperability and scalability of existing standalone simulations in a new distributed environment is of increasing interest in system development.

Reusability [3] of existing simulations can play an important role in simulation development by reducing the development time and improving product quality and reliability. Unlike regular software modules, reuse of simulations and their models are very much restricted to one simulation platform. This is largely due to the fact that simulations require complex time management and detailed model descriptions. Different simulation tools achieve time management differently; and specific time management techniques can only be used with compatible simulation objects. The other important fact is that the ways models are described are almost different in every simulation tool; a model used in one simulation tool cannot be understood and

used by the others. These two important facts have been the tight bottleneck when encouraging reuse of simulations among different platforms. Often, an existing model has to be rewritten in order for it to be used in another simulation tool. To reuse existing simulations, interoperability among various simulations executed on various toolkits must be resolved.

A distributed simulation framework providing a generic time management mechanism is desired as a backbone [4] for the integration of various existing simulations. The High Level Architecture (HLA) [5] meets this objective as a general-purpose simulation interoperability framework developed initially under the Department of Defence (DoD). It has also gained public support as the IEEE Standard 1516 [4]. The distributed design [6] of the HLA promotes reusability and interoperability among HLA compliant simulation components by sharing objects and interactions as well as supporting time management. The Run Time Infrastructure (RTI) [7] is the HLA middleware that provides HLA services to support the distribution and interoperation of the HLA simulation components. In HLA terms, a *federation* defines a simulation formed by *federates*. Federates carry out their roles within the federation (simulation) and are synchronized by the time management services provided by the RTI. The general interface structure of the HLA makes it a good choice as an environment to promote reusability and interoperability among existing simulations.

The Discrete Event System Specification (DEVS) formalism [8] is a mainstream simulation methodology and a mathematically sound M&S framework. It describes a

real system by specifying its time, I/O, state properties and transition functions; and it simulates the behaviour of a real system by determines its next states and outputs given its current states and inputs. DEVS is wildly accepted as a simulation standard, and many DEVS tools and simulations have been developed [9, 10, 11].

The main objective of this research is to find an approach to integrate DEVS simulation with the HLA [24]. The approach allows existing DEVS models to be reused in the HLA environment without modification. Ideally, HLA federation developers can reuse DEVS models as federates without having knowledge of specific DEVS support tools, and DEVS model developers can construct HLA compliant federates without having to learn the HLA. As a proof of concept, the DEVS CD++ toolkit [9] is used in a simple case study involving a DEVS Vending Machine model [12]. Prior to this research, the CD++ toolkit allowed the model to be constructed and executed as a standalone simulation. In the case study, the hierarchical Vending Machine model is split into two DEVS sub-models, and enclosed in wrapper to become separate federates. A third, non-DEVS federate is also introduced. The resulting federation shows that DEVS models can be wrapped to interoperate with other (wrapped) models, as well as with non-DEVS federates. The purpose of integrating DEVS simulation with the HLA is served.

The product of the research is an HLA-based simulation environment that integrates simulations and/or simulation components that are not originally designed to execute under the HLA. The already developed simulations or components can then be reused

under HLA. Cost and time are saved on re-design, re-implementation and re-testing of existing simulations. Furthermore, these conversions do not require any specific knowledge of the tools that run the simulation, and can be easily carried out by HLA developers.

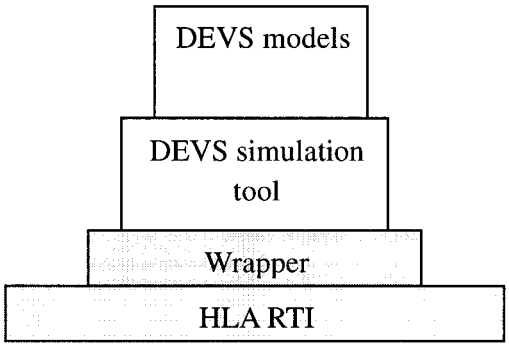


Figure 1-1 Integrating DEVS simulations onto the HLA RTI

Figure 1-1 shows the targeted integration framework. DEVS models are still being executed in the DEVS simulation tool environment. The newly introduced wrapper [13] bridges the originally standalone DEVS tool with the HLA RTI. This framework allows DEVS simulations to be executed in the HLA environment as HLA compliant federates.

The remainder of this document is structured as follows.

Chapter 2 provides a brief overview of related background information. The DEVS formalism and CD++ toolkit are described as the target M&S environment, and the HLA is introduced as the target distributed simulation environment. The Vending Machine simulation used in the case study is also presented.

Chapter 3 reviews research related to simulation reusability and interoperability. It discusses the recent advances on DEVS/HLA [14], some simulation wrapper examples [15, 16, 17], and some highlights of the recent COTS Simulation Package Interoperation Forum (CSPIF) [18] discussion about COTS simulations interoperability. Chapter 3 also provides a critical analysis of these research approaches.

Chapter 4 states the thesis, the research objectives, the proposed solution, the thesis scope and a summary of the contributions made by the research.

The solution is described in Chapters 5, 6 and 7. Chapter 5 details the proposed HLA-based framework and the wrapper development method. The solution is discussed in high-level terms as well as detailed modifications that clearly identify the backbone (RTI) services used, the wrapper (federate) construction, the modifications to the DEVS simulation tool and the inter-process communication (IPC) [19] method used.

Chapter 6 demonstrates the solution by implementing a wrapper case study using the CD++ toolkit and the Vending Machine example. The Vending Machine simulation is split into two sub-simulations and wrapped to interoperate with each other along with a third non-CD++ but HLA compliant customer over the HLA RTI to form a new simulation. The implementation details are explained step by step. Issues such as time management and data conversion are illustrated in detail as well as wrapper design and wrapper creation. The split of DEVS models are also discussed in detail. The

HLA federation object model creation is another important aspect explored in the case study.

Chapter 7 verifies the correctness of the system via a set of test cases. The system performance is discussed when using different data messaging structures. The pros and cons of each design are presented, and solution for performance improvement is also discussed. The effectiveness of the wrapper method using the HLA as a target environment is examined.

Chapter 8 states the conclusions of the research, and suggests some future research directions.

Chapter 2 Background Information

This chapter reviews important concepts and components that fill various roles in the research. The High Level Architecture (HLA) [5] is the distributed simulation framework chosen to support the interoperation of heterogeneous simulation components. The services provided by the HLA are of interests. The Discrete Event System Specification (DEVS) formalism [8] is a commonly used modeling and simulation approach, and the integration of DEVS with the HLA takes both the advantage of efficient event-based simulation and distributed/modulated framework. The CD++ toolkit is an implementation of the DEVS formalism and its DEVS Vending Machine simulation are used in the case study as a proof of concept. The overall integration is realized by using the wrapper method that is a well-known approach for reuse of existing software. In order to make this document self-contained, an introduction to each of these components and concepts is provided below.

2.1 Discrete Event System Specification (DEVS) Formalism

The DEVS (Discrete EVents Systems specifications) [8] formalism for modeling and simulation is a means of specifying a system. DEVS encompasses three basic objects: *real system*, *models* and *processors*. It describes a *real system* in a discrete event fashion using a composite of *models* that can be simulated by the *processors*. DEVS

provides a framework for the construction of hierarchical models in a modular fashion, allowing model reuse, reducing development and testing time.

A DEVS model is a set of instructions for generating data comparable to that observable in the real system [20]. The *structure* of the model is its set of instructions. The *behaviour* of the model is the set of all possible data that can be generated by executing the model's instructions. Each DEVS model is either a behavioural (*atomic*) or a structural (*coupled*) model and can be integrated into a structural hierarchy.

A DEVS *atomic* model is defined as:

$$\mathbf{M = <X, S, Y, \delta_{int}, \delta_{ext}, \lambda, Ta >}$$

Where:

X: the input events set

S: the state set

Y: the output events set

δ_{int} : internal transition function

δ_{ext} : external transition function

λ: output function

Ta: the elapsed time

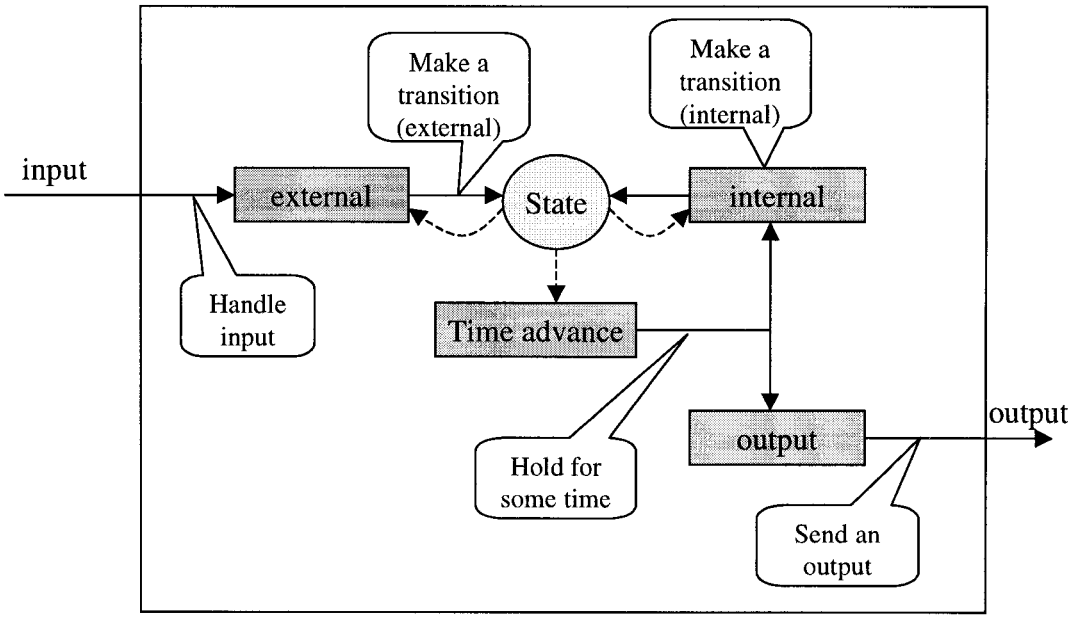


Figure 2-1 DEVS Atomic Model Implements Basic DEVS [21]

A DEVS atomic model consists of an I/O interface and transition functions. Figure 2-1 illustrates how the atomic model proceeds. Each model uses input/output ports in the interface (defined by the X and Y sets) to communicate with other models. The external input events are received through input ports, activating the external transition function (δ_{ext}). Every state has an associated lifetime (Ta), after which, the internal transition function (δ_{int}) is activated. Time advancement is conducted during the internal state changes. These internal events also produce state changes and outputs, and the outputs are transmitted through the output ports.

Atomic models may be coupled into *coupled models*. A coupled model describes how to connect (couple) several atomic and/or coupled models together to form a new model. This new model can itself be employed as a component in a larger coupled model, thus giving rise to hierarchical construction. An example of a coupled model is

given later in the Vending Machine example (section 2.3).

A DEVS coupled model is defined as:

$$\mathbf{CM = < X, Y, D, \{M_i\}, \{I_i\}, \{Z_{ij}\}, select >}$$

Where:

X: the set of input events

Y: the set of output events

D: an index of components, each $i \in D$

M_i: a basic DEVS model (atomic or coupled)

I_i: the set of influencees of model I, for each $j \in I_i$

Z_{ij}: is the output translation mapping $Y_i \rightarrow X_j$ (i.e. the translation function)

Select: the function prescribes which atomic model should be activated first under simultaneous events.

Each coupled model consists of a set of basic models (M_i) connected through the input/output ports of their interfaces. These basic models (components) are treated as the children or imminent dependents of the coupled model. Each component is identified by an index number (D). The models where output values must be sent are called the influencees of each model. The translation function (Z_{ij}) uses an index of influencees created for each model. This function defines which outputs of model M_i are connected to inputs in model M_j.

DEVS processors carry out simulations of DEVS models. The DEVS *processor* is

specialized into two different simulation engines, *simulators* and *coordinators.* The role of a *simulator* is to invoke an atomic model's transition and event functions. A *coordinator* is paired with a coupled model and has the responsibility of translating its children' output events and of managing time for the imminent dependants.

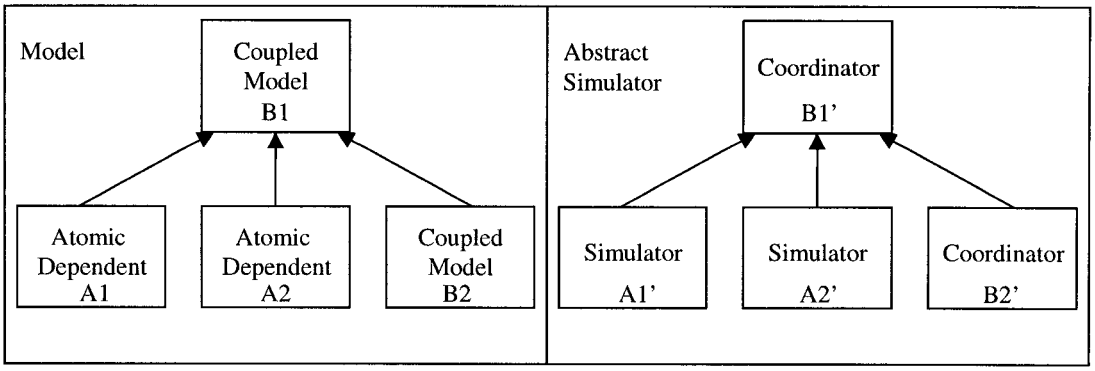


Figure 2-2: Correspondence Between DEVS Models and Processors [22]

Figure 2-2 illustrates the correspondence between the DEVS models and DEVS processors. A coupled model with three children (A1, A2 and B2) and its abstract simulator is shown. Each of the atomic models (A1 and A2) has an associated simulator (A1' and A2') in charge of its events and state changes, and the coupled model (B2) has a corresponding coordinator (B2'). The coordinator (B1') act as the "parent processor" of the simulators/coordinators and it plays the role of the coupled model's processor. In addition, the coordinator performs the time management among all its dependent simulators. This DEVS processors' structure in a DEVS simulation forms a tree structure, and at the root node of the tree resides the root coordinator. The root coordinator is in charge of the overall simulation progress, and is of major concern later in chapters 5 and 6. In the DEVS context, top-level models are the atomic or coupled models whose corresponding processors are the immediate children

of the root coordinator.

2.2 CD++ Toolkit

There are various tools that implement the DEVS formalism, and the CD++ toolkit [9] is among them. The toolkit has been built as a set of independent software components, and each of them are independent of the operating environment chosen. The DEVS models to be executed by the CD++ toolkit are built as a class hierarchy, and each is related with a simulation entity (coordinator or simulator) that is activated whenever the model needs to be executed. New atomic models can be incorporated into this class hierarchy by creating a C++ class derived from a base atomic (for atomic model) class. The derived class overloads the methods in the base atomic class to represent a DEVS specification including external transitions, internal transitions and output functions. The toolkit accepts model descriptions according to the DEVS formalism; both atomic models and coupled models are defined using a specification language following the formal definitions for DEVS coupled models. The defined simulation models are stored in a model description file (the MA file) [23] and are loaded into the CD++ toolkit at run time. During simulator construction, the atomic models' classes whose corresponding models are referenced in the MA file are included in the simulator. To run a simulation, the CD++ toolkit feeds a list of input events from an input file into the simulator and produces the results as a list of output events placed in the output file.

During simulation execution, each lower level processor always forwards the time of its next-to-execute event to its parent coordinator, and waits for a grant to continue its execution. The root coordinator collects the timing requests for all models and makes the final decision as to which coordinator should be granted permission to execute. In other words, the root coordinator is the central controller of the simulation.

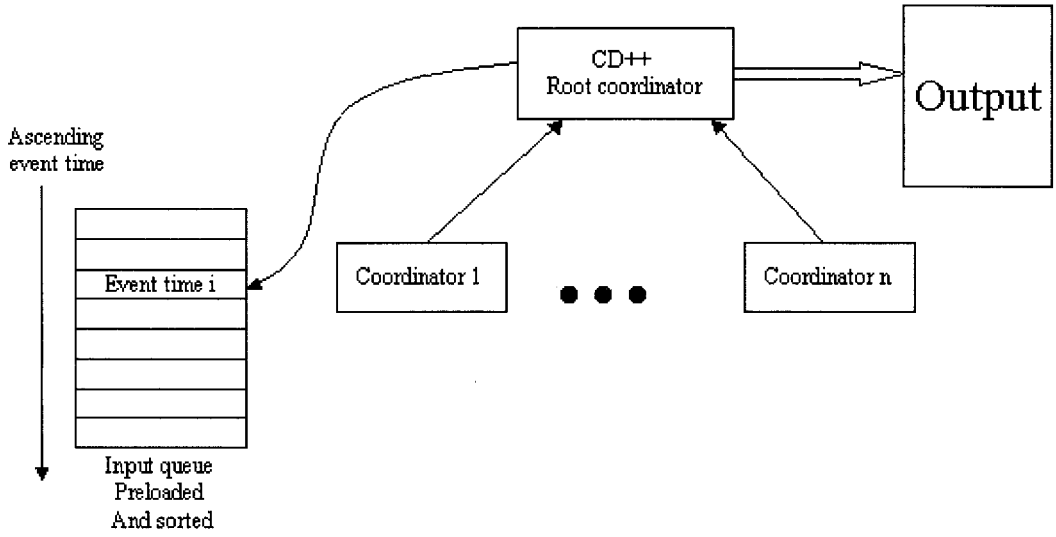


Figure 2-3 Root Coordinator of the CD++ Toolkit

The details of how the root coordinator performs time management and controls the execution sequence is detailed in Figure 2-3. On the left, a queue containing input events is loaded from the input file at start up and the events are sorted by their time from the earliest time to the latest. The root coordinator moves its cursor down through the list to process the event conforming to the current simulation time. The coordinators below the root supply their desired time request to the root coordinator. The root coordinator compares them with the input event at the cursor pointed input event to determine which coordinator should run. The simulation terminates when all input

events are processed and no more next event requests are made to the root coordinator. All outputs produced by the individual processors are forwarded to the root coordinator and eventually placed in the output file.

How the input events are defined, how the building process is handled and how the models are described are discussed in the next section where a Vending Machine model is presented.

2.3 Vending Machine Simulation

A Vending Machine simulation [12], similar to those found in a typical cafeteria, is used in the case study (Chapter 6) to explore the interoperability of DEVS simulations over the HLA RTI. The Vending Machine accepts coins, displays the current balance, allows item selections, dispenses selected items to the customers, and returns change if there is any balance left. The model is one of the many example CD++ models available at the CD++ toolkit Internet site [25].

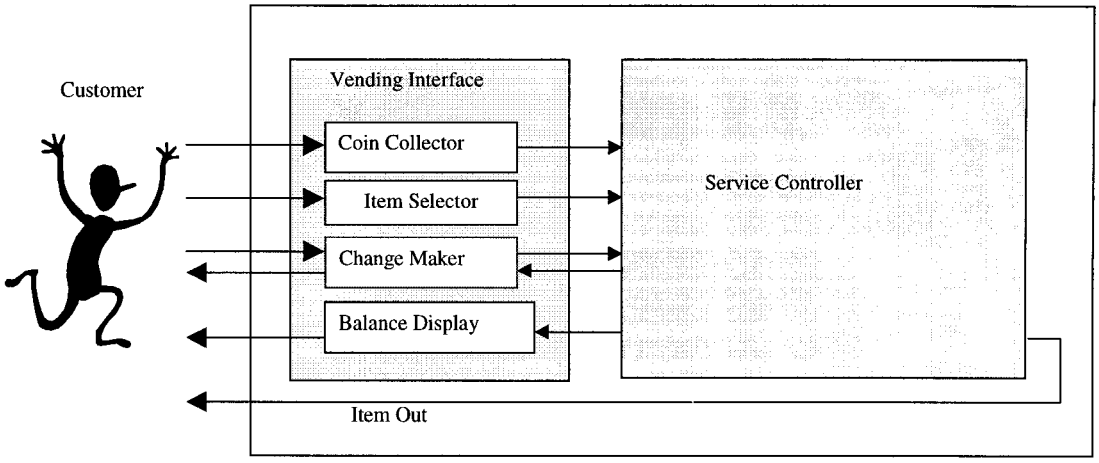


Figure 2-4: Vending Machine Model

Figure 2-4 shows the DEVS model of the Vending Machine. The top level of the model consists of two coupled models. The Vending Interface model, on the left, is a coupled model that forms the Vending Interface to the Customer. The Service Controller, on the right, is a coupled model that carries out all computation associated with the operation of the Vending Machine, and delivers items to the Customer. For simplicity, the details inside the Service Controller are not discussed and the model can be treated simply as a black box that performs necessary operations. All communications between the top-level models and the Customer (shown as big arrows in Figure 2-4) are implemented as input/output events. The couplings of the models are shown in smaller arrows.

A typical Vending Machine use case scenario is shown Figure 2-5. The scenario starts with a Customer inserting coins into the coin collector. Upon receiving the coins, the value of the coins is passed to the Service Controller, the controller then returns the current balance back to the Vending Interface where it is displayed to the Customer.

Once sufficient coins are inserted, the Customer selects the items they want to purchase, and the selections are passed from the Vending Interface to the Service Controller. The Service Controller recalculates the balance, passes the new balance back to the Vending Interface to be displayed and outputs the requested items. The Customer then requests the remaining balance as change, the Service Controller informs the change maker and the remaining balance is refunded to the Customer.

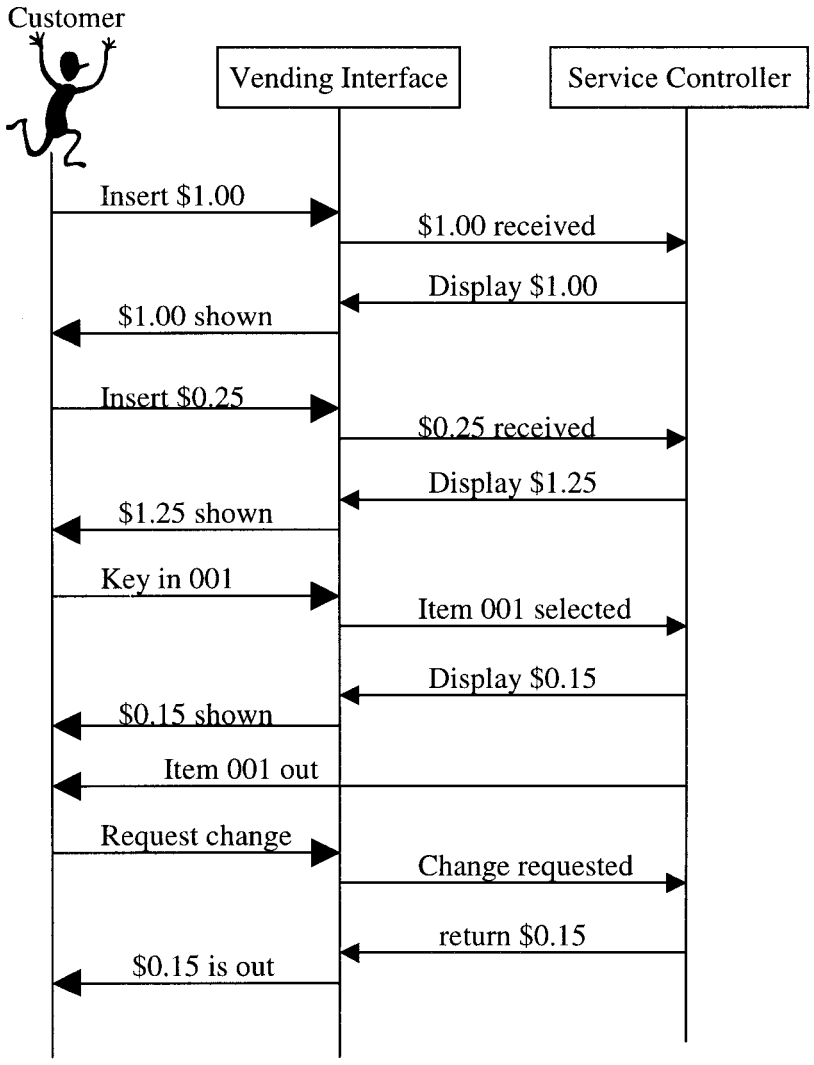


Figure 2-5 Vending Sequences

To execute the simulation, an input file, a model description file (the MA file) and a building process to include model internal functions are needed.

```
[top]
components : VendingInterface
components : ServiceController
in : coin_in ...
out : change_out ...
Link : coin_in cin@VendingInterface
Link : change_out cout@VendingInterface
...

[VendingInterface]
components : coin_collector ...
in : ...
out : ...
Link : ...

[ServiceController]
components : ...
in : ...
out : ...
Link : ...

[coin]
preparation : 0:0:1:0
...
```

Figure 2-6 Sample Model Description File (MA file)

The outline of the MA file for the Vending Machine example is shown in Figure 2-6. At the top level, the simulation has two components, the Vending Interface and the Service Controller. How the components are connected to each other is defined at the "Link" section and what I/O ports are defined is described at the "in" and "out" sections. Following the top-level definitions are the definitions of each individual component that follows the same structure.

Input files are a list of strings, of the form "time port value". The terms in each string define the time of the input action, the port where the input occurs and the value of the input.

Issues arise when tying to reuse such a simulation. It might be desirable to reuse this Vending Machine simulation in a cafeteria simulation, but integrating the simulations may be difficult if the cafeteria simulation is not built to execute in the CD++ toolkit. Furthermore, the Service Controller model of the Vending Machine might be reused in a different DEVS Vending Machine simulation so that it does not have to be rebuilt, but compatibility issues may arise between different DEVS simulation tools. The potential benefits of a flexible and interoperable distributed simulation component have encouraged the extendibility and reusability of standalone simulations.

2.4 High Level Architecture (HLA)

Developing distributed simulations requires a component-oriented architecture that addresses characteristic issues in distributed systems, such as concurrency, synchronization, the open sharing of simulation-related information among components, and the ability to create a single simulation out of components distributed across a network of independent nodes. The interoperation mechanisms that allow distributed components to interact are fundamental to distributed systems, and the presence of standards, like the HLA, to enable interoperability are essential

cornerstones to success. The HLA has its roots in defence programs, but has also gained public support as the IEEE Standard 1516-2000 [5]. The objective of the HLA is to enable the interoperation of distributed simulation components. Ideally, the standard will also encourage and simplify the reuse of simulation components. The HLA provides a framework within which the technologies and issues relevant to distributed simulation can be harnessed to realize ambitious applications of distributed M & S.

The characteristics of distributed simulation present several challenges. In general, a collection of distributed system components must cooperate to accomplish their objective. It is common for components to make decisions based on partial knowledge of the state of other components, and under conditions where the states of other components may be changing concurrently. Components must communicate to share information about their state, and must synchronize when decisions require components to share consistent views of the collective state. It is common for distributed components to share persistent data values, and/or the occurrences of relevant events (typically state changes). In addition to sharing events and data, the role of time adds an extra dimension to distributed simulations. The view of time taken in a simulation is simulation-specific. For example, a simulation of the tectonic motion of continental plates must use a time scale in which time proceeds far more quickly than our human perception of wall-clock time. This is necessary to compress the thousands of years taken to exhibit visible continental drift into a simulation run that produces results within a practical (wall-clock) time. The components in a

distributed simulation must often share their view of (simulation) time, and therefore time management becomes an issue. Furthermore, simulation components must often synchronize their sharing of time to ensure consistency when sharing data and events. The HLA has been designed to address these distributed simulation characteristics.

The HLA provides ten organizational rules, documentation standards for describing interoperability requirements, and a set of services for the Run Time Infrastructure (RTI) (for a more detailed introduction to the HLA, see [7]). In HLA terminology, a distributed simulation component is called a *federate*, and the collection of federates interacting as a distributed simulation is called a *federation*. The HLA organization rules place simple constraints on federates and federations. For example, one of the rules requires that federates only interact using RTI services. The HLA requires that each federate be documented by a Simulation Object Model (SOM) [26]. The SOM provides the interface specification for the federate in terms of the information it generates and makes available to other federates, as well as the information it expects to be generated and made available by other federates. The Federation Object Model (FOM) [26] defines the collective information shared in a federation, and provides a consistent terminology for that information. The RTI is a middleware layer that supports the distribution and interoperation of simulation components at runtime. The RTI provides services to share object *attributes* (data) and *interactions* (events), as well as time management services [7, 27]. A federate that generates timing information to which other federates must synchronize is called *regulating*, while a federate that accepts timing information to which it must synchronize is called

constrained. A federate may be regulating, constrained, regulating and constrained, or neither regulating nor constrained. The look ahead value used in the HLA time management is the amount of time that a federate promises the RTI it will predict attribute updates and interactions. Using look ahead will facilitate higher simulation performance. In addition to the object management and time management, the RTI also provides federation management services that allow federates to dynamically create, join, leave and destroy federations.

2.4.1 Run Time Infrastructure (RTI)

The RTI middleware provides common services to federates and federations. It implements the HLA interface specification and provides the software services necessary to support an HLA-compliant simulation.

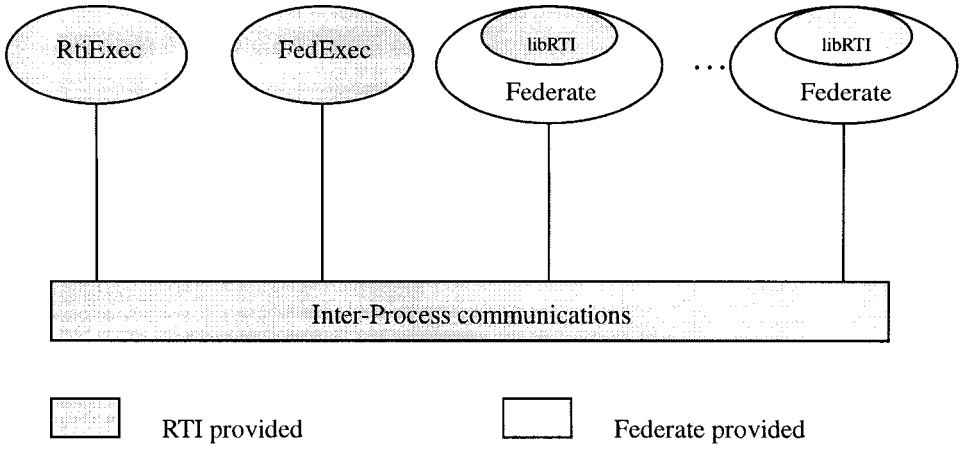


Figure 2-7: Logical View of RTI Components [8]

A high level logical view of an executing HLA federation is shown in Figure 2-7. All

of the components are part of a single federation, with the exception of the RtiExec. The shaded portions are provided by the RTI, while the remainder are federate specific. The Federation Executive (FedExec) manages the federation. It allows federates to join and to resign from the federation, and facilitates data exchange among participating federates. The RTI executive (RtiExec) is a global process that manages the creation and destruction of multiple federations. The RTI library (libRTI) makes RTI services available to each federate.

Federates use libRTI to invoke HLA services. Figure 2-8 illustrates the RTI and federate code responsibilities. Within libRTI, the class RTIambassador bundles the services provided by the RTI. All requests made by a federate on the RTI invoke an RTIambassador method. The abstract class FederateAmbassador identifies the callback functions each federate is obliged to provide. While both the RTIambassador and the FederateAmbassador classes are a part of libRTI, the federate must implement the functionality declared in FederateAmbassador.

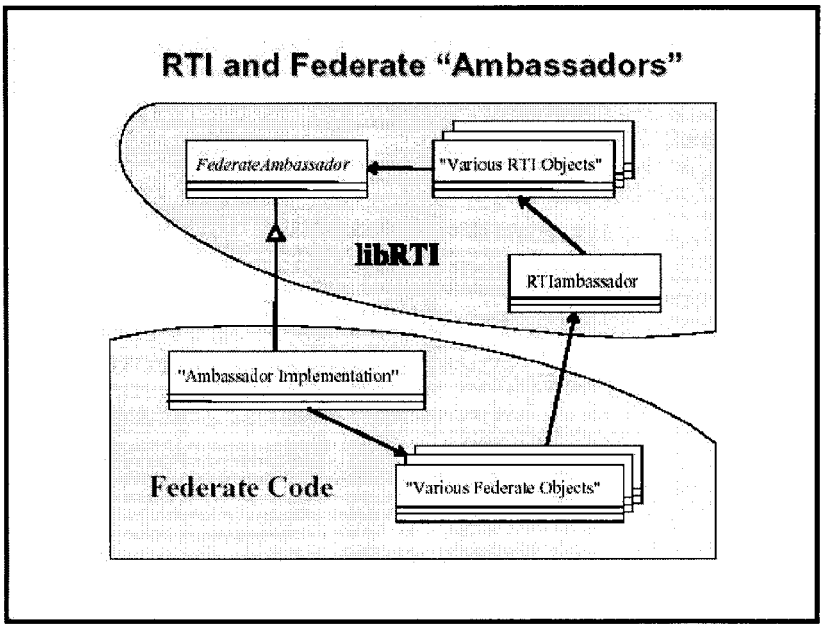


Figure 2-8: RTI and Federate Code Responsibilities [5]

2.4.2 RTI Time Management

The HLA Interface Specification partitions the services provided by the RTI into the following six management areas: federation management, declaration management, object management, ownership management, time management, and data distribution management. In this research, time management is particularly important in achieving interoperability.

The RTI supports both continuous and discrete simulation time management. Only the discrete simulation time management is of interest for this research. At the federate level, a federate requests a time advance by calling nextEventRequestAvailable(time) function. After the request, there are two possible cases. In case one, a federate is granted the time it requested, in which case it simply adjusts the current time to the

requested time and performs any processing that should occur at this time. In the second case, an event with a time stamp earlier than the requested time is received. The federate's previous time advance request is cancelled, and the federate processes the event before proceeding. At the RTI level, the RTI simply waits for all regulating federates to make time requests, and then grants a time advance to the one with the earliest time stamp.

2.5 Wrapper Concept

In this research, the wrapper concept is used to construct HLA-compliant wrappers for CD++ simulations. The wrapper concept provides a proven solution for the integration of software components into new environments [13]. A software wrapper is a layer of software that is introduced to allow a software component to execute in an environment that it was not originally designed and/or constructed to be compatible with. A wrapper provides a bridge by which a software component can be reused in new environments and applications. A wrapper can accomplish several objectives, such as resolving environmental incompatibilities, and modifying the component's behaviour. The complexity of the wrapper depends on the extent of the functionality introduced. Ideally, all new functionality is local to the wrapper, and the original component is not modified. In practice, it may be necessary to modify the original component to provide the necessary hooks to integrate the wrapper.

Chapter 3 State-of-the-Art Research

This section reviews state-of-the-art applications and research exploring the DEVS on HLA, interoperability and reusability. The focus is given to four main representatives in these fields:

- The DEVS/HLA framework [14]
- The AGENT-HLA model. The integration of AGENT simulations onto the HLA [15]
- Wrapping SLX onto the HLA [16]
- CSPIF panel discussion of integrating COTS onto the HLA [18]

3.1 The DEVS/HLA Framework

DEVS/HLA [14] is an implementation of the DEVS formalism over the HLA/RTI. The research is about the detailed design and development of an HLA-compliant simulation environment that supports high-level model building using the DEVS methodology.

The DEVS/HLA pursues a direct mapping of the DEVS simulation protocol on to the HLA. In the process, a mismatch was discovered between the HLA specification and the DEVS formalism. To obtain full expressiveness, the solution presented in the Figure 3-1 was developed. The solution efficiently exploits the functionality of the baseline DEVS/HLA to implement the necessary time-management and coordination

by introducing an explicit time management federate.

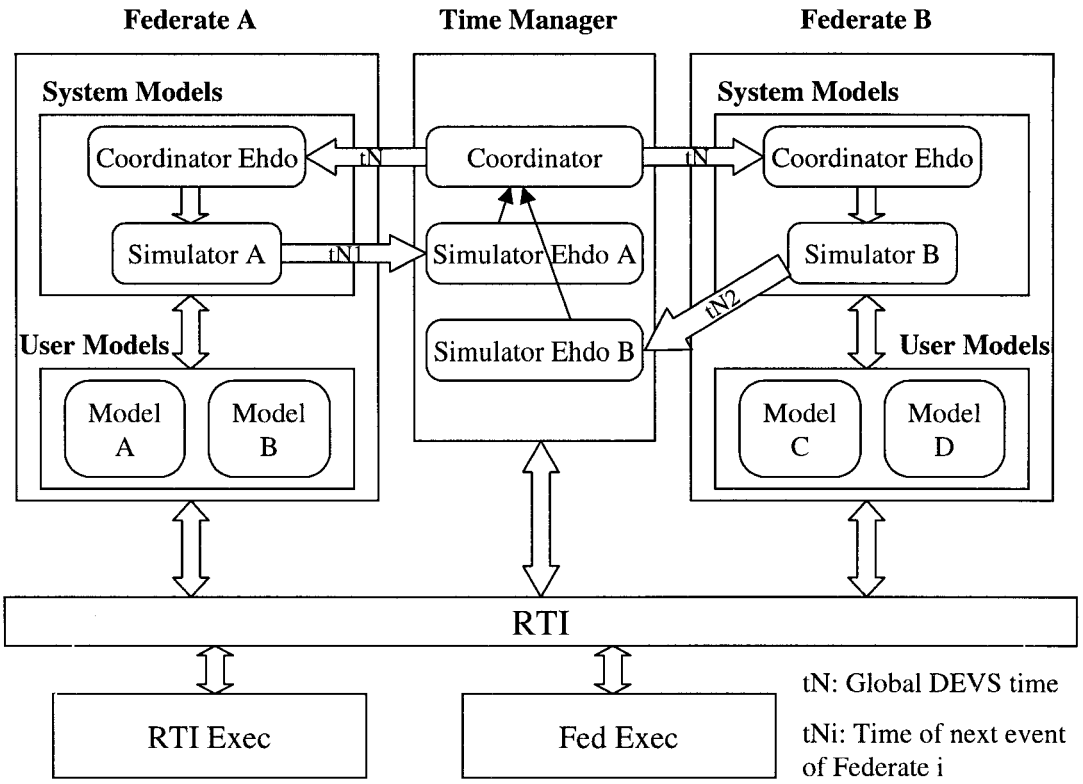


Figure 3-1 Realizing the Coordinator as a Federate [14]

In Figure 3-1, User Models are running in Federate A and Federate B, and the separate Time Manager federate is designed to perform the role of the coordinator of time management for the simulation. From Federate's A and B perspective, it is as if there is a coordinator (Coordinator Ehdo) running locally to make time decisions. "Ehdo" in the figure means the shadow of a real object. This design achieves the time management among the DEVS/HLA federates; however, it restricts the interoperability of DEVS/HLA to interact with other HLA compliant simulations. With a separate federate act as the time manager, other non-DEVS/HLA compliant

simulations would not be able to communicate with the DEVS/HLA coordinator to synchronize time advancements. From the HLA perspective, federates are neither regulating nor constrained since all solution does not use the HLA time management service, and time management is implemented outside of the RTI.

3.2 Simulating Agent-based System with HLA: The Case of SIM AGENT 2002

This SIM AGENT research [15] has made progress in terms of performing distributed simulations of agent-based systems using the SIM AGENT toolkit on top of the HLA RTI. It showed how the HLA can be used to distribute an existing SIM AGENT simulation with different agents being simulated by different federates, and briefly outlined the changes necessary to the SIM AGENT toolkit to allow integration.

An HLA_AGENT library was built to link the SIM AGENT toolkit and the HLA RTI. The HLA_AGENT library was built in a way that the connection to the HLA is transparent to user level simulation; however, additional information specifying the number of federates in the federation and how the objects and agents in their simulation are to be assigned to federates is required in order for the simulation to be distributed over the HLA. It should be noted that there was no additional management federates required in order for the simulation to be executed.

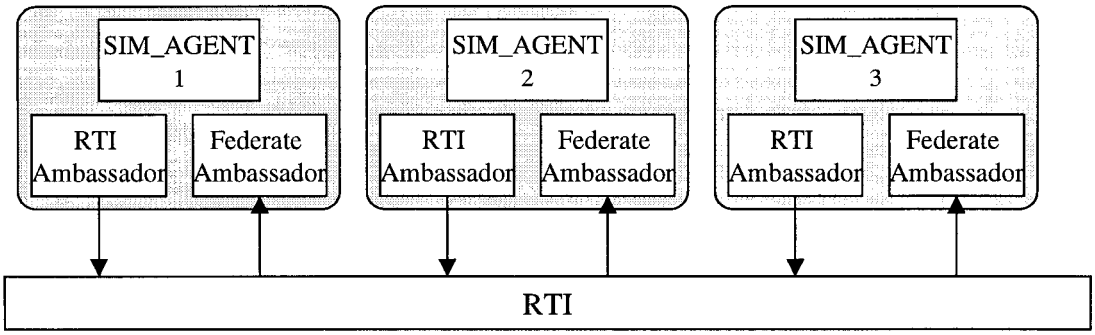


Figure 3-2 SIM_AGENT Federation

Error!

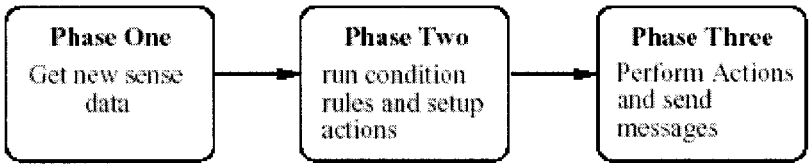


Figure 3-3: Logical Structure of a Simulation Cycle [15]

Figure 3-2 is the high level view of the SIM_AGENT federation. The time management is rather simple in this case. Time is measured in simulation cycles and each simulation cycle consists of the three phases shown in Figure 3-3. There is no synchronization needed within one simulation cycle and a federate running the simulation does one simulation cycle, then waits for a signal to repeat another cycle and so on. RTI time management services such as requesting time advance are not involved in this case.

The SIM AGENT research has achieved interoperability among different simulation AGENTs running on the RTI; however, due to the nature of the simulation, time management of the HLA is not used for the synchronization of the simulation. Although it is very successful in achieving AGENT interoperability over the HLA, the same scenario could not be applied in non-AGENT component situation where time

plays an important role in driving the simulation.

3.3 The HLA-SLX (Simulation Language with Extensibility) framework

The HLA-SLX framework focused on opportunities for the integration of classical stand-alone simulation tools into the High Level Architecture. The simulation tool SLX has been selected as a platform to gain experience with an HLA-based factory simulation.

In order for the SLX model to access RTI functions, SLX implements a set of DLLs that could be included and called from within the SLX models. This approach requires explicit statements in the models to carry out the federation/federate creation and object/interaction publish and subscribe. The DLL also implements other functions of the RTI and makes them accessible from within the SLX model by including the library and calling the corresponding C like functions. Figure 3-4 shows the approach.

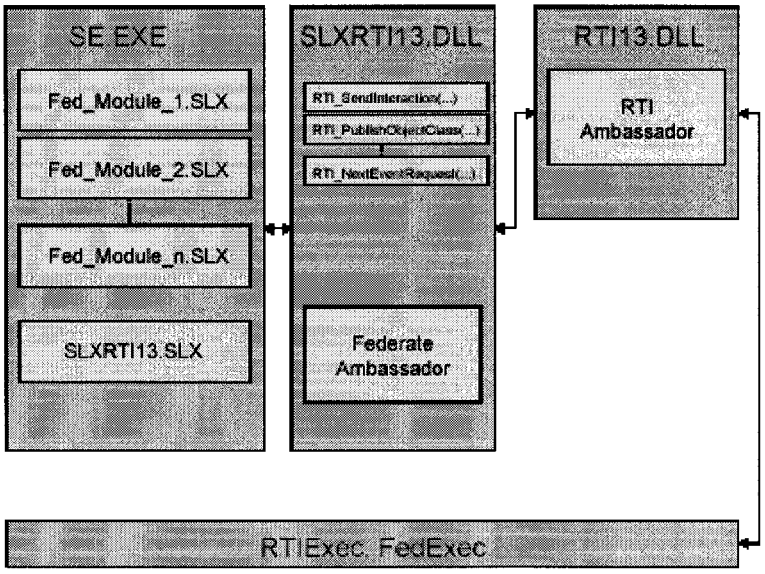


Figure 3-4: SLX Architecture [16]

SLXRTI13.DLL is the SLX-HLA interface developed, which provides one interface to the SLX models and the other interface to the RTI library. It is worth noting that the functions provided in the SLXRTI13.DLL are C like functions instead of the ones provided by the RTI that are object-oriented.

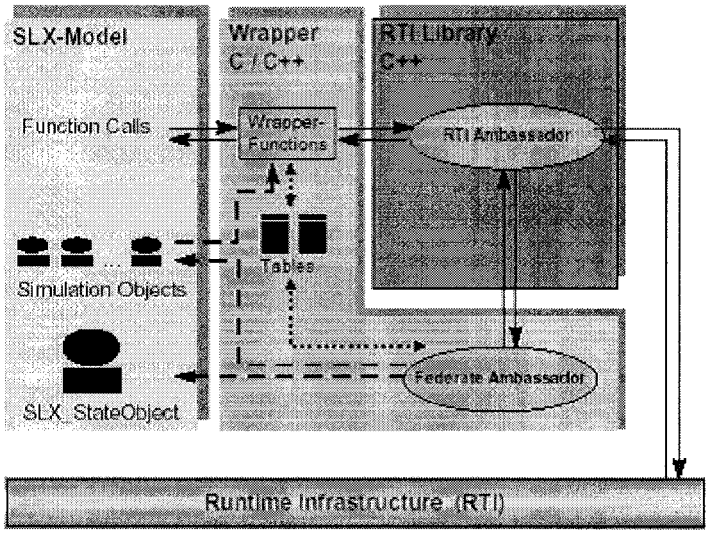


Figure 3-5 Functional View of the Software Structure of the SLX-HLA Interface [16]

Figure 3-5 shows another view of the resulting structure of the design in a more HLA-oriented manner. It is clear that there is a language wrapper being used to convert the object oriented RTI functional calls into a conventional C like syntax to facilitate access from the SLX model. In addition to simple programming language conversion, the language wrapper also performs tasks such as data type conversions between SLX specific data types and data types that are understood and expected by the RTI. Synchronization and time management are also taken care of by a combination of RTI and SLX time management functionality. A drawback to the approach is that a significant amount of HLA-related work still needs to be programmed by SLX model developers, including the initialization of federation, joining of federates, publish and subscribe of interactions, and time advance requests. This requires knowledge of the RTI as well as its rules and regulations.

The work facilitates the integration of SLX with the RTI, but is still distant from making model development transparent to the HLA RTI, and interoperability with other HLA compliant simulations was not discussed.

3.4 CSPIF 2003, HLA-CSPIF DISCUSSION ON COMMERCIAL OFF-THE-SHELF DISTRIBUTED SIMULATION

Commercial-off-the-shelf (COTS) simulation packages are widely used in many areas of industry. Several research groups are attempting to integrate distributed simulation

principles and techniques with these packages to potentially provide *COTS distributed simulation*. This discussion paper [18] presents the views of four panel members at the COTS Simulation Package Interoperation Forum (CSPIF). The panel discussed the technical problems that must be overcome for COTS simulations to interoperate over the HLA.

The intention of the panel discussion was to foster further discussion that will lead to making distributed simulation (such as HLA) available to users of COTS simulation packages.

Two of the most apparent and critical issues discussed by the panel were time management and event transformation. Time management was identified as the key to addressing the issue of synchronizing multiple copies of the simulation such that the causality constraints are not violated. Event transformation addresses the issue of the transparent transformation of an internal event into an external interaction when the event needs to be sent to a remote simulation. An interaction must also be transformed back into an internal event transparently when it reaches the destination simulation. The panel believes some means of allowing the modeller to define the object model needs to be incorporated into the modeling tool of the COTS package.

Some of the panel members' thoughts are very similar to the goal of wrapping the CD++ toolkit to enable interoperability with HLA compliant simulations; however, the panel has only suggested two apparent issues, and they give no concrete examples or products that realize fully automated distribution of COTS package over the HLA RTI.

A number of important issues that were not discussed by the panel include the method for COTS to be able to accept dynamic inputs and reorder event execution, and the ability to obtain true HLA transparency for model developers and alleviate their need to know about the HLA, its rules, and the way HLA operates. The panel also did not discuss concurrency and multiple received interaction issues. FOM/SOM generation was addressed heavily, with suggestions that it be incorporated into the modeling tool of the COTS package. This suggestion may lead to the modification to the COTS packages and may add extra complexity to the integration; however, it is a very interesting suggestion and this research will explore further on the FOM/SOM generation in later chapters during the discussion of wrapper design.

Chapter 4 The Thesis Statement

The state-of-the-art research reviewed in Chapter 3 has exposed several limitations in introducing simulators into an HLA environment. The DEVS/HLA framework approach does not use RTI time management and does not address non-DEVS/HLA interoperability issues. The work of AGENT HLA solved the interoperability of different Agents running on the HLA; however, agent time management is overly simplistic, which makes it difficult to be applied more generally. The SLX HLA uses a wrapper approach to map SLX onto the RTI, however; the interoperability issue was not resolved and it did not achieve true transparency to the SLX developers in order for the models to run on the HLA. The CSPIF panel suggests target system goals; however, the discussion simply highlighted directions for research without providing solutions. Furthermore some of the discussions do not apply in this research and some important issues are missing.

The thesis of this research is that the DEVS CD++ simulation can be directly mapped onto the HLA to interoperate with other HLA compliant simulations using the wrapper approach. The wrapper resolves time management and compatibility issues. It opens the door to the DEVS developer to reuse existing simulations without rework of the models. It provides true transparency for HLA developers to reuse DEVS simulations without the knowledge of the specific DEVS tools. It introduces a method that can integrate simulations without having to worry about simulator incompatibility.

The proposed methodology achieves the goals of the thesis by wrapping the DEVS CD++ toolkit. Small modifications are required at the CD++ level to allow the CD++ simulation toolkit to synchronize and communicate with other simulations. The modifications are independent of the HLA, and could be used to allow CD++ to be integrated with other target environments. The wrapper provides a bridge between the modified CD++ simulation toolkit and the HLA environment. The details of the wrapper design are presented in Chapter 5. To illustrate the application of the methodology, a case study is presented in Chapter 6. Chapter 7 provides a critical analysis of the overall correctness of results and discusses performance issues.

The scope of the thesis is limited to:

- Defining a simulation framework using the RTI Next Generation 1.3 (RTI-NG 1.3) implementation according to the HLA interface specification version 1.3.
- The C++ programming language, Unix pipes and Unix operating system are used in the case study.
- Demonstrating the wrapper method using the DEVS CD++ toolkit case study. For this case study, a Vending Machine simulation is used.
- The use of the RTI time management service is limited to its discrete event time management.
- The performance of the integrated system is not measured against the original standalone system and is not the focus of the research.

The contributions of this research are:

- An HLA-based simulation backbone that supports simulation reuse by integrating simulation tools on to the RTI using the wrapper method. Timing issues and data transformation issues are resolved.
- A wrapper federate architecture designed as the interface layer between the CD++ toolkit and the RTI.
- A general methodology for achieving integration of a DEVS simulation onto the RTI.
- True transparency is achieved during integration. DEVS models do not require any changes in order to be executed on the HLA. HLA federate developers do not require the knowledge of a specific DEVS tool in order to reuse DEVS simulations.
- Analysis of the performance of the system to verify the effectiveness of the wrapper method and suggested solutions to further improve its efficiency.

Chapter 5 The HLA Wrapper

This chapter presents the system level HLA-based wrapper framework and the wrapper development procedure. The target standalone simulation to be reused is encapsulated in a wrapper federate. The resulting HLA compliant simulation is structured into four layers as described in Figure 5-1. The proposed wrapper for incorporating DEVS CD++ simulations into the HLA simulation platform is a major component in this research. Figure 5-2 shows how the wrapper federates may participate in a federation.

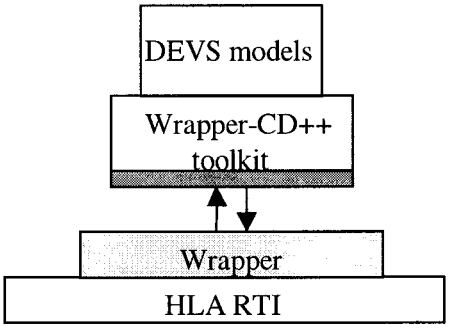


Figure 5-1: High Level View of the Architecture

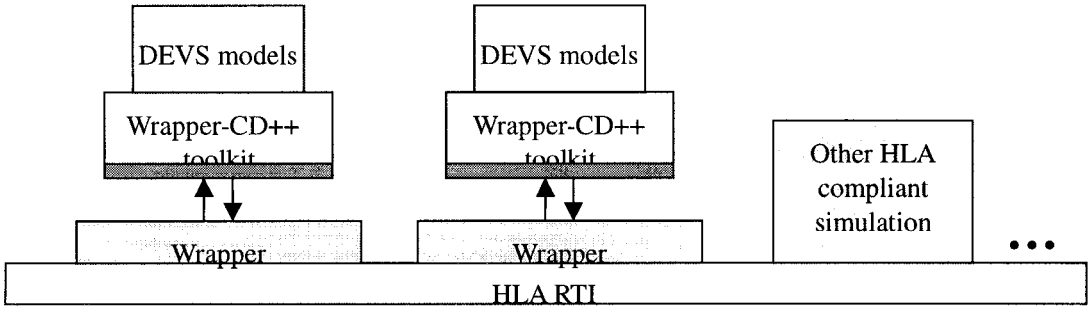


Figure 5-2: Wrapped Simulations Execute in a Federation

As shown in Figure 5-1, originally, DEVS models execute on the standalone CD++

toolkit. After minor modification (the shaded part of the new toolkit), the new CD++ toolkit, called, Wrapper-CD++, is able to communicate with the wrappers via the pipes. The wrapper is an HLA compliant federate that acts as a translator between the HLA and the CD++. The DEVS models running on the top layer remains unchanged in the new environment and the use of the HLA is transparent to the models. At run time, the wrapper federate joins the federation the same as a regular federate does (Figure 5-2), while the wrapped simulation carries out all the simulation behaviour of the federate. As a result, the originally standalone simulation can now participate as a member of the new distributed HLA simulation and reused.

For convention, the original CD++ toolkit will be referred as the *old CD++ toolkit* while the new Wrapper-CD++ toolkit is referred as the *new CD++ toolkit* in later text. The remaining sections in the chapter explore the roles and responsibilities of each layer in the new framework, and the cooperative management of time.

5.1 Inter-Process Communication, Pipes

The solution allows both the new CD++ toolkit and the wrapper run independently as processes. This approach minimizes the changes involved, preserves the integrity of the CD++ toolkit, maximizes system performance, and achieves true transparency for simulation developers. Communication and coordination between the processes is achieved using an IPC mechanism. The arrows connecting the Wrapper-CD++ toolkit and the wrapper in Figure 5-1 represents the IPC mechanism used. Depending on the

targeting operating systems and designer's preference, a variety of IPC mechanisms [28,29,30] could be used, for example pipes, sockets, and semaphores. The pipe is a very common IPC method and is used in this research. At runtime, both the new toolkit and the wrapper execute independently, and synchronize and exchange information by reading from and writing to the pipes. The coordination between the two is achieved by the pipes' blocking mechanism. When an application tries to read from a pipe, it is blocked until there is information available to read. This blocking mechanism during reading makes pipes a key factor for integrating CD++ simulations with other simulation environments. The details of pipe implementation and the coordination between the new toolkit and the wrapper are introduced in Chapter 6.

5.2 Wrapper-CD++, The New Toolkit

One of the key issues to reuse DEVS CD++ simulations is to enable the standalone CD++ toolkit to communicate with other environments. The original CD++ toolkit must be modified to be able to access the pipes and exchange information. The modifications are shown shaded within the Wrapper-CD++ toolkit in Figure 5-1. These modifications enable the new toolkit to communicate and synchronize with the wrapper and are categorized into five main topics:

1. Pipe communication, channel initialization, operation and destruction.
2. Language definition used to exchange information with the wrapper.
3. Time synchronization while processing events.
4. Receiving and executing dynamic events from the wrapper.

5. Waiting, instead of terminating the simulation, when there are no pending events for future processing.

Although the information exchanged with the wrapper is simulation specific and may differ among simulations, the modifications to the CD++ toolkit have been designed to support CD++ messages. The toolkit does not require further modifications for specific simulations, or while being integrated into the HLA. Details of modifications are to be discussed in Chapter 6.

5.3 The Wrapper

The wrapper plays an important role in the integration. The wrapper provides a generic bridge between the new CD++ toolkit and the HLA RTI. It also provides simulation-specific HLA interactions (documented in the SOM). The wrapper enables DEVS CD++ simulations to participate in HLA federations by controlling all access to the RTI services.

A wrapper is an HLA federate, which carries out typical management tasks including:

1. Creating/joining a federation.
2. Declaring the data/events it will send or publish to the federation. Also, declaring data/event that it is interested in receiving or subscribing to/from other federates.
3. Initializing objects/interactions and registering them with the federation.
4. Resigning from the federation.

Other than the basic HLA related tasks listed above, the wrapper also does the following to make use of the CD++ simulation in the context of the entire simulation:

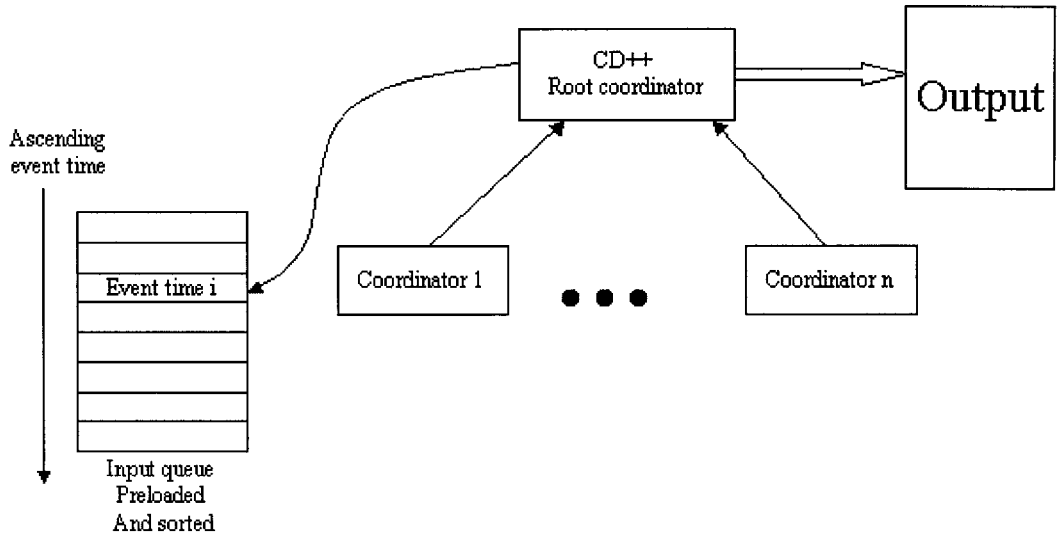
1. Initializing and starting the associated new CD++ toolkit process.
2. Creating, initializing and destroying the pipes.
3. Converting and packaging data for pipe communication.
4. Initializing the federate as time regulating and time constrained with zero look ahead [31].
5. Receiving and buffering interactions from other federates.
6. Requesting a future time advance when there are no pending events to process (in order to keep the simulation going).
7. Identifying data sources and apply the correct data structures.

5.4 Cooperative Time Management

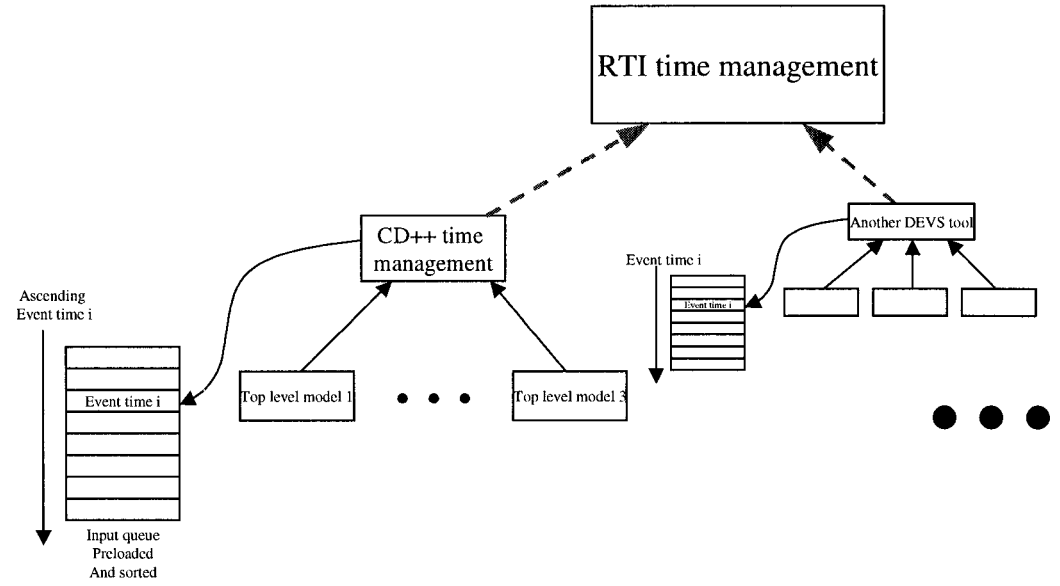
Time management plays a critical role in DEVS simulation. Time management in a new integrated federate (recall Figure 5-1) requires the joint effort of the new toolkit, the pipes, the wrapper and the HLA RTI.

As in the old toolkit, the new toolkit is still responsible for its local time management; however the new toolkit must also account for asynchronous events from other federates. Figure 5-3 illustrates the root coordinator before and after the integration. Recall the old toolkit root coordinator previously (show in Figure 2-2) has been

reproduced in Figure 5-3 (A) to allow easier comparison to the new toolkit. Before integration, the root coordinator performs time management locally for its simulation. After integration, the root coordinator of the new toolkit now has a parent (i.e. the HLA RTI). The new root coordinator must wait for time grant information from the RTI via the pipe before it can go forward. The RTI actually becomes the new root coordinator of the integrated system. Figure 5-3 (B) illustrates the integrated system in the DEVS context where the RTI manages the time exactly as a root coordinator of a DEVS system does.



(A) Before Integration (Figure 2-2)



(B) After Integration

Figure 5-3 CD++ Toolkit Before and After the Integration

The broken red arrow connecting CD++ root coordinator and the RTI time management in Figure 5-3 (B) represents the CD++ simulator extension and wrapper that integrates a DEVS simulation with the HLA RTI.

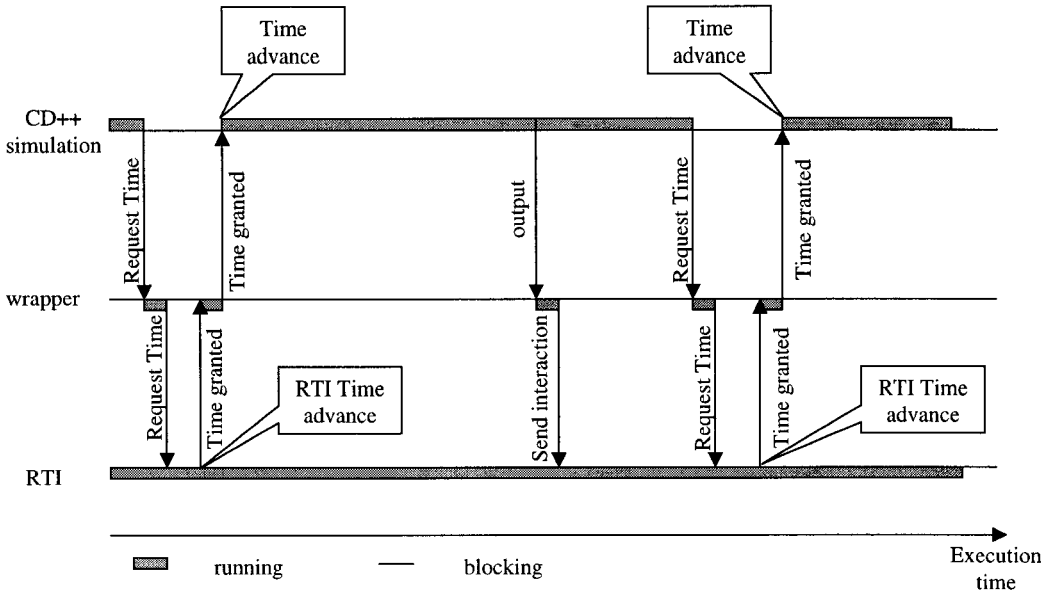


Figure 5-4 Time management over the three layers

The detailed time management coordination among the various components is described from a message sequence perspective in Figure 5-4. It illustrates how the integrated system achieves the time management over the different layers. It consists of four components, namely, the CD++ simulation, the pipes, the wrapper and the RTI. The shading indicates is when the components are running, while the line represents the components being blocked or waiting for messages. From the left, the CD++ simulation executes after the wrapper federate joins the federation. The new toolkit finds the time of its next event to process and makes a time advance request to the wrapper via the pipe. The toolkit is blocked until the wrapper replies with a time grant. The wrapper forwards the time advance request to the RTI. It then waits until a time advance grant is received from the RTI; at which time, it forwards the grant to the toolkit thereby allowing the simulation to proceed. At the RTI level, the RTI provides time management for DEVS simulations via the service call nextEventRequestAvailable(time). The RTI waits until all regulating federates in the

federation have requested a time advance by making the service call, and then grants a time advance to the federate with the earliest request.

The coordinated time management ensures the CD++ DEVS simulation and other DEVS simulations running on the HLA RTI proceed correctly. The detailed implementation of each layer is disclosed in Chapter 6.

Chapter 6 Case Study: Vending Machine

This chapter describes the detailed implementation of a Vending Machine case study. The example demonstrates the correctness of the target framework, the wrapper design and the time management solution proposed in chapter 5. The case study verifies the high level approach shown in Figure 5-1.

This case study illustrates, as a proof-of-concept, that CD++ simulations can interoperate with HLA federates using wrappers. Some key features of the wrapper are:

- Use of the wrapper is transparent to CD++ models.
- Modifications to the CD++ simulation toolkit were easily implemented, and required changes to only two functions.
- Runtime interactions between the CD++ simulation and the wrapper are implemented using pipes.
- The wrappers employ simple HLA interaction mechanisms, which have an intuitive mapping to the semantics of DEVS, and allow CD++ simulations to participate as time-constrained and time-regulating federates.

6.1 Modifications to the CD++ Toolkit

The implementation of the wrapper required simple modifications to the CD++ toolkit. The changes enable simulation time advances to be coordinated through the RTI, allow the simulation to communicate with the wrapper at runtime, and allow a simulation to

continue as a participant in a federation even though the simulation has no pending events (CD++ simulations normally terminate when they have no pending events). The ability to synchronize simulation time advances with external time management is essential if a simulation is to interoperate with other simulations. Communication between the CD++ simulation and the wrapper is implemented using pipes.

Converting CD++ simulations into HLA federates must address several issues. CD++ simulations are standalone programs, and are designed under the assumptions that files supply all inputs and record all outputs, and that the simulation toolkit is in control of all decisions about time management. These assumptions are quite realistic for standalone programs, but are impractical for components in a distributed simulation. The conversion into a federate must account for interfacing with the RTI, asynchronous communication with other federates, the sharing of time management among federates, and following the procedures for participating in federations. The modifications are local to the toolkit only, and do not result in any changes to CD++ models.

6.1.1 Communications with the wrapper, The Channel

To enable interoperability, the new Wrapper-CD++ toolkit must communicate with the wrapper. This section briefly explains the how pipes are used in the case study to serve the purpose.

Pipes are an operating system supplied method for inter-process communication. Pipes

provide a unique blocking mechanism.

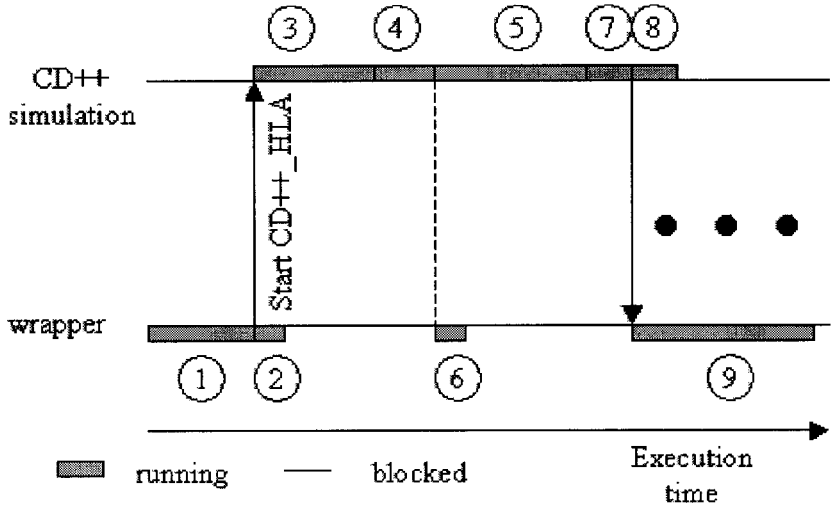


Figure 6-1 Wrapper and CD++ Initialization and Pipe Process

Figure 6-1 illustrates how the pipes are used to coordinate the communication between the new toolkit and the wrapper. The wrapped simulation must join the federation and the wrapper starts by initializing itself, followed by a sequence of RTI service calls to create the federate and join the federation. After HLA specific initialization is performed, the wrapper must initialize the new toolkit. This is done by creating the pipe followed by starting the corresponding new toolkit. In the Unix environment, the new toolkit process can be started by executing:

system("xterm –e newToolkitname newToolkitParms&").

Step 1 in the above figure presents the process. The wrapper and the new toolkit now become two separate processes running independently.

To enable the communication, wrapper calls pipe initialization (step 2 in Figure 6-1) to open the channel for information exchange with the new toolkit. This initialization

operation immediately blocks the wrapper due to the design nature of the pipe. To use a pipe, two parties at both ends must both initialize the pipe before they can proceed. The same blocking mechanism is used while reading and writing to the pipe. Both sides must be present in order for the communication to complete; otherwise, one party is blocked until the other party joins to complete the information exchange. For example, suppose that two processes A and B use pipe P, and A reads from P but B has not yet written to P. In this case, A is blocked until B performs the write. In Figure 6-1, the wrapper is blocked at step 2 until the new toolkit initializes the pipe at step 4.

Step 3 in the figure represents the new toolkit initialization. The new toolkit then initializes the pipe (step 4) resulting in the release of the wrapper. At this point, the wrapper has finished its initialization phase. The wrapper then reads from the pipe to obtain the CD++ simulation status from the new toolkit (step 6) and is blocked again.

Step 5 in the figure represents the new toolkit operation in which it finds the next event to execute and makes a time advance request by writing to the pipe (step 7). The toolkit immediately reads the pipe (step 8) to block until the wrapper forwards information to it. At step 9, wrapper is released when it receives the new toolkit request and forwards the information to the RTI for further processing.

6.1.2 Communication to the Wrapper, The Language

The wrapper presents a well-defined runtime interface to the CD++ simulation, and isolates the simulation from the RTI. The new Wrapper-CD++ toolkit communicates

with the wrapper via a generic data structure defining all necessary information to be communicated.

```
struct{
        int msgId;
        char* time;
        char* port;
        char* value;
}cd_wrapper_msg;
```

Figure 6-2 Data Structure Used to Communicate Between CD++ and the Wrapper

Figure 6-2 shows the communication data structure. In the structure, time is the time stamp of the DEVS event. Port represents the output port of the DEVS event. Value is the data sent out through the port. The msgId identifies the various types of messages that can be exchanged. The msgId values represent different messages sent from the Wrapper-CD++ toolkit to the wrapper and vise versa. The details are described below.

When sent by the toolkit to the wrapper, msgId has three possible values:

1) **Done:** This message indicates that CD++ has processed all local events and is now idle and waiting for inputs. For this message, the time, port and value fields do not contain meaningful values.

2) **TimeAdvanceRequest:** This is a time advance request message. For this message, the port and value fields do not contain meaningful values.

3) **OutputAvailable:** This is an output event that is destined to a federate. Both the port

and the value fields have meaningful values.

Messages from the wrapper to the toolkit could be one of the following three values:

1) **Done:** CD++ shall terminate upon receipt of this message.

2) **TimeAdvanceGrant:** This message is a grant to advance time to the time of the current outstanding TimeAdvanceRequest of the CD++.

3) **NewEventArrived:** This message contains a new event that has arrived as an input. The CD++ simulation should advance its local time to the time of the event and process the message immediately, without any delay. The time of this event is no later than the time of any currently outstanding TimeAdvanceRequest, and this message also cancels any currently outstanding TimeAdvanceRequest.

6.1.3 Wrapper-CD++ toolkit runtime state changes

The original standalone CD++ toolkit executes a simulation by following a simple loop.

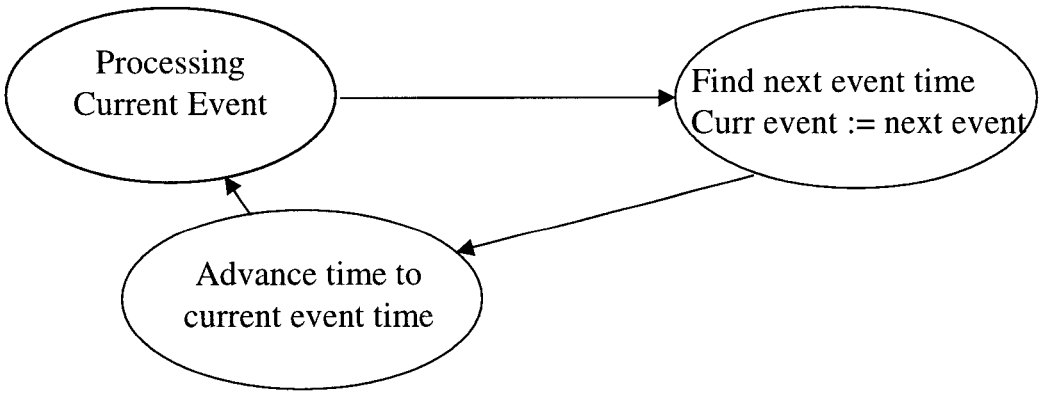


Figure 6-3 Old Toolkit State Change

Figure 6-3 demonstrates the old toolkit state changes. The toolkit always processes the current event, then finds the next available event with the earliest time stamp. The found event is made the current event, and simulation time is advanced to the current event time. This loop repeats until the simulation ends.

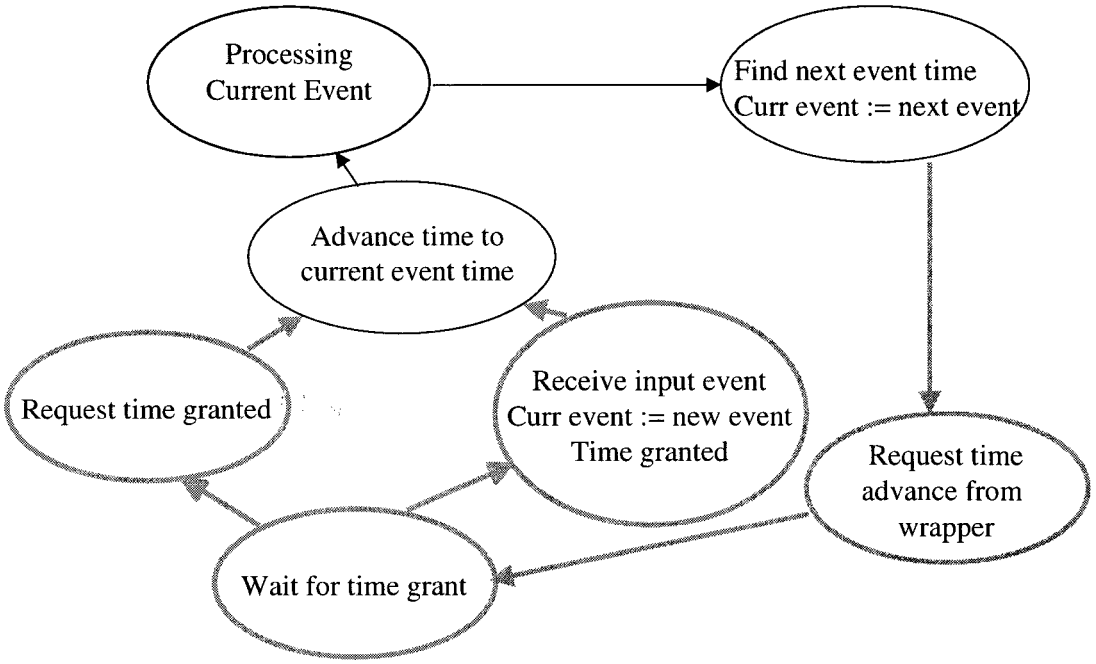


Figure 6-4 State change of the new toolkit

The state changes of the new toolkit are depicted in Figure 6-4. The newly added states and their direction map are shown in read bold. The execution procedures are modified only to request a time advance from the wrapper prior to executing a local event. This triggers a number of state changes. At start up, the toolkit finds the first available local event to be executed and sends a request to the wrapper to advance to the time of that event. If no event is available, it simply sends a "done" message to the wrapper and

enters the idle mode waiting for new event. When the wrapper grants a time request, the toolkit advances to the granted time and the pending event is executed. The toolkit then finds the next event and repeats a request for a time advance. If the wrapper responds to a time request with a new incoming event, the toolkit will execute the new event first, and then request an advance to the time of the next event in the event queue. When the toolkit generates an output event, the time associated with the event must have been previously granted and the toolkit simply sends the output information to the wrapper for subsequent distribution to subscribed federates.

6.1.4 From CD++ to Wrapper-CD++, the Code Modification

The modifications to the CD++ toolkit occur at one class, two functions and three places. All changes happen at the root coordinator level. In the CD++ context, they happen in the proot (root coordinator) class. Other than the necessary data structure definitions and pipe initializations, two functions were affected. The first function is called when the root coordinator receives an internal message. This message indicates that the root coordinator has received responses from all of its children and a decision can be made as to which event shall proceed.

```
1  receive (internalMessage)
2  {
3      if(endOfEvent and nextChange == INF){
4          //no more pending event
5          //exit(); terminate simulation
6          writePipe(done);
7          messageReturn = readPipe();
8          if (messageReturn.msgId == done){
9              exit;
10         }
11         else if (messageReturn.msgId == eventArrival){
12             newEvent = messageReturn.event;
14             process newEvent;
15         }
16         else{
17             error and exit;
18         }
19     }
20     else
21     {
22         nextEvent = Find next event for processing
23         timeToRequest = nextEvent.time;
24         writePipe(requestTime, timeToRequest);
25         messageReturn = readPipe();
26         if (messageReturn.msgId == done){
27             exit;
28         }
29         else if (messageReturn.msgId == eventArrival){
30             currentEvent = messageReturn.event;
31             process currentEvent;
32         }
33         else if (messageReturn.msgId == timeGrant){
34             currentEvent = nextEvent;
35             process currentEvent;
36         }
37         else{
38             error and exit;
39         }
40     }
41 }
```

Figure 6-5 Pseudo Code of the Modification Occur at Receive Function

Figure 6-5 details the changes involved at this function in pseudo code. The modified code is *Italic* and **Bold**. At line 3, if "`endOfEvent and nextChange == INF`" is true, it means that all input events are processed and there are no more pending internal events. Previously, line 5 would terminate the simulation under these conditions. After the modification, line 5 is commented out. The first place where code changes involved is from line 6 to line 18 where new code is added. Instead of terminating, the new toolkit sends a "done" message (line 6) to the wrapper by writing to the pipe. It then immediately reads the pipe for further instructions. Upon wrapper's reply, the code proceeds to line 8 where the message contents are checked. If a termination command is received, the wrapper terminates. If a new event arrives, the wrapper processes the new event (line 11). The second code change in this function starts at line 23. After the root coordinator finds the next local event to process (line 22), it requests a time advance with the time stamp of its next local event (line 24). Line 24 returns control when the wrapper replies. If the requested time is granted (line 33), the toolkit proceeds by execute the next event. If it is a termination command, it exits (although this is not suppose to happen since the federate still has events to process). If the wrapper replies with a new arrival event that has an earlier time stamp than the requested time (line 29), the current event is assigned the new arrival event and is processed.

The second function in the class where code changes are involved is also named ***receive***; except, the parameter is an output message. This function is called when an output event is available.

```
1  receive (OutputMessage)
2  {
3      outputEvent = OutputMessage.event;
4      outputString = createOutput (outputEvent);
5      writeToOutputFile (outputString);
6      writePipe (Output, outputString);
7  }
```

Figure 6-6 Pseudo Code of Receive (outputMessage) Function

Figure 6-6 illustrates the old and new. Once again, the new code is shown in *Italic* and **Bold**. This modification only involves one line of code to forward the contents of output events to the wrapper (line 6) as well as writing them to the output file (line 5). This function does not include any time advancement, as the output event time has already been granted prior reaching this function.

These modifications conclude the changes to the DEVS CD++ toolkit. The changes are simulation independent and no further modification is required when different simulations are executed in the new toolkit.

6.2 Wrapper Design and Implementation

The wrapper is an HLA compliant federate. Its responsibilities are to:

- Enable the wrapped CD++ simulation to communicate with the HLA RTI
- Make time requests and send output for the CD++ simulation
- Inform the CD++ simulation when time grants or interactions are received
- Filter unnecessary inputs that might be received

- Convert between the RTI and toolkit message structures, including time/ports/values

6.2.1 Wrapper operations

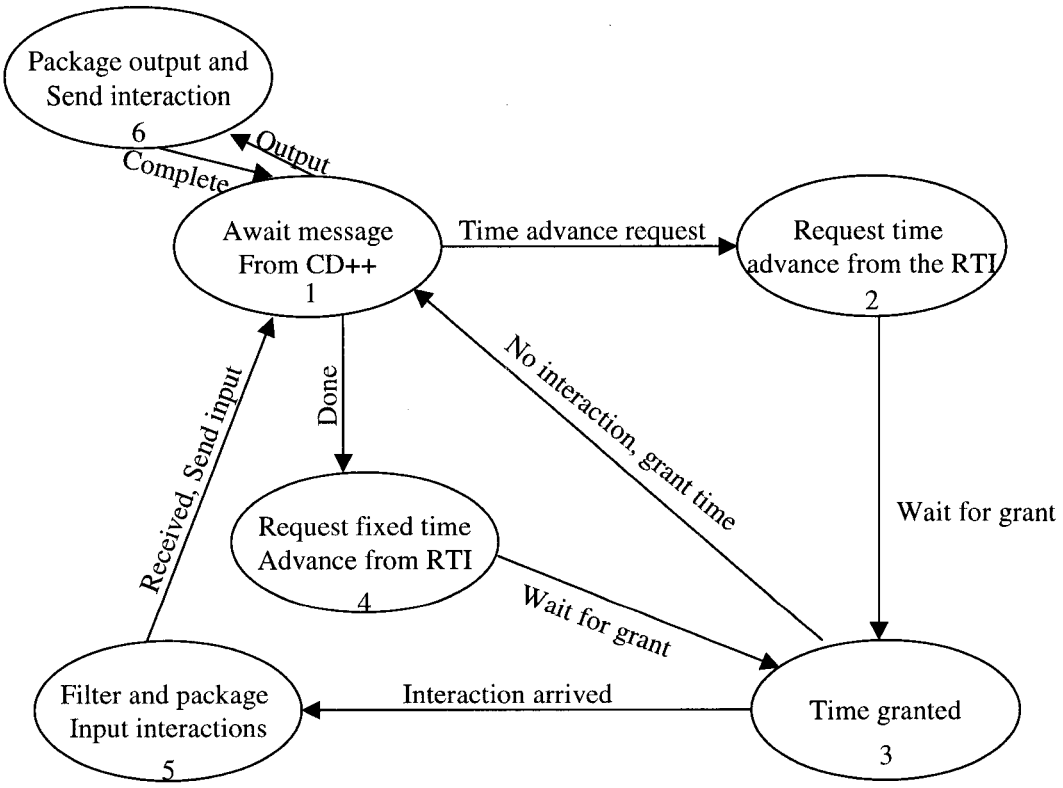


Figure 6-7 Basic Wrapper State Machine

Figure 6-7 illustrates basic wrapper operations by a state machine. At the wrapper level, the execution is a "package and delivery" mechanism. The wrapper first reads from the pipe to get the next event time requested by the CD++ simulation (State 1). The message could have three different meanings:

- If it is a **TimeAdvanceRequest**, the wrapper makes the time request to the RTI and waits for the time to be granted (proceed to state 2). Two things may happen thereafter. If the time is granted (state 3), the wrapper sends a grant message to the CD++ toolkit and tries to read the next message back from the CD++ simulation (proceed to state 1). If the incoming event is an interaction from other federate, the wrapper translates the interaction information into a CD++ message (step 5) sends it to the CD++ simulation (proceed to state 1), and then waits for the next command from the toolkit (state 1).

- If it is a **Done** message, it indicates that this CD++ simulation had finished all of its execution, the wrapper will then request to the RTI a fixed amount of time advance (proceed to state 4) to keep the entire simulation running until a valid incoming interaction is received from another federate, or when the entire simulation terminates.

- If it is an **OutputAvailable** message, the wrapper simply sends the interaction to the subscribed federates (state 6) and waits for the next command from the CD++ toolkit (back to state 1).

The above state machine illustrates the basic wrapper operation procedures. In reality, the design of the wrapper used in this case study is more sophisticated, and the details are explained in 6.2.4.

6.2.2 FOM and Interactions

The HLA rules require that all interactions among federates must be declared in the FOM. The wrapper must also comply with this policy. To ensure the correctness of message transmission among the simulations of federates, each message received from the Wrapper-CD++ toolkit has to be mapped onto the corresponding interactions/objects to be described in the FOM in order for it to be communicated with other federates. The design of the data transformation and transmission has the following characteristics.

Only interactions are being used for messaging. Due to the nature of the communication method (events) used in DEVS, interaction fits into this structure naturally and is easy to understand.

Each interaction used by the wrappers contains a message structure that is understood by all the wrappers. A simple structure shown in Figure 6-8 is used to describe the information required among the different DEVS models. This structure contains three attributes: *eventPort*, *eventValue* and *eventTime*. *EventPort* identifies which output port of the DEVS model this message belongs to. *EventValue* contains the information the port is sending out. *EventTime* is the time of the event, it is not being used since the interaction is timed by the RTI. The *eventTime* is currently used as a debug mechanism to ensure that all time lines conform with each other. The *eventPort* and *eventValue* attributes have the data type of string to facilitate transmission. By defining this structure, the information transmitted between two wrapped federates is transparent to

the RTI. The wrapper is designed to understand this structure and the information in the structure can be easily converted back to meaningful information when the receiving wrapper receives it.

```
struct{
        double  eventTime;
        char*   eventPort;
        char*   eventValue;
}wrapper_rti_msg;
```

Figure 6-8 Data Structure Used in the Interaction Class

Using only one published interaction per-federate can greatly reduce the number of interactions a federate must define, and it also significantly eases the wrapper creation and FOM development.

Each interaction described in the FOM must contain the following information: interaction name, transmission method, interaction type, and any parameters. Since there is only one published interaction per federate, the wrapper's interaction name described in the FOM can follow the convention of federateID_DEVS and is listed in the interaction table of the FOM. The transmission method is always set to be *reliable* and the nature of time regulating federates requires that the interactions be *timestamp* type. Because there is only one structure used in the interaction, the interaction has one parameter. Simply listing the parameter name as federateID_output in the interaction class is sufficient for the FOM. The FOM defaults all newly declared parameters to data

type "any" so that the detailed structure type is hidden from the RTI. Figure 6-9 depicts a sample interaction class declaration found in the FOM.

```
(class federateID_DEVS reliable timestamp
     (parameter federateID_output)
  )
```

Figure 6-9 FOM Interaction Definition

6.2.3 Data Conversions

The wrapper's responsibility as a translator for the CD++ and the HLA RTI requires resolving data incompatibility between the two. The incompatibility exists in terms of time representation and event description.

6.2.3.1 Time Conversion

Different simulation environments use different representations for time. Some of them use integers, some use floats and others may use the form of hh/mm/ss. CD++ and HLA differ in the representations of time. CD++ uses the form of hh/mm/ss/ms, while the HLA uses type RTIfedTime that can be initialized from a double value. In order for the time management to perform correctly, the CD++ time is transformed into HLA time using hh*3600+mm*60+ss+ms/1000 when wrapper requests time advancement. On the other hand, RTI time format is transformed back to the CD++ time format at the

receiver wrapper federate.

6.2.3.2 Data Conversion

CD++ models communicate with each other via events. In the previous discussion, the design of the communication between the Wrapper-CD++ and the wrapper uses the data structure defined in Figure 6-2, while the RTI interaction contains the data structure defined in Figure 6-8. Converting from Figure 6-2 to Figure 6-8 only requires minimal effort.

6.2.4 Other Issues

This section lists some issues and their solutions used in the case study in addition to the general design of the wrapper. These issues include receiving multiple interactions at one time, message filtering, and continuous time advance requests when CD++ has nothing to process.

6.2.4.1 Receiving Multiple Interactions

At any circumstances, there may be a chance that a wrapper federate receives more than one interaction at time X. In a regular HLA federate, the federate processes these interactions one by one in the call back functions. In the wrapper case, it adds extra complexity. The wrapper cannot go ahead and send the information to the

Wrapper-CD++ until a time grant is received. If all the messages are saved and sent to the Wrapper-CD++ in one shot from the pipe, an extra data structure would have to be declared to hold all these messages, the size of the message is also difficult to determine. Furthermore, by receiving many input events at a time, Wrapper-CD++ would also encounter more changes and added complexity.

To overcome the problem, the wrapper is designed to hold any number of messages using a linked list. Whenever an incoming interaction is received, the wrapper's call back function checks whether it is needed (filtering, discussed in detail in the next section) and adds it to the linked list. After the time is granted, the wrapper sends the head of the list to the Wrapper-CD++ toolkit and de-lists the head. The Wrapper-CD++ will return a time request after it processes the message in which case the wrapper sends a second node in the list and receives a new time request. This process repeats until the list is empty. The Wrapper-CD++ finally processes all the messages and determines a new time request. The wrapper then requests an RTI time advance for that time request in order for the simulation to go forward. This approach requires no change at the Wrapper-CD++ level and it can hold as many interactions as possible; however, the looping between the C++_HLA and the wrapper using the pipes may slow down the simulation depending on the number of valid interactions received at one time.

6.2.4.2 Message Filtering

The wrapper has filtering capability that blocks unnecessary messages to the CD++

simulation. Unnecessary messages may occur since there is only one interaction published per wrapper, and outputs for all ports use the same interaction class. The receiving wrapper must know if the message is from a port that it is intended to receive. To verify, the receiving wrapper first extracts the port information from the interaction and compares it with an interaction-port table (shown in Table 6-1). This table is built at compile time based on information about the entire simulation structure, and it lists the ports of an interaction class that are relevant to a federate. Unnecessary interactions are ignored after filtering; however, in the cases where only unnecessary interactions are received, the wrapper must reissue a time request with the previous time stamp since a time advance has been granted but not needed (when it received the unnecessary interaction).

Interaction_name1 port1 port2 port3 Interaction_name2 port1 port4 port5 Interaction_name3 port0 port2 port4 Interaction_name4 port3 port9 port12

Table 6-1 Interaction-Port Table

Table 6-1 shows an example of an Interaction-port table for a wrapper. Each interaction class name is followed by a list of ports that the wrapper should accept for that class. Any interactions received with ports not in the list should be filtered by the wrapper and ignored.

6.2.4.3 Receive Interactions From Non-CD++ Federate

When the wrapper receives an interaction that is not sent from a DEVS CD++ federate, the previous described data conversion in 6.2.3.2 does not apply. In this case, the wrapper has to understand in advance the information it subscribes and the conversion mechanism. This may vary from federate to federate and depends on the non-CD++ federates that are involved in the simulation. Also, because the wrapper communicates with other HLA-compliant simulations, in cases where an object is used instead of an interaction, the wrapper must be able to convert into event-driven behavior using the structure defined in Figure 6-2. This case study has only integrated a very specific HLA-compliant federate and the Vending Machine CD++ simulation, the wrapper can only recognize the CD++ toolkit. When other simulation tools are incorporated into the simulation, the wrapper requires their data structure conversion information as well in order to perform correctly.

6.2.4.4 Continual Time Advance Request When CD++ Has Nothing to Process

Continual time advance is needed in order to keep the simulation going forward. In this case study, all federates involved in the simulation are both time regulating and time constrained. At the RTI level, the RTI must receive time requests from all of the federates in the federation in order to make a time grant decision, if there is one federate that does not request a time advance, the entire simulation will halt. As a result, care

must be taken to ensure that federates continue to request time advances. Of particular concern is the case where Wrapper-CD++ has processed all of its pending events and has no inputs left in the input file. Since the federate must stay alive for future interactions, its wrapper must make a time request in order to keep the simulation going.

An infinite time request would be the ideal solution for the case of concern. Unfortunately, the version of RTI used in this case study does not support the notion of infinite time, and the wrapper must continuously make time advance requests to the RTI. To solve this, the time request is set to be the current local time plus 10000. If this time expires (i.e. is granted) without receiving an interaction, the wrapper again adds 10000 and makes another time request. This is repeated until a valid interaction is received. The time grant associated with the arriving interaction cancels the time advance request, and the message is forwarded to the new toolkit for further processing. Whenever a done message is received from the Wrapper-CD++ again, this situation repeats. When an unnecessary interaction breaks the cycle, the wrapper simply reissues the previous time request to keep up with the looping.

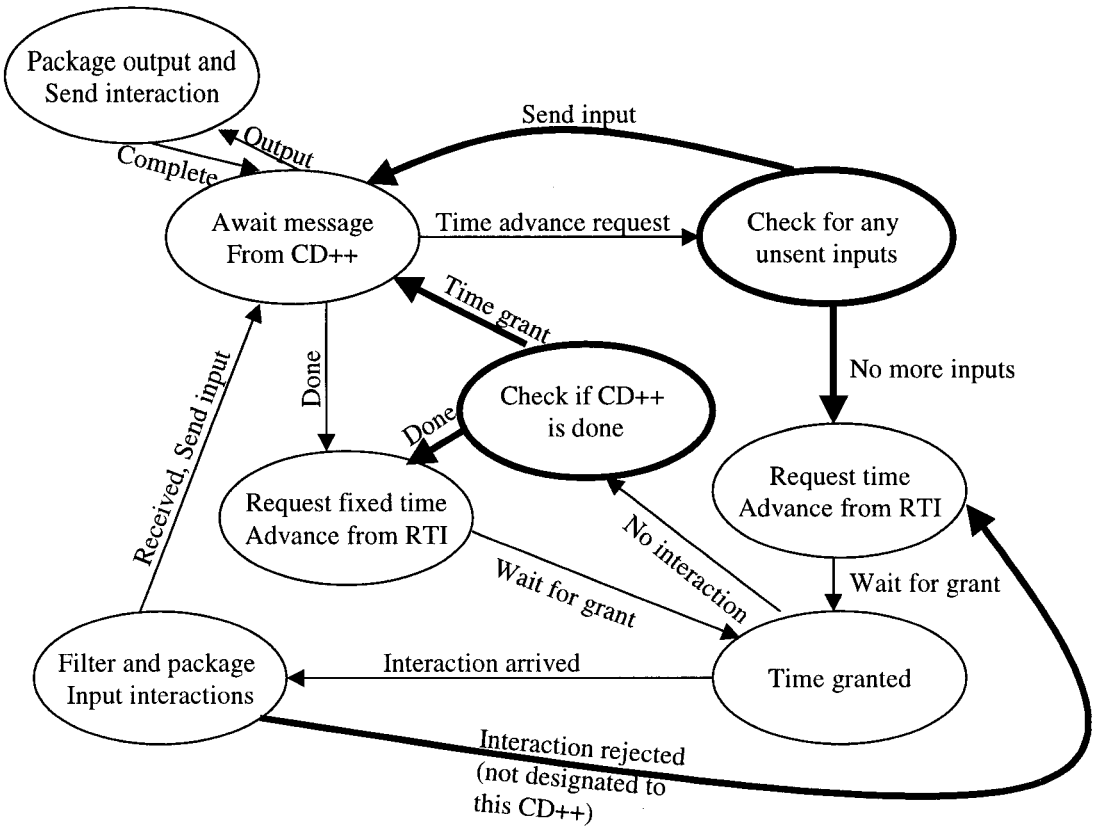


Figure 6-10 Updated Wrapper State Changes

The new state machine of the wrapper looks like Figure 6-10. The state machine for wrapper operation (recall Figure 6-7) must be extended as shown in Figure 6-10 to account for unnecessary interaction, receiving multiple inputs and the continual time advance requests. The newly added states and connections are in **bold**. The reception of multiple inputs is handled by the top right state. The state checks for any queued interactions and forwards them one at a time to the Wrapper-CD++ toolkit. In the middle, when a time advance is granted while CD++ is idle, the wrapper requests another fixed time advance from the RTI. At the bottom, when an interaction not designated to this CD++ is received, the wrapper filters (ignores) the interaction and re-requests the previous time advance from the RTI.

6.2.5 Wrapper Generation

Although wrappers present a well-defined interface to the HLA and to the CD++, each wrapper is subject to change when different CD++ simulations are wrapped. The same situation applies when its subscribed information changes. A wrapper code generator would help in facilitating the process of wrapper creation. This research proposes a wrapper generator solution but the actual generator has not completed. The following information is necessary for a wrapper to be created.

1. Wrapper name: Wrapper name is needed, as it is also the name of the federate.
2. Interaction to publish: Usually wrapper name and a suffix such as int (meaning interaction) i.e. wrapperName_int.
3. Outputs that are to be published and outputs that are final: When the CD++ outputs an event, it could be an output from the entire simulation, or it could be an output to another model. The wrapper must be able to identify the outputs that should be sent to other federates and the outputs that should be placed in the output file. This output file is not the output file produced by the CD++.
4. Interactions to subscribe: Knowing where the inputs are from in order to receive them, usually a table is associated with the subscribed interaction so that only the necessary input are being transferred to the CD++
5. Name of the corresponding CD++ executable: This name is needed in order to start the CD++ executable.

6. Location of the pipes: The location of the pipes must be supplied to the CD++.

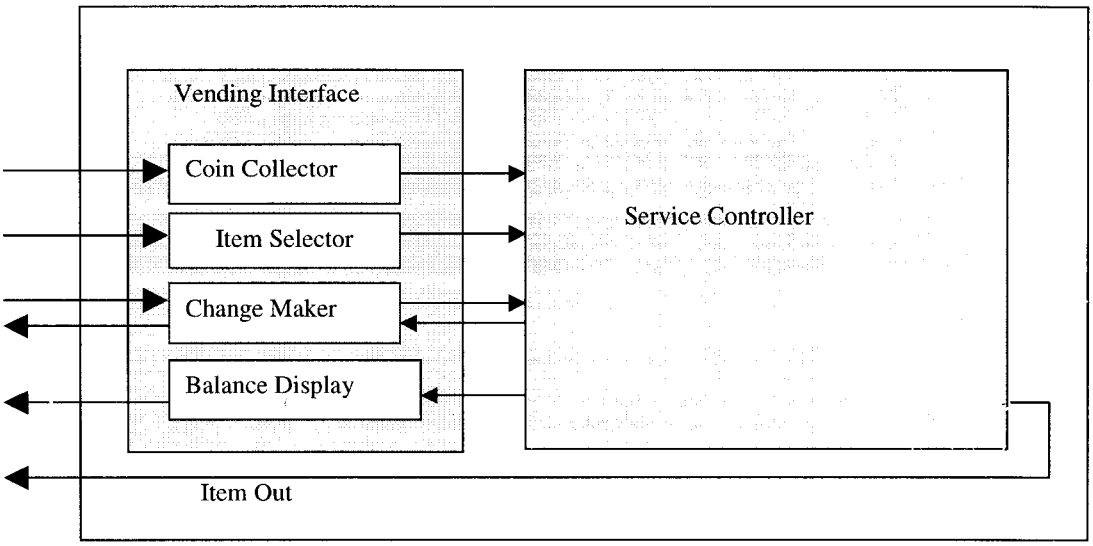
6.3 Splitting CD++ Models

Splitting CD++ coupled models may be needed to achieve distributed simulation of CD++ models and to reuse a subset of existing CD++ simulations. Often, sub-models can be more easily reused than entire simulations. In a car simulation for example, the car engine may be reused in another simulation; in a combat simulation, the tanks and the fighters might be reused in another combat simulation. This situation may require splitting the original simulation into components containing the models to be reused.

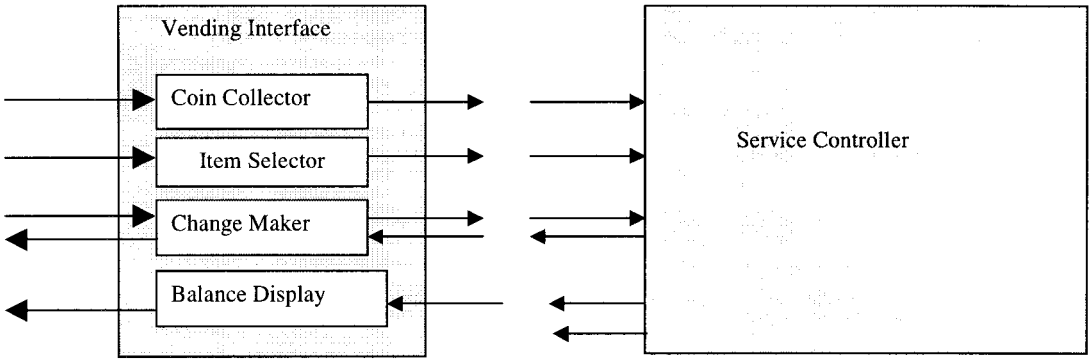
Achieving the split in CD++ is quite simple due to the hierarchical design nature of the CD++ toolkit. A CD++ simulation consists of two parts, the simulation model descriptions and the set of model functions. The model descriptions an placed in the MA file (recall Figure 2-5) and the functions of each model are defined in C++ classes inherited from atomic class and built into the CD++ executable at run time. Splitting a CD++ simulation only requires splitting the model description file. The model functions can be included into the new Wrapper-CD++ executable without any changes, since any unreferenced classes will not be included.

The split of the MA file is straightforward. If a coupled model needs to be taken out, simply create a new MA file can be created by copy and paste of the coupled model description. In the new MA file, the coupled model must be identified as "[top]". Any

atomic models included in this coupled model shall have their preparation time included in the new MA file as well. This process can be done manually by the model designers or may be done automatically by a tool. In this case study, the split of Vending Machine models has been done manually.



Before Split, Original Vending Machine Models



After the Split

Figure 6-11 Split of Vending Machine Models

Figure 6-11 illustrates the Vending Machine models before and after a split. Both the

Vending Interface and the Service Controller coupled models have become top models, and they can both participate in a federation as two independently wrapped simulations.

Chapter 7 Case Study: Verification and Performance Analysis

This chapter presents the results of the Vending Machine case study described in Chapter 6. The correctness of the implementation is verified using different test cases. The performance of the simulation is also discussed.

7.1 Test Set Up

Figure 7-1 shows the distributed simulation system used for testing. The HLA platform is executed above Unix. Each federate is executed on a dedicated workstation connected to a TCP/IP intranet. The DEVS CD++ Vending Machine simulation is split into two pieces. The Vending Interface running on the left workstation, the Service Controller module running on the center and an HLA compliant Customer federate on the right. The customer HLA federate simulates the behaviour of a customer by sending interactions (e.g. inserting coins) to the Vending Machine and receiving interactions (e.g. items) from the Vending Machine. The interactions are recorded for verification purpose.

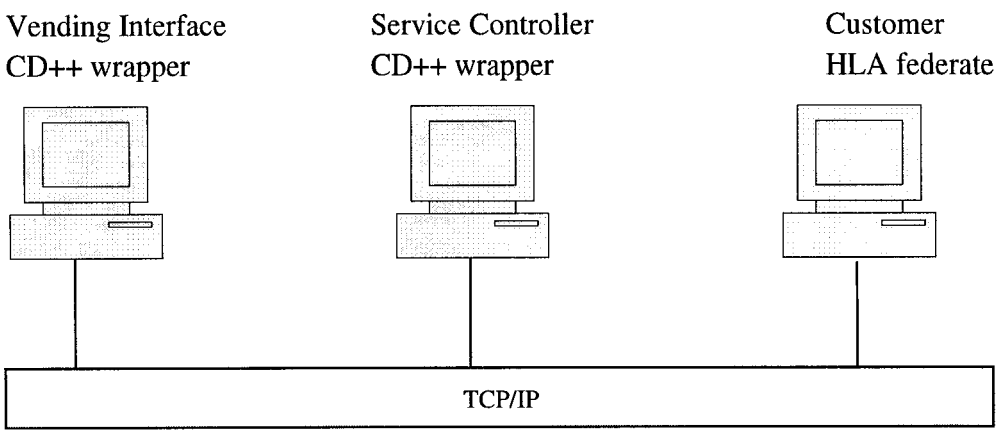


Figure 7-1: Distributed Simulation Platform

The following sections describe the verification of the simulation system. To permit the collection of test data, federates were instrumented to output messages to their runtime consoles.

7.2 Verification

To verify the correctness of the simulation system during the integration, the following tests were conducted:

(a) The wrapper federate, the new toolkit initialization and the pipe communication;

(b) Comparison of the results from standalone CD++ simulation and integrated simulation;

7.2.1 Test Case 1: The Wrapper Federate, The New Toolkit Initialization and The Pipe Communication

The purpose of this test is to ensure that the wrappers and the Wrapper_CD++ toolkit are initialized properly with correct timing parameters and correct pipe communications. The design of the federation allows the start up of the federates in any sequence, such as Vending Interface->Service Controller->Customer or Service Controller->Vending Interface->Customer.

```
FED_VI: CREATING FEDERATION EXECUTION
FED_VI: CREATION OF FEDERATION EXECUTION SUCCESSFUL
FED_VI: ATTEMPTING TO JOIN TimeDemo
FED_VI: SUCCESSFULLY JOINED TimeDemo
FED_VI: published and subscribed to interactions
FED_VI: ENABLING TIME CONTRAINT
FED_VI: Time granted (timeConstrainedEnabled) to: 0.0000000000
FED_VI: ENABLING TIME REGULATION WITH LOOKAHEAD = 0
FED_VI: Time granted (timeRegulationEnabled) to: 0.0000000000
Pipe creation: success
xterm -e cd++ -mvendingInterface.ma -enoEvent.ev
-ovendingInterface.out
Pipe init: Blocked here success
Pipe read: done
FED_VI: REQUEST TIME 10000.000000000
```

Figure 7-2 Screen Capture of Vending Interface Wrapper

Figure 7-2 shows the recorded start up sequences from the Vending Interface wrapper federate. The federate started up by creating and joining federation, it then created the pipes and started the new toolkit process. After conducting pipe initialization, the federate is blocked and waited for information from the new toolkit. After the toolkit has initialized, the federate was unblocked and requests a time advance from the RTI. The italic and bold text in the figure are comments added manually.

```
CD++: A Tool to Implement n-Dimensional Cell-DEVS models
------------------------------------------------------------
Version October 2001 -  Compiled for StandAlone Simulation
Copyright 1998-2001 Gabriel Wainer.
All rights reserved
Author Information:
http://www.sce.carleton.ca/faculty/wainer/celldevs/credits.html
Loading models from vendingInterface.ma
Loading events from noEvent.ev
Running parallel simulation. Reading models partition from
Model partition details output to: /dev/null*
Message log:
Output to: vendingInterface.out
Tolerance set to: 1e-08
Configuration to show real numbers: Width = 12 - Precision = 5
Quantum: Not used
Evaluate Debug Mode = OFF
Flat Cell Debug Mode = OFF
Debug Cell Rules Mode = OFF
Temporary File created by Preprocessor = /tmp/t598.0
Printing parser information = OFF
Simulation technique = Hierarchical
Printing wall-clock time = OFF

Stop at time: Infinity.
LP 0: initializing simulation objects
LP 0: starting simulation (NoTime).
To end  Nothing to process
Request info from pipe
waiting for information from the wrapper
```

Figure 7-3 Text Output Caught at the New Toolkit Console

Figure 7-3 is the screen capture of the new toolkit execution. Because the simulation was wrapped to use as a component in a bigger simulation, this DEVS CD++ simulation has no events in the input file. It prints "To end" and sends "done" to the wrapper, and is then blocked reading from the pipe. Most of the text captured are standard outputs generated by the CD++ toolkit and the new outputs reflect the

changes are mostly at the bottom of the text. Again, the italic and bold text are manually added comments to indicate progress.

The captured text from the Vending Interface wrapper federate and the new toolkit demonstrates that the wrapper initialization, new toolkit initialization and pipe communication are designed and implemented correctly.

7.2.2 Test Case 2: Comparison of the Results from Standalone CD++ Simulation and Integrated Simulation

The most effective and easiest way to verify the correctness of the integrated simulation is to compare its results with the results of the standalone alone simulation. The system set up shown in Figure 7-1 allows such a comparison. The customer HLA federate records the time and value of its output interactions and incoming interactions. Table 7-1 details the recorded (a) outgoing interactions and (b) incoming **accepted** interactions at the customer federate. The incoming interactions have passed the message filtering criteria. VI represents VendingInterface interaction (published by the Vending Interface wrapper federate) and SC represents a ServiceController interaction (published by the Service Controller wrapper federate).

Outgoing Interactions			Incoming **Accepted** Interactions		
Interaction Name	Interaction Timestamp	Value	Interaction Name and port	Interaction Timestamp	Value
Insert_coin	20.020	1.00	VI.display	20.037	1.00
Insert_coin	26.000	0.25	VI.display	26.017	1.25
Select_item	28.000	102	VI.display	29.006	0.00
Change_request	35.001	0	SC.item_out	31.005	2
Insert_coin	150.000	0.50	VI.display	36.027	0.00
Insert_coin	153.000	2.00	VI.change_out	37.044	0.00
Select_item	155.003	101	VI.display	150.017	0.50
Change_request	160.000	0	VI.display	153.017	2.50
			VI.display	156.009	1.50
			SC.item_out	158.008	1
			VI.display	161.026	0.00
			VI.change_out	162.043	1.50
(a)			(b)		

Table 7-1 Outgoing and Incoming interactions of the customer federate

Table 7-1 shows the customer bought 2 items. At time 20 to 35, the customer made four actions: insert $1.00, insert $0.25, select item #2 and request change. In response to the customer actions, the Vending Interface wrapper sends the balance and the Service Controller sends the item to the customer. This sequence is the expected behavior previously discussed in Figure 2-4 in Chapter 2.

To verify the correctness of the integrated simulation, the outgoing interactions were translated into the input events and placed in an input file for the original Vending Machine simulation to process.

Customer Outgoing Interactions			Input events of Vending Machine simulation
Interaction Name	Interaction Timestamp	Value	Input events in rows placed in an input file
Insert_coin	20.020	1.00	00:00:20:020 coin_in 1.00
Insert_coin	26.000	0.25	00:00:26:000 coin_in 0.25
Select_item	28.000	102	00:00:28:000 item_in 102
Change_request	35.001	0	00:00:35:001 request_in -1
Insert_coin	150.000	0.50	00:02:30:000 coin_in 0.5
Insert_coin	153.000	2.00	00:02:33:000 coin_in 2.00
Select_item	155.003	101	00:02:35:003 item_in 101
Change_request	160.000	0	00:02:40:000 request_in -1

(a) (b)

Table 7-2 Interaction to Input Events Translation

Table 7-2 illustrates the translation. Interaction names are translated into input ports used in the original Vending Machine models. For example, Insert_coin interaction is translated to the coin_in port. Time is also converted into hh:mm:ss:ms format (e.g. 150.000 becomes 00:02:30:000). Some special values must also be translated such as the change request. The original Vending Machine simulation use –1, but was defaulted to 0 in the HLA interaction.

Input events	Output events
00:00:20:020 coin_in 1.00	00:00:20:037 out 1.00000
00:00:26:000 coin_in 0.25	00:00:26:017 out 1.25000
00:00:28:000 item_in 102	00:00:29:006 out 0.00000
00:00:35:001 request_in -1	00:00:31:005 item_out 2.00000
00:02:30:000 coin_in 0.5	00:00:36:027 out 0.00000
00:02:33:000 coin_in 2.00	00:00:37:044 change_out 0.00000
00:02:35:003 item_in 101	00:02:30:017 out 0.50000
00:02:40:000 request_in -1	00:02:33:017 out 2.50000
	00:02:36:009 out 1.50000
	00:02:38:008 item_out 1.00000
	00:02:41:026 out 0.00000
	00:02:42:043 change_out 1.50000

(a) (b)

Table 7-3 Input and Output of the Original Vending Machine Simulation

The original Vending Machine simulation takes the input events and produced the output as shown in table 7-3. The original Vending Machine simulation results shown in table 7-3 (b) matches the integrated simulation results shown in table 7-1 (b) after translation of the output ports and timing. The matching results prove the integration to be correct.

7.3 Performance

Performance has not been a significant concern in this research. Because the CD++ environment by itself was being executed at runtime, the performance situation for the wrapped DEVS CD++ simulation should not change significantly after integration; however, it is difficult to precisely measure the performance of the integrated system. The performance is heavily dependant on other integrated components in the distributed simulation, CPU speeds, and network speed.

There is one feature, however, that can be improved to optimize the overall performance of the integrated system. Recall that the single interaction class design requires message filtering as introduced in Chapter 6. The message filter must be used to filter unnecessary interaction and this approach introduces a performance overhead. Often, when a wrapper sends out an interaction, all federates subscribed federates receive the interaction, but there may be only one of them that actually responds to the interaction. This creates extra unnecessary traffic among the federates. To overcome this performance issue, one solution is discussed below.

The solution is to publish one interaction per-port. If a wrapped simulation has 10 output ports, 10 interactions must be defined. This approach removes unnecessary traffic among the federates and improves the overall performance. The disadvantage of the approach is that the number of interactions to be defined increases dramatically. The increased number of interactions will complicate both wrapper generation and FOM generation.

The above solution can resolve the overhead created by the "single interaction class" design and the improvement of the performance should be included in future study.

Chapter 8 Conclusions

8.1 Conclusions

The objective of this thesis is to use the wrapper approach to integrate existing DEVS simulations on to the HLA RTI distributed simulation platform. This integration encourages the reuse of existing DEVS simulations in the HLA environment by achieving true transparent executions to the CD++ simulations and to the RTI environment, neither of the two is aware of the other party.

This work has achieved the following:

- An HLA-based simulation environment to integrate non-HLA-compliant DEVS CD++ simulations.
- A wrapper approach is defined. The wrapper acts as a bridge between the DEVS CD++ simulation and the HLA RTI platform to enable the execution of the CD++ simulations, and interoperability with other HLA compliant simulation.
- A four-layer integration framework is defined and the additions and modifications to each layer are identified. The changes made to the DEVS simulation engine is categorized and the demonstration of a sample wrapper is proven correct.
- The design of the messaging using one interaction makes it exceptionally easy to define the FOM for the federation.

- A case study that integrates DEVS CD++ simulation with HLA compliant simulation under the HLA framework demonstrates the proposed wrapper development method.
- An inter-process communication mechanism is used for the CD++ simulation to communicate with the wrapper. The use of IPC has minimized the changes involved in the CD++ engine, preserved the integrity of the CD++ engine, and achieved true transparency to the DEVS CD++ simulation developers.
- Cooperative time management is achieved across different federates running over the RTI.

8.2 Future Research Directions

Suggested future work for this research includes:

- The nature of the integration requires that the wrapper have the knowledge of the simulation it wraps. Thus; the wrapper has to be recreated for each simulation to be wrapped. Wrapper generation is an important component for future study. This thesis has outlined the necessary information that is required in order for the wrapper to be generated. Due to time constraints, a wrapper auto generator was not built study and remains as a future development. Furthermore, with the integration of various simulation engines, the wrapper must evolve to include their information as well. The development of the wrapper will be a continuous effort as a part of future integration processes.

- Performance of the system is briefly discussed in Chapter 7. Future work can include detailed performance analysis on the wrapped system with variable loads and variable size. The alternate approach suggested to the wrapper interaction design can also be used to measure the performance improvements.

- This research has been focused on the discrete event simulation interoperability. Future work shall be extended to the integration of other types of simulations such as continuous simulation, optimistic simulation and real-time simulation. Furthermore, because HLA supports several types of simulations, a mixture of two different types of simulations interoperating with each other can be attempted.

Reference

[1] B.P. Zeigler, T.G. Kim, et al., (2000), Theory of Modeling and Simulation. New York, NY, Academic Press. URL: http://www.acims.arizona.edu

[2] S. Straßburger, G. Schmidgall and S. Haasis. "Distributed Manufacturing Simulation as an Enabling Technology for the Digital Factory". Journal of Advanced Manufacturing Systems (JAMS). 2:1, 111-126. 2003

[3] M. Ewing, "The economic effects of reusability on distributed simulations", Proceedings of 2001 winter simulation conference, 2001

[4] C. A. Boer, A. Verbraeck, and H.P.M. Veeke. "The Possible Role of a Backbone Architecture in Real-Time Control and Emulation". In Proceedings of the2002 Winter Simulation Conference, 2002

[5] IEEE standard for modeling and simulation (M&S); High Level Architecture (HLA) – Framework and Rules. IEEE Std. 1516-2000

[6] S. Straßburger, "Distributed simulation based on the high level architecture in civilian application domain", Society for Computer Simulation International, ISBN 1-56555-218-0. April 2001

[7] High Level Architecture/ Run-Time Infrastructure. "RTI 1.3-NG-Next Generation Programmer's Guide version 3.2", Department of Defense Modeling and Simulation Office, 2000

[8] B. Zeigler, "DEVS Today: Recent Advances in Discrete Event-Based Information Technology", University of Arizona, 2003, URL: http://www.acims.arizona.edu/EDUCATION/ECE575Fall03/Notes/DevsToday2Col.pdf

[9] G. Wainer, "CD++: a toolkit to define discrete-event models". 2002. In Software, Practice and Experience. Wiley. Vol. 32, No.3. pp. 1261-1306

[10] Jean-Sebastien Bolduc, Python DEVS tool, Mc. Gill University, Canada, URL: www.cs.mcgill.ca

[11] H. Sarjoughian and B. Zeigler DEVSJAVA tool, University of Arizona, U.S.A., URL: http://www.acims.arizona.edu

[12] L. Li, "Vending Machine DEVS simulation using CD++", department internal report, Department of Systems and Computer Engineering, Carleton University, 2002

[13] D. Corman, The IULS Approach to Software Wrapper Technology for Upgrading Legacy Systems", The Boeing Company, Dec 2001

[14] B. P. Zeigler, G. Ball, H. Cho, J.S. Lee, H. Sarjoughian "Implementation of the DEVS Formalism over the HLA/RTI: Problems and Solutions", The University of Arizona, 2000

[15] M. Lees, B. Logan "Simulating Agent-Based Systems with HLA: The Case of

SIM AGENT", 2003

[16] M. Schumann, E. Bluemel "Using HLA for factory simulation", Fraunhofer Institute for Factory Operation and Automation, 1999

[17] S. Straßburger, "Simplex SLX federations", 2001 URL: http://isgsim1.cs.uni-magdeburg.de/hla/fed-barrel.html

[18] S. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, eds."HLA-CSPIF PANEL ON COMMERCIAL OFF-THE-SHELF DISTRIBUTED SIMULATION". Proceedings of the 2003 Winter Simulation Conference, 2003

[19] B.Stearns, Unix named pipes (FIFOs), Computer Review, Winter Quarter 1995, URL: http://www.uga.edu/~ucns/tti/Computer_Review/Winter95/UNIX.html

[20] B.P. Zeigler, H.S. Sarjoughian, "DEVS Component-Based M&S Framework: An Introduction", 2002, URL: http://www.acims.arizona.edu/EDUCATION/refCourse.html

[21] B. Zeigler, "DEVS Framework for Modeling and Simulation", URL: http://www.acims.arizona.edu/EDUCATION/devs3.0PptDocs/Ch1IntroFramework.ppt

[22] G. Wainer, N. Librería, B. Aires, "Methodologies and tools for discrete-event simulation" Argentina. 2003

[23] J. Ameghino, E. Glinsky, G. Wainer. "Applying Cell-DEVS in models of complex systems", Proceedings of the 2003 Summer Computer Simulation Conference. Montreal, QC. Canada. 2003

[24] T. Pearce Simulation-Driven Architecture in the Engineering of Real-Time Embedded Systems 2003 IEEE Real-Time Systems Symposium Work In Progress Session, 2003

[25] CD++ toolkit website. G. Wainer, Carleton University
URL: http://www.sce.carleton.ca/faculty/wainer/wbgraf/index.html

[26] High Level Architecture (HLA) Object Model Development Tool (OMDT) User's Guide. Department of Defense Modeling and Simulation Office, 1998

[27] R. McFarlane, "Time Management in the High Level Architecture" School of Computer Science, McGill University

[28] M. Khambatti, "Named Pipes, Sockets and other IPC", 2001, URL: http://www.public.asu.edu/~mujtaba/Articles%20and%20Papers/cse532.pdf

[29] IPC semaphores, "Acquiring a semaphore", Taligent Documentation URL: http://pcroot.cern.ch/TaligentDocs/TaligentOnline/DocumentRoot/1.0/Docs/books/OS/OS_19.html

[30] "Named Pipes and Mailslots, Windows 2000 Server Resource Kit Online Books", MSDN Library, January 2001

[31] R.M. Fujimoto, "Zero Lookahead and Reeatability in High Level Architecture". In Proc. Spring Simulation Interoperability Workshop. Orlando, FL, 1997