*Proceedings of the 1999 Winter Simulation Conference*
*P. A. Farrington, H. B. Nembhard, D. T. Sturrock, and G. W. Evans, eds.*

# DISTRIBUTED SUPPLY CHAIN SIMULATION IN A DEVS/CORBA EXECUTION ENVIRONMENT

Bernard P. Zeigler

AI and Simulation Group
Department of Electrical and Computer Engineering
University of Arizona
Tucson, AZ 85721, U.S.A.

Doohwan Kim
Stephen J. Buckley

IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598, U.S.A.

## ABSTRACT

The emerging electronic commerce and rapidly changing business environments place strong requirements on a next-generation supply-chain analyzer to simulate the flow of goods through the entire supply chain in a timely manner. Such requirements include scalable and efficient model execution and support for flexible future extensibility based on an open industry standard. This paper presents design considerations for a supply chain modeling and simulation environment to execute in a parallel and distributed manner on a DEVS/CORBA run time infrastructure. We recall that DEVS (Discrete Event System Specification) is a sound formal modeling and simulation framework based on generic dynamic systems concepts that can integrate into a parallel and distributed run time infrastructure. CORBA (Common Object Request Broker Architecture) is an open standard that is rapidly gaining universal business acceptance. It can be employed as middleware to support a heterogeneous, network-centric, distributed computing environment that includes modeling and simulation as well as other business objects. Implementing a distributed supply chain simulator in a DEVS/CORBA execution environment not only may significantly improve execution speed but also may provide advanced supply chain model development capability based on the DEVS modeling and simulation framework.

## 1 INTRODUCTION AND MOTIVATION

The IBM Supply Chain Analyzer (SCA) (Bagchi 1998) is a modeling and simulation software tool that can help companies make strategic business decisions about the design and operation of its supply chain. The supply chain simulation evaluates all the components and variables in a supply chain and helps companies determine which strategies will provide the most flexible and profitable operating environment. The SCA, with a combination of simulation and optimization functions, is designed to model and analyze supply chain issues such as site location, replenishment policies, manufacturing policies, transportation policies, inventory levels, lead times, and customer. The SCA has been successfully implemented to provide competitive edges to its clients ranging from food industries to computer manufacturing companies in recent years.

The emerging electronic commerce and dynamically changing business environment require fast and efficient simulation execution to examine the flow of goods through the entire supply chain in a timely manner. Although the current SCA provides advantageous supply chain modeling and optimization capabilities, it is not implemented for parallel execution in distributed computing environments. It runs on a single platform (Windows NT) and is built upon SIMPROCESS (Swegles 1997), a general purpose business process simulator which is a product of CACI Products Company. SCA preserves all SIMPROCESS capabilities while offering additional supply chain functionality. However, this dependency leads to a drawback since it is difficult to modify the deployment of the SCA tool and it is not based on an open industry standard.

Thus the requirements for a next-generation supply chain modeling and simulation environment should include scalable and efficient model execution and support for flexible future extensibility based on an open industry standard. An attractive combination to meet these requirements is the DEVS modeling and simulation framework in conjunction with the CORBA middleware. The DEVS (Discrete Event System Specification) formalism offers a sound, formal, open modeling and simulation framework based on generic dynamic systems concepts. DEVS has a well defined approach to coupling of components, supports hierarchical, modular model construction, and associated repository reuse. DEVS has also been implemented as highly portable C++/Java based environment executable in both sequential, parallel and heterogeneously distributed platforms (Zeigler 1999). CORBA (Common Object Request Broker Architecture) is an open standard promulgated by the Object Management Group (OMG), a consortium of over 700 companies (Orfali and Harkey 1997). CORBA is rapidly gaining acceptance due to a thriving middleware market consisting of businesses enthusiastically adopting the technology and vendors providing maturing products to support it. DEVS/CORBA refers to conceptual realization of a run-time infrastructure on top of CORBA middleware to support distributed simulation of DEVS components. Combining the advantages of CORBA and DEVS may provide a heterogeneous, network-centric, distributed computing environment that includes modeling and simulation as well as other business objects. Therefore, a DEVS/CORBA implementation, with a suitable distributed/parallel simulation protocol, can support efficient simulation of large scale supply chain models within an open industry middleware standard.

The purpose of this paper is to discuss considerations for the design of a to-be-developed DEVS/CORBA environment to meet requirements such as mentioned above. To set the stage we briefly review some relevant CORBA and DEVS concepts. We then provide an overview of the supply chain analyzer (SCA), its process models and their DEVS representations. With this background, we proceed to discuss the design alternatives and choices for a DEVS/CORBA environment supporting the SCA application.

## 2 REVIEW OF CORBA

Originally intended for local network distributed computing environment, the recent CORBA 2.0 version has extended the architecture's capability to support network-centric computing through the Internet Inter-ORB Protocol (IIOP) services. CORBA provides a common Interface Definition Language (IDL) to standardize the interfaces between object implementations and defines a set of components that allow applications to invoke methods (operations) on object implementations through such interfaces. Object implementations can be written in a variety of languages including C, C++, Java, Smalltalk, and Ada. The Object Request Broker (ORB), or more specifically, an Object Adapter, is responsible for handling an invocation request, finding the target object implementation, delivering the request to the object, and returning the response to the requestor. Relative to a request, the requestor is called the client while the target is called the server. To map between IDL interfaces and the application programs that implement them, vendor-supplied CORBA compilers write code fragments called stubs and skeletons. Stubs on the client side marshal typed data objects from a high-level application representation to a low-level packet representation. Skeletons on the server side demarshal the low-level packet representation back into a typed data object that is meaningful to the application. Supported by the Interface Repository (IR), an online database of meta information about ORB object types, the Dynamic Invocation Interface (DII) provides runtime means for a client to invoke a server's methods without reliance on precompiled stubs. CORBA Object Services define a wide variety of system services with IDL-specified interfaces to augment the functionality of the ORB.

The use of CORBA as communication middleware enhances application flexibility and portability by automating many common development tasks such as object location, parameter marshaling, and object activation. These features enable network-centric, complex, distributed and concurrent applications to be developed more rapidly and correctly. A possible concern to using CORBA as high-level middleware is the performance hit that might be a consequence of its generic and layered design. However, comparison of commercial CORBA products with direct interprocess communication implementations, such as sockets, and with competing middleware has not lent credence to this concern .

## 3 OVERVIEW OF DEVS IN RELATION TO A CORBA IMPLEMENTATION

This section provides a brief overview of DEVS, focusing on the aspects needed to discuss the issues in DEVS/CORBA design. The DEVS formalism (Zeigler et al. 1999) focuses on the changes of variable values – discrete events – and generates time segments that are piecewise constant. In essence the formalism defines how to generate events and their times of occurrence. Based on generic system theoretic concepts, DEVS enables one to express not only pure discrete event models but also discrete event approximations of continuous systems. Advantages of the DEVS methodology for model development include well-defined separation of concerns supporting distinct modeling and simulation layers that can be independently verified and reused in later combinations with minimal re-verification. The resulting divide-and-conquer approach can greatly simplify and accelerate model development leading to greater credibility at reduced effort. DEVS has a well defined concept of system modularity and component coupling to form composite models. It enjoys the property of closure under coupling which justifies treating coupled models as components and enables hierarchical model composition constructs.

### 3.1 DEVS Coupled and Atomic Model Templates

In the DEVS formalism one must specify 1) basic models from which larger ones are built, and 2) how these models are connected together in hierarchical fashion. DEVS-C++ (Zeigler et al. 1997) is a C++ modeling and simulation environment based on the parallel DEVS formalism (Zeigler et al. 1999). There are two major classes from which all user-defined models can be developed – *atomic* and *coupled*. The *atomic* class realizes the basic level of the DEVS formalism, while the *coupled* model embodies DEVS hierarchical model composition constructs.

To specify modular discrete event models requires that we adopt a different view than that fostered by traditional simulation languages. As with modular specification in general, we must view a model as possessing input and output ports through which all interaction with the environment is mediated. When external events, arising outside the model, are received on its input ports, the model description must determine how it responds to them. Also, internal events arising within the model, change its state, as well as manifesting themselves as events on the output ports to be transmitted to other model components. Rather than employing the mathematical formalities, we present basic and coupled models as templates for specifying model interfaces in the spirit of CORBA IDL.

A *basic model* template captures the following information:

- the set of input ports through which external events are received
- the set of output ports through which external events are sent
- the set of state variables and parameters
- the time advance function which controls the timing of internal transitions
- the internal transition function which specifies to which next state the system will transit after the time given by the time advance function has elapsed
- the external transition function which specifies how the system changes state when an input is received. The next state is computed on the basis of the present state, the input port and value of the external event, and the time that has elapsed in the current state.
- the confluent transition function which specifies how the system changes state when an input is received at the same time that an internal event is scheduled.
- the output function which generates an external output just before an internal transition takes place.

DEVS basic models are implemented as the class *atomic* models in DEVS-C++.

Basic models may be coupled in the DEVS formalism to form a *coupled model*. A coupled model, tells how to couple (connect) several component models together to form a new model. A coupled model template captures the following information:

- the set of its components
- the set of input ports through which external events are received
- the set of output ports through which external events are sent
- the coupling specification consisting of:

  1. the external input coupling connects the input ports of the coupled model to one or more of the input ports of the components
  2. the external output coupling connects the output ports of the components to one or more of the output ports of the coupled model
  3. internal coupling connects output ports of components to input ports of other components

DEVS coupled models are implemented as the class *coupled* models in DEVS-C++. A more complete review of the DEVS formalism is provided in (Zeigler et al. 1999).

### 3.2 Closure Under Coupling and IDL Component Interfaces

A coupled model can be expressed as an equivalent basic model in the DEVS formalism. This follows from the fact that the formalism is closed under coupling. (Expressing a coupled model as an equivalent basic model captures the means by which the components interact to yield the overall behavior.) Such a basic model can itself be employed in a larger coupled model as required for hierarchical model construction. Closure under coupling implies that when networking DEVS components, we can get away with one CORBA interface for all model classes, namely for basic model components. Of course, for coupled model components the implementation must appropriately interpret the basic interface methods. Indeed, this is how the DEVS-C++ implementation works. A basic DEVS component interface has an expression in CORBA IDL such as the following:

```
Module DEVS_Component{
   Interface Basic
   {
    boolean initialize();
    double timeAdvance();
    boolean internalTransition();
    boolean externalTransition(in  double
         elapsedTime, in  message);
     boolean confluentTransition(in  message);
     message outputFunction();
   };
};
```

(here *message* is a suitably defined data structure).

### 3.3 Parallel DEVS Simulation Protocol and CORBA Interfaces

In this section we discuss simulation of DEVS models and associated interface definitions in CORBA IDL. Later we return to consider design choices in a full execution environment.

Execution of a coupled DEVS model is mediated by coordinator and simulator objects (Figure 1). Each simulator keeps track of the time-of-last-event, tL and time-of-next-event, tN of its assigned DEVS component.

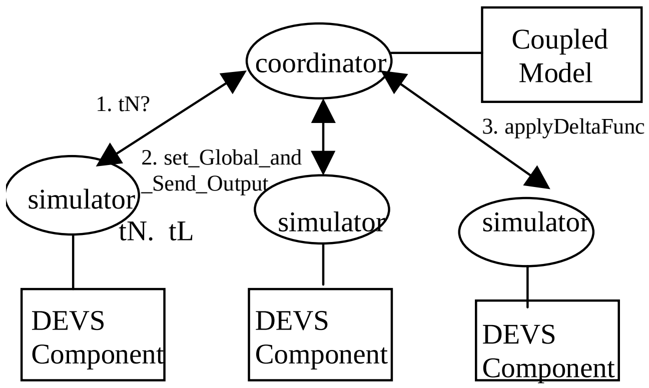

Figure 1: Execution of DEVS Simulation Protocol Through Collaboration of Coordinator and Simulators

Using the parallel DEVS simulation protocol, execution proceeds through iteration of the following cycle, as controlled by the coordinator:

1. Get component times to next event ($tN_i$) and take minimum as the global next event time, $t_N$
2. Tell all components the global $t_N$ and if component is imminent ($tN_i ==$ global $t_N$ ), then generate output message(using the output function)
3. Sort and distribute (using coupling) the output messages
4. Tell all components
   if
   a) component is imminent ($tN_i ==$ global $t_N$ )
   b) or has incoming mail (external events)
   c) or both
   then execute appropriate transition function (a) internal, b) external, c) confluent, respectively

The interaction between coordinator and simulators can be implemented in CORBA by suitable interface definitions and implementations. The interface that a simulator presents to a coordinator can be expressed in an IDL definitions such as:

```
Module Simulator{
   Interface toCoordinator
   {
    boolean start();
    double tN?();
    double set_Global_and_SendOutput
                 (in double tN);
    boolean applyDeltaFunc(in  message);
   };
};
```

Here a simulator can be told to start up (by invoking its component's initialize method). It can respond to a request for the time-of-next-event of its component (by querying for its timeAdvance). Via set_Global_and_SendOutput, it can receive a global tN, determine if its component is imminent, and if so, return its component's output (otherwise returning null). Finally, it can be sent its component's input and told to determine and apply the appropriate version of its component's transition function (or none). The coordinator issues these method invocations in the correct order to implement an iteration of the simulation cycle. The coordinator's interface can be given an IDL definitions such as:

```
Module Coordinator{
   Interface toSimulator
   {
    boolean register
       (in Simulator::toCoordinator SimObjRef);
    boolean startSimulation();
    boolean stopSimulation();
   };};
```

When a simulator is first activated it is provided a unique *toCoordinator* interface object reference by the Object Adapter. It can then send this reference to the coordinator using the *register* method. Having so registered, a simulator can subsequently receive the coordinator's "call back" invocations to implement the simulation cycle.

## 4 OVERVIEW OF SUPPLY CHAIN ANALYZER AND PROCESS MODELS IN DEVS REPRESENTATION

Figure 2 describes the system architecture of SCA as intended for a suitable DEVS/CORBA execution environment. The tool is intended to provide a combination of graphical process modeling, discrete event simulation, animation, activity-based costing, and optimization functions. The SCA employs a new Java-based graphical user interface that allows a user to construct supply chain models through graphical interaction and input model parameters. The model parameters can also be imported or exported in relational database or spreadsheets. In this section, we discuss the supply chain process models and capabilities and how they can be translated in DEVS representation.

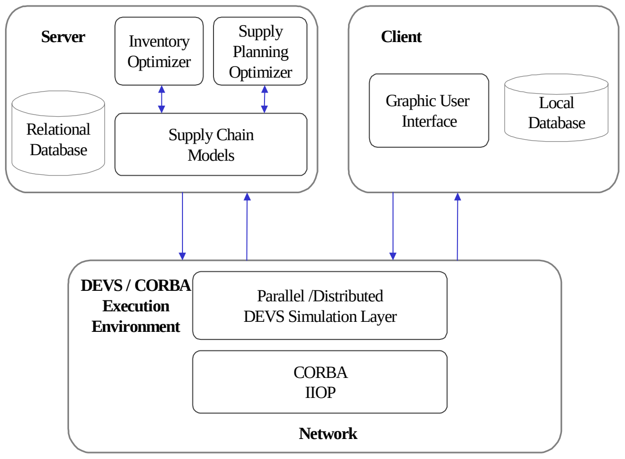

Figure 2: System Architecture of the Distributed Supply Chain Analyzer

### 4.1 SCA Process Models and Components

SCA offers two types of activity process models; those that represent ***manufacturing, distribution, customer, transportation*** and those that perform ***inventory planning, forecasting***, and ***supply planning***.

**Customer** process models external customers that issue orders to the supply chain being modeled. Orders are generated on the basis of customer demand, which may be modeled as a sequence of specific customer orders obtained from historical records or as an aggregated demand over a period of time. The customer process may also contain information on the desired service level and priority for the customer. When forecasting and supply planning activities are included in a supply chain model, the customer process may issue forecasts of future demand to aid these activities.

**Manufacturing** process represents assembly and keeps raw material and finished goods inventory. It can also be used to model suppliers. During simulation, the manufacturing process makes use of modeled information such as, the types of manufactured products, their manufacturing time, bills of material, manufacturing and replenishment policies for components and finished goods, storage capacity, manufacturing and material handling resources, and the order queuing policy.

**Distribution** process models distribution centers, including finished goods inventory and material handling. It can also be used to model a retail store. The inventory replenishment policy, safety stocks, reorder points, material handling resources, storage types and capacities can be modeled for the distribution center or retail store.

**Transportation** process models transportation time, vehicle loading, and transportation costs. Order batching policies (by weight or volume), material handling resources and transportation resources owned by this node may be specified.

**Inventory Planning** process simulates periodic setting of inventory target levels, days of supply targets, and continuous replenishment. Underlying this process is an optimization program developed at IBM Research called the *Inventory Optimizer* that computes and sets recommended inventory levels at various facilities in a supply chain based on desired customer serviceability.

**Forecasting** models product forecasts, including promotional and stochastic demand, for future periods. This process accumulates forecasts and periodically sends them to other processes to create build plans.

**Supply Planning** models the allocation of production and distribution resources to forecast demand under capacity and supply constraints. Underlying this process is an optimization program, *Supply planning Optimizer* that schedules replenishment orders or material shipping.

The typical input parameters for supply chain models include: number and location of suppliers, manufacturers, and distribution centers, stocking level of each product at each site, manufacturing and replenishment policies, transportation policies, supply planning policies, and lead times. After each simulation run, numerous reports are available. They include: Cycle time, serviceability, fill rate, stockout rate, shipments, and revenue, Inventory and WIP, Resource utilization and costs, Returns, incorrect order penalties, and late order penalties, etc. SCA allows users to evaluate various configurations and operational policies of supply chains on the basis of financial tradeoffs by the financial reports. The objective of the financial reports is to come up with the minimum cost design that achieves the desired customer serviceability and revenue targets. The reports are generated by four components; activity based costing, inventory costing, transportation costing, and inter-company financial transactions.

*Activity based costing* (ABC) (O' Guin 1991) is the process of assigning the cost of using resources to the activities that make use of them, and then assigning the cost of an activity to the *cost objects* for which the activity was performed. The cost objects relevant to supply chains include companies, orders, and products. The cost assignments are based on the time-weighted usage of the resources by activities and the activities by orders, which are monitored during the course of the simulation. More detailed description of the four financial components can be found in (Bagchi 1998).

## 4.2 Implementing Supply Chain Models in DEVS Framework

A basic class hierarchy for DEVS with extension for SCA in an object-oriented framework implementation is given in Figure 3. The basic class is *entity* from which class *devs* is derived. *Devs* is specialized into classes *atomic* and *coupled*. The *ActivityNodes* class is derived from *atomic* to

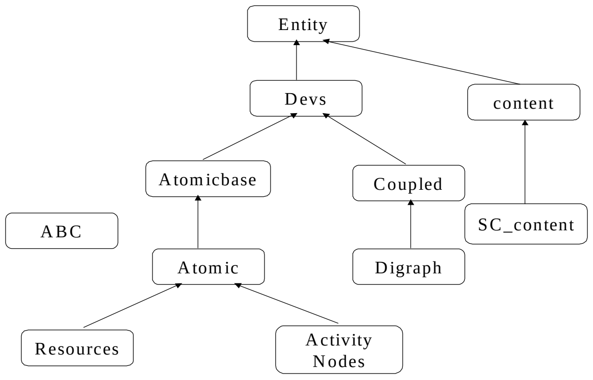

Figure 3: Basic DEVS Class Hierarchy for Supply Chain Models Implementation

provide a template for activity process models with supply chain data definitions. Class *content* is derived from *entity* to carry *ports* and their *values*, where the latter can be any instance of *entity* or its derived classes. *SC_content* is a derived class to represent supply chain entities (plan, order, batch, and forecast) that flow through a supply chain network of activity nodes. Derived from *Atomic*, the *Resources* class represents the agents that add value to entities or perform work at activities such as storage space, transportation, material handling, and manufacturing resources. The ABC class is to support activity based costing functions.

```
Distribution::Distribution(char*name):ActivityNodes
{    add_input_ports (“Ord”, “Deliv”, “Trig”, “Plan”);
     add_output_ports (“Ship”, “Repl”);
     add_state_variables ( … );
     add_processing_time_variables( … );
     add_input_variables( … ); }
double   Distribution::timeAdvance();
void  Distribution::internalTransition();
void  Distribution::externalTransition(messages *,
                         double elapsed_time);
message * Distribution :: output_function()
```

Each activity process in SCA is translated as an *Atomic* model by representing its state and mapping its dynamics into transition functions. Such functions as external transition, internal transition, output function, and time advance function are defined for the activity node. These functions are applied to the state of the model. Port information is specified to connect to other activity processes. The destination for sending an output message is determined by coupling methods that manipulate coupling information. The above sample code describes the *distribution* process represented in DEVS model template in DEVS-C++.

## 5 DEVS/CORBA EXECUTION ENVIRONMENT

Although the interfaces and approach required for an implementation of DEVS in a CORBA environment are clear, there remain many alternatives to be decided in any implementation scenario. Figure 4 illustrates an up-front choice (shown as arrows) between a direct implementation

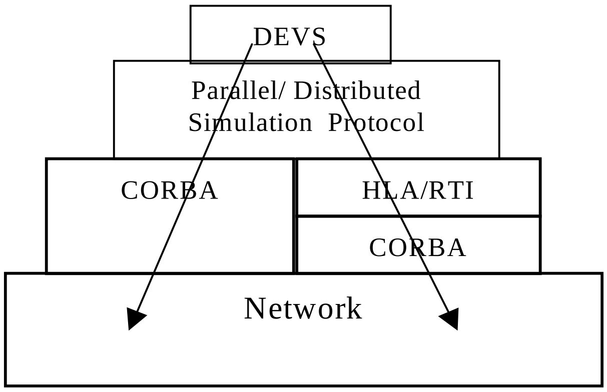

Figure 4: Alternative Implementation Strategies

and one that exploits the existing DEVS/HLA environment (Zeigler et al. 1997)[1]. In the latter, one can replace the existing HLA/RTI by a CORBA-based version with minimum additional effort.

The functional and non-functional requirements of the application domain should be brought in when considering this option. At least in its initial operating capability, the SCA is a purely discrete event modeling environment, and therefore is unlikely to require the object reflection capabilities afforded by HLA (intended for continuous system modeling). Moreover, at least initially, federations containing live components are not contemplated for the SCA. Accordingly, the relevant scaled back functionality of the HLA is presented in the following table:

[1] High Level Architecture (HLA) is a standard that will be required for all defense distributed simulations after 2001. HLA is implemented through a parallel and distributed run time infrastructure (RTI) that supports live as well as simulated entities. DEVS/HLA is an HLA-compliant environments that supports executing distributed DEVS models.

Table 1: Relevant HLA Functionality

| Category | Functionality |
|---|---|
| Federation Management | Create and delete federation executions<br>Join and resign federation executions |
| Time Management | Coordinate the advance of logical time |
| Data Distribution Management | Supports efficient routing of data |

Since public domain and commercial CORBA-based HLA/RTI implementations are becoming available, adopting the DEVS/HLA alternative for the (reduced) functionality is a real option. However, performance issues point in the other direction. In the direct approach we can tune the implementation to the specifics of the DEVS parallel simulation protocol and probably gain efficiency over the mapping of DEVS to the more generic HLA/RTI layer, which itself sits upon CORBA.[2]

A sketch of the direct implementation architecture is given in Figure 5. The IDL interfaces for coordinator and simulators, defined earlier, enable the two kinds of objects

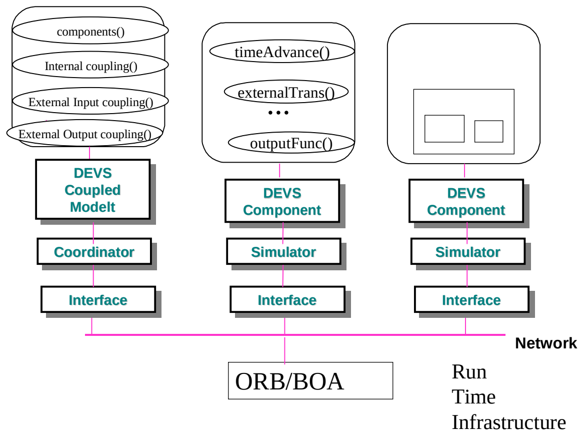

Figure 5: Direct Implementation Architecture

to collaborate to perform DEVS simulation. Simulators collaborate with their assigned model components to supply the information required for time management and data exchange as mediated by the coordinator. Another implementation choice arises here since the latter collaboration may, or may not, be mediated through CORBA. In the former case, we treat model components as arbitrary implementations of the toSimulator CORBA interface. This allows including as components, not only models coded in DEVS-C++, but also of other languages (such as DEVSJAVA) and legacy codes as well (Zeigler 1998). In the latter case, we eliminate the extra CORBA mediated pathways between simulators and their assigned components by implementing the toSimulator interface directly in DEVS-C++. A similar choice arises in the case of the coordinator and its interface to its assigned coupled model at the network level. The tradeoff here is between flexibility and potential performance benefits. Note that the implementation of the coupled class in DEVS-C++ is based on closure under coupling and therefore permits the use of the basic model interface for coupled model components.

## 5.1 Exploiting CORBA Functionality and Services

CORBA-based implementation of DEVS opens the door to exploiting the full range of CORBA functionality and services, both existing and under development. Some important benefits of CORBA features are delineated in Table 2.

Table 2: CORBA Features and DEVS/CORBA Benefits

| CORBA Feature | Benefit for DEVS Implementation |
|---|---|
| Language-neutral Data Types | Marshalling and de-marshalling of method arguments greatly simplify message exchange among components over the network (contrast this with HLA that does not support this level of data transparency). |
| Local/remote Transparency | Similarly, model components can communicate in the same way whether on a single processor or over a network between processors with different bit-level architectures |
| Dynamic Method Invocation | Not needed for DEVS itself, since DEVS has well defined and therefore statically compilable interfaces. However, it supports late binding access to databases and other business object is supported. |
| High Level Binding | As already indicated, use of IDL interfaces enables coupling of components in arbitrary languages and wrapping of legacy code. |
| Self Describing Meta Data | CORBA offers access to configuration meta data that may be exploited for understanding model behavior and monitoring simulation progress. |

[2] Indeed, problems with the definition of the HLA specification itself would be obviated (Zeigler 1999).

In addition the open CORBA environment makes it possible to embed DEVS/CORBA in a larger network-centric environment. A wide and extensive variety of additional services are defined in CORBA 2.0 that are being incorporated by CORBA vendors in the ORB offerings. These include: Lifecycle Service, Naming Service, Event Service, Persistent Object Service, Concurrency Control Service, Transaction Service, Security Service, Trader Service, Query Service, Licensing Service, Properties Service, Time Synchronization Service, Collection Management Service, and Startup Service. These services may be of significant value when DEVS models and simulations are embedded into network-centric collaboration with larger web and other business objects. Further, they offer a panoply of alternative implementations that may enhance functionality and performance of DEVS/CORBA itself.

## 6 CONCLUSIONS AND FUTURE WORK

We have discussed the prototype design of distributed supply chain simulation based on a DEVS/CORBA run time infrastructure. By employing DEVS/CORBA execution environment for parallel and distributed supply chain simulation, one can expect to achieve the following benefits. First, DEVS provides a sound modeling and simulation framework for advanced supply chain model development, which supports easier-to-achieve and more powerful, future extensions. Secondly, DEVS supports parallel and distributed simulation to significantly improve the speed and flexibility of model execution. Finally, CORBA middleware is an industry standard enabling integration into a heterogeneous, distributed computing environment that supports simulation on (LAN/intranet/internet) networked platforms and connectivity of simulation to other web and business enterprise objects. Many alternatives for implementation of DEVS in a CORBA environment are enabled by CORBA features and services, only a few of which have been enumerated here. In future work, we will be implementing prototype alternatives and evaluating their performance.

## REFERENCES

Bagchi, S. S.J. Buckley, M. Ettl, and G.Y. Lin Experience Using the IBM Supply Chain Simulator. *Proceedings of the 1998 Winter Simulation Conference,* ed. D.J. Medeiros, E.F. Watson, J.S. Carson, and M.S. Manivannan. Institute of Electrical and Electronic Engineers, Piscataway, New Jersey, 1998.

O'Guin, M., *The Complete Guide to Activity Based Costing,* 1991, Prentice Hall.

Orfali, R. and D. Harkey, *Client/Server Programming with Java and CORBA.* 1997, New York, NY: Wiley Computer Publishing.

Swegles, S., Business process modeling with SIMPROCESS. In *Proceedings of the 1997 Winter Simulation Conference,* ed. S. Andradottir, K.J. Healy, D.H. Withers, and B.L. Nelson. Institute of Electrical and Electronic Engineers, Piscataway, New Jersey. 1997.

Zeigler, B.P., *et al.*, *The DEVS Environment for High-Performance Modeling and Simulation.* IEEE C S & E, 1997. 4(3): p. 61-71.

Zeigler, B.P., et al. Distributed/Real Time Simulation: Integrating Legacy and Modern Codes. in Int'l Conf. Env. Sys., Soc. Aut. Engr. 1998. Danvers, MA.

Zeigler, B.P. *Implementation of the DEVS Formalism over the HLA/RTI: Problems and Solutions*. in *SIW*. 1999. Orlando, FL. (See the HLA web site: www.hla.mil for further information.)

Zeigler, B.P., T.G. Kim, and H. Praehofer, *Theory of Modeling and Simulation*. 2 ed. 1999, New York, NY: Academic Press.

## AUTHOR BIOGRAPHIES

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

**DOOHWAN KIM** is a post doctoral research scientist at IBM Thomas J. Watson Research Center in Yorktown Heights, NY. He received his Ph.D. degree from University of Arizona in Electrical and Computer Engineering. His research interests include discrete event modeling, distributed simulation, and high performance computing.

**STEPHEN J. BUCKLEY** is a Research Staff Member at the IBM Thomas J. Watson Research Center in Yorktown Heights, NY. He is currently the manager of the Supply Chain Analysis department, which developed the Supply Chain Simulator. He received the Ph.D. degree from MIT in Computer Science. In addition to simulation, his interests include algorithms, scheduling, and robotics.