# Simulation Interoperability Across Parallel DEVS Models Expressed in Multiple Programming Languages

**Thomas Wutzler**
**Max-Planck Institute for Biogeochemistry**
**Hans Knöll Str. 10**
**07745 Jena, Germany**
**thomas.wutzler@bgc-jena.mpg.de**

**Hessam S. Sarjoughian**
**Arizona Center for Integrative Modeling & Simulation**
**Dept. of Computer Science & Engr.**
**Ira A. Fulton School of Engr.**
**Arizona State University, Tempe, Arizona, USA**
**sarjoughian@asu.edu**

**Keywords**: CORBA, DEVS, Distributed Simulation, Interoperability

**ABSTRACT**: Research efforts have focused on developing methods and technologies to support interoperability among different simulations. In addition, modeling and simulation frameworks have also been extended to support distributed simulation. In this research we present a many-to-many approach to model to simulator mapping. The goal is for a Parallel DEVS model described in one programming language to be executed using a simulator implemented in another programming language. This approach supports simultaneous execution of a set of DEVS-based models written in different programming languages. A prototype distributed simulation environment consisting of the DEVSJAVA and Adevs has been developed to simulate DEVS and non-DEVS models described in any of the Java, C++, and Visual Basic programming languages.

## 1 INTRODUCTION

Interoperability among simulators continues to be of key interest within the simulation community. A chief reason is the existence of a legacy of large simulations, which are developed in different programming languages. For example, simulation of ecological processes have been developed in programming languages including C, C++, Fortran, Java, and Visual Basic.

In the domain of natural and social sciences, often mathematical and experimental data are directly represented in (popular) programming languages instead of first casting them in appropriate modeling and simulation frameworks. Since computer programming languages are intended to be generic and not specialized for simulation, they do not offer basic simulation artifacts (e.g., causal output to input interactions and time management) that are essential for separating simulation correctness vs. model validation [1]. The consequence is, therefore, custom-built simulations where the separation between models and simulators is missing or otherwise obscure.

Fortunately, these legacy *programming-code* models often have well-defined mathematical formulations, which facilitate their conversion to *simulation-code* models. The translation from programming-code to simulation-code models can be valuable since the latter can benefit from simulation model development, comprehension, modification, execution, and reuse.

A key advantage of executing simulation models using well-defined simulation protocols is that a simulator can execute the models independent of their realizations in particular programming languages. To achieve exchange of models requires a modular design with well-defined interface specifications and a mechanism to execute the models within a concerted simulation environment [2]. There are various approaches of exchanging model implementations and a concerted execution of the models. Techniques range from highly specialized coupling solutions [3], the use of blackboards for message exchange [4], modeling frameworks (e.g., [5]), XML-based descrip-

tions of models [6], to the usage of a sophisticated simulation middleware [7].

In this work, we propose an "abstract model" for the parallel Discrete Event System Specification (DEVS) [8] to support a new way to allow the concerted execution of a set of DEVS-based models written in different programming languages. For example, a DEVS-compliant model of forest growth implemented in Java (e.g., DEVSJAVA [9]) may be executed together with a soil carbon turnover model implemented in C++ (e.g., Adevs [10]). Furthermore, the abstract model allows the execution of models written in a programming language for which no simulator has been developed. An example of this could be a model written in Visual Basic, but simulated in DEVSJAVA once it is wrapped inside a DEVS component.

In the remainder of this paper, we will describe the concept and a realization of executing DEVS (or DEVS-compliant) simulation models expressed in one programming language, but executed in a simulation environment implemented in another programming language. We exemplify this approach for discrete-event and optimization models.

## 2 BACKGROUND AND RELATED WORK

A key advantage of a well-defined modeling and simulation framework is support for building large, complex simulation models using system-theoretic concepts. Systems-theory and its foundational concept of hierarchical composition from parts lends itself naturally to object-based modeling and distributed execution. Furthermore, the combination of systems-theory and object-orientation offers a potent basis for developing scaleable, efficient modeling and simulation environments. A well-known approach to system-theoretic modeling and simulation is the Discrete Event System Specification framework [8].

In this paper we focus on the Parallel DEVS formalism [8, 11] since it is well suited to provide the basic mechanism to allow the interoperation and concerted simulation of heterogeneous models for the following reasons. First, sub-models can be combined using input and output ports and their couplings. Second, it enjoys closure under coupling, which allows a modular hierarchical assembly of sub-models. Third, DEVS can reproduce the other major discrete-time (DTSS) and approximate continuous modeling paradigms (DESS) that are commonly used in describing ecological and other natural systems. The DEVS formalism is independent of a programming language in which it may be realized. There exist a variety of DEVS simulation engines supporting several programming languages. However, interoperability issues among DEVS simulation engines arise due to differences in underlying platforms (programming languages). For example, implementation differences between DEVSJAVA and Adevs prevent sharing and reuse of the DEVS models.

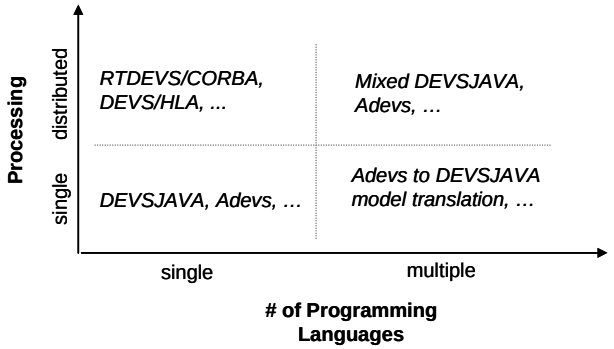


Figure 1: Placement of the multi-programming language approach among other approaches.

Aside from basic research in developing simulation environments such as DEVSJAVA to support combined logical- and real-time simulations (Figure 1, bottom left), there has also been interest in distributed simulation with one DEVS-implementation (Figure 1, top left) (RTDEVS/CORBA [12] and DEVS/HLA). It is reasonable to expect distributed simulation where models are described in different programming languages. This is important since a primary objective of reusing model implementation is to avoid rewriting models expressed in different programming languages into a single programming language (Figure 1, bottom right). The interoperation between heterogeneous models can be considered a special case of distributed simulation because sub-models will run in different processes that must communicate (Figure 1, top right). Given these last two considerations, it becomes attractive to avoid manual translation of models from one programming language to another, especially for complex, large-scale models that are common in the natural sciences.

The DEVS framework also imposes fewer constraints on the participating sub-models compared to simulation middleware HLA. This observation holds for the class of time-stepped logical- or real-time models that are investigated in this research.

In this paper we present an approach in which we use an abstract Parallel DEVS model in logical time to establish an interface for model interoperation in DEVSJAVA and Adevs. We provide adapters for both models and two simulation engines that support the abstract model. Both atomic models and the execution of coupled models can be mapped to this abstract model. We show three useful applications of this approach:

- The interoperation between different DEVS simulation engines.
- The implementation of sub-models in a computer language for which there is no DEVS simulation engine.
- The integration of non-DEVS models within a DEVS simulation.

Since DEVS can reproduce time-stepped and approximate continuous systems, it acts as a generic interface for coupling discrete-event, discrete-time, and continuous models. Each component model needs to specify ports, initialization, state transitions, time advance, and output functions. Models can be hierarchically combined to form a coupled model. Simulators and coordinators take care of the correct simulation of the coupled model. Although there are a variety of extensions to Parallel DEVS, in this work we consider logical-time simulations.

# 3 APPROACH

## 3.1 An Abstract Model for Alternative DEVS Model Implementations

Different implementations of the DEVS formalism share the same semantics due to the DEVS mathematical specification, but they generally differ in the underlying software design. In order to allow an abstraction for different implementations, we have defined an abstract model as shown in Figure 2. The operations of this abstract model, which include mediation between simulators, can be realized with a middleware.

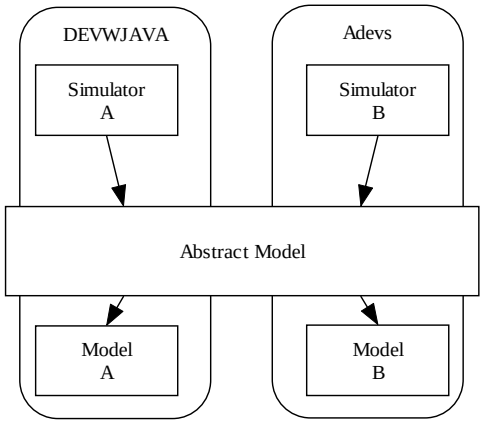


**Figure 2**: Abstraction of different DEVS implementations. Usually, simulator directly invoke operations on the model. In the presented approach, the model implementations are wrapped by an abstract model.

We specified the abstract parallel DEVS model operations in OMG-idl (Figure 3) and used CORBA to invoke these operations expressed in different programming languages. Note that we defined one basic interface instead of defining interfaces for simulators. Furthermore, we have not defined an interface for coupled models since the execution of a coordinator of a coupled model is specified as an atomic model (see Section 3.2.1). The advantages of this approach will be discussed in Section 5.

```
interface DEVS{      // OMG-idl (CORBA)
   void doInitialize( );
      // begin of simulation
   double timeAdvance( );
      // time for next output and internal transition functions
   Message outputFunction( );
      // produce outputs for current time
      // is not allowed to change the state of the model
   void internalTransition( );
      // state transition
   void externalTransition( in double elapsedTime, in Message msg );
      // state transition with input event
   void confluentTransition( in Message msg );
      // input event at time of internal transition}
```

**Figure 3**: Operations of the abstract DEVS model.

## 3.2 Extending Simulators

To support the functionality of the above abstract model, it is useful to extend the internals of existing DEVS simulation engines that are intended to simulate models expressed in multiple programming languages. This is because a simulator should not be aware of the remote model implementations which may be assigned to it. The simulator, therefore, needs to have access to a *Model*

*Proxy* as shown in Figure 4. The *Model Proxy* will translate its method invocations to the abstract model (i.e., *DEVS Interface* element shown in Process A). While the translation is syntactical in nature (i.e., preserves Parallel DEVS model semantics), it is generally non-trivial since the *Model Implementation* could be significantly different depending on the constructs of a chosen programming language. An example of complex syntactical translation is message contents (or input and output events) of a model implementation that is expected by a simulator intended to execute it.

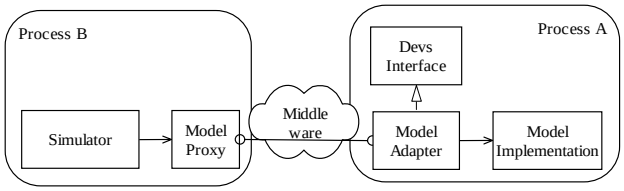


**Figure 4**: Adapting specific DEVS implementations to the abstract model.

Via a middleware the operations of the abstract model in the remote process (Process A) are invoked by the model proxy. If the actual model implementation (shown in Process B) cannot support the abstract model, a *Model Adapter* is needed to translate the invocations of the abstract model to the invocations of the *Model Implementation*. This again, is a syntactical mapping, which might be non-trivial due to difference in message formats and ways in which objects may be created and destroyed (e.g., garbage collection during simulation cycles). The abstract model here, corresponds to the DEVS-Interface, but additionally includes semantic constraints (see discussion of real time DEVS at the end).

In this setting, therefore, the *Simulator* and the *Model Implementation* remain unchanged. The *Model Proxy* and the *Model Adapter* need to be developed only once for a given simulation engine. Afterwards, the simulator can simulate any abstract model implementation, and all its models can be represented as abstract models.

### 3.2.1 Execution of Coupled Models as Atomic Models

The abstract model corresponds only to an atomic model in a straightforward way. However, a coordinator that executes a coupled model can be seen as an atomic model itself. It performs the state transition of a coupled model. This corresponds to the closure under coupling. The simulation cycle of the coordinator is as shown below.

1. initialize model
2. while( next event )
   a. compute input output for next event time
      i. invoke imminent component output function
      ii. distribute outputs
   b. invoke transition function
   end while

The invocation of the atomic model methods has to preserve this order of the coordinator's functions. The DEVS simulation protocol guarantees that the initialization function is invoked at the beginning of the simulation, and that one of the transition functions is invoked once during one simulation cycle. Hence, we could map 1. to the initialize operation and 2. to all of the transition functions. However, the outputs are valid only between 2.a.i. and 2.b. because output function is invoked right before the next internal transition. In order to allow the output function to return valid values, we had to break the simulation cycle right before b. The following algorithm specifies the same semantics of the coordinator's simulation cycle.

1. initialize model
2. if( next event )
   compute input output for next event time
   a. invoke imminent component output function
   b. distribute outputs
   end if
3. while next event:
   a. invoke transition function
   b. if( next event ) compute in/out for next ev. time
      i. invoke imminent component output function
      ii. distribute outputs
      end if
   end while

With this we constructed the following model adapter. The adapter employs its own coordinator of the coupled model. The doInitialize() function performs (1.) and (2.) on the coordinator. Each transition function (internalTransition(.), externalTransition(.), or confluentTransition(.) ) performs (3.) on the coordinator. The outputFunction() and the transition functions invoke the corresponding methods of the coordinator. This allows execution of a coupled model as an abstract one. The simulator is not aware of actually executing a coupled model.

### 3.2.2 Integration of Non-DEVS Functionality

In addition to interoperating with various DEVS simulation engines, the abstract model can be used to integrate non-DEVS functionality as well. In this case, the model adapter maps the non-DEVS functionality to the basic model interface. Three examples of this integration follow.

A) One is the integration of time-stepped models. The time-advance function will always return the time until the next time step. The internal transition executes the transition. The external transition will only store the inputs for the next transition.

B) A second use is the integration of continuous time models that specify the calculation of derivatives but have no notion of DEVS yet. The model adapter will employ quantization [13]. The transition functions will invoke the calculation of the new derivatives within the model implementation. Next, the transition functions will update a quantized integrator and calculate the time until the next boundary crossing. After an ordinary internal transition, an internal output transition is scheduled. The time-advance function will return the calculated time until the next boundary crossing. The output function of the model adapter will return the output of the continuous model, but only if it is in the output phase.

C) Another use is the integration of functions that do not depend on time as in an optimization procedure (this means a Mealy-type passive model). The external transition function of the model adapter will invoke the original function and immediately schedule an internal transition in phase "output." Within the internal transition the model is again set to passive state (i.e., with a time-advance of infinity). The output function will return the result of the function invocation, but only if it is in the output phase.

## 4 EXAMPLE APPLICATIONS

### 4.1 Interoperation Between Different DEVS Simulation Engines

We implemented the model proxy, the model adapter, and the adapter of coupled models in the two DEVS simulation engines DEVSJAVA and Adevs. Next, we simulated a simple model consisting of an experimental frame (ef) and a processor (p) (see Figure 5) using the Adevs and DEVSJAVA simulation engines.

The generator is an atomic model which generates jobs at some given time intervals. The transducer matches jobs coming in at its two input ports and calculates the time difference. The experimental frame is a coupled model. It has an output port for jobs created by the generator, an input port of the finished jobs, two external couplings, and one internal coupling. The processor accepts jobs when in passive phase and outputs finished jobs after its processing time.

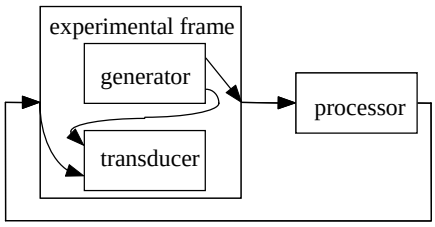


**Figure 5**: Experimental frame (ef)-processor (p) model.

We implemented the experimental frame in DEVSJAVA and the processor and the combined ef-p models in Adevs (Figure 6). The ef model adapter was constructed and started in a DEVSJAVA server process. Within a C++ program, a model proxy was constructed with the CORBA reference of the DEVSJAVA model adapter. The model proxy could be used in an Adevs simulation like any other Adevs atomic model. As noted earlier, the experimental frame appears to the Adevs simulator as an atomic model.

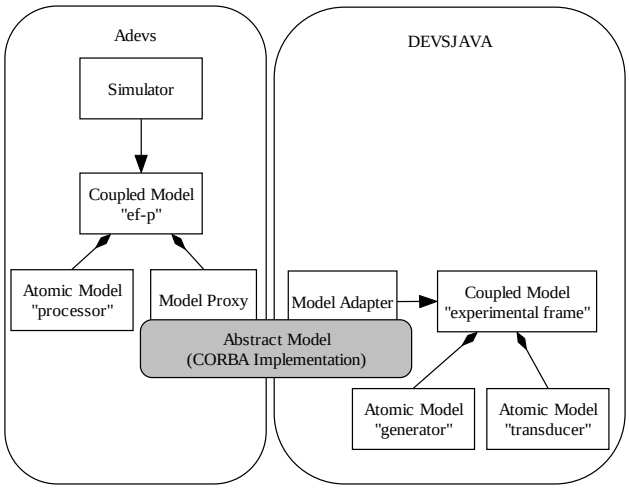


**Figure 6**: Implementation of the ef-p model with the experimental frame submodel specified and executed in a simulation engine different from that of the ef-p model.

### 4.2 Model Implementation Without Corresponding Simulation Engines

To illustrate simulating DEVS-compliant models which have no corresponding simulation engines, we implemented the atomic model of a processor (see the previous section) within VBA routines of an MS Excel workbook.

The class of the Visual Basic model descends directly from the portable object adapter classes generated by VBORB (An object request broker for Visual Basic) [14]. It implements the abstract model. The processing time of the processor was obtained from a cell of a workbook. Therefore, the user could easily change the model behavior before or during the simulation. Within a start-up routine of the workbook, a CORBA object was constructed with this VB model class. In this example the model itself implemented the DEVS Interface, so no model adapter was needed.

A DEVSJAVA model proxy was constructed with reference to the CORBA object and was used as any other DEVSJAVA atomic model.

### 4.3 Simulation of Non-DEVS Models

In the following example a user buys power from two different providers offering two different price regimes depending on the ordered amount (Figure 7). Given a power demand (c) we want to know which amount of power we have to order from the oil provider (x) and which amount from the solar provider (y = c - x), so that the overall price is minimized. This is an optimization problem for the power price: p = x * f(x) + (c-x) * g(c-x), where f(x) and g(y) are the unit cost functions depending on the ordered amount of power.

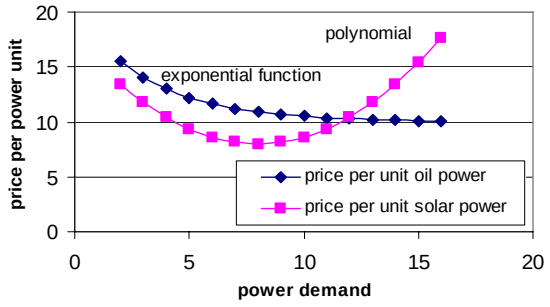


**Figure 7**: Cost functions of the optimization example.

Suppose the search for a minimum of the price function is not possible in a closed form. Hence, an iterative optimization procedure is used. We modeled the problem within MS Excel and its Solver module. We implemented a VBA function that determines the best proportion of power from the oil provider and the minimum average price per power unit.

The DEVS Interface gives us the ability to utilize this optimization procedure within a DEVSJAVA simulation. The optimization model is modeled as an atomic model according to 3.2.2 C. A simple experiment consisting of a generator, PowerOptimizer, and Observer is devised. The generator outputs are received as powerDemand input events by the PowerOptimizer. The observer model records the outputs of both models with time as shown below.

```
<input time='0' powerDemand='20' />
<input time='0' oilRatio='49.9087' unitPrice='9.294' />
<input time='4' powerDemand='14' />
<input time='4' unitPrice='8.81' oilRatio='0' />
<input time='6' powerDemand='16' />
<input time='6' unitPrice='9.42044' oilRatio='41.13' />
```

## 5 DISCUSSION

The approach presented for combining heterogeneous DEVS models has the advantage of placing only few constraints on the participating sub-models. Hence, few or no changes are necessary for the DEVS models that are written for specific simulation engines. Often relatively simple adapters will suffice. Other approaches (e.g., HLA) place much more constraints on the sub-models. For example, HLA is intended to support all simulations as well as physical systems with support for different types of simulation protocols (e.g., combined conservative and optimistic simulations). An advantage of the proposed approach is the free availability of both CORBA implementations and DEVS simulation engines for the main programming languages.

Correctness of model execution, therefore, is grounded in the DEVS formal specification. The developer of a DEVS simulator has to show that the model adapter is compliant to a common abstract model. This ensures every model is a legitimate Parallel DEVS model in logical time. Interoperability between different DEVS simulation engines using one abstract model allows use (or implementation) of different engines without having a “coupled” abstract model. Instead we mapped the execution of a coordinator to an atomic model. Hence, the interoperability takes place at the model level, not at the simulator level. The

advantage is that only few constraints need to be placed on models that were not specifically designed for a DEVS-simulator. The price is that we lose degrees of freedom in distributing the simulation because the structure of a remote coupled model is transparent to the simulator. A standardization method for interoperating DEVS engines can alternatively specify abstract simulators and abstract coordinators for the purpose of interoperability between simulation engines.

We note that the abstract model does not rely on any particular middleware such as CORBA, although the choice of a middleware has key importance including performance and interoperability robustness. Thus, the abstract model may be implemented, for example, with COM or Web services.

In the context of this work, we have accounted for logical-time DEVS models. The approach presented in this paper, however, is also applicable for real-time DEVS models. In this case, the interface can stay the same, but the semantics of the abstract model need to comply with real-time DEVS specifications. The transition functions are required to return immediately and do computational or observing work within a second thread or process (an activity). The external transition of the example optimization model would schedule an internal transition after a time greater than zero that reflects an optimistic estimate of the time to do the optimization calculation. It would then start the optimization activity but return immediately. After calculating the “optimal price” activity in real-time, it is returned with the output function. The implementation of the activity is left to model specification. RTDEVS/CORBA can simulate any DEVSJAVA model in real time. Hence, with the DEVSJAVA model proxy it will be able to simulate a Parallel DEVS abstract model in real time.

## 6 CONCLUSIONS

In this paper we considered the need for simulating common types of pure or DEVS-compliant models. We showed that a combination of DEVS-based simulation engines can be used to carry out a concerted simulation of heterogeneous DEVS, DTSS and DESS, and timeless models. Distributed simulation across models codified in different programming languages was achieved via an abstract model and interoperability services of the CORBA middleware.

The presented approach places few constraints on the submodels and thus aids flexible and non-tedious simulation model integration. We successfully demonstrated the usability of the approach for (1) the interoperation between different DEVS simulation engines, (2) the implementation and simulation of models in a computer language that has no DEVS simulation engine available, and (3) the integration of non-DEVS models or functionality within a DEVS simulation.

The work presented in this paper remains to be further investigated to support real-time DEVS models as well as using HLA.

Although in this paper we presented basic examples, in an ongoing project we will demonstrate the concerted application of more complex ecological models, implemented in Java and C++, that are time stepped or are specified by differential equations.

## 7 ACKNOWLEDGMENTS

This work was funded by a doctoral scholarship of the German Academic Exchange Service.

## 8 REFERENCES

[1] H. S. Sarjoughian and B. P. Zeigler, "DEVS and HLA: Complementary Paradigms for Modeling and Simulation?" *Transactions of the Society for Modeling and Simulation International*, vol. 17, pp. 187-197, 2000.

[2] B. Acock and J. F. Reynolds, "Introduction: modularity in plant models," *Ecological Modelling*, vol. 94, pp. 1-6, 1997.

[3] S. Valcke, D. Declat, R. Redler, H. Ritzdorf, and R. Vogelsang, "The PRISM Coupling and I/O System: OASIS 4," MPI Meteorology Hamburg, 2005.

[4] J. Liu, C. Peng, Q. Dang, M. Apps, and H. Jiang, "A component object model strategy for reusing ecosystem models," *Computers and Electronics in Agriculture*, vol. 35, pp. 17-33, 2002.

[5] C. Hillyer, J. Bolte, F. van Evert, and A. Lamaker, "The ModCom modular simulation system," *European Journal of Agronomy*, vol. 18, pp. 333-343, 2003.

[6] D. Pullar, "SimuMap: a computational system for spatial modelling," *Environmental Modelling & Software*, vol. 19, pp. 235-243, 2004.

[7] HLA, "High Level Architecture." http://hla.dmso.mil: Defense Modeling and Simulation Office, 1999.

[8] B. P. Zeigler, H. Praehofer, and T. G. Kim, *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*, Second Edition ed: Academic Press, 2000.

[9] ACIMS, "DEVSJAVA Software," http://www.acims.arizona.edu, 2005.

[10] J. J. Nutaro, "Adevs (A Discrete EVent System simulator) C++ library," 2005.

[11] A. Chow, "Parallel DEVS: A Parallel, Hierarchical, Modular Modeling Formalism and Its Distributed Simulator," *SCS Transactions on Simulation*, vol. 13, pp. 55-102, 1996.

[12] Y. K. Cho, "RTDEVS/CORBA: A Distributed Object Computing Environment For Simulation-Based Design of Real-Time Discrete Event Systems," in *Electrical and Computer Engineering*, vol. PhD. Tucson: University of Arizona, 2001.

[13] E. Kofman, "Discrete event simulation of hybrid systems," *Siam Journal On Scientific Computing*, vol. 25, pp. 1771-1797, 2004.

[14] M. Both, "VBOrb - A Visual Basic Object Request Broker," vol. 2005: http://www.martin-both.de/vborb.html, 2003.