Thomas Wutzler[1,2,3] & Hessam S. Sarjoughian[2]

Interoperability among Parallel DEVS Simulators and Models Implemented in Multiple Programming Languages[†]

[1]Max-Planck Institute for Biogeochemistry

Hans Knöll Str. 10

07745 Jena, Germany

thomas.wutzler@bgc-jena.mpg.de

[2]Arizona Center for Integrative Modeling & Simulation

School of Computing and Informatics
Arizona State University, Tempe, Arizona, USA

sarjoughian@asu.edu

[3]corresponding author Tel: +49 3641 576271 Fax: +49 3641 577274

Keywords: DEVS, Interoperability, Distributed Simulation, Middleware, Scalability

---

[†] Manuscript received 29.7.2006, revised 19.5.2007, accepted 22.6.2007[†]

Abstract

Flexible, yet efficient, execution of heterogeneous simulations benefits from concepts and methods that can support distributed simulation execution and independent model development. To enable formal model specification with submodels implemented in multiple programming languages, we propose a novel approach called the Shared Abstract Model (SAM) approach, which supports simulation interoperability for the class of Parallel DEVS-compliant simulation models. Using this approach, models written in multiple programming languages can be executed together using alternative implementations of the Parallel DEVS abstract simulator. In this paper, we describe the SAM concept, detail its specification, and exemplify its implementation with two disparate DEVS-simulation engines. We demonstrate the simplicity of integrating simulation of component models written in the programming languages Java, C++, and Visual Basic. We describe a set of illustrative examples that are developed in an integrated DEVSJAVA and Adevs environment. Further, we stage simulation experiments to investigate the execution performance of the proposed approach and compare it with alternative approaches. We conclude that application domains in which independently developed heterogeneous component models that are consistent with the Parallel DEVS formalism benefit from a rigorous foundation while also being interoperable across different simulation engines.

1 Introduction

Interoperability among simulators continues to be of key interest within the simulation community [1, 2]. A chief reason is the existence of legacy simulations which are developed using a variety of software engineering paradigms that are jointly executed using modern, standardized simulation interoperability infrastructures such as HLA [3] and DEVS-BUS [4]. The latter one supports the Discrete Event System Specification (DEVS) modeling and simulation approach [5]. Based on general-purpose simulation interoperability techniques and high performance computing technologies, these approaches offer robust means for a concerted execution of disparate models. However, use of such approaches can be prohibitive in terms of time and cost for redevelopment of existing models. For example, in natural and social sciences application domains, often mathematical and experimental data are directly represented in (popular) programming languages including C, C++, C#, Fortran, Java, and Visual Basic [e.g.6, 7] instead of first being cast in appropriate modeling and simulation frameworks. Since computer programming languages are intended to be generic and not specialized for simulation, they do not offer some simulation artifacts — such as causal output to input interactions and time management — that are essential for separating simulation correctness vs. model validation [8, 9]. The consequence is often, therefore, custom-built simulations where separation between models and simulators are weak or otherwise difficult to understand. Fortunately, these legacy *programming-code* models often have well-defined mathematical formulations, thus 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 rich modeling concepts and artifacts which in turn enable rich simulation model formulation, development, execution, and reuse. A key advantage of using a well-defined simulation protocol is that it allows a simulator to execute models independent of their realizations in particular programming languages. Achieving model exchange requires a modular design with well-defined interface specifications and a mechanism to execute the models within a concerted simulation environment [10, 11]. Various approaches exist for exchanging model

implementations and their concerted execution. Techniques range from highly specialized coupling solutions [12], the use of blackboards for message exchange [13], modeling frameworks [14], XML-based descriptions of models [15], to the usage of standardized simulation middleware [16].

In this work, we propose the *Shared Abstract Model* (SAM) interoperability approach, which provides a novel capability for concerted execution of a set of DEVS-based models written in different programming languages. For example, a DEVS-compliant adaptation of a model of forest growth [17] implemented in Java may be executed together with a soil carbon dynamics model [18] implemented in C++, using the DEVS simulation engines DEVSJAVA [19] and Adevs [20, 21]. 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 component, that implements the Abstract Model.

In the remainder of this paper, we will describe the SAM concept and its realization for executing DEVS (or DEVS-compliant) simulation models that are expressed in one programming language, but that are executed in a simulation environment implemented in another programming language. We exemplify this approach for discrete-event and optimization models. Finally, we will examine the scalability of the Abstract Model with respect to the number of couplings between the models, which shows potential applicability toward large-scale simulations using high performance computing platforms.

2 Background and Related Work

For many years significant advances have been achieved toward simulation interoperability. The importance of simulation interoperability lies in systematic capabilities to overcome differences between simulations that may have vastly different or partially formal underpinnings – i.e., the types of dynamics each simulation can express could be very different. Other key benefits are support for development, execution, and management of large-scale simulations and emulations. Interoperability, therefore, is chiefly used for federating disparate simulations, software applications, and physical systems with humans in the loop. Therefore, simulation interoperability must deal with differences between syntax and semantics at a common level of abstraction among simulations and software applications that are treated as simulations. A common level of model expressiveness may exceed or constrain the kinds of simulations that are federated (for an example, see [22]). In this context, interoperability serves as a standard such as High Level Architecture [3, 23]. Nonetheless, interoperability concepts and methods lend themselves to distributed execution. Examples of these methods are numerous and they are generally used for specific classes of simulation models.

A key advantage of a well-defined modeling and simulation framework is to support building large, complex simulation models using system-theoretic concepts. Such a framework can provide a unified basis for analysis, design, and implementation of simulation models for complex systems [24, 25]. 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 [5].

In the context of the DEVS framework, the concept of DEVS-Bus [26] was introduced to support distributed execution and later extended using HLA [27, 28] and ACE-TAO ORB in logical and (near) real-time [4, 29]. A recent development executes DEVS Models using Service Oriented Architecture (SOA) [30]. However, these approaches distribute models executing using a single simulation engine. Considering a modeling framework such as DEVS, interoperability for a class of DEVS-based simulation models (e.g., Python DEVS [31] and DEVS-C++ [5]), can address a different kind of need – handling differences between alternative realizations of a formal model specifications and simulation protocols. The aim of this research, therefore, is on interoperability and enabling standardization while making use of common interoperability concepts and technologies.

In this paper we focus on the Parallel DEVS formalism which extends classic DEVS [5, 32]. The formalism is well suited to provide the basic mechanism for interoperation of simulation models for the following reasons. First, models can be combined using input and output ports and their couplings. These models can have arbitrary complexity (structure and behaviour) based on a generic, yet formal specification. Second, it allows concurrency with the closure under coupling property. Concurrency is important for handling multiple external events arriving simultaneously from one or more models and handling simultaneously scheduled internal and external events. Closure under coupling ensures correctness of input/output exchanges among components of hierarchical coupled 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. Fourth, Object-Oriented DEVS, which supports model inheritance, provides a basis for model extensibility and distributed execution in logical and/or real-time.

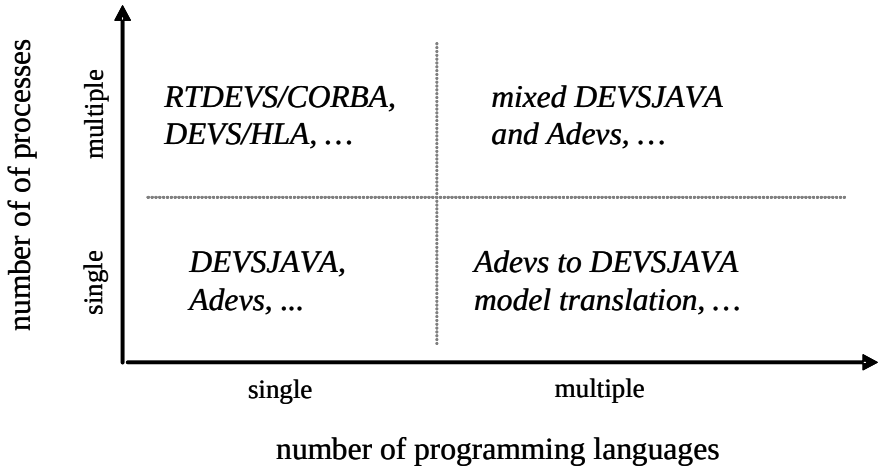


Figure 1: DEVS-based simulators realizations, given multiple programming languages and processors.

The DEVS formalism is independent of programming languages or software design choices. Indeed, there exist a variety of DEVS simulation engines implemented in several programming languages and distributed using HLA, CORBA middleware technologies, or Web-Services [33, 34]. However, interoperability issues arise among DEVS simulation engines. For example, implementation and design choice differences between DEVSJAVA and Adevs

prevent sharing and reuse of the DEVS models. In the context of this paper, we refer to models implemented in different programming languages and executed using different simulation engines as *heterogeneous models*, given the same DEVS formal modeling framework.

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 given a single DEVS abstract simulator implementation (Figure 1, top left) (RTDEVS/CORBA [4] and DEVS/HLA [29]). Moreover, distributed simulations where models are implemented in different programming language are also of interest as noted in the previous section. This is because a primary objective of reusing model implementation is to avoid recoding models expressed in different programming languages into a single programming language (Figure 1, bottom right). The interoperation between heterogeneous models can be considered as a general case of distributed simulation because different implementations of DEVS will run in different processes that communicate with HLA or general-purpose middleware such as CORBA (Figure 1, top right).

Given these last two considerations, we can consider three approaches that enable mixed DEVS-based simulation interoperability: (i) adding translations directly into the models to account for differences between programming languages and alternative simulation engine designs, which can be automated to a large extent [35], (ii) mapping different DEVS simulations to a middleware that is less formal than DEVS itself, and (iii) extending the DEVS coupling interface schemes (i.e., the syntax and semantics of the DEVS ports and couplings) to support interoperation among different models and distinct simulators implemented in different programming languages.

In the earth sciences, the most common approach for executing coupled models in high performance computing is approach (i). All different component models are combined and compiled in ad-hoc fashion by one team into a single model [36]. Compared to the type of interoperability proposed in this paper, this approach helps customizing performance of simulations, but model development is very laborious and inflexible. Furthermore, this kind of support for interoperability impedes the independent development of the component models by different research groups. This disadvantage combined with weak support for model verification and validation are major obstacles in using approach (i) and thus meeting a growing need for alternative approaches in environmental modeling and simulation [11, 37, 38].

The HLA simulation middleware approach (ii) enables the joint simulation of different kinds of models. Compared to the type of interoperability proposed in this paper, it emphasizes simulation interoperability and capabilities (e.g., data distribution management) instead of theoretical modeling and abstract simulator concepts and specifications [8, 22]. HLA standard is considered more powerful in terms of supporting any kind of simulation that can be mapped into HLA Object Model Template, including interoperation with physical software/systems. In particular, any non-simulated federate may be federated with simulated federates using a common set of HLA services (e.g., time management). However, it is difficult with HLA to ensure simulation correctness as described in by Lake et al. [22].

Our primary focus, therefore, is on DEVS simulation interoperability (iii) which is based on formal syntax and semantics and where simulation correctness among heterogeneous DEVS-based simulations is ensured. This is useful given the simplicity and universality of the DEVS framework for time-stepped logical- and real-time models. Also it is well suited for complex, large-scale models that are common in the natural sciences and can be described

rigorously in the DEVS, DTSS, and DEVS formalisms. One technique for implementing this approach is to design wrappers for the DEVS-models written in different programming languages. These can be used by simulators to allow for example cellular atomic models written in C++ and C# to exchange messages [39]. In this approach, simulators use wrappers written for different implementations of atomic models to directly communicate with one another.

In contrast to the above approaches, we present an *Abstract Model* of the Parallel DEVS formalism in logical time to establish an interface for model interoperation between multiple implementations of the DEVS Abstract simulators. The core of the SAM approach is an Abstract Model Interface which is based on the DEVS simulation protocol. This Abstract Model Interface has a fundamental role in enabling concerted simulation of disparate models executing using distinct simulators. The Abstract Model approach requires the DEVS simulation engines to provide adapters that support the Abstract Model Interface for their native models. In this way the disparity between different atomic/coupled models and their distinct simulators is accounted for. We describe a realization of this approach in terms of the DEVSJAVA and Adevs simulation engines. We show the benefits of the Abstract Model with examples highlighting (a) interoperation between different DEVS simulation engines, (b) implementation of models in a programming language for which there is no DEVS simulators, and (c) integrating 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.

Before proceeding further, we provide a brief description of the abstract DEVS simulator [5, 32]. Every atomic component model $M=\langle X, S, Y, \delta_{ext}, \delta_{int}, \delta_{conf}, \lambda, ta\rangle$ is simulated by a *simulator* and every coupled component model $N=\langle X , Y, D, \{M_{d\in D}\},\{I_{c\subseteq D\cup\{N\}}\},\{Z_{c,d}\}\rangle$ is simulated by a *coordinator*. Hence, the DEVS simulation engine constructs a hierarchy of simulators/coordinators, which corresponds to the hierarchy of atomic/coupled models. The coordinator at the root of the hierarchy is called *root coordinator*. It manages the global clock and controls the execution of the simulators/coordinator hierarchy. The simulators and coordinators of one simulation engine share a common input/output interface (i.e., input events and ports *X* and output events and ports *Y*), which is used by the parent coordinator.

The DEVS simulation proceeds in cycles based on discrete events occurring in continuous time. Coordinator and simulator advance their time based on the time of last event (*tL*) and the time of next event (*tN*).The *tN* for a simulator is determined based on time advance of the atomic models. The *tN* for coordinators is determined by taking the minimum *tN* of all its simulator and coordinator children. The simulation cycle is controlled by the root coordinator. The root coordinator starts the cycle by obtaining the minimum *tN* of all simulators (global *tN*). Second, it advances global time. Next, it asks every atomic simulator to call its model's output function λ. The simulator does this if it is imminent, i.e. simulator's *tN* is equals to global time. Then, all output events are sent as input events to their respective destinations. Subsequently, all imminent simulators are told to execute one of the models three

transitions functions. Lastly, *tL* and *tN* of the root coordinator are updated. When all atomic models have their time of next events scheduled for infinity, the root coordinator simulation cycle terminates.

A simulator decides which of the three transition functions (δ_{ext} , δ_{int} , δ_{conf}) of a model is to be invoked by comparing model's time advance function (*ta()*) to global *tN*. Time advance function specifies the relative time until a next internal transition function in the model – i.e., a scheduled next event time. The external transition (δ_{ext}) is invoked by the simulator if inputs (i.e., X) for the model arrive at times prior to the model's next event time. The internal transition (δ_{int}) is invoked once the model's time advance is expired, but no external events arrive prior to or at this time. The confluent transition (δ_{conf}) is invoked if the model's time advance is the same as the time when external events arrive. If there is no input event to the model and model's time advance is not yet expired, no action is taken. Lastly, simulator *tN* and *tL* are updated based on models *ta()*, and coordinator *tN* is updated to the minimum *tN* of its simulator and coordinator children. The responsibility of a coordinator is to exchange input and output events (*X* and *Y*) between itself and its simulators and coordinator children using external input/output and internal couplings (i.e., $\{Z_{c,d}\}$) defined in a coupled model.

All events in DEVS are bags of messages. Each bag may contain several messages of varying types for each port at a given time instance. It enables modeler to deal with parallel external input or output events. DEVS can also handle zero time advances, i.e., a complete simulation cycle taking zero time. To avoid infinite loops there must be at least one atomic model in each feedback loop that generates a non-zero time advance after a finite number of zero time steps.

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 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 Abstract Model Interface is shared by all participating simulators and models. Adapters account for disparities between the implementations. The operations of this Abstract Model can be realized with a middleware. A simulator usually directly invokes operations on the model. In the presented approach, the method invocations are mediated by the Abstract Model.

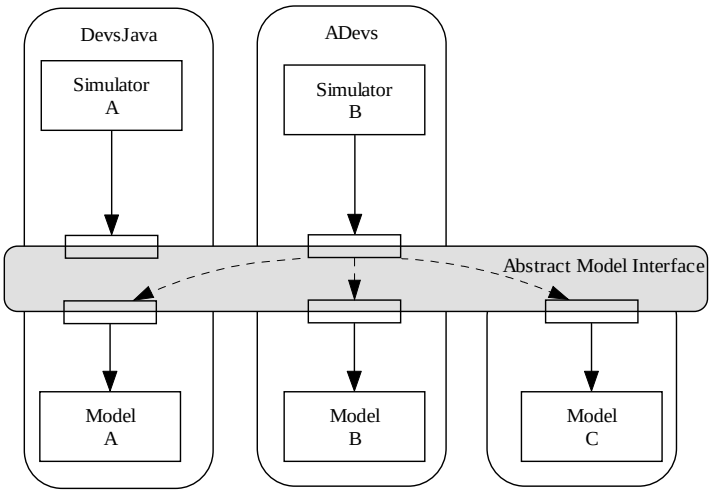


Figure 2: Abstraction of different DEVS implementations.

We specified the Abstract Model Interface in OMG-idl (Listing 1) and used CORBA to invoke these operations expressed in different programming languages. It is important to note that one basic interface is defined for models 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 can be specified as an atomic model. This will be explained down in section 3.2.2.

```
interface DEVS {// OMG-idl (Corba)
    // begin of simulation
    double doInitialize()

    // time of next internal transition without inputs
    // value also returned by doInitialize and State functions.
    double timeAdvance()

    // produce outputs for current time
    // is not allowed to change the state of the model
    Message outputFunction()

    // state transition
    double internalTransition()

    // transition with input event before timeAdvance
    double externalTransition(in double e, in Message msg)

    // input event at time of internal transition
    double confluentTransition(in Message msg)
};
Message: bag { inputPort -> value }
```

Listing 1: The Abstract Model Interface specification.

3.2 Adapting DEVS simulation engines

To support the functionality of the above Abstract Model, existing DEVS simulation engines are required to provide adapters to the Abstract Model Interface. A simulator does not need to know that some of the submodels are executed in a remote process. This is achieved by using a *Model Proxy* as shown in Figure 4. The *Model Proxy* translates its method invocations to invocations of the *Abstract Model Interface*. While the translation is only

syntactical in nature and preserves Parallel DEVS model semantics, the handling of message contents can be non-trivial.

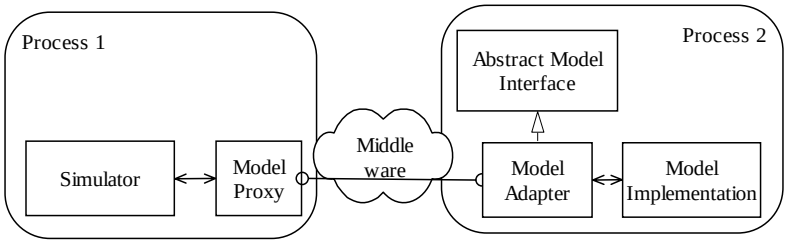


Figure 3: Adapting a specific DEVS implementations to support the Abstract Model approach.

The invocations of the Abstract Model Interface can be mediated to a remote process using a middleware. If the actual model implementation, as shown in Process 2, cannot directly support the Abstract Model, a *Model Adapter* is needed to translate the invocations of the Abstract Model to the invocations of the *Model Implementation*. 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 implementation of the Abstract Model, and all models specified for the simulation engine can be represented as implementations of the Abstract Models.

3.2.1 Model Proxy and Model Adapter for Atomic Models

The implementation of the model proxy and the model adapter for atomic models is straightforward, because the abstract model interface actually corresponds to an atomic model. The model proxy was implemented in both example implementations DEVSJAVA (Listing 2) and Adevs by extending the atomic model. All invocations are simply translated to invocations of the Abstract Model Interface. The only non-trivial issue, is the translation of message contents, for which an additional software component is employed. Error handling has been omitted from Listing 2 for compactness reasons.

```
void initialize() {
    ta = devsMod.doInitialize();
    if( ta == devsBridge.DEVS.TA_INFINITY )
            passivate();
    else
            holdIn("active",ta);
}
void deltext(double e, MessageInterface x){
    MsgEntity[] msg = trans.devs2CorbaInputs(x);
    ta = devsMod.externalTransition(e, msg);
    if( ta == devsBridge.DEVS.TA_INFINITY )
            passivate();
    else
            holdIn("active",ta);
}
void deltcon(double e, MessageInterface x){
    MsgEntity[] msg = trans.devs2CorbaInputs(x);
    ta = devsMod.confluentTransition(msg);
    if( ta == devsBridge.DEVS.TA_INFINITY )
            passivate();
    else
            holdIn("active",ta);
}
void deltint() {
    ta = devsMod.internalTransition();
```

```
    if( ta == devsBridge.DEVS.TA_INFINITY )
           passivate();
    else
           holdIn("active",ta);
}
MessageInterface out() {
    MsgEntity[] msg = devsMod.outputFunction();
    MessageInterface devsMsg = trans.corba2DevsOutputs(msg);
    return devsMsg;
}
```

Listing 2: DEVSJAVA implementation of the model proxy.

3.2.2 Model Adapter for Coupled Models

While the model adapter for atomic models is as straightforward as the model proxy, the model adapter for coupled models is a clever piece. It makes use of the DEVS closure under coupling property, which states that each coupling of systems defines a basic system [5]. Our basic idea is to consider the execution of a coordinator as the execution of an atomic model. Hence, the model adapter for a coupled model employs a coordinator to wrap the execution of the coupled model according to the specification of the Abstract Model. Therefore, the simulation cycle of a coordinator is specified as a DEVS-model within the model adapter. Although this use of the Abstract Model for coupled models is unusual given the simulator/coordinator separation, this approach has several advantages, which will be discussed in section 6.

In the following the term "processor" is used as a generic term for both simulator and coordinator. In DEVS-environments DEVSJAVA and Adevs, the processors exhibited the following methods in their interfaces.

- ComputeIO(t): with global time t, first, invokes ComputeIO function of associated processors (which eventually call the output function of the imminent models), and second, distributes the outputs to other processors and the parent coordinator.
- DeltFunc(t, m): with global time t and message m, adds given message bag to the inputs of the coordinator, distributes these inputs to the corresponding processors, and invokes DeltFunc of the associated processors (which eventually execute transition functions of the models) for time t and updates the times of last and next event (*tL*, *tN*).

Usually these methods are called by the parent-coordinator. In order to execute a coordinator within an Abstract Model, these methods have to be called from the methods of the model adapter in the same correct order as by the parent-coordinator. The crucial point is that each processor receives all inputs before its delta-Function is invoked. The parallel DEVS simulation protocol guarantees that the output function is called exactly once before the internal or confluent transition function. The functions tL, tN, and getOutputs return the coordinator's last event time, next event time and the bag of external outputs respectively. With these conditions, the coordinators' methods can be mapped to the Abstract Model as shown in Listing 3. Error handling and message translation have been omitted from the listing for compactness reasons. In section 4.1.3. it will be demonstrated that the execution order is kept correct.

```
double doInitialize(){
    coord.initialize();
    return timeAdvance();
}
double timeAdvance(){
    return coord.tN() – coord.tL();
}
double internalTransition(){
    // ComputeIO called before by outputFunction
    coord.DeltFunc( coord.tN(), [empty set] );
    return timeAdvance();
}
double externalTransition( e, x ){
    // ouputFunction was not called before
    coord.DeltFunc(coord.tL() + e, x );
    return timeAdvance();
}
double confluentTransition( x ){
    // ComputeIO called before by outputFunction
    coord.DeltFunc( coord.tN(), x );
    return timeAdvance();
}
MsgEntity[] outputFunction(){
    coord.ComputeIO( coord.tN() );
    return coord.getOutputs();
}
```

Listing 3: DEVSJAVA implementation of the model adapter for coupled models.

The implementation of the coordinator can be potentially very different from the description in Listing 3 given other DEVS simulation engines. However, all the implementations are to exhibit the division into a part of calculation and distribution of processors outputs on the one hand, and a part of execution of the transition on the other hand. Hence, the description in Listing 3 can guide the development of model adapters for coupled models of other simulation engines.

3.2.3 Integration of Non-DEVS Functionality

In addition to interoperating various DEVS simulation engines, the Abstract Model can also be used to integrate non-DEVS functionality. In this case, the model adapter maps the non-DEVS functionality to the Abstract Model. Three examples of this integration follow.

A) One use case is the integration of time-stepped models. The time-advance function has to 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 case 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 [40]. 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 case is the integration of functions that do not depend on time, which means a Mealy-type passive model for example an optimization procedure. 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 a 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

The first detailed example demonstrates how a coupled model, which is developed and implemented in on DEVS simulation engine, is used as a component model within a larger coupled model within another DEVS simulation engine.

4.1.1 The ef-p Example Model

The ef-p model is simple coupled model of three atomic models (Figure 4). The atomic and coupled models are shown as blocks and couplings between them are shown as unidirectional arrows with input and output port names attached to them. The generator atomic model generates job-messages at fixed time intervals and sends them via the Out port. The transducer atomic model accepts job-messages from the generator at its Arrived port and remembers their arrival time instances. It also accepts job-messages at the Solved port. When a message arrives at the Solved port, the transducer matches this job with the previous job that had arrived on the Arrived port earlier and calculates their time difference. Together, these two atomic models form an experimental frame coupled model. The experimental frame sends the generators job messages on the Out port and forwards the messages received on its In-port to the transducers Solved-port. The transducer observes the response (in this case the turnaround time) of messages that are injected into an observed system. The observed system in this case is the processor atomic model. A processor accepts jobs at its In port and sends them via Out port again after some finite, but non-zero time period. If the processor is busy when a new job arrives, the processor discards it.

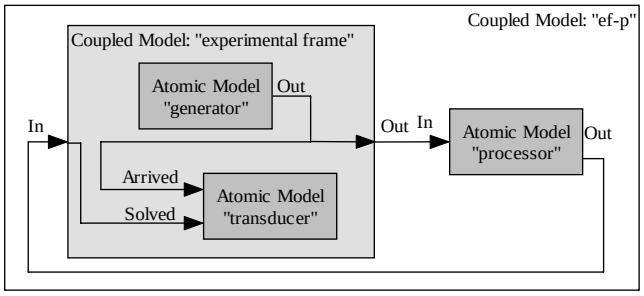


Figure 4: Experimental frame (ef)-processor (p) model.

4.1.2 Implementation of the ef-p Example Model using DEVSJAVA, Adevs and CORBA

This and the following examples were developed and run on several personal computers running the operating system Windows XP. All examples have been tested on a single machine and in addition also with running the

component models on different machines. We partitioned the models into two different simulation engines according to Figure 5. Rounded boxes represent operating system processes; white angled boxes represent simulators, dark gray boxes represent models, light grey shapes represent interoperation components, and arrows represent interactions.

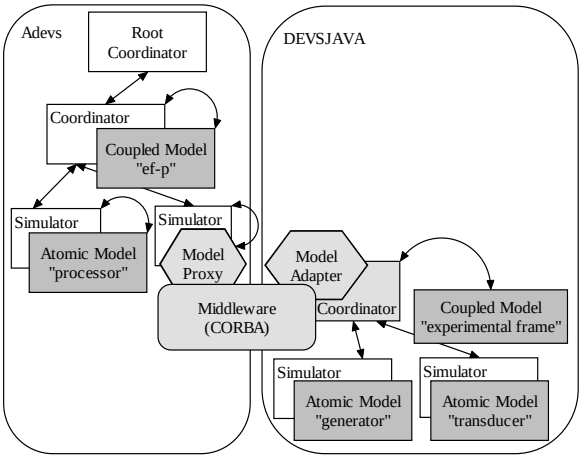


Figure 5: Distributed setup of the ef-p model.

First, we implemented the model proxy and the model adapters for the atomic and coupled models as described in section 3.2 in the two DEVS simulation engines DEVSJAVA and Adevs using the programming environments eclipse version 3.1 and Visual Studio version 7.1. Next, we implemented the ef coupled model in DEVSJAVA and the processor in Adevs in the usual way as using only one DEVS simulation engine. Finally, we set up and performed the simulation of the ef-p model. The experimental frame coupled model, a message translator, and the model adapter were constructed and started in a DEVSJAVA server process (Listing 4a). Further the CORBA-Object of the model adapter was constructed published using the SUN Object Request Broker (ORB) and naming service which is part of the JDK 1.5 (SUN 2006). The CORBA-stub of this adapter was then obtained in the C++/Adevs client process using the ACE/TAO ORB version 1.5 (Schmidt 2006). Together with a message translator the model proxy was constructed (Listing 4b). Finally, this model proxy was used as any other Adevs atomic model within the Adevs simulation (Listing 4c).

```
(a)  digraph ef = new Ef();
     MessageTranslator trans = new StringMessageTranslator();
     DEVS efAdapter = new DevsDigraph( ef, trans );

(b)  MessageTranslator *trans = new StringMsgTranslator();
     atomicDevs *efProxy = new atomicDevs(efAdapterStub._retn(),trans);

(c)  staticDigraph *coupledModel = new staticDigraph();
     coupledModel->add(proc);
     coupledModel->add(efProxy);
     coupledModel->couple(proc, proc->outPort("out"),efProxy,efProxy->inPort("in"));
     coupledModel->couple(efProxy,efProxy->outPort("out"),proc,proc->inPort("in"));
     devssim sim(coupledModel);
     sim.run(100);
```

Listing 4: Constructing and using the remote experimental frame sub-model.

4.1.3 Execution of the the ef-p Example Model

The execution of the coupled ef-model by the model adapter using a coordinator appears to the model proxy as the execution of an atomic model. The following section illustrates that the execution order is kept by an execution trace. The example is a complex confluent transition of the model setup according to Figure 5. The generator produces job messages at time intervals of 5 seconds and the processor has a processing time of 10 seconds. Before the confluent transition at time 10 the processor had accepted the first job at time 0 and neglected the second one at time 5. Next event times are 10 for the processor, 10 for the generator and Infinity for the transducer. Next event time for the ef coupled model is the minimum of the time advance for the generator and the transducer models (i.e., ta()=10).

First, the output-function of the Adevs model proxy is invoked. The model proxy invokes the output function of the DEVSJAVA model adapter (Figure 6). The signatures of the model adapter correspond to the Abstract Model (Listing 1). The signatures of the coordinator and the simulators are according to DEVSJAVA implementation and correspond to the methods described in section 3.2.2. Most of the complexity is hidden within the coordinator. There are only two calls between processes, i.e. between model proxy and model adapter. The model adapter tells its coordinator to compute and distribute the outputs. The coordinator does this by first, letting the simulator of the only imminent component (generator) invoke the generators output function via computeIO. Second, the coordinator calls the simulator's sendMessages method. This causes the simulator of the generator (g) to put messages on the transducers (t) input and on the coordinators output port. Finally, the model adapter is able to return the outputs of the coordinator as the result of the output function.

The Adevs coordinator of the ef-p model routes the output of the processor (job 1) to the model proxy. Because the model proxy is imminent, its confluent transition is invoked. The model proxy invokes the confluent transition of the model adapter. Next, the model adapter places the external input at the coordinator's inputs by calling putMessages. Next, it invokes the coordinator's transition function DeltFunc with the next event time. The coordinator first routes the input job to the simulator of the transducer. Next, it asynchronously invokes the transition functions of the simulators of the generator and the transducer. The generator's simulator has no external inputs but it is imminent, hence it executes the internal transition, which schedules the next job. The transducer is in passive state, hence its simulator executes the external transition with a bag of two inputs. When both generator's and transducer's transitions have completed, the coordinator updates its times of last and next event.

After control flow has returned to the model adapter, the model adapter asks the coordinator for the times of last and next events and returns the time difference. Finally the model proxy holds the atomic model in active state for this time.

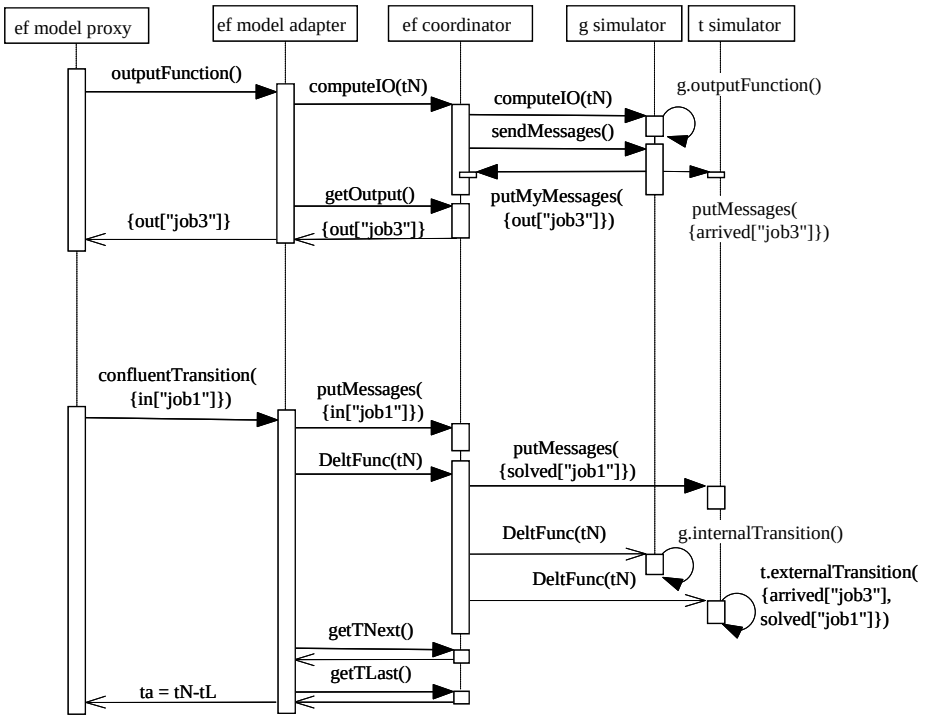


Figure 6: Sequence diagram of an output and a subsequent confluent transition.

4.2 DEVS-Compliant Models without Simulators

The second application example illustrates the simulation of a DEVS-compliant model that has been implemented without a corresponding DEVS simulation engine. We implemented the processor model (see the previous section) within VBA routines of a workbook of MS-Excel version 2003. The class of the Visual Basic processor model descended directly from the portable object adapter classes which were generated by VBORB, an object request broker for Visual Basic [41]. This processor directly implemented the Abstract Model. Hence, no model adapter was needed. 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 and published. Within a DEVSJAVA simulation a model proxy was initialized with the CORBA-stub of this processor and coupled to an experimental frame model. For the DEVSJAVA simulation it was completely transparent, that the processor submodel was executing a remote process of the Excel-worksheet.

4.3 Non-DEVS Models

The third application example demonstrates integration of Non-DEVS functionality as described in section 3.2.3c. We wrapped an optimization problem, i.e. a timeless function, by the Abstract Model Interface. The optimization problem minimized the price for power supply by ordering the power from different provides with different pricing schemes. The proportions of the power that were ordered by the different providers were optimized depending on the amount of required total power. The problem was modeled within an MS-Excel worksheet and solved by employing the MS-Excel solver. Further, a simple DEVSJAVA experiment was devised that consisted of the proxy of the optimizer model, a generator model, and an observer model. The latter two submodels and the coupled model

were specified in DEVSJAVA. The generator submodel produced events of changes in the amount of total required power. This output was connected to the optimizer model. The outcomes of the optimization, i.e. shares of power and price, were coupled to the observer model. The observer logged the inputs together with the time of receiving them. On execution the optimization was carried out whenever the generator produced changes in power demand, and the observer logged the result of the optimization at the same points in logical time when the changes in required power took place.

5 Simulation Experiments on Model Performance and Scalability

This section describes some experimental results on the performance of the Abstract Model approach using a set of ef-p models, which were presented in section 4.1.1. The first aim for these experiments was, to show that the Abstract Model approach scales well with the number of component models and links. The second aim was, to demonstrate that performance gains are achieved by distributing the simulation to several machines using the Abstract Model. In these performance studies the overall coupled model was composed of n=2..64 ef-p component models. We set up a suite of experiments including four scenarios (see Table 1). In the *Baseline* scenario, all models are simulated using DEVSJAVA in a single operating system process without using the Abstract Model. In the *Local* scenario, the Abstract Model was used to mediate interoperation between component models. The component models and their model adapters were executed in two operating system processes that were different from the process of the root-coordinator (Figure 7a). All processes executed on a single machine. In the *Remote* scenario, the root coordinator process executed on one machine while the two model processes executed on another machine. In the *Distributed* scenario, all processes executed on different machines (i.e., each process executes on its own dedicated machine).

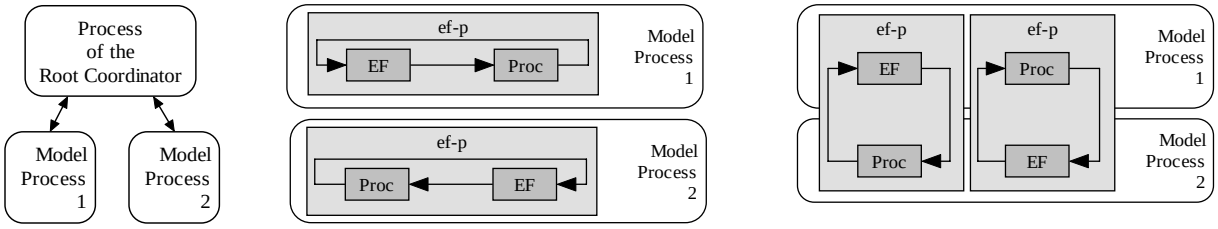
Table 1: Scenarios for measuring the performance impact of the Abstract Model.

	InProc	BetweenProc
Baseline	n=2,4,8,16,32,64	
Local	n=2,4,..,64	n=2,4,..,64
Remote	n=2,4,..,64	n=2,4,..,64
Distributed	n=2,4,..,64	n=2,4,..,64

Two kinds of overhead are important to consider when interoperating (heterogeneous or distributed) simulations. The first is due to time synchronization and the second to message passing among distinct processors (or logical processes). The second is strongly correlated with the number of nominal data links [26].

To capture these kinds of overhead, *InProc* and *BetweenProc* configurations were devised (Figure 7). Gray boxes represent models, Arrows represent couplings (message passing), and round boxes represent operating system processes. The atomic models in InProc and BetweenProc configurations were the same, only the coupling to ef-p models varied. In the former InProc configuration (Figure 7b), each ef-p model was executed within one operating

system process and message passing between the component models of each ef-p model occurred within only one operating system process. In this configuration only synchronization messages were exchanged across process boundaries because there were no nominal data links between processes. In the latter BetweenProc configuration (Figure 7c), the component models of each ef-p model resided in different processes and the couplings (ef-p coupled model) were specified in the root coordinator's process. In this configuration the component models of each ef-p model additionally exchanged data messages across process boundaries. The number of nominal data links increased by two with each additional ef-p model.



(a) (b) (c)

Figure 7: Allocation of model components to different operating system processes.

The Local, Remote, and Distributed scenarios were compared against the Baseline scenario and against one another to quantify the impact of the Abstract Model on total simulation execution times. Scaling effects were studied within each scenario by varying the number of ef-p models that were simulated together with one root coordinator.

First, Time synchronization overhead was studied by comparing the DEVSJAVA scenario and the Local InProc scenario.

$$overhead_{Synchronization} = (time_{Local\_InProc} - time_{Baseline}) / time_{Baseline} * 100\%$$

Second, Message passing overhead was studied by comparing the local InProc scenario with the local BetweenProc scenario.

$$overhead_{Message\ Passing} = (time_{Local\_BetweenProc} - time_{Local\_InProc}) / time_{Local\_InProc} * 100\%$$

Finally, the speedup of distributing the models on different machines was studied by comparing the remote and the distributed scenario.

$$speedup = time_{Remote} / time_{Distributed}$$

Performance studies were performed on an IBM T42 computer with 1GB main memory and a 1.7 GHz Pentium processor. For the remote and distributed scenarios, two identical computers (1GB main memory, 2.8 GHz Pentium processor) were connected to the T42 with a 100 Mbps network connection. Execution times were measured using the `System.nanoTime()` function. Other processes (Virus protection etc.) were still running during the performance studies, causing an additional random component in execution times. All atomic and coupled models were implemented and executed using JAVA version 1.5 and DEVSJAVA version 2.7.2 that was extended with multi-threading. To help measure execution times, 10^4 flops were added to the Proc model transition function.

5.1 Results of the Performance Studies

In all scenarios, the execution time basically increased linearly as the number of simulated ef-p models were increased (see Figure 8a and 8b). The execution time of models executing in different processes at the same computer was 2 to 4 times longer than the Baseline scenario execution time. For example, in the case of having 64 ef-p models, the local BetweenProc execution time was 3.5 times of the Baseline scenario corresponding to a 250% overhead. The overhead was due to synchronization and message passing (Figure 8c), each roughly accounting for half of the total overhead. No increasing trend of the overhead was observed due to the scaling of the number of the simulated models and the associated higher number of data links.. In the case of 16 ef-p models, the execution time for the local InProc scenario was by chance exceptionally high.

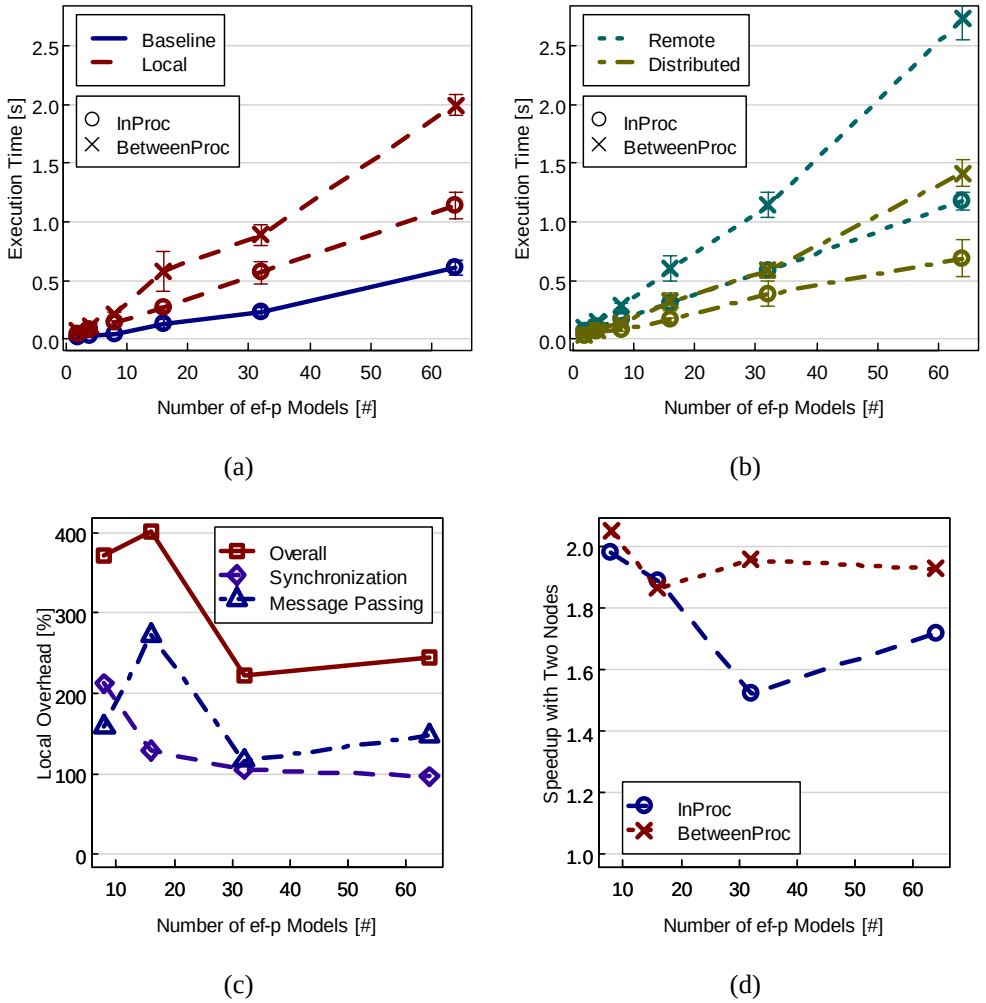


Figure 8: Execution times (a,b), overheads (c) and Speedup (d) of the performance experiments

With models residing on one different machine (remote scenario), the execution times of the InProc configuration were similar to the case where models were executed on the same machine (local scenario). However, for the

BetweenProc configuration, the execution time was higher than in the local scenario (compare Figure 8a and 8b). Distributing the models on two computers reduced execution times by two-third to nearly half within both scenarios (see Figure 8b). This corresponds to a speedup of 1.5 to 2 (see Figure 8d). It is noted that the speedup does not decrease with the number of simulated models.

6 Discussion

High performance computing often deals with complex models that are built from component models of different scientific fields and often by different teams, organizations, or institutions. Two commonly used approaches for simulation of the coupled model are (i) translating the component models into one complex model and (ii) using simulation middleware (e.g., HLA). The first approach impedes independent component model development and the second approach may not formally ensure simulation correctness. The approach of utilizing DEVS to couple component models (iii) presented in this paper is capable of overcoming both shortcomings at the same time. First, the component models can be implemented independently with different simulation engines and different programming languages and run as different processes allowing distributed execution. Second, the DEVS formalism formally ensures simulation correctness. The precondition for the application of the presented approach is that all component models comply with the (parallel) DEVS formalism. The execution of the coupled model by several communicating processes naturally leads to distributed simulation platform settings.

In order to overcome differences in various DEVS implementations and programming languages, we proposed the Abstract Model and specified its interface in a meta-language. There are also **alternative approaches** of abstracting these differences.

```
interface Simulator1{
    void start();
    double tNext();
    message setGlobalTNextAndOutput(in double gtN);
    void DeltFunc(in message);
};
interface Simulator2{
    double startAndGetTNext();
    message setGlobalTNextAndOutput(in double gtN);
    double DeltFuncAndGetTN(in message);
};
struct Coupling{
    string outport;
    string inport;
    Simulator3 destination;
};
interface Simulator3{
    void informCouplings(sequence<Coupling> sc);
    void putInputMessages( in message x );
    void start();
    double tN();
    //invokes putIn/OutputMessages of dest.
    void setGlobalTNAndSendOutputs(in double gtN);
    // input messages were set already
    void DeltFunc();
};
interface Coordinator3 : Simulator3{
    void putOutputMessages( in message x );
}
```

Listing 5: Use of abstract interfaces for simulators instead of models

As shown in Listing 5 (Simulator 1) an abstract interface can be defined for simulators instead of models [42]. The simulator is asked at each simulation cycle to generate outputs if it is imminent. There is only one transition function and the simulator decides itself based on global time and inputs which kind of transition is to be performed. A performance optimization may be achieved by integrating the invocation of *tN* function into the start function or DeltFunc , which calls δ_{ext} δ_{int} or δ_{conf} of the model (Listing 5, Simulator 2). In yet another approach (Listing 5, Simulator 3), the simulators are informed about their coupling information and send messages directly to the other simulators instead of the coordinator [e.g. 39].

However, the **SAM approach exhibits a better performance** than using an abstract interface for simulators because the SAM utilizes the sparseness of the DEVS formalism, which is grounded in the asynchronous way of model execution. Performance is determined to a great degree by the number of communications between the different operating system processes (possibly executed on different nodes) and only to a smaller degree by in-process sub-routine invocations. One invocation of a method of the abstract model interface or abstract simulator interface respectively corresponds to one inter-process communication. While there are two invocations of the abstract simulator2 interface at each simulation cycle (and even more invocations with simulator1 and simulator3), the Abstract Model interface is only invoked in the subset of simulation cycles where there are input messages to the model or if the model is imminent.

Both time **synchronization and message passing overhead** of the Abstract Model were about 120%. This is worse compared to the shared memory version of the DEVS-BUS (synchronization overhead 20-25%), but much better than the HLA/RTI implementation of the DEVS-BUS (5,000-6,000%). The message passing overhead does not increase with the number of messages and the number of nominal data links (which is true for the DEVS-BUS). One major advantage of the Abstract Model compared to the DEVS-BUS is the exploitation of parallelism. The execution of the model on two computers shows almost the same performance as in the case where no Abstract Model is used (Figure 8). Performance can improve further when a model can be partitioned appropriately to execute on a larger number of processing nodes.

The execution of the models can be migrated to a grid. The configuration of the model servers can be done by a naming service and the factory pattern. There was no need to change the configuration of the client (the process of the root coordinator) to switch between the local, remote, and distributed scenario in the performance experiments. The locations of the component models are completely transparent to the simulation. In addition to showing reasonable performance characteristics, the SAM approach for combining heterogeneous DEVS models has the advantage of making it **straightforward for sub-models to participate in a coupled simulation**. Hence, few or no changes are necessary for the DEVS models that are written for specific simulation engines, for which model adapters have been developed. Often relatively simple adapters will suffice. We demonstrated the simplicity of specifying component models and the setup of the coupled simulation (section 4.1.2). Additionally, we demonstrated how easy legacy models can be integrated into a coupled simulation by developing specific DEVS-compliant model adapters (sections 4.2 and 4.3). Other approaches (e.g., directly using HLA) place more constraints on the sub-models. This is because 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). A

further advantage of the proposed approach is the **free availability** of both CORBA implementations and DEVS simulation engines for common programming languages which are used in industrial-strength settings as well as research and development.

Correctness of model execution, in the presented approach, is grounded in the DEVS formal specification. The synchronization among distributed models and coordinators is handled by the DEVS protocol. Within DEVS the models have no notion of "global" time. However, the transitions of the remote model are invoked in correct order (see background section). In order to support this functioning, the developer of a model adapter for a specific DEVS simulation engine has to show that the model adapter is compliant to the Abstract Model. This ensures that every model that is legitimate in logical time in the original simulator is also legitimate in the heterogeneous case. Interoperability between different DEVS simulation engines using the SAM approach allows employing (or implementing) 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 benefits are that only few constraints need to be placed on models that were not specifically designed for a DEVS-simulator and that less messages between processes are required (see section performance considerations above). The coupled structure of a remote model is transparent to the simulator. Nonetheless, from the modeling perspective, every hierarchical coupled model can be systematically mapped onto a flat model without any side-effect on the approach presented here.

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

A recent work that is aimed at combining different DEVS models is **DEVS Modeling Language** (DEVSML). It relies on semi-automatic translation DEVS models to their XML counterpart [35]. This environment is being applied toward system acquisition test and evaluation [30]. In terms of execution, the environment supports synthesizing coupled DEVS models. The translation of the models that are specified using DEVSML to simulation code is far more complex in comparison with the approach developed in this paper, and currently only the single programming language JAVA is supported.

In the context of this work, we have accounted for logical-time DEVS models. The approach presented in this paper, however, is in principle also applicable to **real-time DEVS** models [43]. In this case, the interface of the Abstract Model can stay the same, but the semantics (execution) of the Abstract Model needs to comply with real-time DEVS specifications. Activities, which are defined as abstractions of tasks in a real system, are part of the real-time DEVS transition functions. Each activity takes non-zero wallclock time to be completed and thus cannot return immediately. An activity is to be computed within a finite time period. For example, the external transition of the optimization model (section 4.3) 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, the price 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.

The first **use-case of the SAM approach** is the interoperation of several DEVS models that are implemented in different DEVS simulation engines or different programming languages. This was possible before already by the DEVS-Bus/HLA implementation [29] or using HLA directly. However, the SAM approach requires far less implementation complexity and in many cases less performance overhead. The second use case is distributing a simulation of DEVS-models that are implemented in the same simulation engine. This was also possible before with several methods (DEVS-Bus, DEVS/CORBA [44], CD++ [39]). Which of these methods is most efficient in performance depends on the number of models and the number of links between the models and the implementation of the DEVS simulation engine. The SAM approach also allows to include non-DEVS component models by designing model adapters. This gives the SAM approach the potential to develop to a simulation middleware that ensures simulation correctness. The usage of HLA or MPI to design a distributed model from scratch will lead to solutions that are more efficient in performance in most cases. However, we think that the usage of the Abstract model for setting up an ad-hoc integration of existing models will be much easier, flexible, scalable, and better maintainable in most cases.

The SAM approach places only few constraints on **partitioning** a complex model into component models and **allocating models on different machines**. Both structure and allocation of a component model is completely transparent to the simulator/coordinator that simulates the component model. This gives possibility of extending the approach to include variable structure [45] or to do dynamic re-allocation of component models during run-time [46]. However, this requires extending the model proxy described in section 3.2.1. Furthermore, the SAM approach presented here can aid the DEVS Standardization effort [47].

6.1 Outlook

To further the work presented, it is important to employ the proposed approach with high performance computing for conducting large-scale simulations. As part of this effort, simulation services such as message filtering, remote parameterization, and automated stop/save/recovery of simulations may be developed and supported. Given the interest in simplifying and automating distributed simulation configuration, management, and execution, another research direction is devising a testbed with metrics to help evaluate alternative simulation design experiments. This may be done by considering performance features of different simulation engines, complexity of data transformations, communication bandwidth, and service-oriented realizations. Another further research in automated partitioning and configuration of models for distributed simulation is of high interest [46]. In particular, generic model partitioning offers a basis for assigning execution of model components to simulation engines given domain-independent computational cost modeling and thus improving performance and efficiency of complex multi-scale biological system modeling. Further, the proposed approach is planned to be applied to simulating forest carbon dynamics that consists of component models of forest growth, soil carbon dynamics, and carbon in wood products [48]. This research includes developing SAM model adapters to account for mixed continuous/discrete dynamics described with differential equation and discrete-time system specifications, as outlined in section 3.2.3.

7 Conclusions

The Shared Abstract Model (SAM) interoperability approach presented here supports DEVS-based simulations and their extensions consistent with the DEVS formalism. With this approach, component models can be developed independently in alternative DEVS simulation environments and programming languages. The correctness of disparate simulations is ensured based on DEVS formalism. The SAM can be implemented using sound distributed middleware concepts and techniques. The Abstract Model supports desirable scalability trait. It utilizes the sparseness inherent in the DEVS formalism and inter-process-communications and exhibits good simulation speedup. The basic interface developed on top of the DEVS formalism enables flexible and non-tedious integration of new component models. Furthermore, the SAM approach may be extended with standardizing message formats and simulation services. We conclude that the proposed interoperability approach supports DEVS-based simulator interoperability providing several advantages and thus that it is indispensable for the use in application domains where heterogeneous simulation models are executing on parallel/distributed platforms.

Acknowledgement

This work was funded by a doctoral scholarship of the German Academic Exchange Service. The authors thank the anonymous reviewers on an earlier version of this paper. Their critiques and suggestions helped with the organization of the paper and the presentation of the materials contained therein.

8 References

[1] Fujimoto, R. M. *Parallel and Distributed Simulation Systems*: John Wiley and Sons, Inc., 2000.

[2] Righter, R., and J. C. Walrand. "Distributed Simulation of Discrete Event Systems." *Proceedings of the IEEE* 77, no. 1 (1989): 99-113.

[3] IEEE. "HLA Object Model Template, Version IEEE 1516.2-2000." IEEE, 2000.

[4] Cho, Y. K., X. L. Hu, and B. P. Zeigler. "The RTDEVS/Corba Environment for Simulation-Based Design of Distributed Real-Time Systems." *Simulation: Transactions of the Society for Modeling and Simulation International* 79, no. 4 (2003): 197-210.

[5] Zeigler, B. P., H. Praehofer, and T. G. Kim. *Theory of Modeling and Simulation 2nd Edition*: Academic Press, 2000.

[6] Sitch, S., B. Smith, I. C. Prentice, A. Arneth, A. Bondeau, W. Cramer, J. O. Kaplan, S. Levis, W. Lucht, M. T. Sykes, K. Thonicke, and S. Venevsky. "Evaluation of Ecosystem Dynamics, Plant Geography and Terrestrial Carbon Cycling in the LPJ Dynamic Global Vegetation Model." *Global Change Biology* 9, no. 2 (2003): 161-85.

[7] Thornton, P. E., B. E. Law, H. L. Gholz, K. L. Clark, E. Falge, D. S. Ellsworth, A. H. Golstein, R. K. Monson, D. Hollinger, M. Falk, J. Chen, and J. P. Sparks. "Modeling and Measuring the Effects of Disturbance History and Climate on Carbon and Water Budgets in Evergreen Needleleaf Forests." *Agricultural and Forest Meteorology* 113, no. 1-4 (2002): 185-222.

[8] Sarjoughian, H. S., and B. P. Zeigler. "DEVS and HLA: Complementary Paradigms for Modeling and Simulation?" *Simulation: Transactions of the Society for Modeling and Simulation International* 17, no. 4 (2000): 187-97.

[9] Zeigler, B. P., and H. S. Sarjoughian. "Implications of M&S Foundations for the V&V of Large Scale Complex Simulation Models, Invited Paper." Paper presented at the Verification & Validation Foundations Workshop, Laurel, Maryland, VA. Society for Computer Simulation, https://www.dmso.mil/public/transition/vva/foundations, John Hopkins Univ., October 2002.

[10] Reynolds, J. F., and B. Acock. "Modularity and Genericness in Plant and Ecosystem Models." *Ecological Modelling* 94, no. 1 (1997): 7-16.

[11] Filippi, J. B., and P. Bisgambiglia. "Jdevs: An Implementation of a DEVS Based Formal Framework for Environmental Modelling." *Environmental Modelling & Software* 19, no. 3 (2004): 261-74.

[12] Valcke, S., E. Guilyardi, and C. Larsson. "Prism and Enes: A European Approach to Earth System Modelling." *Concurrency And Computation: Practice And Experience* 18, no. 2 (2006): 231-45.

[13] Liu, J., C. Peng, Q. Dang, M. Apps, and H. Jiang. "A Component Object Model Strategy for Reusing Ecosystem Models." *Computers and Electronics in Agriculture* 35 (2002): 17-33.

[14] Hillyer, C., J. Bolte, F. van Evert, and A. Lamaker. "The Modcom Modular Simulation System." *European Journal of Agronomy* 18, no. 3-4 (2003): 333-43.

[15] Pullar, D. "Simumap: A Computational System for Spatial Modelling." *Environmental Modelling & Software* 19, no. 3 (2004): 235-43.

[16] HLA. "HLA Framework and Rules." IEEE 1516-2000, IEEE, 2000.

[17] Nagel, J. "TreeGrOSS: Tree Growth Open Source Software - a Tree Growth Model Component." Programmdokumentation, Niedersächsischen Forstlichen Versuchsanstalt, Abteilung Waldwachstum, 2003.

[18] Liski, J., T. Palosuo, M. Peltoniemi, and R. Sievanen. "Carbon and Decomposition Model Yasso for Forest Soils." *Ecological Modelling* 189, no. 1-2 (2005): 168-82.

[19] ACIMS. 2005. "DEVSJAVA Modeling & Simulation Tool." *Arizona Center for Integrative Modelling and Simulation*, <http://www.acims.arizona.edu/SOFTWARE/software.shtml#DEVSJAVA>. (7 August 2007).

[20] Nutaro, J. J. 2005. "Adevs (a Discrete Event System Simulator) C++ Library." <http://www.ece.arizona.edu/~nutaro/index.php>. (7 August 2007).

[21] Nutaro, J. J. "Parallel Discrete Event Simulation with Application to Continuous Systems." Ph. D. dissertation, Electrical and Computer Engineering Dept, University of Arizona, 2003.

[22] Lake, T. W., B. P. Zeigler, H. S. Sarjoughian, and J. J. Nutaro. "DEVS Simulation and HLA Lookahead." Paper presented at the Simulation Interoperability Workshop, Orlando, FL IEEE, Spring 2000.

[23] IEEE. "HLA Federation Development and Execution Process, Version IEEE 1516.3." IEEE, 2003.

[24] Kim, T. G., S. M. Cho, and W. B. Lee. "DEVS Framework for Systems Development, Unified Specification for Logical Analysis, Performance Evaluation, and Implementation." *Discrete Event Modeling and Simulation Technologies: A Tapestry of Systems and Ai-Based Theories and Methodologies*, edited by H. S. Sarjoughian and F. E. Cellier, 131-66: Springer, 2001.

[25] Davis, P. K., and R. H. Anderson. "Improving the Composability of Department of Defense Models and Simulations." Santa Monica, CA: RAND, 2004.

[26] Kim, Y. J., J. H. Kim, and T. G. Kim. "Heterogeneous Simulation Framework Using DEVS Bus." *Simulation: Transactions of the Society for Modeling and Simulation International* 79 (2003): 3-18.

[27] Dahmann, J., M. Salisbury, C. Turrel, P. Barry, and P. Blemberg. "HLA and Beyond: Interoperability Challenges." Paper presented at the Simulation Interoperability Workshop, Orlando, FL IEEE, 1999.

[28] Fujimoto, R. "Time Management in the High-Level Architecture." *Simulation: Transactions of the Society for Modeling and Simulation International* 71, no. 6 (1998): 388-400.

[29] ACIMS. 2005. "DEVS/HLA Software." *Arizona Center for Integrative Modeling and Simulation*, <http://www.acims.arizona.edu/SOFTWARE/software.shtml#DEVS/HLA>. (7 August 2007).

[30] Mittal, S., J. L. R. Martín, and B. P. Zeigler. "DEVS-Based Simulation Web Services for Net-Centric T&E." Paper presented at the Summer Computer Simulation Conference SCSC'07, San Diego 2007.

[31] Bolduc, J.-S., and H. Vangheluwe. 2002. "A Modeling and Simulation Package for Classic Hierarchical DEVS." *Modelling, Simulation and Design lab, McGill University in Montreal, Quebec, Canada.*, <http://moncs.cs.mcgill.ca/MSDL/research/projects/DEVS/PythonDEVS/PythonDEVS.pdf>. (7 August 2007).

[32] Chow, A. C. H. "Parallel DEVS: A Parallel, Hierarchical, Modular Modeling Formalism and Its Distributed Simulator." *Simulation: Transactions of the Society for Modeling and Simulation International* 13, no. 2 (1996): 55-67.

[33] Cheon, S., and B. P. Zeigler. "Web Service Oriented Architecture for DEVS Model Retrieval by System Entity Structure and Segment Decomposition." Paper presented at the DEVS Integrative M&S Symposium, Huntsville, AL 2006.

[34] Kim, K. H., and W. S. Kang. "A Web Services-Based Distributed Simulation Architecture for Hierarchical DEVS Models." *Artificial Intelligence and Simulation*, 370-79, 2004.

[35] Mittal, S., and J. L. R. Martín. "DEVSML: Automating DEVS Execution over SOA Towards Transparent Simulators Special Session on DEVS Collaborative Execution and Systems Modeling over SOA." Paper presented at the DEVS Integrative M&S Symposium DEVS' 07 2007.

[36] Raddatz, T. J., T. J. Schnitzler, E. Roeckner, W. Knorr, C. Reick, R. Schnur, and P. Wetzel. "Modelling the Carbon Cycle Response to Anthropogenic CO2 Emissions: Uncertainties and Constraints." *Geophysical Research Abstracts* 7, 04642 (2005).

[37] Papajorgji, P., H. W. Beck, and J. L. Braga. "An Architecture for Developing Service-Oriented and Component-Based Environmental Models." *Ecological Modelling* 179, no. 1 (2004): 61-76.

[38] Roxburgh, S. H., and I. D. Davies. "Coins: An Integrative Modelling Shell for Carbon Accounting and General Ecological Analysis." *Environmental Modelling & Software* 21, no. 3 (2006): 359-74.

[39] Lombardi, S., G. A. Wainer, and B. P. Zeigler. "An Experiment on Interoperability of DEVS Implementations." Paper presented at the SIW 2006.

[40] Kofman, E. "Discrete Event Simulation of Hybrid Systems." *Siam Journal on Scientific Computing* 25, no. 5 (2004): 1771-97.

[41] Both, M. 2006. "Vborb - an Visual Basic Object Request Broker." <http://www.martin-both.de/vborb.html>. (7 August 2007).

[42] Zeigler, B. P., D. Kim, and S. J. Buckley. "Distributed Supply Chain Simulation in a DEVS/Corba Execution Environment." Paper presented at the Proceedings of the 1999 Winter Simulation Conference 1999.

[43] Kim, T. G., S. M. Cho, and W. B. Lee. "A Real-Time Discrete Event System Specification Formalism for Seamless Real-Time Software Development." *Discrete Event Dynamic Systems* 7 (1997): 355-75.

[44] Kim, K. H., and W. S. Kang. "Corba-Based, Multi-Threaded Distributed Simulation of Hierarchical DEVS Models: Transforming Model Structure into a Non-Hierarchical One." *Computational Science and Its Applications - Iccsa 2004, Pt 4*, 167-76, 2004.

[45] Hu, X., X. Hu, B. P. Zeigler, and S. Mittal. "Variable Structure in DEVS Component-Based Modeling and Simulation." *Simulation: Transactions of the Society for Modeling and Simulation International* 81, no. 2 (2005): 91-102.

[46] Park, S., C. A. Hunt, and B. P. Zeigler. "Cost-Based Partitioning for Distributed and Parallel Simulation of Decomposable Multi-Scale Constructive Models." *Simulation: Transactions of the Society for Modeling and Simulation International* 82, no. 12 (2006): 809-26.

[47] Wainer, G., B. Zeigler, H. Sarjoughian, and J. Nutaro. "DEVS Standardization Study Group Terms of Reference." Simulation Interoperability Standards Organization, 2004.

[48] Wutzler, T. "To Build a Spatial Model of Forest Growth Using Stand-Based Forest Inventory." *International Conference on Modeling Forest Production - Scientific Tools, Data Needs and Sources, Validation and Application*, edited by H. Hasenauer and A. Mäkelä, 503. Vienna, Austria: Department of Forest- and Soil Sciences BOKU University of Natural Resources and Applied Life Sciences, Vienna, 2004.

Author Bios:

Thomas Wutzler is a PhD student at Max-Planck Institute for Biogeochemistry, Jena, Germany and was a visiting scholar in 2005 at the Arizona Center for Integrative Modeling and Simulation (ACIMS), Tempe, USA.

Hessam Sarjoughian is Assistant Professor of Computer Science and Engineering at Arizona State University, Tempe and Co-Director of the Arizona Center for Integrative Modeling and Simulation. His research includes modeling and simulation methodologies, model composability, network co-design, and agent-based simulation.