*Proceedings of the 2011 Winter Simulation Conference*
*S. Jain, R. R. Creasey, J. Himmelspach, K. P. White, and M. Fu, eds.*

# MODELLING AND SIMULATION-BASED DESIGN OF A DISTRIBUTED DEVS SIMULATOR

Eugene Syriani [†,‡]

Hans Vangheluwe [◇,‡]

[†] Dept. of Computer Science
University of Alabama
Tuscaloosa, AL 35401

[◇] Dept. of Mathematics and Computer Science
University of Antwerp
Antwerp, Belgium

Amr Al Mallah [‡]

[‡] School of Computer Science
McGill University
Montréal, Canada

## ABSTRACT

Distributed, discrete-event simulators are typically deployed on different computing and network platforms using different implementation languages. This hampers realistic performance comparisons between simulator implementations. Furthermore, algorithms used are typically only present in code rather than explicitly modeled. This prohibits rigorous analysis and re-use. In this paper, the structure and behavior of a distributed simulator for the DEVS formalism is modeled explicitly, in the DEVS formalism. Simulation of this model of the simulator allows for the quantitative analysis of reliability and performance of different alternative simulator designs. In particular, using a model of a distributed simulator allows one to simulate scenarios such as failures of computational and network resources, which can be hard to realize in reality. We demonstrate our model-based approach by modeling, simulating and ultimately synthesizing a distributed DEVS simulator. Our goal is to achieve fault tolerance whilst optimizing performance.

## 1 INTRODUCTION

Simulation may put high demands on computational and memory resources. Distributed environments can help overcome many of the limitations imposed by single processor implementations of large-scale tasks. In this paper we focus on the simulation of models in the Discrete Event system Specification (DEVS) (Zeigler 1984) formalism. Our experience shows that this formalism, thanks to its modularity and compositionality is highly suitable for large-scale tasks, such as model transformation (Syriani and Vangheluwe 2008).

A distributed environment for the simulation of DEVS models is attractive for several reasons. Although not the primary goal of distributed simulation, model execution time can be reduced. Also, the limited memory issue of a single machine can be overcome and models with an extremely large state-space can still be handled. Our main goal in distributing DEVS models is inter-operability. This allows one to handle geographically distributed users and/or resources (*e.g.,* databases or specialized equipment) and to exploit greater data handling capability through specialized nodes. Furthermore, desirable properties such as fault-tolerance can be supported in a natural way.

The following section reviews the classic DEVS formalism as well as its simulation protocol. Section 3 proposes a DEVS model representing a DEVS simulator. This model is extended to that of a distributed DEVS simulator together with preliminary fault-tolerance capabilities. From this model, one can synthesize or implement a distributed DEVS simulator, on dedicated middleware. Section 4 outlines our implementation. In Section 5, the modeled distributed simulator is calibrated by means of values gathered from the implemented distributed DEVS simulator. Subsequently, simulation experiments on the modeled distributed

DEVS simulator allow one to determine optimal values for the model's parameters to achieve goals such as performance or fault-tolerance. Section 6 compares some of the current distributed DEVS implementations and shows how our modeling and simulation-based approach can be considered as a generalisation of these. Section 7 concludes.

## 2 CLASSIC DEVS

We introduce the classic *Discrete EVent system Specification* (DEVS) formalism and review its simulation protocol. The notation introduced here will be used for the remainder of the article.

### 2.1 Formalism

The DEVS formalism was introduced in the late seventies by Bernard Zeigler as a rigorous basis for the compositional modeling and simulation of discrete event systems (Zeigler 1984). It has been successfully applied to the design, performance analysis, and implementation of a plethora of complex systems such as peer-to-peer networks (Xie, Boukerche, Zhang, and Zeigler 2008), transportation systems (Lee, Lim, and Chi 2004), and complex natural systems (Filippi and Bisgambiglia 2004).

A DEVS model is either *atomic* or *coupled*. An atomic model describes the behavior of a reactive system. A coupled model is the composition of several DEVS sub-models which can be either atomic or coupled. Sub-models have *ports*, which are connected by channels. Ports are either *input* or *output*. Ports and channels allow a model to send and receive events between models. A channel must go from an output port of some model to an input port of a different model, from an input port of a coupled model to an input port of one of its sub-models, or from an output port of a sub-model to an output port of its parent model.

We denote an **atomic DEVS** model by: $\langle S, X, Y, \delta_{int}, \delta_{ext}, \lambda, \tau \rangle$ . $S$ is a set of sequential **states**. $X$ is a set of allowed **input events**. $Y$ is a set of allowed **output events**. There are two types of transitions between states: $\delta_{int} : S \to S$ is the **internal transition function** and $\delta_{ext} : Q \times X \to S$ is the **external transition function**. Associated with each state are $\tau : S \to \mathbb{R}^+ \cup \{+\infty\}$, the **time-advance** function and $\lambda : S \to Y$, the **output function**. In this definition, $Q = \{(s,e) \,|s \in S, 0 \leq e \leq \tau(s)\}$ is called the **total state space**. For each $(s,e) \in Q$, $e$ is called the **elapsed time**, the time the system has spent in a sequential state $s$ since it last underwent a transition.

Informally, the operational semantics of an atomic model is as follows: the model starts in its initial state $(s_0, e_0)$. It will remain in a given state for as long as the time-advance of that state specifies or until input is received on some port. If no input is received, after the time-advance of the state expires, the model first (before changing state) produces an output event as specified by the output function. Then, it instantaneously transitions to a new state specified by the internal transition function. However, if an input event is received before the time for the next internal transition, then it is the external transition that is applied. The latter depends on the current state, the time elapsed since the last transition, and the input event.

We denote a **coupled DEVS** model named $C$ by: $\langle X, Y, N, M, I, Z, \texttt{select} \rangle$, where $X$ and $Y$ are as before. $N$ is a set of **component names** (or labels) such that $C \notin N$. $M = \{M_n | n \in N, M_n$ is a DEVS model (atomic or coupled) with input set $X_n$ and output set $Y_n\}$ is a set of DEVS **sub-models** (or components). $I = \{I_n \mid n \in N, I_n \subseteq N \cup \{C\}\}$ is a set of **influencer** sets for each component. $Z = \{Z_{i,n} | \forall n \in N, i \in I_n, Z_{i,n} : Y_i \to X_n \vee Z_{C,n} : X \to X_n \vee Z_{i,C} : Y_i \to Y\}$ is a set of **transfer functions** between connected components. $\texttt{select} : 2^N \to N$ is the **select** or tie-breaking function. $2^N$ denotes the power set of $N$ (the set of all sub-sets of $N$).

The connection topology of sub-models is expressed by the influencer set $I_n$ of the components. Note that for a given model $n$, this set includes not only the external models that provide inputs to $n$, but also its own internal sub-models that produce its output (if $n$ is a coupled model). Transfer functions ($Z_{i,n}$) represent output-to-input translations between components. They can be thought of as channels that perform appropriate type translations. For example, a "departure" event output of one sub-model is translated to an "arrival" event on a connected sub-model's input. The `select` function takes care of conflicts as explained below.

The semantics of a coupled model is, informally, the parallel composition of the semantics of all its sub-models. A priori, each sub-model in a coupled model is assumed to be an independent process, concurrent to the rest. There is no explicit method of synchronization between processes. Blocking does not occur, except if it is explicitly modeled by the output function of a sender and the external transition function of a receiver. There is however a *serialization* whenever there are multiple sub-models that have an internal transition scheduled at the same time (this set is referred to as the **imminent set**). The modeler controls which of the conflicting sub-models undergoes its transition first by means of the `select` function.

Our work uses the `pythonDEVS` (Bolduc and Vangheluwe 2001) DEVS simulator, grafted onto the object-oriented scripting language Python.

## 2.2 The DEVS Simulation Protocol

A simplistic but highly modular simulation protocol for DEVS models combines every DEVS model (atomic and coupled) with a **solver**. The simulation proceeds through the interaction (message passing) between the solvers (Kim, Seong, Kim, and Park 1996) as shown in Figure 1.

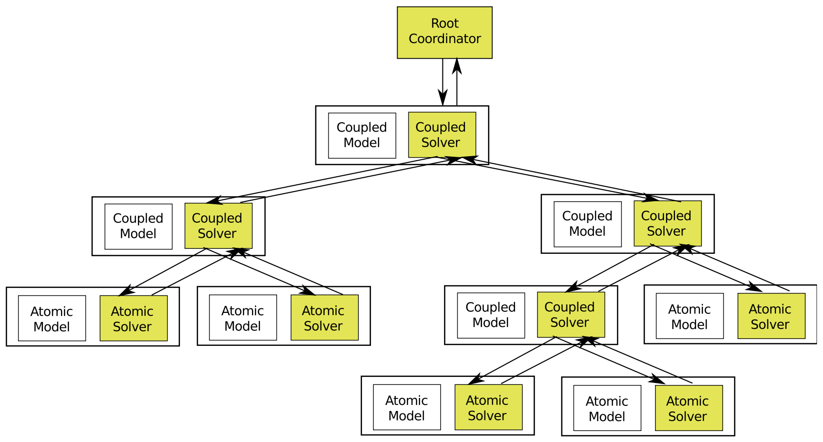


Figure 1: Hierarchical Simulation Protocol.

The solvers have access to their respective models through an appropriate model-solver interface. In an atomic DEVS **atomic solver**, the last event time $t_L$ as well as the local state $s$ are kept. In a **coordinator** (the solver of coupled DEVS models), only the last event time $t_L$ is kept. The next event time $t_N$ is stored in both types of solvers and is sent as output when the solver is queried. This requires consistent (recursive) initialization of the $t_L$s. The $t_N$ allows one to check whether the solvers are appropriately synchronized. The operation of an abstract simulator (atomic solver or coordinator) involves handling four types of messages sent between a *source* and a *target* solver (associated with their respective models):

- `INIT`: $(s_0, source, target, t)$ message holding the state at which the model and the simulation time $t$ are (re)set;
- $\star$: $(source, target, t)$ message to indicate that an internal transition is due, at simulation time $t$;
- `X`: $(x, source, target, t)$ message to carry the external input information $x$, at simulation time $t$;
- `Y`: $(y, source, target, t_N)$ message to carry the output information $y$; and
- `DONE`: $(source, target, t_N)$ message to acknowledge that one of the above messages has successfully been handled.

When a coordinator receives a $\star$ message, it selects an imminent component $i^*$ from the imminent set by means of the `select` function specified for the coupled model it is associated with. The message is then routed to $i^*$. When an atomic solver receives a $\star$ message, it generates an output message `Y` based on the old state of the atomic model it is associated to. It then computes the new state by means of the internal transition function of the atomic model it is associated to. Note how DEVS output messages are only produced while executing internal events. When a solver outputs a `Y` message, it sends it to its parent coordinator. The coordinator sends the output, after appropriate output-to-input translation ($Z_{i,n}$), to each of the influencees of $i^*$ (if any). If the coupled model $C$ itself is an influencee of $i^*$, the output, after appropriate output-to-output translation ($Z_{i,C}$), is sent to $C$'s parent coordinator.

When a coordinator receives an `X` message from its parent coordinator, it routes the message, after appropriate input-to-input translation, to each of the affected components. When an atomic solver receives an `X` message, it executes the external transition function of its associated atomic model.

After processing an `X` or `Y` message, a solver sends a `DONE` message to its parent coordinator to prepare a new schedule. Once a coordinator has received `DONE` messages from all its components, it sets its next-event-time $t_N$ to the minimum $t_N$ of all its components and sends a `DONE` message to its parent coordinator. This process is recursively applied until the top-level **root coordinator** receives a `DONE` message.

To run a simulation experiment, the initial values of $t_L$, $s$, and $t_N$ must first be set in all simulators of the hierarchy. If $t_N$ is kept in the simulators, it must be recursively set too. Once the initial conditions are set, the main loop which sends a $\star$ message and waits for a `DONE` message, is executed until some simulation termination condition is satisfied.

In addition to the classic DEVS formalism, the parallel DEVS formalism (Chow and Zeigler 1996) is also widely used. It removes the sequentialization (through $select$) of simultaneous internal transitions. A confluent transition function $\delta_{con} : Q \times X \rightarrow S$ is added to the classic atomic DEVS model. It is triggered when an atomic model receives an external event at the time of its internal transition. Note that in `pythonDEVS`, which implements classic DEVS, the external transition takes precedence over the internal transition. In parallel DEVS, the event set $X$ is now a set of bags of events since atomic simulators may output events concurrently. As for the coupled model, the $select$ function is removed.

## 3 MODELLING A SIMULATOR

In this section, we show how the DEVS simulator described above can be explicitly modeled in DEVS. First, each of the simulation entities is represented as an atomic DEVS model. Then, we model a distributed simulation engine for an arbitrary DEVS model. The model explicitly represents the simulation entities, the different machines, and the communication layer. Figure 2 shows the architecture of a cluster integrating all the different modeled entities. A star on an in-port is a shorthand notation to represent channels incoming from all the components of the same type as the source of the channel (*e.g.,* every Simulator's log port is connected to the log_in port of the Log). A star on an out-port is a shorthand notation to represent channels outgoing to all the components of the same type as the target of the channel (*e.g.,* the control_out port of the Master is connected to every Simulator's control_in port). Dots between ports with a generic label are a shorthand notation to represent as many ports on the host component as there are components of the same type as the source/target of the channel (*e.g.,* the out-port pattern labeled simID on Machine denotes one such out-port per AS, CO, and RC).

### 3.1 The Simulation Entities

The simulation model is composed of Atomic Solvers, Coordinators, and a Root Coordinator, each modeled as an atomic DEVS block. Solvers and coordinators hold their corresponding model in their state. Each of these simulator models has one in-port receive and as many send out-ports as there are machines (this is needed since classic DEVS lacks variable structure). Sending and receiving *simulation messages* $\star$, `X`, `Y`, and `DONE` (encoded as events) is performed via these ports. Additional ports handle *reallocation messages*, *control messages* (stop and resume), and *logging messages* for fault-tolerance purposes. A simulator receives a *(re-)allocation message* to indicate on which machine the simulator operates.

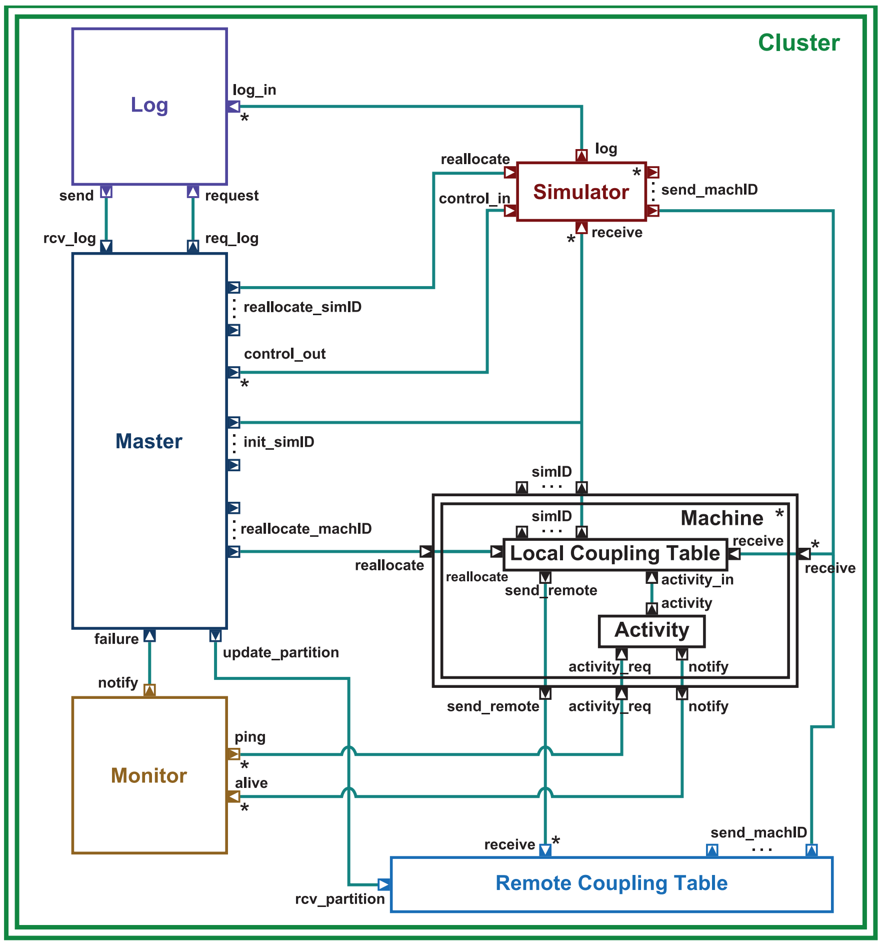


Figure 2: A DEVS model representing a distributed environment for a DEVS simulation.

### 3.1.1 The Atomic Solver Model

An Atomic Solver (AS) is an atomic DEVS model. Its state is composed of:

- the atomic DEVS model $M$ it simulates;
- a unique identifier id such that map(id) uniquely identifies $M$;
- the identifier of the parent of $M$, parentId;
- the last event time $t_L$;
- the next event time $t_N$;
- the output set $\Lambda$ of $M$;
- a bit-vector activePorts identifying which out-port is currently active (its size is determined by the number of machines); and
- the mode $\mu$ the AS is in: *PAUSED* or *RUNNING*.

An AS is reactive. It waits for either a $\star$ event to process the internal transition function of $M$ or an X event to process the external transition function of $M$. The state of the AS is updated in $\delta_{ext}$ as described in Algorithm 1. It also receives a special simulation message for initialization, INIT, holding the state the AS should be (re)set to. The DEVS model $M$ is set at instantiation-time of the AS and activePorts is set to the zero vector. Its mode is triggered by a control message $\chi$ received from the control_in port and, from the reallocate in-port, it receives reallocation message $\rho$ indicating the (possibly) new active out-port. In the algorithm, we distinguish structural elements of the AS from those of its model $M$ by annotating the latter with $\bullet^M$. The internal transition function $\delta_{int}$ clears $\Lambda$. If $\mu = \textit{RUNNING}$ and either $\Lambda \neq \emptyset$ or one of $\star$ or `X` was received, then $\tau(s)$ is the real execution time spent for applying $\lambda^M$, $\delta^M_{int}$, or $\delta^M_{ext}$. On the other hand, if `INIT` or $\chi$ was received, then $\tau(s) = 0$. Otherwise, the time advance of the AS is infinity. In case $\Lambda \neq \emptyset$, the output function $\lambda$ first sends `Y`: $(\Lambda, \texttt{id}, \texttt{parentId}, t_L)$, followed by `DONE`: $(\texttt{id},\texttt{parentId}, t_N)$. Note that the output is sent via the currently active out-port. The output function is extended to allow logging for handling fault-tolerance (see Section 3.3): whenever a simulation message is received, the new state of the AS is serialized and output through the log port.

### 3.1.2 The Coordinator Model

A *Coordinator* (CO) is an atomic DEVS model. Its state is the same as for the AS with additional information:

- $M$ is now a coupled DEVS model;
- the list of events to be processed $L$;
- the list of children (simulators this CO coordinates down the simulators hierarchy) `children`;
- the list of children still processing an event `activeChildren`; and
- $\Psi$ is the output set of `X` messages for its children (not to be confused with $\Lambda$).

The external transition function of the CO is modified from the one of an AS as described in Algorithm 2. `activePorts`,`children`, and `activeChildren` are set at instantiation-time of the CO.

The internal transition function $\delta_{int}$ clears $\Lambda$ and $\Psi$. The time advance of the CO is infinity unless $\mu = \textit{RUNNING}$ and either $\Lambda \neq \emptyset$, $\Psi \neq \emptyset$, `INIT` was received, or $\chi$ was received. In this case, $\tau(s) = 0$.

If `INIT` was received, the output function of the CO produces `INIT`: $(\texttt{nil}, id, \texttt{children}, t)$, where $t$ is the same time as in the `INIT` message the CO received. Having $s_0 = \texttt{nil}$ in the `INIT` message means that the simulator is starting (usually sent only once). If $\Psi \neq \emptyset$, the CO sends `X`: $(\Lambda, id, \texttt{activeChildren}, t)$, where $t$ is the same time as in the `X` message the CO received. Otherwise, if $\Lambda \neq \emptyset$, the output function first produces `Y`: $(\Psi, id, \texttt{parentId}, t)$, where $t$ is the same time as in the `X` or `Y` message the CO received. However, the CO sends $\star$: $(\Lambda, id, i^*, t)$ when it received a $\star$ message with time $t$. Finally, if $\texttt{activeChildren} = \emptyset$ then `DONE`: $(id, \texttt{parentId}, t_N)$ is sent.

**Algorithm 1** The $\delta_{ext}((s,e),x)$ of an AS.

```
UPON RECEIVE INIT: $(s_0, \textit{source}, \textit{target}, t)$ do
  if $s_0 \neq \texttt{nil}$ then
    $s^M \leftarrow s_0$
  else
    $\texttt{id}, \texttt{parentId}, \Lambda \leftarrow \textit{target}, \textit{source}, \emptyset$
    $t_L \leftarrow t - M.e$
    $t_N \leftarrow t_L + \tau^M(s^M)$
    $\mu \leftarrow \textit{PAUSED}$
  end if

UPON RECEIVE X:$(x, \textit{source}, \textit{target}, t)$ do
  $s^M \leftarrow \delta^M_{ext}((s_M, t - t_L), x)$
  $t_L \leftarrow t$
  $t_N \leftarrow t_L + \tau^M(s^M)$
  $M.e \leftarrow 0$

UPON RECEIVE $\star$:$(\textit{source}, \textit{target}, t)$ do
  $\Lambda \leftarrow \Lambda \cup \{\lambda^M(s^M)\}$
  $s^M \leftarrow \delta^M_{int}(s^M)$
  $t_L \leftarrow t$
  $t_N \leftarrow t_L + \tau^M(s^M)$
  $M.e \leftarrow 0$

UPON RECEIVE $\chi$ do
  if $\chi == \texttt{PAUSE}$ then
    $\mu \leftarrow \textit{PAUSED}$
  else if $\chi == \texttt{RESUME}$ then
    $\mu \leftarrow \textit{RUNNING}$
  end if

UPON RECEIVE $\rho$ do
  activePorts $\leftarrow (0, \ldots, 0)_{|\texttt{activePorts}|}$
  activePorts$[\rho] \leftarrow 1$
```

### 3.1.3 The Root Coordinator Model

A *Root Coordinator* (RC) is also an atomic DEVS model. Its state consists of:

- `id`, `activePorts`, and `children` as defined before;
- the current simulation time $T$;

- a termination condition $\theta$.

**Algorithm 2** The $\delta_{ext}\left((s,e),x\right)$ of a CO

```
UPON RECEIVE INIT: $(s_0, source, target, t)$ do
  if $s_0 \neq$ nil then
    $s^M \leftarrow s_0$
  else
    $\mathtt{id}, \mathtt{parentId}, \Lambda, \Psi \leftarrow target, source, \emptyset, \emptyset$
    $t_L \leftarrow t$
    $t_N \leftarrow +\infty$
    $\mu \leftarrow$ PAUSED
  end if

UPON RECEIVE X:$(x, source, target, t)$ do
  activeChildren $\leftarrow$ activeChildren $\cup \left\{i | \mathtt{map(id)} \in I_i^M\right\}$

  $\Psi \leftarrow \bigcup_{i \in \mathtt{activeChildren}} Z^M_{\mathtt{map(id)},i}(x)$
  $t_L \leftarrow t$

UPON RECEIVE $\star$:$(source, target, t)$ do
  immList$\leftarrow \{i | (i,T) \in L \wedge T = t\}$
  $i^* \leftarrow \Xi(\mathtt{immList})$
  activeChildren $\leftarrow$ activeChildren $\cup \{i^*\}$
  remove$(i^*, L)$
  $t_L \leftarrow t$

UPON RECEIVE Y:$(y, source, target, t)$ do
  activeChildren $\leftarrow$ activeChildren $\cup \left\{i | i \in I^M_{source} - \{\mathtt{map(id)}\}\right\}$
  $\Psi \leftarrow \bigcup_{i \in \mathtt{activeChildren}} Z^M_{i,source}(y)$
  $\Lambda \leftarrow \Lambda \cup \left\{Z^M_{i,\mathtt{map(id)}}(y) | i \in I^M_{\mathtt{map(id)}}\right\}$
  $t_L \leftarrow t$

UPON RECEIVE DONE:$(source, target, t)$ do
  $L \leftarrow L\tilde{\cup}\{(source, t)\}$    // replace $source$ entry if it already exists in $L$, otherwise add entry to $L$
  activeChildren $\leftarrow$ activeChildren $- \{source\}$
  $t_N \leftarrow \min(t_N, t)$
```

When the external transition function receives DONE: $(source, target, t)$ $T$ is set to $t$. However, if $\theta$ is not satisfied, the simulation is stopped. The time advance function $\tau$ of the RC is always infinity except when DONE was received (in which case it evaluates to 0). The output function returns INIT: $(\mathtt{nil}, \mathtt{id}, \mathtt{id}, 0)$ when the simulation starts and $\star$: $(id, \mathtt{children}, T)$ when DONE was received.

## 3.2 Communication Between Simulators

Each simulation entity runs on a machine. This is modeled by a channel from the simulator to the machine being active, determined by the non-zero dimension of activePorts. There can be at most one active channel per simulator at a time.

The Local Coupling Table (LCT) holds a table mapping each simulator running on the machine to a unique port. When it receives an event, the latter is forwarded to the appropriate port of the target after some time delay. The delay time for local search is sampled from a parameterized uniform distribution (typically in the order of milliseconds). However, if the target is not in the local table, it is forwarded to the send_remote out-port. The events received are stored in a queue to handle concurrency (reception of remote and local events). In order to ensure sequential execution of simulators on the same machine, the LCT waits for a call-back from the simulator currently processing before the next event in the queue is sent. Since ASs always send a DONE message after the reception of a simulation message, the LCT expects such a message before sending the next event. As for COs, they only send a DONE message after the reception of a simulation message that induces activeChildren to be empty. Therefore the output function of the CO is extended to send a RETURN message after a simulation message is received. An LCT models the intra-machine communication of simulators.

The Remote Coupling Table (RCT) has a similar behavior to the LCT. Additionally, it holds a table mapping each simulator in the cluster to the machine it is running on. The parameterized delay time is typically longer than for an LCT, taking in consideration network communication delays (typically in the order of tens to hundreds of milliseconds). However, the event queue does not depend on call-backs. An RCT models inter-machine communication of simulators.

Machines are modeled as coupled models comprising two atomic sub-models: LCT and Activity. The state of an Activity is either *ACTIVE* or *FAILED*. The Activity model generates failures on the machine. After some time (specified in the time advance), it sends a *failure message* to the LCT. Note that it is possible to model machine replacement by allowing the Activity to send a *revival message*. When the LCT receives a failure, it is passivated (the time advance evaluates to infinity).

### 3.3 Fault-tolerance Entities

When running a simulation in a distributed environment, several fault-tolerance issues must be handled. Among them is machine failure. Gracefully recovering from such failures is important, especially for long-running simulations. The cost of re-starting these after a machine failure may be prohibitive and hence, the simulation should be made fault-tolerant. Mechanisms such as state restoration and resource reallocation will be used. There are three major components modeled as atomic DEVS models that ensure fault detection and correct restoration: the Monitor, the Log, and the Master servers.

The Monitor server monitors each machine to detect failures. At regular time intervals, it pings all the machines through its ping out-port. The Activity then receives this request from activity_req. After some small delay, it sends back an acknowledgement via its notify out-port. Note that if the state of the Activity is *FAILED*, then no acknowledgement will be output. The monitor accumulates all responses within a certain timeout. It continues pinging (at the regular frequency) as long as it receives responses from all the machines (from its alive in-port). However, if a timeout is reached before receiving responses from all machines, the Monitor considers the machines that have not responded yet as having failed. It subsequently notifies the Master model.

The Log server receives the log messages from the AS, CO, and RC entities through its log_in in-port. The log message of a simulator, identified by id, is LOG: $(\mathtt{id}, m, s)$, where $m$ is the last simulation message received and $s$ is the resulting state after $\delta_{ext}$ or $\delta_{int}$ is applied. At the level of the Log model, a cleanup mechanism removes unnecessary traces from the log. Whenever a third entry from the same simulator is received, the external transition function of Log removes the first one. This is sufficient since state restoration is only applied from the latest previous state.

The Master server coordinates the whole environment. Its state holds the simulation hierarchy (coupling of all the AS, CO, and RC models) and the resource allocation (which simulator is currently running on which machine). Before the simulation starts, the master sends an INIT message to all the simulators. Recall that this message provides knowledge of the parent of the simulator, the machine it will be running on, its children (in case of a CO or a RC), and the initial simulation time (for the RC). The Statechart in Figure 3 expresses the behavior of the Master. After initialization, the Master sends the control message $\chi = \mathtt{RESUME}$. Subsequently, the RC sends a $\star$ message to its children and the simulation runs as long as the termination condition is not yet satisfied. When it receives a failure notification (from Monitor), the Master first sends $\chi = \mathtt{PAUSE}$ to all the simulators to halt the simulation. It also requests from Log, the last saved state of the simulators formerly running on the failed machines. We call a simulator (AS, CO, or RC) failed if it is allocated to a failed machine (*i.e.,* the Activity is in *FAILED* state). In the mean time, the Master repartitions the simulators. The output function sends the appropriate $\rho$ message to the failed simulators. Note that the repartition may also need to reallocate simulators that were running on non-failed machines. It notifies the RCT and as well as each machine about the new allocation of resources. Upon receiving the log entries from Log, the Master then sends an INIT message to the failed simulators in order to restore their state to the previous "safe" state. Finally, it sends $\chi = \mathtt{RESUME}$ to all the simulators to continue the simulation from their current (or newly modified) state.

We have modeled the Master, Log, and Monitor components as three different servers. It is possible to consider them as one single server. Note that the Master could even be modularly split up further. This is a design consideration.

### 3.4 Generic Instantiation and Parametrization

The above model of a distributed DEVS simulator was implemented in pythonDEVS. To be able to simulate this model, several simulation experiments are provided as a library. It instantiates the Cluster coupled DEVS model which, in turn, creates the necessary ASs and COs according to the given host DEVS model. The necessary in-ports, out-ports, and channels are created. For experimental purposes, the Cluster system expects several parameters: The initial host model $M$; the number of machines in the cluster $n$; the initial partition of resources specifying the node location of every simulation entity; the initial states of all the AS, CO, and RC models, referred to by $s_0$ (the start time of the simulation can be specified in the initial state of RC); the termination condition $\theta$ of the simulation, specific to $M$; the distribution of the delay $\Delta_{LCT}$ for LCT to respond; the distribution of the delay $\Delta_{RCT}$ for RCT to respond; the distribution of the delay $\Delta_{LOG}$ for Master-Log communication (typically very fast); the distribution of the delay $\Delta_{MON}$ for Master-Monitor communication (typically very fast); the ping frequency $p$ of the Monitor; and the distribution of the delay $\Delta_{ACT}$ for the Activity to notify that machine has not failed (typically $\Delta_{ACT} \approx \Delta_{LCT}$).

We carried out an experiment where the only varied parameter is the time before the Monitor times-out. The simulation collects performance results for different timeout values. Performance can be measured, for example, by the number of log entries in the Log server since every simulation operation of AS, CO, and RC is logged with a fixed frequency.

The termination condition for the simulation experiments of the modeled DEVS simulator is satisfied when either $\theta$ is satisfied at the RC level or if all Activity models are in *FAILED* state.

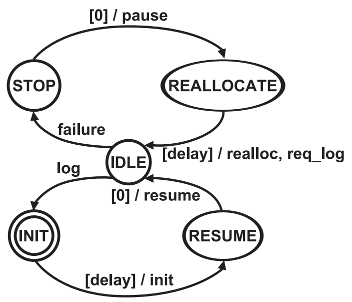


Figure 3: Modal behavior of the Master

## 4 A DISTRIBUTED DEVS SIMULATOR

We have chosen the RMI (Remote Method Invocation) as a middleware layer for our implementation for several reasons. For our purpose of implementing a sequential classical DEVS simulation protocol, RMI simplifies distributed computing, through the transparency it provides over remote procedure calls. It also hides all internal implementations of the lower level communication to maintain remote references locally. Given the nature of our particular simulations, any type of object can be sent between the solver objects as an event. This flexibility in object types would be much more tedious to implement using primitive TCP. In the following, we briefly describe the concrete realization of the distributed simulator. We use "Python Remote Objects" (PyRO), a RMI-based middleware solution similar to Java RMI. PyRO is implemented in Python, which makes it compatible with our `pythonDEVS` sequential DEVS simulator implementation of which we re-use parts.

### 4.1 Simulation Tracing - Log Server

As described previously, the DEVS simulation is meant to produce a trace which describes events occurring at certain times, as well as the way they are handled. This trace can be produced from all solver objects executing on different machines. To aggregate these individual traces into a single simulation trace, a *Log server* is used. The Log server also captures the communication between solvers. It is used to calculate the network delay variables (LCT, RCT) by logging the time at which messages were sent and received. This communication trace allows for estimation of the network and the framework overheads. The log server also stores the most recent states of the solvers for fault tolerance purposes.

### 4.2 Instantiation

To run the simulation, the initial step is to start a name server on one of the machines on the cluster. Then, a server containing a factory object (solver factory) which can host solver objects is instantiated on each of the cluster machines. It will subsequently register itself with the name server. Through the name server, the solver factory can be discovered by the simulation engine clients. These clients instantiate solver objects, atomic or coupled, destined to live on the factory object's machine.

A remote reference is created for each factory and is used to create the solver objects. These solver objects are passed the log server reference so they can send their traces to it. The simulator has been implemented to accept a mapping object, describing which machine in the cluster a solver should be running on. The simulator can then instantiate the solvers according to the partitioning mapping locations. If no specific mapping was provided, they are instantiated locally. Figure 4 illustrates the overall architecture of the implementation.

### 4.3 Simulation Protocol

The simulation protocol was implemented in an asynchronous fashion. This allows for better performance of the overall simulation. For example, when an event is destined to multiple solvers, it is broadcast in an asynchronous fashion. In turn, the receiving solvers can receive and process the event simultaneously, and then through a callback respond asynchronously.

## 5 CALIBRATION AND OPTIMIZATION

The simulation experiments we perform use an input model $M$. For this paper, we use an abstracted model of traffic in a city. A city (coupled DEVS) is divided into several districts (coupled DEVS). A district encapsulates a road network with houses and offices (both atomic DEVS). Roads (coupled DEVS) are bidirectional and thus consist of two one-way road segments (atomic DEVS). In the road network of a district, roads can connect to an intersection (coupled DEVS composed of eight road segments) at specific points, with or without a traffic light (atomic DEVS). Some districts communicate via highways (coupled DEVS) modeled as sequences of roads. At periodic intervals, houses generate cars to go to a predefined office. Note that a house and its corresponding office can be in different districts. After some random time interval, a car leaves the office and returns back home. The simulation ends when all cars are back home or are involved in a car accident.

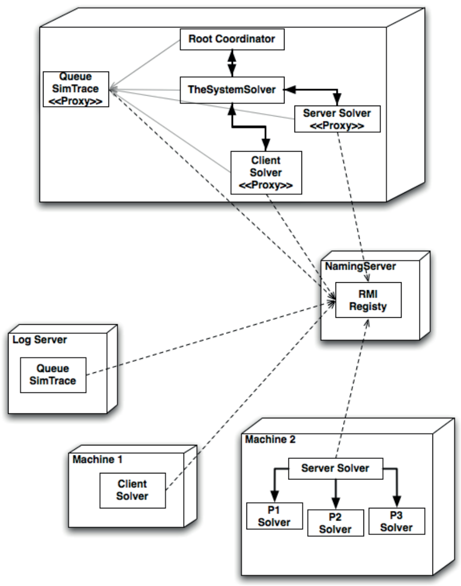


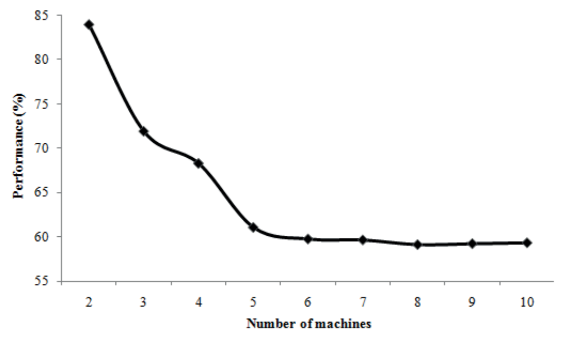
Figure 4: The RMI architecture of a distributed DEVS simulator using PyRO.

The DEVS simulator used to simulate model $M$ is itself modeled as a DEVS model as described in Section 3. The simulator model was calibrated with model execution parameters from the PyRO-based distributed simulator described previously.

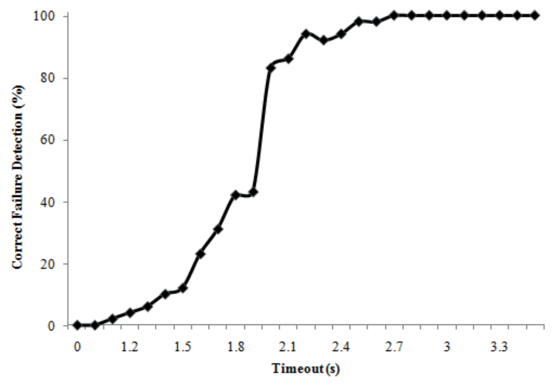
### 5.1 Optimization of Performance Metrics

One of the most important configuration questions in the distributed implementation, with regards to efficient fault-tolerance, is the timeout to set on solver calls, before assuming that a machine has crashed or has a fault. This timeout really depends on the model being simulated and the partitioning that is used for the solvers. The timeout value might also be different at different solver levels. Setting an arbitrary constant value may not be sufficient, as it has to take into account the network message passing time and other specific properties.

To accomplish a realistic simulation, several parameters need to be calculated from the middleware and the cluster. Network statistics for a specific cluster can be gathered using the current implementation of the log server. As discussed, the log server keeps track of both the simulation and communication traces. Communication trace analysis allows one to estimate remote and local delays for message passing between different solvers. Each solver outputs a communication trace both before sending a message to another



(a)



(b)

Figure 5: (a) Effect of adding machines; (b) master reaction delay on correct detection of machine failure.

solver and when it receives one. Traces are augmented with extra information to allow calculating latency and are classified as local or remote messages. For example, the trace message produced at each solver is augmented with the following parameters:

- the name of the model in the solver,
- the global current time at which this trace was produced,
- the local time at the solver machine when the message was received,
- the local time at the machine when the trace was produced.

After several experiments on the cluster simulating the city example, we analysed the trace information to calculate the distribution of remote and local delays. These values were then used as parameters in the modeled simulator to simulate message passing delays.

Figure 5(a) illustrates the performance behavior of the simulator depending on the number of machines used. In our experiments, the city model has five districts and 10 bridges, totaling 1000 coupled DEVS models and 10,000 atomic DEVS models. For our example, the partitioning of the city models on the different available machines adheres to one constraint: the coupled DEVS representing a district and all its sub-models are always on the same machine. Performance was measured by taking the total simulation time of a complete simulation experiment. The graph shows a decrease in performance when the number of machines used increases. This is because the network communication between components adds an overhead. The graph hence shows that, for our model, the optimal number of machines is two. (The case where there is only one machine is ignored since it is not distributed). Our goal is anyhow not to increase performance but rather maximize interoperability using specialized nodes. For example, in some cases parts of the partition would be fixed. Then, such analysis becomes more valuable. Furthermore, the performance levels off when more than six machines are used. This is due to the limited number of components in the modeled simulator.

The graph in Figure 5(b) allows one to find the optimal timeout for the given configuration. In this case the minimal timeout with $100\%$ reliable fault detection is 2.7 seconds. This is significantly less than the monitor frequency which is 7 seconds.

## 6 RELATED WORK

The DEVS formalism and its simulation protocol as described in Section 1 are well suited for local execution on a single machine. In addition, many approaches for distributing DEVS simulation have been proposed and implemented (*e.g.,* (Seo, Park, Byounguk, Cheon, and Zeigler 2004, Cheon, Seo, Park, and Zeigler 2004, Zhang, Zeigler, and Hammonds 2006, Sunwoo, Kim, Hunt, and Park 2007)). Thanks to the modularity and hierarchical structure of DEVS, distributing a DEVS model execution on several processors can be achieved without modifying the simulation models.

The distributed architecture for DEVS can be divided into layers. The *application layer* (highest level of the system) is the modeling and simulation problem under study. The DEVS model lies in the *modeling layer*. At the *simulation layer*, the protocol to simulate the DEVS model is implemented. The lowest level layer is the *middleware layer* used to implement the communication between compute nodes.

James II (Uhrmacher, Röhl, and Himmelspach 2003) is a dynamic simulation framework where a model of a DEVS simulator can be simulated. The purpose of this is to allow enhancing the DEVS formalism with dynamic restructuring capabilities. That is, the simulation model (composed of atomic solvers and coordinators) is modified at run-time to, for example, re-partition the model distributed on several machines.

## 6.1 DEVS/Grid

As introduced in (Seo, Park, Byounguk, Cheon, and Zeigler 2004), DEVS/Grid makes use of the distributed computing paradigm: grid computing. Together with the simulator, the simulation layer comprises a model partitioner, deployer, and activator. The partitioner divides the DEVS model in partitions containing at least one DEVS block. Partitioning is computed based on the construction of a cost tree using a hierarchical reconstruction of the states of coupled models. These partitions are dispatch by the deployer to an activator on each host resolved by Grid middleware services. The message received consists of the activator's id, the blocks in the partition and the coupling information (between different partitions). The activator then creates a simulator for each DEVS block (atomic or coupled) and starts it. At that time, the necessary communication channels are automatically created based on the coupling information (how the ports are connected).

Simulators communicate at the middleware layer. A DEVS message is sent from the simulator to a proxy which encodes the message as a “Globus message”. Messages are exchanged via sockets following the Globus protocol (widely used in grid computing).

The DEVS/Grid simulation protocol does not use coordinators, in contrast with the conventional abstract simulator as defined in (Zeigler 1984). Coordinators have two roles (in parallel DEVS). One is to maintain the coupling relations between ports, but this is taken care of by the grid activators, statically and dynamically. The other role of a coordinator is to synchronize simulation time and control simulation behavior of its (inner-)solvers. For that, the DEVS/Grid protocol distinguishes protocol messages from user messages. The protocol message $TA$ is for publishing the next event time $t_N$ of a DEVS model, while $done$ is used for expressing the termination of a simulator. User messages are those generated by DEVS models.

The simulation protocol works as follows. After publishing a $TA$ message, a simulator is blocked until it receives all $TA$ messages from the other simulators. It advances its simulation time by the minimum $t_N$ received. Thereafter, output is computed (if needed) and the simulator delivers the corresponding output message to the linked models via the communication layer. Upon receiving DEVS messages, each simulator publishes a $done$ message and performs its delta transition function concurrently depending on the message (event) received. It then computes its new $t_N$ and the simulation cycle starts again.

## 6.2 DEVS/RMI

DEVS/RMI (Zhang, Zeigler, and Hammonds 2006) combines `DEVSJAVA` (the implementation of a DEVS modeling and simulation framework, like `pythonDEVS`) with Java remote object technology Java/RMI.

In DEVS/RMI the DEVS model is extended with three additional key components (that can be modeled as DEVS blocks): the *configurator* for model partitioning, the *controller* for simulation control, and the *monitor* for failure detection and status verification of hosts. The configurator analyses the model received to then apply a partition / repartition algorithm. Partitioning can be done statically by assigning model partitions to remote locations during initialization phase. Moreover, dynamic partitioning is performed at run-time, passing models as parameters to remote machines. When a new partition plan needs to be computed, the controller stops the simulation until the partition information is received from the configurator. Thereafter, the controller re-configures the simulation environment and continues the simulation. Since Java/RMI supports persistent object migration natively, migrated or newly created simulators need not restart the simulation from time zero; the simulation continues seamlessly. The monitor collects information about each running model in the network, measures their activities and sends that information to the configurator.

The simulation algorithm remains the same as the conventional one. However remote interfaces must be extended (using proxies) in order to communicate via Java/RMI and pass serialized messages. It is important to note that simulators may be created remotely or locally, depending on what is specified in the (sub-)model "where" parameter.

## 6.3 DEVS P2P

DEVS P2P allows DEVS models to be simulated on a peer-to-peer network (Cheon, Seo, Park, and Zeigler 2004). Each DEVS block is coupled in a coupled block with "virtual" DEVS ports correctly routed according to the corresponding source/target coupling. The channels are mapped to pipes over the network.

The middleware used is JXTA. Consequently, a mapping from the virtual DEVS channels to JXTA pipes needs to be processed as described in (Cheon, Seo, Park, and Zeigler 2004). DEVS P2P supports the same hierarchical partitioning and restructuring as implemented in DEVS/Grid. In DEVS P2P peers are wrappers for their local simulator (atomic or coupled). Each simulator is assigned to the active partitions of the model only, which decreases the number of simulation processes and control messages. At run-time, when a new simulator/peer is created, the new pipe is published to the network. Then, when a source pipe needs to communicate, it advertises its discovery pipe and only then the I/O communication can be established.

The peer, solver and coupled simulator protocols are described in (Sunwoo, Kim, Hunt, and Park 2007). The peer algorithm simply serves as an interface for communication between other peers over the network, wrapping message I/O handling. The solver protocol remains almost unchanged with the difference that message I/O is sent or received via the peer it is located on. The coupled simulator protocol however, handles the coupling relation between DEVS blocks internally. The simulation time synchronization is also managed at the coupled level. The coupled simulator also processes message I/O via the peer. In the case of a remote sub-model, it is the peer that communicates via the network. All messages peers exchange are serialized to JXTA messages in XML format.

When a peer delivers a $TA$ or $done$ message, it is published to all connected peers. Given that it is in JXTA message format, the identifier of the peer is also passed and the JXTA layer will ensure communication with the specified peer. Reconfiguration and load balancing is managed in that layer as well. Caching is used to decrease network communication overhead.

## 6.4 Discussions

Distributed DEVS is achieved in layers. As shown in the previous section, the three approaches differ at the *middleware layer*, but also at the *simulation layer*.

DEVS P2P strongly relies on JXTA Pipe Interface as middleware to support distributed execution of DEVS. DEVS/ Grid relies on a grid infrastructure "Globus" for its communication layer. The most neutral approach is DEVS/RMI which uses the RMI technology (in this case Java/RMI). The advantage of using RMI is the portability of the distributed simulation framework. Because RMI supports persistent data migration on remote machines, DEVS/RMI natively supports dynamic model structure change. In DEVS/Grid, the grid layer requires explicit specification when migrating the state of a simulator. As for DEVS P2P, this is handled through JXTA services. Additionally, each block is coupled with a "virtual" block to communicate between peers. Similarly, Grid resolves simulator mapping on computing nodes

Table 1: Comparing different distributed DEVS approaches.

| | **DEVS/Grid** | **DEVS P2P** | **DEVS/RMI** | **DEVS/PyRO** | **Simulator Model** |
|---|---|---|---|---|---|
| **Middleware** | Globus | JXTA | Neutral: RMI | PyRO/RMI | DEVS |
| **Portability** | No | No | Yes | Yes | Yes |
| **Dynamic re-configuration** | Grid computa-tion | JXTA message ser-vice | JAVA/RMI | PYRO/RMI | DEVS Component (Master) |
| **Mapping to node** | Resolved by grid | Extend each block to communicate with peer | Host IP passed as parameter for each block | Machine Name on Cluster is passed for each block | DEVS Component (RCT) |
| **Partitioning** | GMP | AHMP | Any | Any | Any |
| **Message exchange** | GIIS | JXTA message for-mat | Serializable object | Pickled Objects | DEVS Event |
| **Simulation protocol** | Explicit message delivery | Modified to com-municate through peer | Unchanged, inter-facing with RMI technology | Unchanged, RMI Based | DEVS Protocol |

by the activator. In DEVS/RMI, the location of the simulator is specified statically as a parameter in the block. This can then be changed by the controller at run-time. As far as model partitioning is concerned, DEVS/Grid uses the Generic Model Partitioning Algorithm (GMP) to a cost tree that it constructs. DEVS P2P uses a more cost-efficient algorithm, the Autonomous Hierarchical Model Partitioner (AHMP). DEVS P2P and DEVS/Grid use a message exchange format respective to their middleware constraint, while RMI supports serialized objects to be exchanged. DEVS P2P simulation protocol forces all none I/O messages are exchanged locally between the simulator and its peer. In the DEVS/Grid environment, the simulation protocol must explicitly deal with delivering messages to remote simulators. As for DEVS/RMI, the simulation protocol must implement the RMI interface, without changing the conventional protocol. DEVS P2P and DEVS/Grid need an additional layer for simulation time management. These comparisons are summarized in Table 1.

We have added our modeled distributed DEVS simulator and its realization in the form of the DEVS/PyRO implementation to this comparison in the last two columns of the table.

## 7 CONCLUSION

In this article, we have explicitly modeled the structure and behavior of a distributed DEVS simulator, as a DEVS model. From this DEVS model, a distributed DEVS simulator was realized (*i.e.,* partially synthesized). This simulator runs over RMI using *PyRO*. The actual performance data obtained from this implementation (simulating the model of traffic in a synthetic city) was used to obtain realistic parameter values to be used in the DEVS model of the simulator. Isolating deployment platform parameters, such as timeouts, optimal values were found by simulating multiple alternative models of the DEVS simulator. These variables were finally used to calibrate the real simulator.

In the future, we want to completely automate the synthesis of simulators from their DEVS models. Also, we plan to synthesize DEVS/RMI rather than the rather inefficient PyRO.

## ACKNOWLEDGMENTS

The authors acknowledge the Natural Sciences and Engineering Research Council (NSERC) of Canada for partial support of this work.

## References

Bolduc, J.-S., and H. Vangheluwe. 2001, June. “The Modelling and Simulation Package `pythonDEVS` for Classical Hierarchical DEVS”. MSDL technical report MSDL-TR-2001–01, McGill University.

Cheon, S., C. Seo, S. Park, and B. P. Zeigler. 2004, April. “Design and Implementation of Distributed DEVS Simulation in a Peer to Peer Network System”. In *ASTC’04*, edited by H. Unger, 18–22. Arlington, USA: SCS.

Chow, A. C.-H., and B. P. Zeigler. 1996. “Parallel DEVS: a Parallel, Hierarchical, Modular Modeling Formalism”. *TSCS* 13:55–67.

Filippi, J.-B., and P. Bisgambiglia. 2004. “JDEVS: an implementation of a DEVS based formal framework for environmental modelling”. *Environmental Modelling and Software* 19 (3): 261–274.

Kim, K. H., Y. R. Seong, T. G. Kim, and K. H. Park. 1996. “Distributed Simulation of Hierarchical DEVS Models: Hierarchical Scheduling Locally and Time Warp Globally”. *TSCS* 13 (3): 135 – 154.

Lee, J.-K., Y.-H. Lim, and S.-D. Chi. 2004, February. “Hierarchical Modeling and Simulation Environment for Intelligent Transportation Systems”. *Simulation* 80 (2): 61–76.

Seo, C., S. Park, K. Byounguk, S. Cheon, and B. P. Zeigler. 2004, April. “Implementation of Distributed high-performance DEVS Simulation Framework in the Grid Computing Environment”. In *ASTC’04*, edited by H. Unger. Arlington, USA: SCS.

Sunwoo, P., S. H. J. Kim, C. A. Hunt, and D. Park. 2007, November. “DEVS Peer-to-Peer Protocol for Distributed and Parallel Simulation of Hierarchical and Decomposable DEVS Models”. In *ISITC’07*, edited by H. Kim, S.-S. Song, M. Strzelecki, J. Choi, D.-U. An, and S. Kim, 91–95. Jeonju, Korea: IEEE Computer Society.

Syriani, E., and H. Vangheluwe. 2008, July. “Programmed Graph Rewriting with Time for Simulation-Based Design”. In *ICMT 2008*, edited by A. Vallecillo, J. Gray, and A. Pierantonio, Volume 5063 of *LNCS*, 91–106. Zürich, Switzerland: Springer.

Uhrmacher, A. M., M. Röhl, and J. Himmelspach. 2003, October. “Unpaced and Paced Simulation for Testing Agents”. In *15th European Simulation Symposium*, edited by A. Verbraeck and V. Hlupic, 71–80. Delft, The Netherlands: SCS.

Xie, H., A. Boukerche, M. Zhang, and B. P. Zeigler. 2008. “Design of A QoS-Aware Service Composition and Management System in Peer-to-Peer Network Aided by DEVS”. In *DS-RT*, edited by D. Roberts, A. E. Saddik, and A. Ferscha, 285–291.

Zeigler, B. P. 1984. *Multifacetted Modelling and Discrete Event Simulation*. Academic Press.

Zhang, M., B. P. Zeigler, and P. Hammonds. 2006. “DEVS/RMI-An Auto-Adaptive and Reconfigurable Distributed Simulation Environment for Engineering Studies”. *Journal of Test and Evaluation* 27 (1): 49–60.

## AUTHOR BIOGRAPHIES

**EUGENE SYRIANI** is an Assistant Professor in the Department of Computer Science at the University of Alabama. He holds a Ph.D in CS and a B.Sc. in Math and CS from McGill University. His current research interests are model transformation, model-driven engineering methodology, and simulation-based design. His email address is esyria@cs.mcgill.ca.

**HANS VANGHELUWE** is a Professor in the department of Mathematics and Computer Science at Antwerp University, Belgium, an Associate Professor in the School of Computer Science at McGill University, Montréal, Canada. He holds a D.Sc. degree, M.Sc. degrees in Computer Science and in Theoretical Physicsfrom Ghent University in Belgium. He is Associate Editor of the International Journal of Critical Computer-Based Systems, of Simulation: Transactions of the Society for Computer Simulation, and of the International Journal of Adaptive, Resilient and Autonomic Systems. His email address is hv@cs.mcgill.ca.

**AMR AL MALLAH** is a Software Development Engineer at Microsoft. He holds a B.Sc. and M.Sc. in Computer Science from McGill University, Montréal, Canada. His main research interest is in model-based testing of model transformations. His email address is amr.almallah@mail.mcgill.ca.