Studying performance of DEVS modeling and simulation environments using the DEVStone benchmark

Gabriel Wainer, Ezequiel Glinsky and Marcelo Gutierrez-Alcaraz

SIMULATION 2011 87: 555 originally published online 20 January 2011
DOI: 10.1177/0037549710395649

Simulation

Simulation: Transactions of the Society for Modeling and Simulation International
87(7) 555–580

DOI: 10.1177/0037549710395649
sim.sagepub.com

Studying performance of DEVS modeling and simulation environments using the DEVStone benchmark

Gabriel Wainer, Ezequiel Glinsky and Marcelo Gutierrez-Alcaraz

Abstract

The Discrete Event System Specification (DEVS) formal modeling and simulation (M&S) framework (which supports hierarchical and modular model composition) has been widely used to understand, analyze and develop a variety of systems. Numerous DEVS simulators have been developed; nevertheless, evaluating the performance of such simulators is a complex process and it has been usually done using ad hoc methods. DEVStone, instead, is a synthetic benchmark that can be used to automate the evaluation of the performance of DEVS-based simulators. DEVStone generates a suite of models with varied structure and behavior automatically. To do so, it uses a standardized mechanism that can be the basis for comparisons between simulation software tools. As a proof of the concept, we present various tests in which DEVStone was used to study the efficiency of five different simulation engines. In this case, we compared various versions of the CD++ simulator, and then compared its performance with the 'A Discrete Event System Simulator' (ADEVS) M&S tool. This is the first effort in which these simulation tools have been thoroughly compared with a very demanding set of experiments. The use of DEVStone allowed a standardized and exhaustive method to compare different features of the simulation software. We show how the basic ideas used for DEVStone facilitates performance analysis for upgrades and updates of a given simulation engine, while also providing a common metric to compare different M&S environments.

Keywords

DEVS, modeling and simulation tools, simulator performance evaluation, synthetic benchmarks

1. Introduction

In recent years, various modeling and simulation (M&S) methodologies have provided well-developed, well-proven approaches to problem solving. In particular, one of the M&S techniques that has gained popularity is Discrete Event System Specification (DEVS): a sound, formal framework based on generic dynamic systems theoretical concepts, which supports provably correct, efficient, event-based simulation.[1]

A DEVS model is described as a composite of submodels that can be behavioral (atomic) or structural (coupled), which can be interconnected in a hierarchical, modular fashion. DEVS has been successful for modeling a wide range of application domains. For instance, it has been used for the analysis of multi-agent robots,[2] fire spreading[3–5] and other large ecosystems,[6] prototyping and testing embedded system designs,[7,8] urban traffic analysis,[9,10] decision support systems, logistics and supply chain,[11–13] computer architecture,[14,15] and many others. DEVS was successfully applied in such a variety of applications due to the ease of model definition, improved composition and reuse, and hierarchical coupling. DEVS includes explicit specification of the model timing, and uses a discrete event approach for simulation.

Department of Systems and Computer Engineering, Carleton University, Ottawa, ON, Canada.

Corresponding author:
Gabriel Wainer, Department of Systems and Computer Engineering, Carleton University, 4456 Mackenzie Building, 1125 Colonel By Drive, Ottawa, ON, Canada K1S 5B6
Email: gwainer@sce.carleton.ca

This provides precision and speedups in the execution time, as models advance triggered by instantaneous asynchronous events in contraposition with time-stepped approaches,[16] and a variety of DEVS tools are now available (see http://www.sce.carleton.ca/faculty/wainer/standard/tools.htm).

Despite this variety of applications and the performance gains reported by DEVS simulators research, there is a common misconception that the hierarchical nature of DEVS models may degrade the efficiency of model execution. The arguments in this sense are primarily anecdotal and purely based on opinions, as there is no scientific proof or any thorough studies to give evidence on the truth of those arguments. This is exacerbated by the fact that existing performance studies only focus on executing test scenarios for a particular application domain.

Likewise, the traditional method to study the performance simulation tools is through these ad hoc simulations, which are usually focused on a given domain (which, in general, is of interest for the particular tool being studied). All of these studies are skewed by the application of choice, and no general conclusions can be obtained based on them (in particular, because in many cases the simulation software is already adapted to the particular simulation application). These performance results are usually very meaningful for the application domain, but one cannot use them to assess the overall qualities of a given solution. More importantly, such results cannot be applied to generic M&S tools (such as all DEVS environments, or most existing M&S software), in which multiple applications for different could be developed by different users. If we want to give a general recommendation about a given simulation environment, we cannot use the results obtained for a particular application.

Another problem with the current methods for studying the performance of M&S tools is that most of them require constant modifications and upgrades. Nevertheless, there is no standardized mechanism to assess the performance of the environment when the software is upgraded, or to compare different versions of a particular engine. If a new release of a simulation tool is built, how do we compare it with the previously existing versions? In general, this is also done through ad hoc testing, but there are no well-established mechanisms for regression performance testing.

We decided to attack these open issues by proposing a new synthetic benchmark that can be used to analyze the performance of DEVS simulators thoroughly. The benchmark, called DEVStone,[17] is a synthetic model generator whose accuracy relies on the execution of a large pool of models with varied options. The main advantage of DEVStone is its flexibility and configurability. The DEVStone framework proposes to generate a set of synthetic models, in which the main difference is their size, complexity and computational cost. The synthetic generator focuses heavily on the structural aspects of the simulation environment. DEVStone can be adapted easily, and the idea is to be able to replicate the structure of a wide variety of application categories, which, in turn, could be used for studying different domains of application. This variety provides a robust test set and uniform means for obtaining metrics, making it possible to analyze the efficiency of a simulation engine with relation to the characteristics of a category of models of interest.

In order to proof the concept, DEVStone was implemented on CD++[18] and 'A Discrete Event System Simulator' (ADEVS),[19] two tools that implement DEVS theory. We show how the benchmark can be used to run exhaustive tests, and in this case, we discuss the results obtained when analyzing the overhead of hierarchical DEVS simulators (comparing it with non-hierarchical algorithms). We also discuss how the benchmark can be employed to assess how performance is affected when new releases of a given simulation software are issued. In this case, we use varied versions of CD++, which has been revised and extended several times (including standalone,[18] parallel,[4,20] and real-time engines[21]). Before DEVStone, CD++ performance tests were done as in other simulation environments, i.e. using ad hoc applications in different domains: urban traffic;[10,22] complex physical systems;[23,24] and different ecological, artificial, and biological systems.[6,24] Although those results gave a good idea about the performance for the given domain, it is obvious that they cannot be used as a generic measure for the overall performance of the tool (algorithms that worked very well for urban traffic did not work for biological systems, etc.). Finally, we show some performance results obtained when comparing CD++ and ADEVS. This is the first thorough comparison between two DEVS simulation engines (and DEVStone allowed us to carry out this comparison using an exhaustive method, instead of simply comparing particular simulation examples). Although most of the analysis has been done using the CD++ and ADEVS tools (which are the ones we have chosen and have available), the ideas can be applied to any generic simulation software (and the implementation is straightforward for any modular M&S framework), as discussed in the following sections.

We describe the synthetic generator, and then we show how it can be used to analyze the performance of different simulation engines (in particular, ADEVS and various versions of CD++), in order to show the feasibility of the approach. The goal is to discuss the proposed method and its advantages, and to show how different tests can be easily automated according to the

needs of the user. The examples in this paper show a subset of the numerous tests that can be performed with DEVStone, discussing some interesting results we found in the process of checking our proposal. The performance analysis is not the main goal of this article, but the results obtained also permitted us to meet our first goal, which is to characterize the execution time of DEVS simulators, clarifying the misconceptions about the performance of hierarchical DEVS. The results show that hierarchical models execute with high performance, providing empirical evidence about this fact. We also show how to apply the benchmarking tool to test the performance of upgraded simulators, demonstrating that the benchmark can be used to determine which directions and decisions should be taken when updating a simulator. Furthermore, we want to show how DEVStone can be used to measure and improve other existing DEVS simulation tools. DEVStone can thus be use as a 'standard' simulation metrics against which other simulators can be compared; by selecting the right subset of models this can be done without difficulties.

2. Background

DEVS is a formal M&S framework based on dynamic systems theory concepts.[1] DEVS is a mathematical formalism with well-defined concepts of hierarchical and modular model construction, coupling of components, and support for discrete event model approximation of continuous systems. A significant advantage of DEVS is the explicit separation of concerns between the M&S mechanisms, which enables independent verification of each layer.

The hierarchical nature of DEVS allows existing models to be coupled in a modular fashion in order to build larger systems. Since the formalism is closed under coupling, a coupled model can be treated as a basic DEVS component, enabling reuse of previously tested components. DEVS modular specifications view a model as having input and output ports through which every interaction occurs. The DEVS modeler combines the outputs of one model with the inputs of another, merging different simulation components and connecting the system with experimental data.

An atomic DEVS model is formally described as follows:

$$M = \langle X,\ S,\ Y, \delta_{\text{int}},\ \delta_{\text{ext}}, \lambda,\ ta \rangle$$

where X is the set of input events, S is the state set, Y is the set of output events, δ_{int} is the internal transition function, δ_{ext} is the external transition function, λ is the output function, and ta is the time advance function. At any time, the system is in some state $s \in S$. In the absence of external events, the system stays in state s for the time specified by $ta(s)$, which can be any real value in the interval $[0,\infty)$. When that time is consumed, the system outputs the value $\lambda(s)$ and changes immediately to the state $s' = \delta_{\text{int}}(s)$. If an external event $x \in X$ is received before the expiration time $ta(s)$, the new state s' of the system is determined by $\delta_{\text{ext}}(s, e, x)$, where e is the time elapsed since the last transition. The model remains in the new state s' until the time $ta(s')$ is expired or an external event is received.

A DEVS coupled model, composed of several atomic or coupled submodels, is formally described as

$$\text{CM} = \langle X,\ Y,\ D,\ \{M_i\},\ \{I_i\},\ \{Z_{ij}\} \rangle$$

where X is the set of input events, Y is the set of output events, D is an index for the components of the coupled model, and for all $i \in D$, M_i is a basic DEVS (i.e. an atomic or coupled model), I_i is the set of influencees of model i (i.e. models that can be influenced by outputs of model i), and for all $j \in I_i$, Z_{ij} is the i to j translation function. Coupled models are defined as a set of basic components (atomic or coupled) interconnected through the models' interfaces. The influencees of a model define to which model outputs must be sent. The translation function converts the outputs of a model into inputs for other models. To do so, an index of influencees is created for each model (I_i). This index defines that the outputs of the model M_i are connected to inputs in the model M_j, where j is an element of I_i.

These specifications, further discussed by Zeigler et al.[1] and Wainer[25] and other relevant literature, have been used by numerous developers to create different DEVS tools. The formal aspects of DEVS permit building efficient and correct simulation software with ease and, consequently, the development of DEVS tools has been very popular in the last few years. Some of the existing tools include DEVS/SOA[26] (based on the High Level Architecture (HLA) and DEVS, allowing the building of large-scale distributed modeling and simulations), and DEVSJAVA[27] (a DEVS-based M&S environment written in Java that supports parallel execution). It also includes DEVSim++[28] (an object-oriented software to simulate DEVS models implemented as basic classes in C++ that can be extended), JAMES II[29] (a framework to create your own tools or to integrate it as a back-end into any software), JDEVS[30] (a DEVS M&S environment written in Java), and a few others. Although this is not a comprehensive list, it shows the variety of DEVS tools available.

A current effort is trying to standardize the DEVS M&S framework and related tools,[31] with the goal of

allowing these tools to interoperate. Two different interoperability objectives have been identified:

- Standardizing DEVS model representation: the question is how to build a platform-independent DEVS model representation. The specifications must be independent of computer programming languages, allowing software specifications to be derived from them. Different proposals based on XML notation have been proposed and are under study.
- Standardizing interoperability middleware: the idea is to allow for interfacing different simulation environments, providing synchronization for the same simulation run on different DEVS simulators (and, in particular, across a distributed network).

Considering these numerous efforts, the need for a common benchmark to test the multiple implementations becomes apparent. In particular, the DEVStone benchmark to be introduced in the following section provides a unified mechanism for comparing different standardized tools and environments.

As discussed earlier, we have implemented the benchmark on two existing tools: ADEVS[19] and CD++.[25] ADEVS provides a C++ library based on the Parallel DEVS[1] and Dynamic DEVS formalisms.[29,32,33] Parallel DEVS is an extension to classic DEVS that introduces better handling of simultaneous events, and Dynamic DEVS introduces the idea of dynamic structure models, which can change their transition functions, links, or complete subcomponents during runtime. Users can use the classes in the library to build their own models. The models are constructed based on a template class in C++. The engine is implemented as a library in ADEVS; therefore, it is enough to declare the library in the header of the file to start developing ADEVS-based simulations. Every atomic or coupled model in ADEVS is composed of two files:

- A library file (.h) where the name of the model, input and output ports, and local variables are defined for the particular atomic model.
- A source code file (.cpp) where the actual model is built based on a template. Common elements of the class include constructor/destructor, internal transition, external transition, time advance, and output functions.

The second tool used in the DEVStone experiments we present later is CD++, a M&S environment that implements DEVS and Cell-DEVS theory. Different CD++ simulation engines support standalone, parallel, and real-time simulation of DEVS models. The method for defining DEVS models in CD++ is different from that used for ADEVS models. CD++ includes a specification language for describing coupled models, setting parameters and external input events, and atomic models are developed in C++.

CD++ simulation engine is built following DEVS abstract simulation algorithms, and implemented as a C++ class hierarchy that corresponds to simulation entities, as defined by Zeigler et al.[1] and shown in Figure 1.

Two basic abstract classes, *Model* and *Processor*, define the behavior of the atomic and coupled models and the corresponding simulation mechanisms. The *Atomic* class implements the behavior of atomic components, whereas the *Coupled* class implements

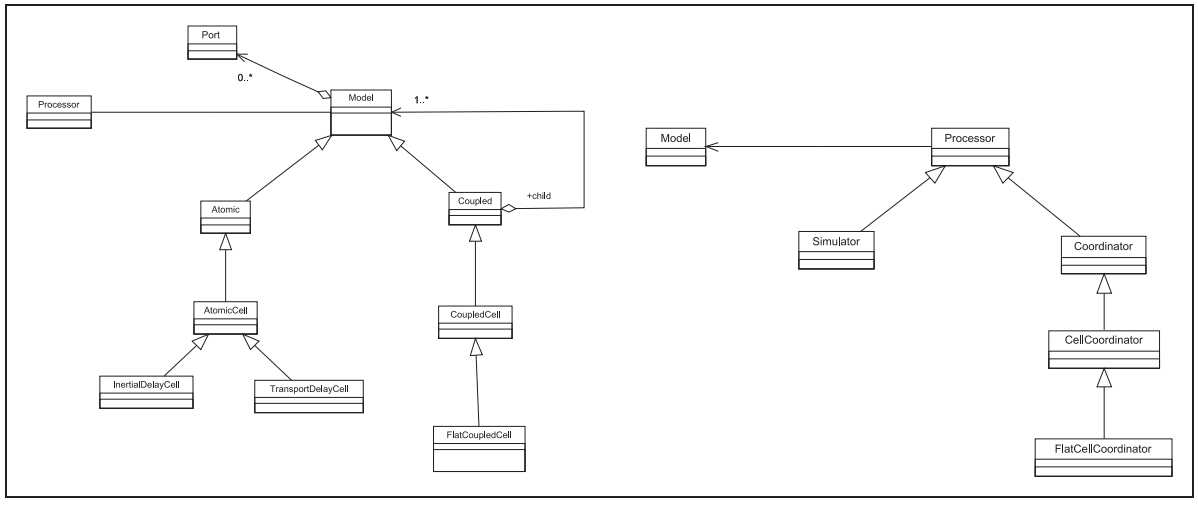

Figure 1. CD++: (a) model hierarchy; (b) processor hierarchy.

coupled models. A *Simulator* object manages an associated atomic model, handling the execution of its δ_{int}, δ_{ext} and $\lambda(s)$ functions. A *Coordinator* object manages an associated coupled object. The *Root Coordinator* manages global aspects of the simulation environment and it is involved with the topmost-coupled component. The simulation process is based on the message exchange among *Processors*. Each message contains information to identify the sender, the receiver and a timestamp. Different types of messages are exchanged between simulators and coordinators to advance the simulation process, which include synchronization and content messages.

3. DEVStone: A synthetic model generator

In the previous section, we discussed the main ideas regarding DEVS models and their abstract simulators. These simulators execute efficiently, using a discrete-event approach (advancing time according to the occurrence of events). The hierarchical structure of the simulation algorithms results in not needing a centralized event list (and the tree organization of the simulation engine organizes the data in a tree-like fashion without explicit definition of the data structure). This allows enhanced model definition and high performance; nevertheless, there is a common preconception that this hierarchical organization might degrade the efficiency of model execution. As discussed earlier, these arguments are based primarily on intuition, as there are no studies providing a means to compare the efficiency of DEVS simulators, nor comparing different versions of a particular simulation engine, and most simulators (DEVS or other) are tested by executing different scenarios for a given application domain. For instance, the IWA/COST simulation benchmark focuses on sludge wastewater treatment control strategies.[34] Another example is Knight et al.,[35] where the authors use a part of the Space and Missile Defense Battle Lab to evaluate the performance of different HLA/RTI distributed simulation software. Various tools (Verilog compilers, flight simulators, and varied multiphysics software) provide their own self-tailored benchmarks to compare with other competing software applications. In most cases, these results are meaningful for the domain, but do not provide general answers about generic M&S software tools. In the same sense, numerous previous studies have been carried out studying the performance of DEVS environments. Nevertheless, most of these studies only focused on performance results for a given tool, and were usually limited to a given type of models of interest. Some examples include Troccoli et al.,[36] where the authors presented performance studies of Cell-DEVS models in a parallel simulation environment. Amehino et al.[23] focused on a watershed model to show performance improvements in parallel and distributed architectures. DEVS was shown to be more efficient than the continuous counterparts were for simulating natural, and artificial systems, such as a photovoltaic system (a period of 3 years was simulated in less than 1 minute[30]). However, those studies do not provide a thorough analysis for the execution of models with different characteristics; neither do they give a common metric to compare results among different DEVS simulators.

The main problem in analyzing the performance of generic simulation tools is the wide variety of models that users can create using such environments. The models can have different structures, levels of complexity, and degrees of interaction, all of which affect performance. An interesting approach is provided by the ARGESIM[37] benchmark suite, which proposes a varied set of models to test a simulation software system. The benchmark suite includes a number of continuous model simulations (a third-order stiff system modeling lithium-cluster dynamics, a constrained pendulum, a fuzzy controller for a tank system, and a row of colliding spheres). It also includes discrete-event simulations (a crane crab, supply chain, a restaurant business, a flexible manufacturing system, and an emergency department simulation), logical models (dining philosophers, a two-state model), and spatial models (pollution, epidemics). Although this variety of models is extremely useful, the specifications of the models are subject to the interpretation of the modeler. Likewise, it still provides a subset of different applications, and it does not include any method to automate the creation of complex models.

The DEVStone synthetic benchmark, instead, provides a mechanism to deal with these issues. It can be used to generate automatically a wide variety of models of different shapes and sizes, which can then be used to test different features of different simulators. It has been built following a flexible reconfiguration approach, so the users can adapt it easily to test different features of their simulation tools. It can thus be used to compare all kinds of simulation engines, or different versions of a given simulation software, providing standard metrics that can be used for comparing the different tools. Although it has been created with a focus on DEVS hierarchical models, it can be adapted to be used for non-DEVS engines and non-hierarchical models. It has been proved that DEVS is the most generic discrete-event system specification, thus any existing discrete-event model can be represented as a DEVS model. Also, there have been numerous efforts showing how to translate varied M&S methods (Petri nets, finite state machines [FSM], Modelica, bond graphs, timed automata, statecharts, etc.) into DEVS models, and numerous

libraries have been built.[23] Following these ideas, one can translate the DEVStone models into the corresponding equivalent (for instance, we can eliminate the hierarchical models for testing flat M&S techniques, and can use a single flat atomic model to test those; adapting the following ideas for modular environments is straightforward).

The core idea of the benchmark is to create varied synthetic models with different structure that are representative of those found in real-world applications. To do so, DEVStone generates models with different size, structure and computational complexity, resembling different kinds of applications. Hence, it is possible to analyze the efficiency of a simulation engine with relation to the characteristics of a category of models of interest. The tool can be used to assess the efficiency of several DEVS simulation engines, and provides a common metric to compare the results using different tools. The synthetic model generator produces a variety of models with diverse structure and performing a mix of common operations, focusing on those that have an impact on performance, namely the size of the model and the workload carried out in the transition functions. A DEVStone generator builds varied models using the following parameters:

- Type: different structure and interconnection schemes between the components.
- Depth: the number of levels in the modeling hierarchy.
- Width: the number of components in each intermediate coupled model.
- Internal transition time: the execution time spent by internal transition functions.
- External transition time: the execution time spent by external transition functions.

Internal and external transition functions are programmed to execute an amount of time specified by the user, which execute Dhrystones.[38] The Dhrystone synthetic benchmark, intended to be representative for system programming, uses published statistics on the use of programming language features, and it is available for different programming languages (Ada, C++, Java, Pascal, etc.). Dhrystone code consists of a mix of instructions using integer arithmetic; therefore, it is a good choice for analyzing models such as DEVS in which we use discrete state variables. Any simulation tool based on a programming language in which Dhrystones can be defined and executed can be adapted to execute the DEVStone benchmark. In particular, simulation software without an internal transition function should set this time to zero, and the simulation can be executed.

DEVStone models use two key parameters d, the depth, and w, the width, with which a DEVStone model of any given size can be implemented, where each depth level, except the last, will have $w-1$ atomic models, and each atomic model provides a customizable Dhrystone running time. The inner model of such a scheme is composed of a coupled model that embeds a single atomic model. In general, with a depth d and width w, we build a coupled model with d coupled components in the hierarchy, all of which consist of $w-1$ atomic models (in the lower level of the hierarchy, the coupled component consists of a single atomic model). The model can be conceived of as a coupled component that wraps w atomic components and another coupled component, which in turn has a similar structure. The connection with the exterior is done by one input and one output links. The input feeds the first coupled component; the coupled component then builds links from the single input each of its subcomponents. Non-hierarchical M&S tools can use DEVStone by defining $d=0$ and use a single-level model to test the performance.

DEVStone uses four different types of internal and external structures:

- **LI** models, with a low level of interconnections for each coupled model;
- **HI** models with a high level of input couplings;
- **HO** models with high level of coupling and numerous outputs; and
- **HOmod** models with an exponential level of coupling and outputs.

Figure 2 shows the structure of a LI model in which we have four layers ($d=4$) of coupled components, where each layer contains three submodels ($w=3$). The arrows in the figure represent the input and output ports for a given coupled model; the solid white boxes represent coupled components, and the shaded gray boxes represent atomic components. The Coupled Component #0 in Figure 2(a) consists of one coupled and two atomic components. The lower levels in the hierarchy (Coupled Component #1, Coupled Component #2) use the same internal structure. Coupled Component #3 is a "leaf" model, which contains one atomic child (#7).

The main performance metric we are interested in computing is the execution time. In the case of a simulation tool based on message passing (like all DEVS software), we can also record the number of messages (and their type) in order to analyze the internal features of the simulation (for instance, classifying the number of synchronization and data messages involved in the simulation). We can easily perform stress tests by generating very large models (helping to test memory

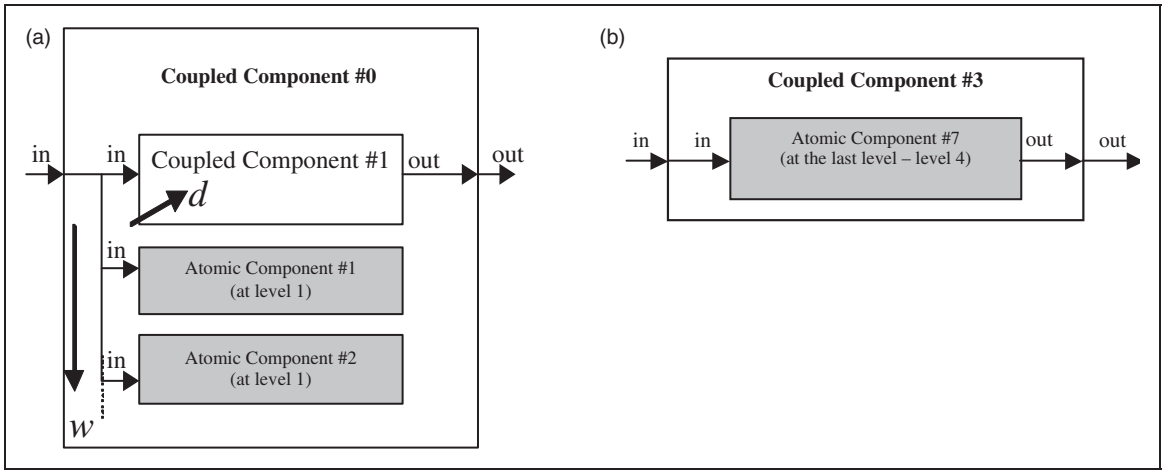

Figure 2. Example of a LI model: (a) top level; (b) level 4.

utilization, and the size of the models that can be executed).

As we know the model structure, and the time each component is supposed to spend executing transition functions, we can compute its theoretical execution time of every given model. First, we need to devise the number of atomic and coupled models in the structure. For instance, for the LI models just discussed, if the model has a depth of d levels and a width w models per level, we have d coupled components with $w-1$ atomic components each (except for the innermost one, which only has one atomic component). Consequently, the total number of atomic components is

$$\#Atomic\ Models = (w-1)*(d-1)+1$$
$$\forall\ d>1(\#\ Atomic\ Models = d\ otherwise)$$

Since these models follow a predefined interconnection pattern, we can anticipate the message routes triggered by an external event and the time spent in transition functions. LI models produce one external event for each coupled component; external events trigger the external transition function and, subsequently, an internal transition is scheduled. Thus, the number of transition functions to be triggered equals the number of atomic components in the model, as shownin the following:

$$\begin{aligned}\#Internal\ Transitions &= \#Atomic\ Models\\ \#External\ Transitions &= \#Atomic\ Models\end{aligned} \quad (1)$$

The difference between the three predetermined interconnections types, LI, HI, and HO, resides in the quantity of internal and external couplings for each level. LI models use just one external input and one external output coupling. Instead, HI and HO types

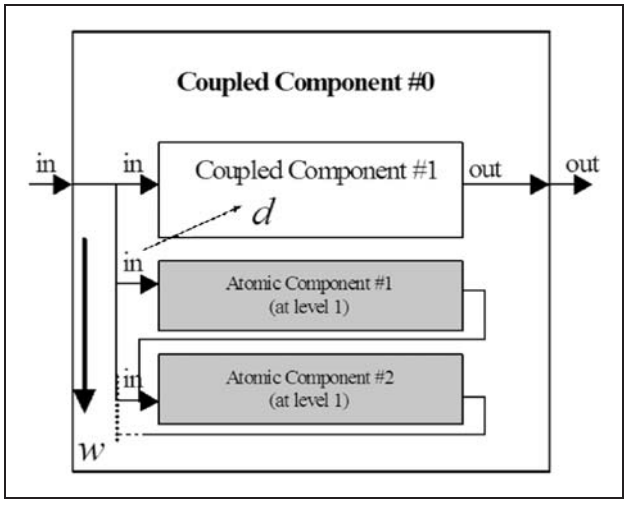

Figure 3. HI model structure.

have more interconnections among the components and the outputs, respectively. HI models connect the output port of an atomic block i to the input port of the next atomic block $i+1$, as seen in Figure 3.

Therefore, there are more messages interchanged upon the reception of an external event. In a model with depth d, and width w, we have

$$\begin{aligned}&\#AtomicModels = (w-1)*(d-1)+1\\ &\quad \forall d>1(\#Atomic\ Models = d\ otherwise)\\ &\#Internal\ Transitions = ((w-1)+(w-2))*(d-1)+1\\ &\#External\ Transitions = ((w-1)+(w-2))*(d-1)+1\end{aligned} \quad (2)$$

For each external event, each coupled model forwards the received message to its $w-1$ atomic children and to its coupled child. This process of forwarding

messages is repeated in each coupled component except for the leaf component, which forwards the messages to its single atomic child.

The **HO** type models (Figure 4) have a more complex interconnection scheme with the same number of atomic and coupled components. HO coupled models have two input and two output ports in each level. The second input port in the coupled component is connected to its first atomic component.

This atomic model connects its output to the second output of its parent coupled model, thus, the resulting number of interconnections is similar to the HI type of model, Equations (1) and (2) are identical for this type, with the difference that there are more messages interchanged between levels in HO types than in the HI type of modes. For this model type we have

$$\begin{aligned} &\#Atomic\ Models = (w-1)*(d-1)+1 \\ &\quad \forall d > 1 (\#Atomic\ Models = d\ otherwise) \\ &\#Internal\ Transitions = ((w-1)+(w-2))*(d-1)+1 \\ &\#External\ Transitions = ((w-1)+(w-2))*(d-1)+1 \end{aligned} \tag{3}$$

These equations do not take into account the time that is required for message passing among atomic models, coupled models, the simulator through the coupled coordinator, and the Root Coordinator (which is one of the interesting metrics the benchmark can compute). Thus, although the ideal execution time would be the same for HO models, the overhead in transmitting more messages would be higher (and this is a metric DEVStone can easily compute).

The HOmod models increment the message traffic, and they exponentially explode the interchange of messages among coupled models. They use a second set of $(w-1)$ models where each one of the atomic components triggers the entire first set of $(w-1)$ atomic models. These in turn have their outputs connected to the second input of the coupled model within the level. With such interconnections, the inner model receives an amount of events that has an exponential relationship between the width and the depth at each level. External events are forwarded by each coupled component to its $w-1$ atomic children and to its coupled child, and the process is repeated in each coupled module until the arrival to the leaf component.

For instance, in the case of Figure 5, we use $w=3$, which creates three atomic models, and a second set of

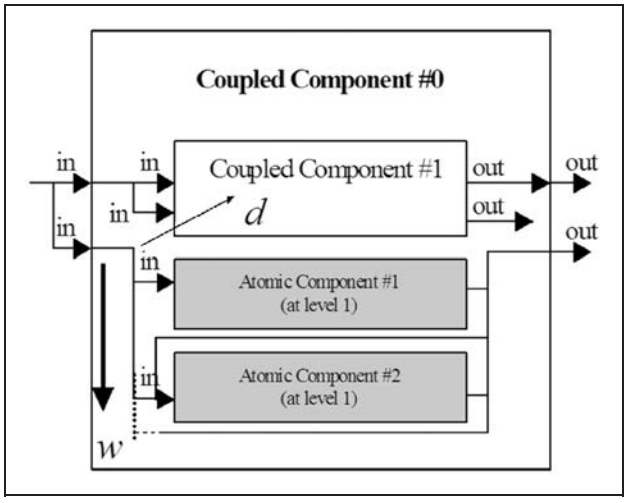

Figure 4. HO model structure.

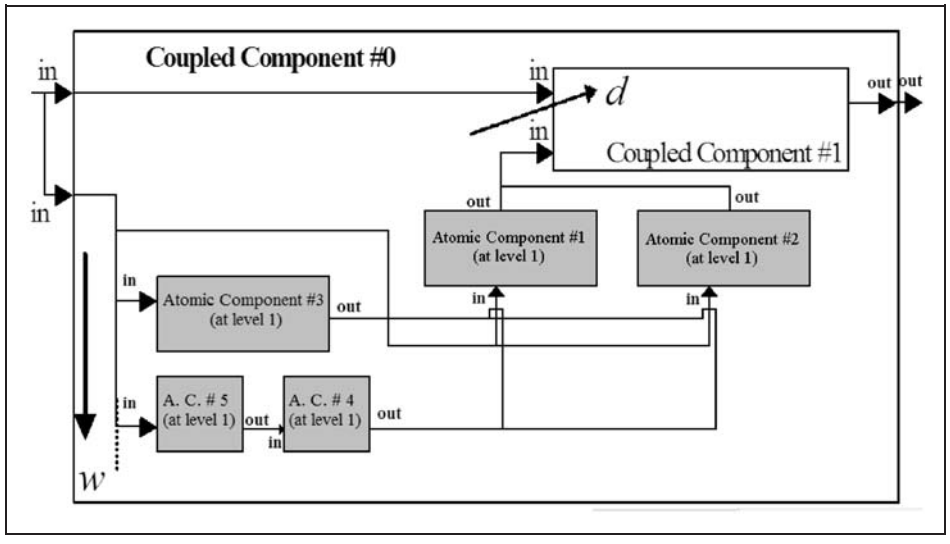

Figure 5. DEVStone HOmod model (shown explicitly for w = 3).

$w-1$ models connected to a coupled model. In this case, every input will be initially transmitted to AC #3, #5, #1, and #2. These trigger four internal and four external transitions. Then, model AC #4 triggers one more external transition; and the results of their internal transition will be spread to models AC #1 and #2. The output of AC#3 will also activate AC #1 and #2, totalling nine activations for this level. Each of the nine outputs is spread to the coupled component #1, which will trigger another nine transitions transmitted to its internal component #2, totalling 27 transitions.

Consequently, $\forall\ d>1$ (*# Atomic Models = d otherwise*) the behavior of the **HOmod** model is given by

$$\begin{aligned}\#\text{Internal Transitions} &= n_{it} = (w-1)^{2*(d-1)} \\ &+ \left(2*(w-1)+\sum_{1}^{w-2} i\right)*(d-1)+1 \\ \#\text{External Transitions} &= n_{\text{et}} = (w-1)^{2*(d-1)} \\ &+ \left(2*(w-1)+\sum_{1}^{w-2} i\right)*(d-1)+1\end{aligned} \tag{4}$$

In the example of Figure 5, having $w=d=3$, $n_{\text{it}}=n_{\text{et}}=2^4+(2\ *\ 2+1)\ *\ 2+1=27$.

DEVStone can be used in any simulator with capabilities for defining and executing Dhrystone code. We can also use single-layered models for comparison with tools with non-hierarchical structures. Likewise, if the chosen modeling technique does not support the execution of internal transitions, we can compare them with DEVS models without internal transitions scheduled.

3.1. DEVStone implementation

The models introduced in the previous sections were implemented in both CD++ and ADEVS software tools. As we can see, we need to implement the Atomic model to be used at the leaf level, and then a mechanism to create the corresponding coupled models. In this section, we will explain how this has been implemented in both environments.

The atomic component of DEVStone can be expressed as a simplified DEVS model:

DEVStone Atomic component = $< X, Y, S, \delta_{\text{int}}, \delta_{\text{ext}}, \lambda, ta >$where

$X=\{\text{in}\}$;
$Y=\{\text{out}\}$;
$S=\{\text{ø}\}$; (the atomic component has no internal states).
δ_{int} = number_of_dhrystones_internal; (the function simply uses a predefined value included as an external parameter; this value defines the number of Dhrystones to execute, this data comes from the model at runtime, and once executed, the model goes to passive mode);
δ_{ext} = number_of_dhrystones_external; (the function uses this value, which defines the number of Dhrystones to execute, this data comes from the model at runtime);
$\lambda=\{1\}$; (the output function simply generates an output value to be forwarded to the rest of the models; this value is not used by any of the models); and
ta is defined as an external parameter that the user can define for external transitions (internal transitions passivate).

Figure 6 shows a code snippet for the atomic model definition in CD++.

This atomic model definition starts by defining a constructor, where the input ports *in*/*out* are included. *DEVStoneAtomic* is a class derived from *Atomic* (which is used in CD++ for defining all of the atomic models). We also include a *preparationTime* argument, which can be used to define varied time advances in the external transition function (if this value is not defined, a default value of 1 ms is used). Then, we search the external coupled model file definition (*getParameter* is a method to do so), and search for the external values for the time advance, and number of Dhrystones to be used in internal/external transitions. The external transition function starts by recording some statistical information, and then it activates the Dhrystone function using the external parameter previously read. The model will remain active during the *preparationTime* defined by the user (as this is a discrete-event simulator, and the time advance function jumps directly to the next event, in the system, the choice of this time does not affect the performance). When this time is consumed, the output function is triggered (and an output is generated), and then the internal transition computes statistical information, executes the desired number of Dhrystones, and passivates waiting for the next external event.

Figure 7 presents the definition of the same atomic model in ADEVS. The model starts by defining input/output ports (*arrive*/*depart*) and the constructor initializes the parent atomic model. The external transition is activated when an external input arrives in the model, which in turn activates the Dhrystone and it computes the execution time. As ADEVS uses Parallel DEVS, all of the new events must be iterated, added, and included in the *line* list. The output transition is activated when the time for the first event arrives. The function takes the first element in the queue, and sets the departure time for it (while transmitting the value on the output port).

```
DEVStoneAtomic::DEVStoneAtomic( const string &name ) : Atomic( name )
    , in( addInputPort( "in" ) ) , out( addOutputPort( "out" ) )
       , preparationTime( 0, 0, 0, 1 )                                    {

       string time(getParameter( description(), "preparation" ) ) ;
       string timeIntDelay(getParameter( description(), "intDelay" ) ) ;
       string timeExtDelay(getParameter( description(), "extDelay" ) ) ;

       if( time != "" )           preparationTime = time ;
       if( timeIntDelay != "" )   intDelay = atoi ( timeIntDelay.c_str() );
       if( timeExtDelay != "" )   extDelay = atoi ( timeExtDelay.c_str() );

}

Model &DEVStoneAtomic::externalFunction( const ExternalMessage &msg )    {

    ParallelMainSimulator::Instance().incExternalRuns();
    dhrystoneRun (extDelay);
    holdIn( AtomicState::active, preparationTime );

}

Model &DEVStoneAtomic::internalFunction( const InternalMessage & )  {

    ParallelMainSimulator::Instance().incInternalRuns();
    dhrystoneRun (intDelay);
    passivate();
}

Model &DEVStoneAtomic::outputFunction( const CollectMessage &msg )  {
       sendOutput( msg.time(), out, 1 ) ;
}
```

Figure 6. Atomic model definition in CD++.

Then, this value is used by the internal transition, which activates the Dhrystone function, and consumes the first element in the bag before scheduling the internal transition.

Finally, Figure 8 shows the coupled model definition generated for CD++, which corresponds with that presented in Figure 2. This coupled component is generated through a PERL script that mimics the shape of the model to be executed and generates a coupled model with the corresponding structure. This script can be easily modified to change the internal structure or connections between components for doing different tests. For CD++, DEVStone was implemented as an automatic model generation tool based on a common template for each model. The coding was done in PERL, and it consists of a file for each model. Based on a common template for every coupled component per model, each PERL script uses multiple loops to form the final model using the desired depth and width, and DEVStoneAtomic components are embedded in CD++. All of these software pieces are publicly available for use and modification at http://cell-devs.sce.carleton.ca.

The coupled components of any DEVStone model are defined in the model, and generated following the procedure described in the previous section. On the other hand, the DEVStone benchmark model developed for ADEVS has a similar atomic component structure as that developed for CD++. However, due to the rather tight integration between the modeling environment (C++) and the simulator (C++ libraries), some challenges arise at the time of the implementation. One of them relates to the generation of the coupled and atomic components, since each one comprises a unique C++ file, meaning that for any model to be created in the ADEVS version of DEVStone new code needs to be created in C++. Another important problem that is generated from the first problem deals

```
// Assign locally unique identifiers to the ports
const int DEVStoneAtomic::arrive = 0;
const int DEVStoneAtomic::depart = 1;

DEVStoneAtomic::DEVStoneAtomic():Atomic<IO_Type>(), t(0.0), t_spent(0.0) {}
// Initialize the parent Atomic model; set clock to zero;
// no time spent on a external event so far

void DEVStoneAtomic::delta_ext(double e, const Bag<IO_Type>& xb) {

        start=clock();dhrystoneRun(extDelay); end=clock();
        diff = (double) (end - start) / CLOCKS_PER_SEC;

        t += e;      // Update the time spent on the exevent at the front of the line
        if (!line.empty()) t_spent += e;

       // Add the new external events to the back of the line.
       Bag<IO_Type>::const_iterator i = xb.begin();
       for (; i != xb.end(); i++)  {
             line.push_back(new Exevent(*((*i).value)));
             line.back()->tenter = t;   // Record time at which exevent entered line.
       }

}

void DEVStoneAtomic::delta_int()  {
        start=clock(); dhrystoneRun(intDelay); end=clock();

        diff = (double) ((end - start) / CLOCKS_PER_SEC);
        t = INF;  // passivates the model
        t_spent = 0.0; // Reset the spent time
        line.pop_front();  // Remove the departing exevent from the front of the line.
}


void DEVStoneAtomic::output_func(Bag<IO_Type>& yb) {
       Exevent* leaving = line.front();         // Get the departing exevent
       leaving->tleave = t + ta();       // Set the departure time
       IO_Type y(depart,leaving); // Eject the exevent
       yb.insert(y);
}

double DEVStoneAtomic::ta()  {
   if (line.empty()) return DBL_MAX; // next event is at infinity
       return line.front()->twait-t_spent;  // Otherwise, return the remaining time

}
```

Figure 7. Atomic model definition in ADEVS.

with multiple C++ and libraries that are created automatically and the way they need to be compiled and linked.

Since there is no separation of atomic and coupled components, both of them need to be created at the same time as individual C++ files. To overcome this problem, DEVStone for ADEVS was coded as PERL scripts for each model type as well as in the CD++ version. The scripts contain loops based on a template for each model type and the desired width and depth generate a *component* file and a *component* library file, for each atomic and coupled component. The linking among components is done in the coupled component files and the atomic component model, and library files are mainly used as libraries (only the names of variables and component labels are changed). Still, ADEVS does not provide exactly the same functionality as CD++; it can read data at runtime but it cannot use this data as an internal parameter. This is problematic with the input of the number of Dhrystones in the internal and external transition functions. As a result, the number of Dhrystones is entered in the creation of the simulator as a library file by the PERL script. The final external linking of the model and the creation of the simulator are done in an additional file that is also

```
[top]
components:  comp0 comp01@Atom comp02@Atom
out : out                   in : in
link : in in@comp0          link : in in@comp01
link : in in@comp02         link : out@comp0 out

[comp0]
components : comp1 comp11@Atom comp12@Atom
out : out                   in : in
link : in in@comp1          link : in in@comp11
link : in in@comp12         link : out@comp1 out
...

[comp01]
preparation : 00:00:00:000        intDelay : 0        extDelay : 0

[comp02]
preparation : 00:00:00:000        intDelay : 0        extDelay : 0
...
```

Figure 8. Model files generated by DEVStone for a LI model.

created by the generator by keeping track of the outer coupled and atomic components of the model. A *makefile* script is also generated by the PERL scripts to automate the compiling and linking of all of the C++ files.

In the following sections, we show how the benchmark can be used to collect different metrics of interest. Although we have discussed that DEVStone focuses on execution time performance (in terms of total execution time, and the number of messages involved in the simulation process), it can be used to obtain other metrics too. Most existing benchmarks can be used with multiple purposes (for instance, in addition to studying the execution time, a benchmark can be used to measure the amount of memory used, the number of I/O operations executed, the amount of time spent in communications, etc.), as they provide the same set of operations to be executed repeatedly. In the following sections, we show how to make use of the benchmark structure to obtain different metrics (according to the user's needs) which were meaningful for the simulation environments under study.

4. Case study 1: Virtual-time simulators

As discussed earlier, DEVStone can be easily used to compare the performance of different versions of the same simulation software with ease. At present, there are numerous versions of the CD++ software, which include a standalone CD++ simulator (running in single-processor simulations) and various parallel versions (based on Parallel DEVS[39]) allows complex models to be executed, which often requires high performance computing power. Parallel CD++ was built on top of Warped,[40] which provides different optimistic synchronization algorithms. It also implements an unsynchronized protocol (called NoTime), which is used to compare the pure overhead of new algorithms. In particular, the NoTime kernel is useful in evaluating the performance overhead of DEVS simulations, because a DEVS simulation algorithm provide its own synchronization method (therefore, one can run a parallel DEVS simulation, and the synchronization provided by Warped is not needed, although it can improve the overall performance if activated). Therefore, we use the NoTime kernel to compute the pure overhead introduced by the simulation kernel.

In this section, we show how DEVStone can be applied to characterize the overhead of different version of a given simulation tool. In addition, allowing one to test the usefulness of the benchmark, the experiments presented here are the first systematic effort at characterizing the performance of DEVS M&S environments.

The first tests we present here show a basic analysis of the overhead of three CD++ simulators: (i) the standalone CD++ (named as *original* in the various figures), (ii) a parallel CD++ with unsynchronized kernel (that is, a kernel using only DEVS simulation synchronization algorithms), and (iii) parallel with optimistic kernel. The idea is to compare the overhead

generated by increasing levels of synchronization in the kernel. We compared the execution times results with the theoretical execution time for each type, computed as follows:

$$\begin{aligned} Total\ theoretical\ time = [&(\#External\ Transitions \\ &* Time\ InExternal\ Transition) \\ &+ (\#Internal\ Transitions * TimeInInternalTransition)] \\ &* Number\ Of\ Ext\ Events\ To\ Top\ Component \end{aligned} \tag{5}$$

DEVStone was used to generate different model structures, and the models were executed using 10 external events at a constant rate. Each of these events triggered a known number of external and internal transition functions defined by Equations (1)–(4). Table 1 shows the parameters we used for different the tests executed in this case, which were used to compute the overall execution time and the number of messages involved in the simulation (which is an even more meaningful metric when one considers that the models could execute in a distributed environment).

As we can see, the user can define a wide variety of test cases with different arguments, including model type, structure and time spent on transition functions (e.g. model E is of HI type, it is composed of three levels, with six components per level). DEVStone allows one to define these test cases with ease, and to define different scenarios to test a given tool thoroughly. This table shows the variety of tests that one can build; all of them are automated, and they are based on the arguments used to generate the corresponding DEVStone structure. Other parameters could be used for different tests. For instance, one could use extended model types (with loopback connections or larger number of internal connections) and varied time for the time advance and transition functions (in order to explore the differences in having completely different execution time for the internal or external transition functions and the influence of the function delays). Other options would include very large models (to study cases of memory exhaustion), and varied time advance functions (in order to study the influence of different time advance values in the scheduling algorithms in the simulator; in all of these cases, *preparationTime* is used to trigger an internal transition immediately after the previous one has finished its computation). These modifications can be automated, and the method to create the models is the same for any simulator (the idea is to follow up the procedure to generate a DEVStone, as discussed in Section 5, and to execute the benchmarks in the different simulation engines). DEVStone is generic and these arguments can be easily changed, while keeping a standard way of testing different versions of a given simulator or different simulation engines. For instance, Simulation A in Table 1 could be used for many different simulators, and in all cases, this test should use an LI model of three layers, each of them containing 10 submodels, and using 50 ms of execution time in each of the transition functions. This time should be generated using the Dhrystone benchmark to provide a uniform set of instructions on any of the simulation software applications.

Table 1. Simulation parameters

Simulation	Model Type	Depth	Width	δ_{int}	δ_{ext}
A	LI	3	10	50 ms	50 ms
B	LI	10	3	50 ms	50 ms
C	LI	5	5	50 ms	50 ms
D	LI	10	10	50 ms	50 ms
E	HI	3	6	50 ms	50 ms
F	HI	6	3	50 ms	50 ms
G	HI	5	5	50 ms	50 ms
H	HI	6	6	50 ms	50 ms
I	HO	3	6	100 ms	0 ms
J	HO	6	3	0 ms	100 ms
K	HO	5	5	50 ms	50 ms
L	HO	6	6	50 ms	50 s

The following figures summarize the worst execution times for a large number of simulations for each of these cases (variation between these and the best execution cases was below 1%, as the models are deterministic and the tests were run in dedicated equipment; the variations are mainly due to the operating system overhead). The experiments were executed in a single processor, allowing us to measure the pure overhead incurred by the different simulation engines.

The variety of tests that can be easily executed using DEVStone allow one to analyze different aspects of the simulation engines with ease. As expected, the original version provided the best execution time, whereas the parallel unsynchronized kernel (NoTime) always outperformed the parallel optimistic (Time Warp) version. Figure 9(a) illustrates the execution time of LI models. The relatively simple structure and size of Simulations A, B and C resulted in small differences between the theoretical and actual execution times. In Simulation D, which contains more than 80 models, the difference is more noticeable. The same tendency is observed in the execution of HI and HO models, especially for larger models.

The most relevant results for these tests are related to the performance of the hierarchical simulation

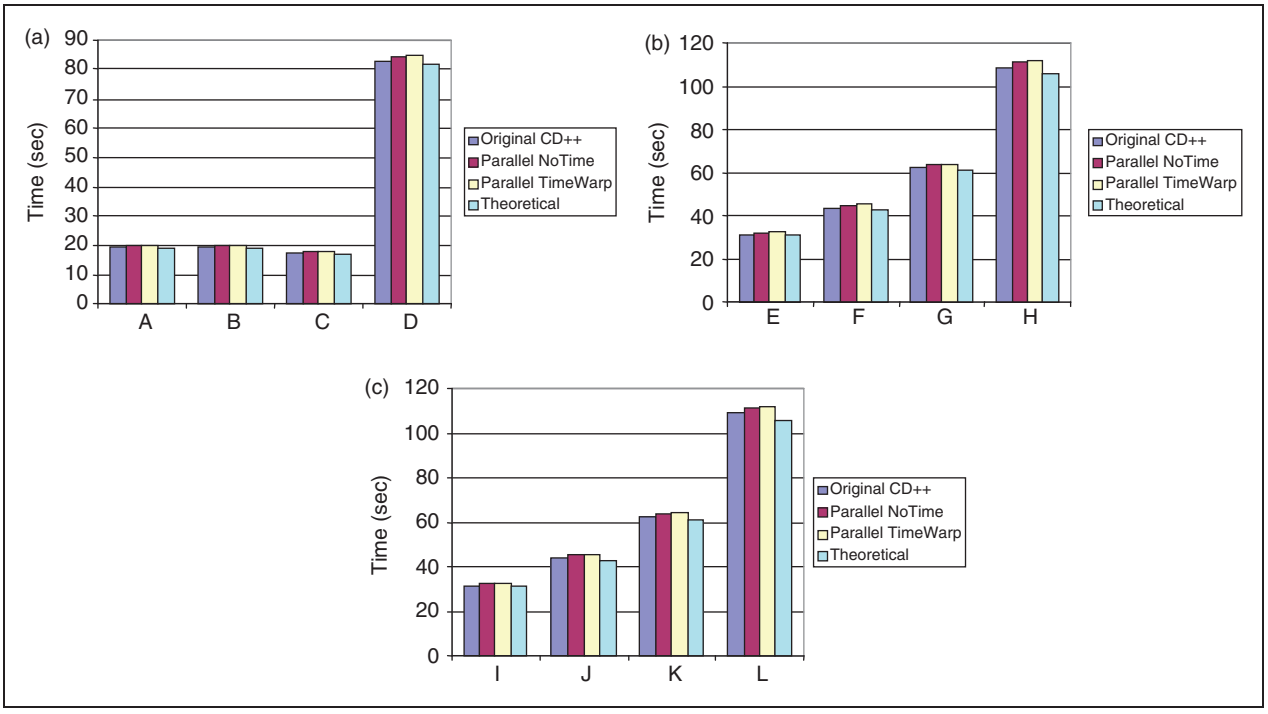

Figure 9. Execution times of different models using different simulation engines with (a) LI, (b) HI, and (c) HO.

application. Although we show that a flat simulator provides better performance (in the following sections in this paper), we can see that the amount of overhead for the original version (a hierarchical simulator) is only between 1% and 3% in every case (and this overhead is below 5.5% for the most complex problems running on top of the Warped middleware). These results show that a hierarchical simulation engine introduces constrained overhead, while providing ease for modeling.

From this point, unless stated otherwise, the analysis is focused only on the original CD++ (equivalent results are observed in the other versions).

The different DEVStone tests also permit one to determine the impact of structure on the overall performance. For instance, Simulations C (LI), G (HI), and K (HO) have 5×5 components each; and the same workload in the internal and external transitions. Nevertheless, we can see an increase in the overhead: 1.25% for Simulation C, 2.2% for Simulation G, and 2.4% for Simulation K. This is due to the more complex internal structures that the HI and HO models have.

DEVStone also allowed us to easily generate and run models with a large number of components (approximately 10,000 models), which resulted in higher overhead ratios, as seen in Table 2.

Running extensive tests through DEVStone allowed us to find the results presented in Table 2. As we can see, there are a large number of internal messages interchanged, which causes the overhead. This version of CD++ uses a hierarchical modeling method, and a hierarchical simulation algorithm as presented by Kim et al.[41] This algorithm creates a one-to-one correspondence between DEVS models and execution engines called *processors*: *simulators* execute atomic models, and *coordinators* execute coupled models (as previously shown in Figure 1). Following this idea, Figure 10(a) shows a two-level model; the top-level (*Coupled Model* #1) model includes three atomic (#1–3) and one coupled model (*Coupled Model* #2); this coupled model is composed of two atomic models (#4 and #5). Figure 10(b) illustrates the processor hierarchy built to run this model (*Coordinator* #1 schedules the activity of *Coupled Model* #1; *Coordinator* #2 is in charge of *Coupled Model* #2, and every *Atomic* model is associated with a *Simulator*). As we can see, every time the *Root Coordinator* needs to send an external event to *Simulators* #4 and #5, it must send a message to *Coordinator* #1 first. The same occurs when *Model* #5 sends an output to *Model* #3.

The number of these intermediate coordinators can be arbitrarily large depending on the model, as seen in Table 2 (which also shows the number of messages DEVStone generated in order to process a single external event). For instance, models R and S have identical structure, but the internal interconnections of their

Table 2. Simulated models using the hierarchical simulation approach

	Model			
Simulation parameter	R	S	T	U
Number of components per level	100 components	100 components	150 components	150 components
Hierarchy depth	100 levels	100 levels	75 levels	75 levels
Model type	LI	HO	LI	HO
Number of atomic components	9,802	9,802	11,027	11,027
Number of simulators	9,802	9,802	11,027	11,027
Number of coupled components	99	99	74	74
Number of coordinators	99	99	74	74
Number of root coordinators	1	1	1	1
Number of messages exchanged per single external event	79,220	3,484,718	89,416	2,958,468
Execution overhead	9.96%	10.84%	9.42%	10.27%

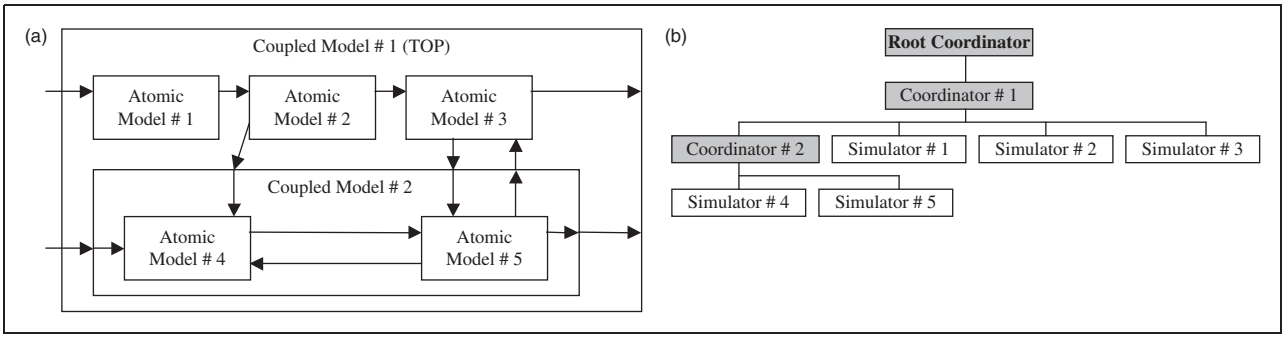

Figure 10. (a) Sample model structure and (b) associated processor hierarchy.

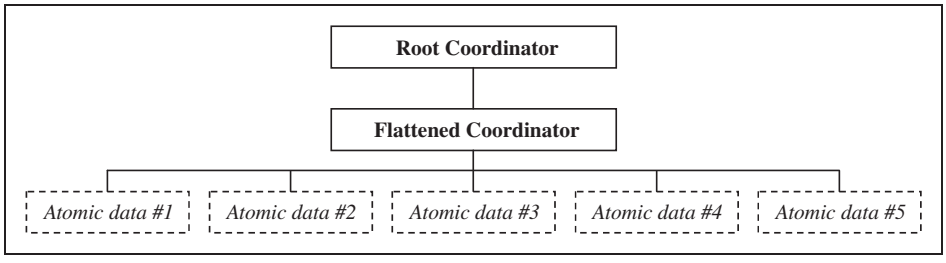

Figure 11. Flat simulator approach for Figure 10.

components results in a remarkable difference in the number of messages involved (the same happens when we compare models T and U). These results provided a hint to optimize the simulation technique: reducing intermediate coordinators by flattening the simulation hierarchy as suggested by Kim.[41] The idea is to create a flat coordinator that triggers δ_{int}, δ_{ext}, and $\lambda(s)$ functions for each atomic component, transforming the hierarchical model into a flat structure by mapping the ports for all atomic and coupled components in the hierarchy. Component links are handled by the flat coordinator, which forwards the events as needed. Finally, it must handle the interaction with the *Root Coordinator*, as seen in Figure 11.

We applied DEVStone to study this new version of CD++. Table 3 shows the execution results for the same parameters presented in Table 2. As simulators and coordinators disappeared, and one flat coordinator is created regardless of the number of components, the resulting overhead has been reduced from 10% to close to 5%.

Regardless of the type of model (LI or HO) and their structure, the simpler simulator hierarchy and fewer messages exchanged during the simulation reduced

the overhead; a 38.3% for small LI models, up to 47.7% for larger LI or HO models with the same number of components (7×5 and 7×9 models).

In order to analyze the performance degradation purely due to overhead in the simulation engine, we executed several examples with empty transition functions (such as that presented in Figure 8(c)). In these experiments, the execution time solely depends on the message exchange.

Figure 12 shows the pure improvement due to the new flat simulator. The figure presents the performance obtaining by changing the width of an LI model with a fixed depth of 8; and the performance of HO models with variable depth and fixed width of 8. Regardless of the model's width and depth, the flat simulator reduces the execution time by 52.4–54.7%. For HO models, the improvement in performance becomes more noticeable when the depth of the model increases, as the impact of the intermediate coordinators that are eliminated from the hierarchy results in fewer messages being exchanged, and more efficient simulation.

DEVStone allowed us to execute numerous tests such as these, and we could find the same trend in all synthetic models; the flat technique reduces the overhead incurred by the hierarchical approach by between 40% and 58%, as a result of the reduction of the number of exchanged messages.

Table 3. Simulated models using the flat simulation approach

	Model			
Simulation parameter	R	S	T	U
Number of components per level	100	100	150	150
Hierarchy depth	100	100	75	75
Model type	LI	HO	LI	HO
Number of atomic components	9,802	9,802	11,027	11,027
Number of simulators	0	0	0	0
Number of coupled components	99	99	74	74
Number of coordinators	0	0	0	0
Number of root coordinators	1	1	1	1
Number of flat coordinators	1	1	1	1
Number of messages exchanged per single external event	4	9,804	77	11,029
Execution overhead	4.51%	5.64%	4.38%	5.21%

5. Case study 2: Real-time simulators

CD++ was recently extended to allow real-time execution.[21] In real-time systems such as this one, correctness depends not only on computation results, but also on the time at which the results are produced.[42] A correct answer after its deadline is regarded as an unsuccessful response. The real-time CD++ extension allows the execution of events triggered by the real-time clock, and it provides numerous I/O device drivers in order to enable interaction between the models and their surrounding environment.

The virtual-time simulation discussed in the previous section provided us with encouraging results in terms of the overheads involved for real-time execution. Nevertheless, overhead can be a serious impediment in actual implementations of systems with real-time constraints. Therefore, this new version of the simulation needed a completely new set of tests in order to be able to determine the performance of the new real-time engine. DEVStone allowed us to automate the generation of those tests with ease, allowing us to study the performance of the real-time simulator thoroughly[17] by analyzing different synthetic models and their execution in real-time. The idea was to generate DEVStone tests, and use them to execute virtual-time and real-time simulations, computing again the execution time and the number of messages involved. Then, we compared the time required to process a single event in virtual-time against the worst-case response in real-time. This is computed as the time the simulator takes from the

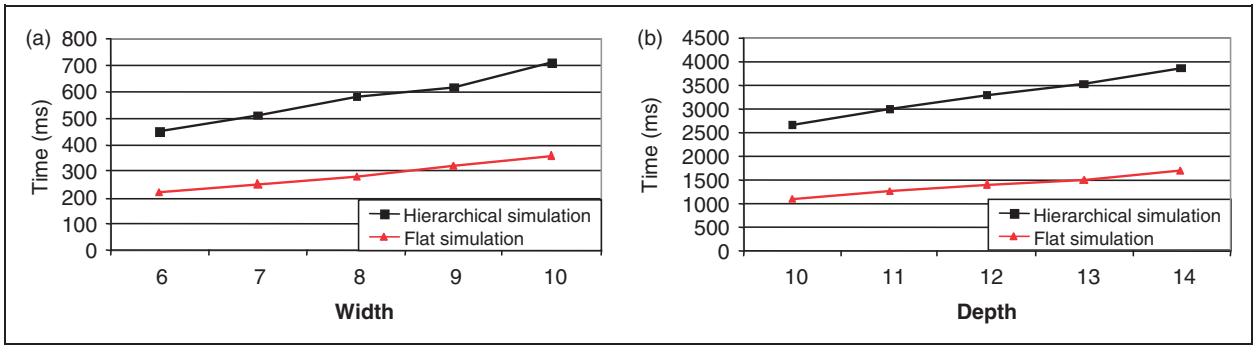

Figure 12. Execution time comparison for hierarchical and flat simulators: (a) LI with variable width; (b) HO with variable depth.

moment an external input is received until a response is obtained (analyzing end-to-end deadlines, in which an input is received in an external input port, and an output is expected in a given external output port). Again, using the synthetic structure of the model generated by DEVStone helps in determining the worst-case response time, as one can follow the path of execution of the messages in the runtime system, and the total execution time (theoretical and real) of a given model. This example shows how to use DEVStone to compute a different meaningful metric when needed: the worst-case execution time represents a meaningful measure since real-time systems deal with predictable responses in critical applications (and average compute times are not a valid mechanism for comparing the desired objectives). We present the execution results for different stress tests in which external input messages are injected and the output of the model is compared with a predefined deadline. (For instance, in Figure 10(a), we compute the time from the moment when an event is received in a given input port, until the corresponding output, defined by the user, is seen in the corresponding output port.) RT-CD++ allows the user to define the model's deadline (which can coincide or not with the time advance function), being able to automate the testing of the success ratio (i.e. the number of missed deadlines versus the total number of input events received from the environment).

DEVStone was used to automate the generation of synthetic models to measure the ability of RT-CD++ to execute those models (using different structures and very demanding situations). The real-time simulations received a fixed number of external events generated at a constant rate, and each model was expected to deliver responses to those inputs before a given deadline. Each input produces a single output, and we measure the real-time required to compute each output. The success ratio and the worst-case response times were computed as

$$\textit{Percentage of success} = \frac{(\textit{number of events} - \textit{number of missed deadlines}) * 100}{\textit{number of events}} \quad (5)$$

$$\textit{Worst} - \textit{case response time} = \max(r_1, r_2, \ldots, r_N) \quad (6)$$

where r_i is the response time for the ith event, and N is the number of events received.

DEVStone was used to build large LI and HO models (with 25–50 components) under a simulation scenario in which external events arrive at high frequencies with strict deadlines. The idea is to increase the complexity of the model, while not changing the input event frequency or their deadlines, trying to find out the cases where the simulator is unable to meet its deadlines. The simulator was highly overloaded, having external events arrive every 30 ms and deadlines (d_i) set at 60 ms after the arrival of an external event (e_i), as shown in Figure 13.

The model can respond at any time between the occurrence of the arrival of a new event (which triggers an external transition), and the response should be received 60 ms after that. This includes the consumption of the time advance function, triggering of the output function, computing of the output function and generation of the output in the corresponding output port). RT-CD++ has been modified to allow the user to define such deadlines and record whether the deadline has been missed or not. If the simulation algorithms are inefficient, this would imply missing a large number of deadlines, and this is the main objective of the study.

DEVStone was used to characterize the performance of this new simulator thoroughly. Figure 14 presents some tests that show that it is possible to execute LI models with up to 35 components and different shapes with a 100% success ratio (and minimum worst-case response time). Under these conditions, when simulating a LI model with 40 active components in its structure, we obtained 89% and 100% success (executing wide models was more efficient than the deep models with the same number of components). In contrast, only 12% and 18% of the deadlines were met for the equivalent HO.

As we know the DEVStone structure with precision, we can use Equations (1) and (3) to compute the number of transition functions triggered for each model type. For LI models of this size, each external event triggers 80 transition functions, whereas 240 are triggered in the equivalent HO models. This larger number results in greater overhead, and a larger number of messages to be exchanged (which explains the low success ratio for complex model types).

Most of the experiments presented up to now have studied the execution time for different models, focusing on structure, size, and complexity. DEVStone can

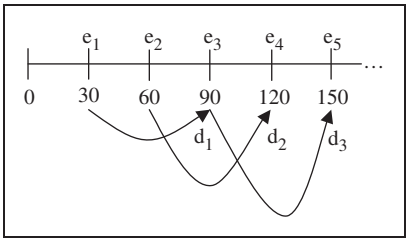

Figure 13. Event arrival and deadlines.

be used for an alternative analysis focusing on the performance of the target environment. DEVStone was used to study the effect of different inter-arrival periods (i.e. the frequency of event arrivals) on the execution performance. We generated different DEVStone coupled models, and then triggered external events arriving at different paces (20–180 ms). For instance, Figure 14(b) shows one of those experiments, where we analyze the success ratio for 8×8 HO models receiving 100 events with deadlines set at 1000 ms. As we can see in the figure, when the inter-event period is reduced significantly, the success ratio is worse (because small inter-event times do not allow the simulator to process all of the messages for event e_k and the next event e_{k+1} arrives before that, 10 unprocessed messages accumulate, and there is an evident degradation of performance).

Finally, Figure 14(c) shows the real-time performance of the flat versus the hierarchical simulator for an HO model with variable depth (various tests were generated in a similar fashion; this figure shows the result for width = 9). In this scenario, the flat simulator meets all of the deadlines for up to the 13×9 HO model (97 components). In contrast, the hierarchical approach had an inferior percentage of success (87%) and showed worse response times, even for smaller models (7×9; 49 components). For larger models, the difference between hierarchical and flat simulators becomes more noticeable.

DEVStone models allowed us to test these varied cases with ease, showing that the flat technique outperforms the hierarchical, reducing the incurred overhead by up to 50% and therefore providing improved response times and a better success rate. A main cause for this is the reduced number of messages exchanged in the flat simulation mechanism.

6. Case study 3: Comparing ADEVS and CD++

DEVStone was applied to compare the performance of the ADEVS and CD++ simulator. This is the first attempt at a documented effort comparing two DEVS simulators, which has been possible thanks to the basic idea of this flexible and reconfigurable benchmark. The two simulators were compared using similar block structures, fixing the different parameters in order to be able to obtain similar metrics. CD++ was compiled with GCC 2.95.3, and ADEVS used the GNU Compiler version 3.4.2; both without debugging information and default compile-and-link options. In this case, we computed the execution time for each of the cases.

DEVStone allowed us to find some non-documented limitations in both tools, namely:

- ADEVS depth cannot be greater than 195 levels (due to a limit of the GCC compiler, which finds too many nested loops inside the executable). CD++ does not seem to have a limit on the depth (we executed 4000 levels and 3 components per level were read by the CD++ simulator without problems).

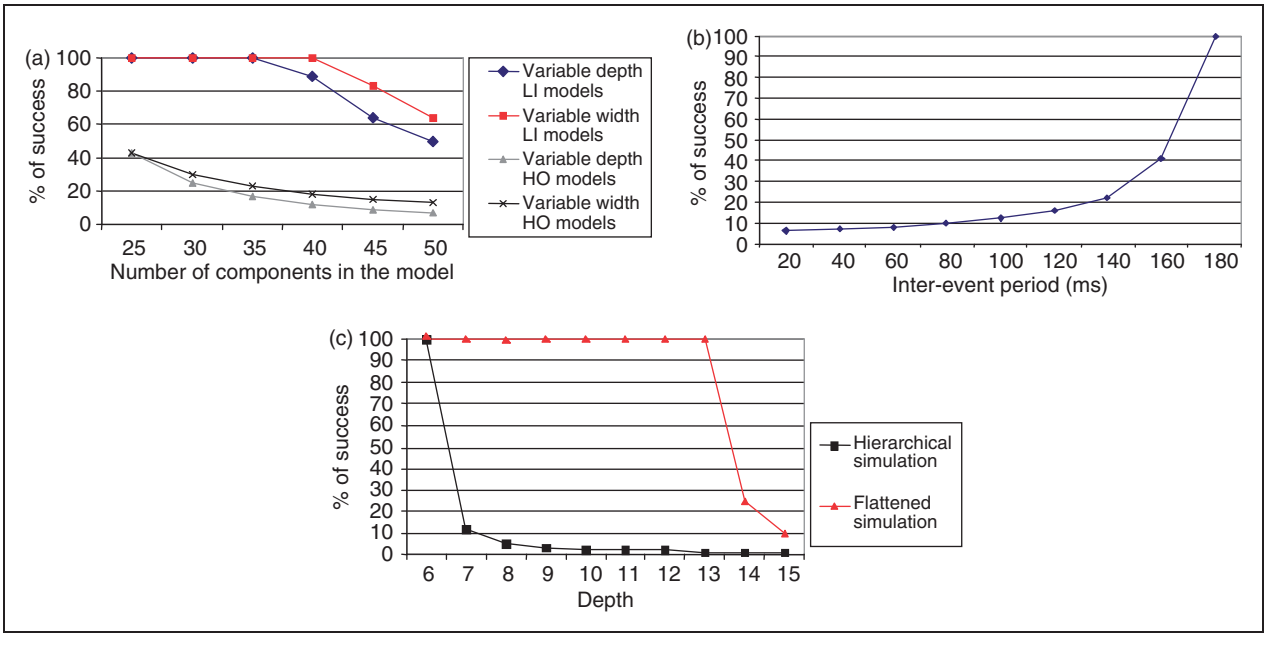

Figure 14. Comparison of model execution with variable depth and width. (a) Percentage of success for varied models. (b) Success ratio for varied inter-arrival periods. (c) Success ratio for hierarchical/flat simulators.

- CD++ uses a construction to define the components of a single level (the *width* of the level), and all of the components can be defined on a single text line. For widths of more than 1839 components inside each level, the simulator generates a message that the components are too large. The script described in Section 3, which is used to generate the coupled model automatically, was thus modified to split the components in multiple lines of 1500 atomic blocks.
- The minimum depth and minimum width for any model generated by DEVStone for CD++ or ADEVS is 2.
- For extreme cases (i.e. models of 195 × 1839 components), it is possible to initialize the simulation, but it is not possible to run the simulation to any time longer than 0 s, due to memory limitations (the process is killed by the Out-Of-Memory kernel service). It seems that, whenever the simulator requests massive amounts of memory to the operating system (beyond the available physical memory and some of the virtual memory), the OS decides to terminate process as a potentially harmful.

Considering that CD++ builds models dynamically, and ADEVS uses a library to generate a compiled version of the model, we were first interested in studying the performance of both simulators during the transient initialization period. To do so, we used DEVStone to measure the initialization time in both CD++ and ADEVS (showing how the benchmark is flexible enough to analyze yet another interesting performance metric without special modifications). To do so, we considered the worst-case scenario for both simulators when executing a model that is 195 levels deep and contains 1839 components on each level. The simulation results for LI models are presented in Figure 15 and Table 4. We considered the initialization time only, i.e. simulation time is $T_{sim} = 0$. From the graph, it is obvious that ADEVS outperforms CD++ in terms of initialization speed. This is because in CD++, the simulator and the model are two completely separated entities (the compilation of the simulator takes only a couple of minutes and can be used for any model; in our case the simulator was compiled only once and reused for all the simulations in this paper). On the other hand, ADEVS needs to be recompiled every time a model (atomic or coupled) changes. Therefore, for small to medium sized models, even minor fixes required recompilation, whose time surpassed the simulation time. Subsequently, we used DEVStone to understand these differences better, combining DEVStone with a profiler.

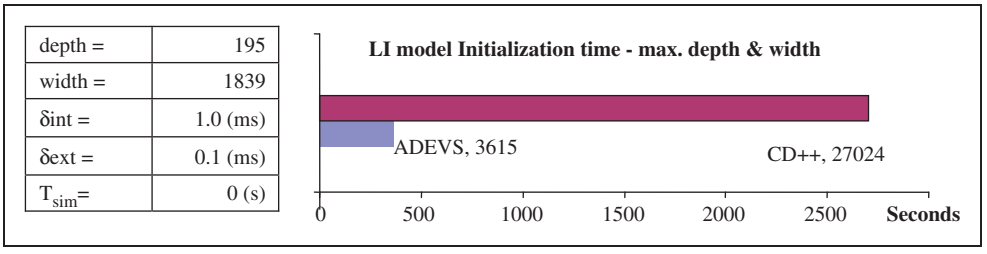

Figure 15. DEVStone initialization time LI model.

Table 4. Profiler output for the simulation of LI models

Time (%)	Cumulative (s)	Self (s)	Calls	Self K/call	Total K/call	Name
29.09	4,633.2	4,633.2	1,427,848	0	0	ProcessorAdmin::processor (basic_string<. . .> const &)
17.91	7,486.26	2,853.06	450,903,011	0	0	basic_string<. . .>::compare (basic_ string<. . .> const &,unsigned int, unsigned int) const
15.86	10,012.31	2,526.05	162,000,611	0	0	basic_string<. . .>::rep(void)const
9.59	11,539.19	1,526.88	3,307,266,338	0	0	basic_string<. . .>::length(void) const
7.91	12,798.64	1,259.45	989,592,590	0	0	basic_string<. . .>::data(void) const
7.57	14,004.23	1,205.59				String_char_traits<char>::compare (char const *, char const *, unsigned int)
6.85	15,095.7	1,091.47	287,281,638	0	0	Processor::description(void) const
5.05	15,899.58	803.88	1,065,955,666	0	0	basic_string<. . .>::Rep::data(void)

According to the initial result of the profile, a large part of the initialization time (20.09% of it) is spent inside the *processor* block. Another important source of delay is the comparison of symbols (done by C++ libraries while loading the model into memory). Based on these results, we did a more thorough analysis using DEVStone combined with GNU profiler, who provides a *Call Graph* that indicates functions inside the program and the *relative* time that the computer takes running those functions. While running DEVStone, we obtained the results shown in Table 5.

According to the output of the profiler, CD++ spends most of the time looking at and comparing the new and incoming symbols, or model names, which explains the initialization overhead. Most of the workload is spent in library functions that are specific to the compiler, which are used for symbol parsing, look-up, and compare. This is heavily dependent on the libraries used to compile CD++ (and better performance can be obtained by simply porting the CD++ simulator to a newer version of GNU C++ or better libraries).

As ADEVS takes time compiling very deep models, it would be useful to compare the performance of both simulators using a model with maximum depth and minimum width. An event file was created that provides 10 external events evenly spaced every 0.250 (s) and the results of the first simulation with such files are shown in Figure 16.

In this case, CD++ proves to be faster than ADEVS, mainly because in CD++ the models are loaded in memory on demand, and processed accordingly. In the first scenario (where the internal transition function time equals the external transition function) ADEVS takes almost 50% more time. This is because ADEVS (a) loads the entire model-simulator to memory, (b) runs the simulation, and (c) flushes the

Table 5. Call graph of the CD++ simulator

Index	Time (%)	Self (s)	Children (s)	Times called	Function name
		0.00	81,691.17	1/1	Main [1]
[2]	98.5	0.00	81,691.17	1	MainSimulator::run(void) [2]
		...	...	...	...
[4]	85.4	2,853.06	67,920.51	450,903,011	Basic_string<...>::compare(basic_string<...> const &, unsigned int, unsigned int) const [4]
		1,496.24	50,534.74	3,240,893,692/3,307,266,338	Basic_string<...>::length(void) const [7]
		...	...	...	...
		51,569.68	0	3,307,266,338/162,000,611	Basic_string<...>::length(void) const [7]
[5]	83.8	69,496.80	0	162,000,611	Basic_string<...>::rep(void) const [5]
		...	...	...	...
		1,496.24	50,534.74	3,240,893,692/3,307,266,338	Basic_string<...>::compare(basic_string<...> const &, unsigned int, unsigned int) const [4]
[7]	64.1	1,526.88	51,569.68	330,7266,338	Basic_string<...>::length(void) const [7]
		51,569.68	0	3,307,266,338/162,000,611	Basic_string<...>::rep(void) const [5]
		2,316.60	23,254.22	713,924/1,427,848	MainSimulator::loadLinks(Coupled &, Ini &) [9]
		2,316.60	23,254.22	713,924/1,427,848	Coupled::addInfluence(basic_string<...> const &, basic_string<...> const &, basic_string<...> const &, basic_string<...> const &) [11]
[10]	61.7	4,633.20	46,508.43	1,427,848	ProcessorAdmin::processor(basic_string<...> const &) [10]
		1,817.76	43,273.98	287,282,416/450,903,011	Basic_string<...>::compare(basic_string<...> const &, unsigned int, unsigned int) const [4]
		...	...	...	...
		0.12	25,768.79	356,962/356,962	MainSimulator::loadLinks(Coupled &, Ini &) [9]
[11]	31.1	0.12	25,768.79	356,962	Coupled::addInfluence(basic_string<...> const &, basic_string<...> const &, basic_string<...> const &, basic_string<...> const &) [11]
		2,316.60	23,254.22	713,924/1,427,848	ProcessorAdmin::processor(basic_string<...> const &) [10]
				...	...

program from RAM. Reading and writing the whole file to and from the harddrive explains the increase in simulation time. Checking memory usage while running DEVStone showed that ADEVS takes up to 99% of the available memory right from the beginning while CD++ increases the memory usage incrementally.

DEVStone provides a feasible methodology to analyze the performance of these different simulators and to study their performance thoroughly. A large number of simulations were executed with different models of similar values for every model. We were able to run numerous tests based on changing four main parameters: the model type, variations in the width of the model, variations in the depth of the model, and variations in the real-time running internal and external transition functions.

By changing the width of the model, we can focus our analysis on the time that the simulator spends sending messages back and forth to atomic blocks within each level.

In Figure 17 we can see that ADEVS clearly outperforms CD++ by a significant margin for large models (and this difference is larger when the internal and external transition times are equal). One cause

Simulation Parameters

		1		2		3	
Depth	Width	δ_{int}	δ_{ext}	δ_{int}	δ_{ext}	δ_{int}	δ_{ext}
195	2	0.1ms	0.1ms	0.1ms	1ms	1ms	0.1ms

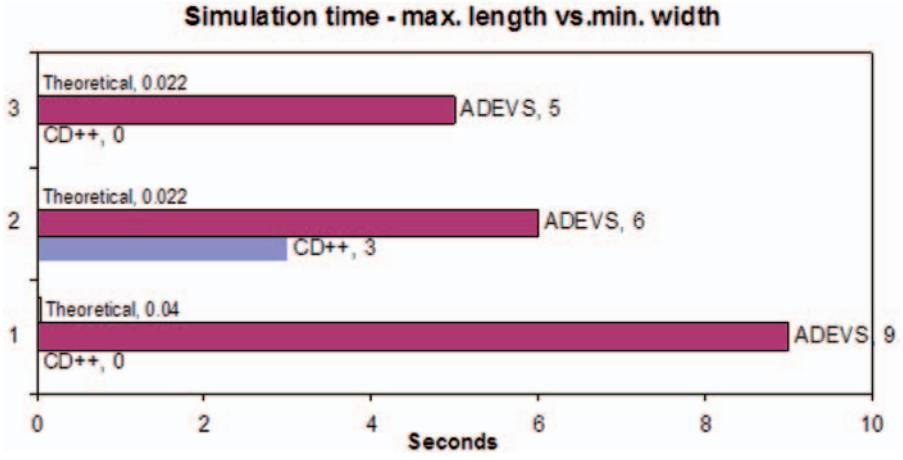

Figure 16. Comparing ADEVS and CD++ setup times (rounded to seconds).

Simulation Parameters

		1		2		3	
Depth	Width	δ_{int}	δ_{ext}	δ_{int}	δ_{ext}	δ_{int}	δ_{ext}
10	100-600	0.1ms	0.1ms	0.1ms	1ms	1ms	0.1ms

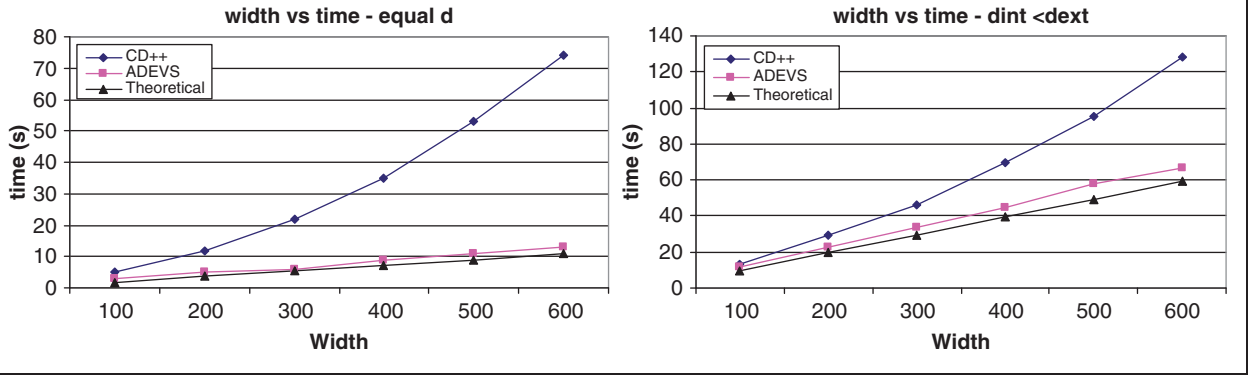

Figure 17. CD++ vs. ADEVS. LI models: (a) $\delta_{int} = \delta_{ext}$; (b) $\delta_{int} < \delta_{ext}$.

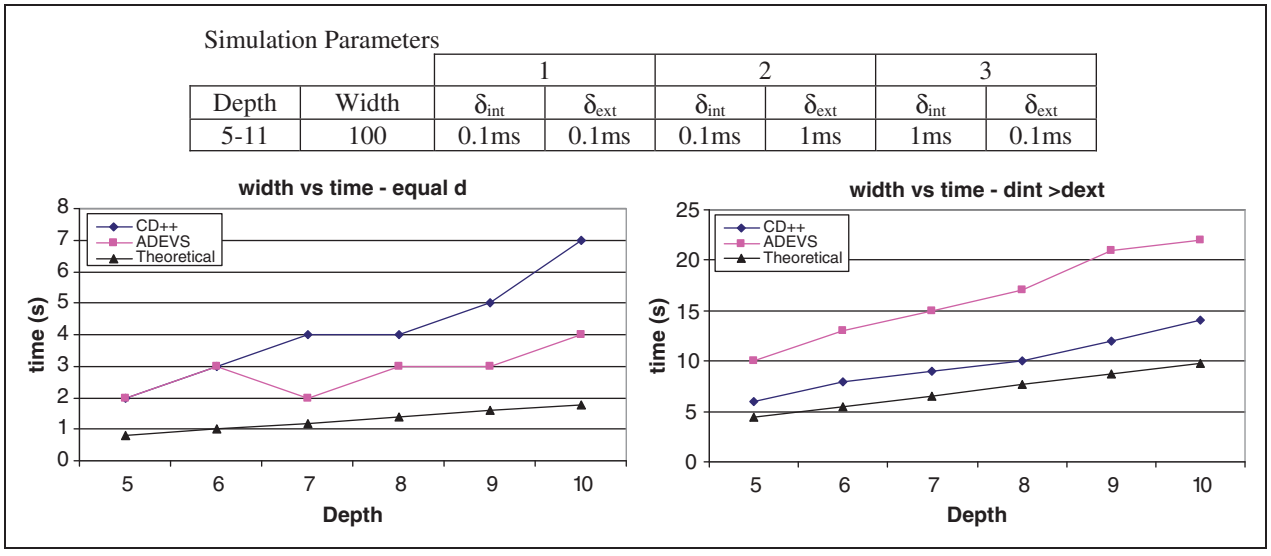

Figure 18. CD++ vs. ADEVS. HI models: (a) $\delta_{int} = \delta_{ext}$; (b) $\delta_{int} > \delta_{ext}$.

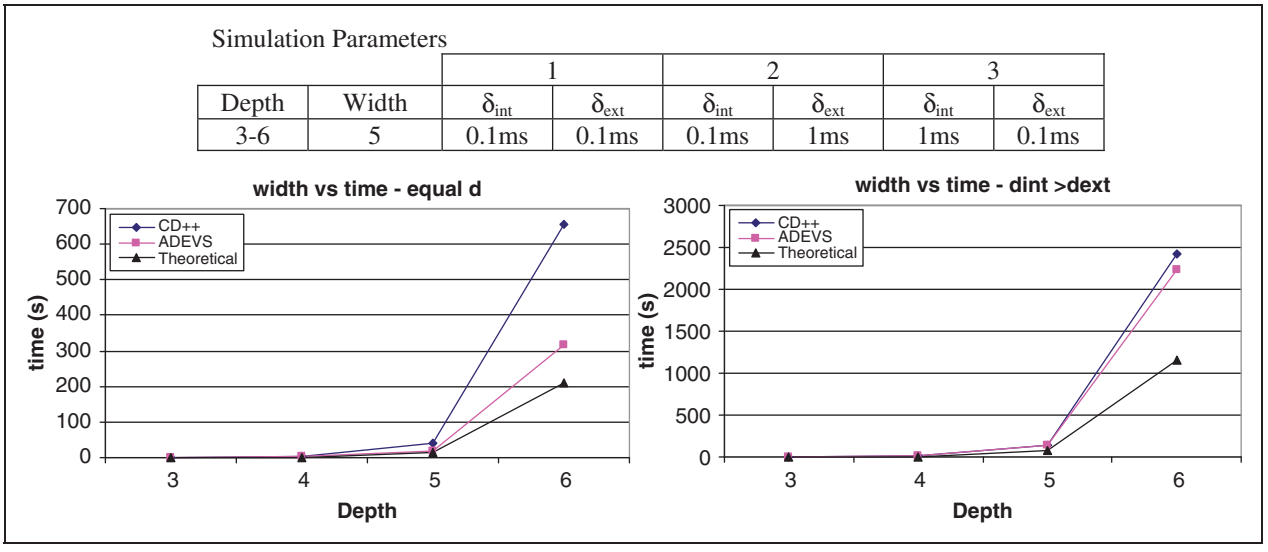

Figure 19. CD++ vs. ADEVS. HOmod models: (a) $\delta_{int} = \delta_{ext}$; (b) $\delta_{int} > \delta_{ext}$.

for this is that CD++ uses a temporary file to store intermediate results, whereas ADEVS stores them in memory.

We also executed a similar experiment in which we changed the depth of the levels and kept the width constant. The same three different sets of internal and external transition functions were used. For variable depth HI models with constant width the following simulation parameters used were as shown in Figure 18.

In Figure 18 we can see that when the times spent in transition functions are equal ADEVS executes faster than CD++ (as we saw before). Nevertheless, whenever the time spent in internal transitions is larger than the external transition, CD++ surpasses the performance of ADEVS.

For HOmod models, the parameters need to be changed, due to the exponential growth of messages between coupled components, however the time given to each transition function is kept equal. Following the previous test scenarios the test was repeated varying the depth this time, the parameters of the simulation were as shown in Figure 19.

In this last test, CD++ and ADEVS present a very close performance for models where the internal transition function is greater than the external transition function. For the rest of the cases ADEVS performs better than CD++, although depending on the model

and complexity of it. Along with this trend, CD++ offers equivalent performance for models that consume equal time in the transition functions as well as models that have larger loads in the external transition function. The main reason for this difference is that CD++ searches the model coupling every time a message is routed. This is flexible and it allows dynamic changes; it also permits having a smaller data structure in memory; nevertheless, the parsing tree used for this coupling causes the simulation to run slower than ADEVS, where the coupling is fixed (and resolved at compile time; any changes in the model requires a complete recompilation of the model). Therefore, in this particular case, ADEVS outperforms CD++, which pays the cost of the internal interpreted language for coupled model and Cell-DEVS definition (two facilities that ADEVS does not count with).

A summary of the conclusions gathered through the realization of the comparative benchmarking analysis is given as follows:

- Based on the results obtained by DEVStone we demonstrated that the benchmark proves to be flexible enough to create and analyze different scenarios and compare different implementations of DEVS-based simulators.
- It is possible to classify simulators by their performance, for example, CD++ shows better performance than ADEVS when it comes to models that deal with intensive code in the internal transition function.
- ADEVS takes advantage of the tight integration of the model and the simulator at run time, although the price is paid at compilation time; it also offers less flexibility compared to CD++ when it comes to model development and modification.

The DEVStone benchmark allows us to have a performance-measuring tool that allows us to compare different types of implementations of the DEVS formalism. Future developers of CD++ can measure the performance of their upgraded version versus the original or even the performance of CD++ against ADEVS. There is no doubt that the particular implementations of the DEVStone benchmark for both CD++ and ADEVS can be improved for code efficiency or for simulation performance depending on what the end-user requires.

7. Conclusions

Evaluating the performance of a simulation tool is typically a tedious and complex process, which requires the execution of a wide variety of models with different characteristics. Our main goal was to provide a means of evaluating the efficiency of existing simulation engines with focus on DEVS-based tools, and facilitating a qualitative and objective comparison of different tools.

To do so, we defined and developed DEVStone, a synthetic model generator that supports the process of evaluating the performance of simulation engines. DEVStone relies on executing a collection of models with different characteristics. In order to emulate several degrees of complexity in their structures, we identified four types of models that correspond to three interconnection patterns. In addition, each atomic component usually executes code in its transition functions; we proposed the use of Dhrystone code to mimic the task to be performed by these components. DEVStone makes it possible to: (i) create models with different sizes, shapes, and behavior; (ii) generate an arbitrarily large number of such models; and (iii) execute those models using the simulator(s) under study. After defining a DEVStone atomic model, the remaining tasks can be done automatically.

Our framework provides a common metric to compare the results that were obtained using the different simulation tools, and also enables an analysis of the efficiency of successive versions of the same simulator, such as upgrades or fixes. We used the CD++ toolkit to show how to apply the proposed benchmark; also, we performed a comparison benchmark against ADEVS. Although we restricted our case study to the existing CD++ and ADEVS simulation engines, the same ideas may hold for other DEVS-based simulators. Using DEVStone, we showed that hierarchical simulation techniques are able to simulate models with low overhead, even for models with complex structure. By means of the proposed framework, the performance of both virtual-time and real-time hierarchical simulators was shown to be satisfactory. Moreover, the results demonstrated that the flat simulation technique could improve the efficiency in some cases, especially when the model structure is particularly large or complex. Regardless of the size and complexity of the models, the flat simulator outperformed the hierarchical one. In general, the charts illustrate that the overhead incurred by the flat simulator is reduced by up to about 55% of the overhead incurred by the hierarchical approach. In any of these cases, the hierarchical structure of the DEVS models is maintained unchanged (and only the simulation engines are improved to generate speedups in the simulation).

DEVStone thus provides a systematic way to assess the performance of a simulation engine, reducing the time required to measure its efficiency. It is possible to analyze the efficiency of any DEVS simulator with relation to the size, the behavior, and the structure of the model under execution. An example with the ADEVS

simulator was presented. A precise performance characterization of a simulator allows modelers to consider the actual overhead of the tool based on solid results, and then analyze the feasibility of executing timed models with specific timing constraints.

The results presented here show how the benchmark can be used for studying this kind of application (and there is a variety of different modifications and improvements that users can introduce based on the ideas presented here). The main contributions and scope of this proposal can be summarized as follows:

(i) Prior to this work, there were no synthetic benchmarks for studying M&S software. This is the first successful attempt in defining and implementing one, and in showing how to use it in different contexts.
(ii) This is the first objective- and application-independent mechanism proposed to evaluate M&S environments, which can be employed in different contexts (providing a fair method for comparing simulation engines).
(iii) The benchmark is flexible and can be easily adapted, as has been shown throughout the article. DEVStone also provides numerous mechanisms for testing different options, and the user can easily modify it to create the next generation of synthetic benchmarks for M&S with ease. This opens a new field in the community, which should decide what other important things should be included in such a benchmark (some ideas are discussed at the end of this section).
(iv) DEVStone was used to execute very complex simulations (some of the most complex simulations included over 350,000 components). The atomic model's behavior is synthetic and based on the execution of complex Dhrystone code, instead of simple sets of instructions that could be optimized by a compiler. Finally, the models can be combined into a variety of coupling schemes. The tests can include varied parameters for each of the components in an automated fashion.
(v) These complex simulations were used to carry out the first existing comparative analysis between different DEVS M&S tools.

The use of DEVStone allowed us to address the misconception that hierarchical M&S environments (such as those used for defining DEVS models) can have performance issues derived from the hierarchical structure of the models involved in the simulation. Although DEVStone provides a flexible and generic method, the scope is limited to the kind of models presented in this work (for instance, DEVStone has not been built with the aim of testing the performance of tools for cellular automata, Petri nets, FSM, etc.). Nevertheless, the benchmark is defined using a flexible approach, allowing the users to focus on the aspects they need to improve, opening the doors for other similar research in related areas. An expert in a given field could use the basic ideas and design to adapt the benchmark to their own kind of environment. For instance, in the case of modeling hierarchical applications, the user can use DEVStone to test different implementations easily, and then use the most adequate for their needs. The model structure defined provide a mechanism to easily automate the model creation, and the computation of the theoretical results; this structure could be modified and expanded with ease (as we did with the HOmod models; HOmod models did not exist in earlier versions of the benchmark, and including this new kind of model was done in a straightforward manner). These ideas could be used to define more complex scenarios (for instance, to define cases with feedback in the couplings by including such definitions in the coupled models). It could also be expanded to contain various kinds of atomic models (for instance, other models with a larger number of input/output ports, others with varied time advance functions, etc.). In any case, the benchmark should be standardized and expended by a community of users. The benchmark should be complemented by a combination of sample models to be used as a standardized performance measure. This article provides the grounds for the formation of such a community.

Acknowledgement

This work has been partially funded by NSERC.

Conflict of interest statement

None declared

References

1. Zeigler B, Kim T and Praehofer H. *Theory of Modeling and Simulation*, 2nd edn. New York: Academic Press, 2000.
2. El-Osery A, Burge J, Jamshidi M, Fathi M and Akbarzadeh M-R. V-LAB: A virtual laboratory for autonomous agents - SLA based controllers. *IEEE Syst Man Cybernet* 2002; 32: 791–803.
3. Barros F and Ball GL. Fire modeling using dynamic structure cellular automata. In: *III International Conference on Forest Fire Research*, *III International Conference On Forest Fire Research*, 14*th Conference on Fire and Forest Meteorology*, 1998, Vol. I, 879–888.
4. Liu Q and Wainer G. Parallel environment for DEVS and Cell-DEVS models. *SIMULATION* 2007; 83: 449–471.
5. Filippi J-B, Morandini F, Balbi J and Hill DRC. *SIMULATION* 2010; 86: 629–646.

6. Wainer G. Applying Cell-DEVS methodology for modeling the environment. *SIMULATION* 2006; 82: 635–660.
7. Cunning SJ, Schulz S and Rozenblit JW. An embedded system's design verification using object-oriented simulation. *SIMULATION* 1999; 72: 238–249.
8. Kim J-K, Kim YG and Kim TG. DHMIF: DEVS-based hardware model interchange format. In: *Proceedings of the European Simulation Symposium*, Marseille, France, 2001.
9. Lee J and Chi S. Using symbolic DEVS simulation to generate optimal traffic signal timings. *SIMULATION* 2005; 81: 153–170.
10. Wainer G. Developing a software tool for urban traffic modeling. *Software Pract Exper* 2007; 37: 1377–1404.
11. Gambardella LM, Rizzoli AE and Zaffalon M. Simulation and planning of an intermodal container terminal. *SIMULATION* 1998; 71: 107–116.
12. Pérez E, Ntaimo L, Bailey C and McCormack P. Modeling and simulation of nuclear medicine patient service management in DEVS. *SIMULATION* 2010; 86: 481–501.
13. Byon E, Pérez E, Ding Y and Ntaimo L. Simulation of wind farm operations and maintenance using discrete event system specification. *SIMULATION* 2010; in press.
14. Daicz S, Troccoli A and Wainer G. Experiences in modeling and simulation of computer architectures using DEVS. *SIMULATION* 2001; 18: 179–202.
15. Chen Y and Sarjoughian HS. A component-based simulator for MIPS32 processors. *SIMULATION* 2010; 86: 271–290.
16. Zeigler BP, Song H, Kim T and Praehofer H. DEVS framework for modeling, simulation, analysis, and design of hybrid systems. In: *Hybrid Systems II (Lecture Notes in Computer Science*. Vol. 999*)*, Berlin: Springer, 1995, 529–551.
17. Glinsky E and Wainer G. Performance analysis of real-time DEVS models. In: *Proceedings of the SCS Winter Simulation Conference*, San Diego, CA, 2002.
18. Wainer G. CD++: A toolkit to develop DEVS models. *Software Pract Exper* 2002; 32: 1261–1306.
19. Nutaro J. ADEVS website, http://www.ornl.gov/~1qn/adevs/index.html (accessed 1 October 2010).
20. Wainer G, Jafer S and Liu Q. Advanced parallel simulation of DEVS models in CD++. In: Wainer G and Mosterman P (eds) *Discrete-Event Modeling and Simulation: Theory and Applications*. London: Taylor and Francis, 2010.
21. Moallemi M and Wainer G. Designing an interface for real-time and embedded DEVS. In: *Proceedings of* 2010 *Symposium on Theory of Modeling and Simulation*, *DEVS'*10, Orlando, FL, 2010.
22. Díaz A, Vázquez V and Wainer G. Application of the ATLAS language in models of urban traffic. In: *Proceedings of the Annual Simulation Symposium*, Seattle, WA, 2001.
23. Ameghino J, Troccoli A and Wainer G. Models of complex physical systems using Cell-DEVS. In: *Proceedings of the* 34*rd Annual Simulation Symposium*, Seattle, WA. 2001.
24. Ameghino J, Wainer G and Glinsky E. Applying Cell-DEVS in models of complex systems. In: *Proceedings of the SCS Summer Computer Simulation Conference*, Montreal, Canada, 2003.
25. Wainer G. *Discrete-Event Modeling and Simulation: A Practitioner's Approach*. Boca raton, FL: CRC Press, 2009.
26. Mittal S, Risco-Martín JL and Zeigler BP. DEVS/SOA: A cross-platform framework for net-centric modeling and simulation in DEVSUnified process. *SIMULATION* 2009; 85: 419–450.
27. Sarjoughian HS and Zeigler BP. DEVSJAVA: Basis for a DEVS-based collaborative M&S environment. In: *Proceedings of the SCS International Conference on Web-based Modeling and Simulation*, San Diego, CA, 1998.
28. Kim TG. DEVSim++: C++ based simulation with hierarchical modular DEVS models. *User's Manual CORE Lab*. Taejon, Korea: EE Dept, KAIST, 1994.
29. Himmelspach J, Ewald R and Uhrmacher AM. A flexible and scalable experimentation layer for JAMES II. In: *Proceedings of the Winter Simulation Conference*, Miami, FL. 2008.
30. Filippi J-B, Bernardi F and Delhom M. The JDEVS environmental modeling and simulation environment. In: *Proceedings of the the IEMSS'*02 *Conference on Integrated Assessment and Decision Support*, Lugano, Switzerland, 2002.
31. Wainer G, Al-Zoubi K, Mittal S, Risco-Martín JL, Sarjoughian H and Zeigler BP. An Introduction to DEVS standardization. In: Wainer G and Mosterman P (eds) *Discrete-Event Modeling and Simulation: Theory and Applications*. London: Taylor and Francis, 2010.
32. Barros FJ. Modeling and Simulation of dynamic structure heterogeneous flow systems. *SIMULATION* 2002; 78: 18–27.
33. Sun Y and Hu X. Performance measurement of dynamic structure DEVS for large-scale cellular space models. *SIMULATION* 2009; 85: 335–351.
34. Copp JB. *The COST Simulation Benchmark*. *Description and Simulator Manual*. Luxembourg: Office for Official Publications of the European Communities.
35. Knight P, Corder A, Liedel J, Gidens J, Drake R, Jenkins C, et al. Evaluation of run time infrastructure (RTI) Implementations. In: *The Huntsville Simulation Conference*, Huntsville, AL, 2002.
36. Troccoli A and Wainer G. Performance analysis of cellular models with parallel Cell-DEVS. In: *Proceedings of the SCS Summer Computer Simulation Conference*, Florida, 2001.
37. Breitenecker F, Wassertheurer S, Popper N and Zauner G. Benchmarking of simulation systems—the ARGESIM comparisons. In: *First Asia International Conference on Modeling and Simulation*, Pukhet, Thailand, 2007.
38. Weicker RP. Dhrystone: A synthetic systems programming benchmark. *Communications of the ACM* 1984; 27: 1013–1030.
39. Chow A and Ziegler B. Parallel DEVS: A parallel, hierarchical, modular modeling formalism. In: *Proceedings of*

the SCS Winter Simulation Conference, Orlando, FL, 1994.
40. Martin D, McBrayer T, Radhakrishan R and Wilsey P. *Time Warp Parallel Discrete Event Simulator*. Technical report, Computer Architecture Design Laboratory, University of Cincinnati, 1997.
41. Kim K, Kang W, Sagong B and Seo H. Efficient distributed simulation of hierarchical DEVS models: transforming model structure into a non-hierarchical one. In: *Proceedings of the 33rd Annual Simulation Symposium*, Washington, DC, 2000.
42. Liu J. *Real-time Systems*. Englewood Cliffs, NJ: Prentice-Hall, 2000.

Gabriel Wainer, SMSCS, SMIEEE, received the MSc (1993) and PhD degrees (1998, with highest honors) from the University of Buenos Aires, Argentina, and Université d'Aix-Marseille III, France. In July 2000, he joined the Department of Systems and Computer Engineering, Carleton University (Ottawa, ON, Canada), where he is now an Associate Professor. He is the author of three books and over 220 research articles, has edited four other books, and has helped to organize over 100 conferences, including being one of the founders of the ICST SIMUTools Conferences. He is the Vice-President Publications, and was a member of the Board of Directors of The Society for Computer Simulation International (SCS). He is Special Issues Editor of the *Transactions of the Society for Computer Simulation International* (SCS), member of the Editorial Board of *Wireless Networks* (Elsevier), *Journal of Defense Modeling and Simulation* (Sage), and the *International Journal of Simulation and Process Modeling* (Interscience). He has been the recipient of various awards, including the IBM Eclipse Innovation Award, a Leadership award by the Society for Modeling and Simulation International, various Best Paper awards. He has also been awarded Carleton University's Research Achievement Award (2005–2006) and the First Bernard P. Zeigler DEVS Modeling and Simulation Award.

Ezequiel Glinsky received his MASc (2004) in Electrical Engineering from Carleton University (Canada) with a Senate Medal for Outstanding Academic Achievement. Previously, he received a BSc (2000) and MSc (2002) from the Universidad de Buenos Aires (Argentina), where he has served as Research Assistant and Lecturer. He has also worked as a Lecturer in several universities in Argentina. His areas of research are real-time and parallel simulation using DEVS. He has worked in Microsoft Argentina since 2006, currently as Business Group Lead for Servers and Tools. Before joining Microsoft, he led development teams in Latin America in different companies.

Marcelo Gutierrez-Alcaraz became a Professional Engineer in 2003 in Bolivia after receiving a degree in Electrical Engineering from the Universidad Mayor the San Andres. He worked in automation and control systems in different projects in Bolivia and was the co-founder and CEO of MPH Co., a start-up company that builds specialized equipment and provides maintenance in Power Electronics. In 2007 he obtained the degree of MASc in Electrical Engineering at Carleton University in Ottawa, Canada; during his postgraduate studies he was a Scholar of the Organization of American States (OAS) and LASPAU. Since July 2007 he is a PhD student working with MasterVolt B.V. and TU Delft in the design and development of a highly integrated Generator Set for Maritime and Mobile Applications.