Simulation

Simulation: Transactions of the Society for Modeling and Simulation International
1–18
© The Author(s) 2017
DOI: 10.1177/0037549717690447
journals.sagepub.com/home/sim

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

José L. Risco-Martín[1], Saurabh Mittal[2], Juan Carlos Fabero Jiménez[1], Marina Zapater[1] and Román Hermida Correa[1]

Abstract
The discrete event system specification formalism, which supports hierarchical and modular model composition, has been widely used to understand, analyze and develop a variety of systems. Discrete event system specification has been implemented in various languages and platforms over the years. The DEVStone benchmark was conceived to generate a set of models with varied structure and behavior, and to automate the evaluation of the performance of discrete event system specification-based simulators. However, DEVStone is still in a preliminary phase and more model analysis is required. In this paper, we revisit DEVStone introducing new equations to compute the number of events triggered. We also introduce a new benchmark with a similar central processing unit and memory requirements to the most complex benchmark in DEVStone, but with an easier implementation and with it being more manageable analytically. Finally, we compare both the performance and memory footprint of five different discrete event system specification simulators in two different hardware platforms.

Keywords
Discrete event system specification, DEVStone, synthetic benchmarks, performance, memory usage

1 Introduction

In the last four decades, various modeling and simulation (M&S) methodologies have provided excellent approaches to solve problems. Among them, one of the M&S techniques that has gained popularity is the discrete event system specification (DEVS): a sound, formal definition, based on theoretical concepts of generic dynamic systems, which supports efficient event based simulation, verification, and validation.[1,2]

DEVS divides the system into basic models called atomic models and composite models called coupled models. Atomic models define the behavior of the system, whereas coupled models specify the structure. We can distinguish between classic DEVS and parallel DEVS (PDEVS).[2] In classic DEVS, when two or more models are scheduled for state transitions at the same time, one of the models is chosen according to a *select* function provided in the coupled model specification. PDEVS is an extension of classic DEVS which allows all of the imminent components to be activated and to send their output to other components. Removing the *select* function and adding a new *confluent* transition function, PDEVS introduces the possibility of managing simultaneous events in a natural manner.

DEVS has been successfully used for modeling a wide range of application domains. For example, it has been used in urban traffic analysis,[3] logistics and supply chains,[4] computer architectures,[5,6] embedded system designs,[7] unmanned aerial vehicles,[8] decision support systems,[9] etc. Because of the ease of model definition, model composition, reuse, and hierarchical coupling, DEVS has always been successfully applied in a variety of applications.

In contrast to time-stepped discrete time simulation, DEVS advances time through the concept of minimum

[1]Department of Computer Architecture and Automation, Complutense University of Madrid, Spain
[2]Complex Systems Research Lab, Dunip Technologies, LLC, USA

Corresponding author:
José L. Risco-Martín, Department of Computer Architecture and Automation, Complutense University of Madrid, C/Prof José García Santesmases 9, 28040 Madrid, Spain.
Email: jlrisco@ucm.es

time to the next event, thereby advancing time asynchronously and achieving a significant speedup over the former method.[2] As a result, the DEVS formalism has been implemented in major object-oriented programming languages, like Lisp, Scheme, C++, Java, Python, and SmallTalk, leading to many DEVS simulation engines across the globe, like DEVSJAVA,[10] DEVS-Suite,[11] COSMOS,[12] CD++,[13] PyPDEVS,[14] aDEVS,[15] JAMES-II,[16] DEVSim++,[17] and xDEVS,[18] to name but a few.

This variety of simulation engines has generated an extensive study on the DEVS performance, commonly focused on particular application domains. However, after several years of research, a final version of a discrete event simulation benchmark was published, named DEVStone.[19–21] DEVStone can be used to automatically generate a vast variety of models with different shapes and sizes. These models can then be simulated to test different features with respect to the corresponding simulator.

These benchmarks incorporate several benefits but some of them suffer from shortcomings in their mathematical descriptions, like the formal computation of the total number of events triggered. In this paper, we reconsider these benchmarks. Firstly, we include the computation of the total number of events triggered inside each benchmark. Secondly, we fix some equations that in the work by Wainer et al. did not give the exact number of transitions. In[21], these Equations are (2), (3) and (4).[21] It is worthwhile to mention that these errors have not affected the reliability of the previous papers results, since these models have been always used to compare wall-clock execution times. Finally, we define an additional benchmark, which demands the same computational effort as the more complex model in DEVStone, but is analytically more manageable, as is demonstrated in the research work of this paper.

The remainder of this paper is organized as follows: we show a brief description of the DEVS formalism and introduce several DEVS simulation engines in Section 2. The DEVStone benchmark is revisited in Section 3, including all of the contributions to the benchmark performed in this work. In Section 4 we describe our experimental infrastructure and methodology. In Section 5 we present experimental results, including a comparison of up to five simulators and more than 1400 DEVStone models. Finally, we present conclusions in Section 6.

2 DEVS: Formalism and simulation engines

2.1 The discrete event system specification

DEVS is a general formalism for discrete event system modeling based on set theory.[2] The DEVS formalism provides the framework for information modeling which gives several advantages to analyze and design complex systems: completeness, verifiability, extensibility, and maintainability. Once a system is described in terms of the DEVS theory, it can be easily implemented using an existing computational library. As stated in Section 1, the PDEVS approach was introduced, after 15 years, as a revision of classic DEVS. Currently, PDEVS is the prevalent DEVS, implemented in many libraries. In the following, unless it is explicitly noted, the use of DEVS implies PDEVS.

DEVS enables the representation of a system by three sets and five functions: input set (X), output set (Y), state set (S), external transition function (δ_{ext}), internal transition function (δ_{int}), confluent function (δ_{con}), output function (λ), and time advance function (ta).

DEVS models are of two types: atomic and coupled. Atomic models are directly expressed in the DEVS formalism specified above. Atomic DEVS processes input events based on their model's current state and condition, generates output events and transitions to the next state. The coupled model is the aggregation/composition of two or more atomic and coupled models connected by explicit couplings. Particularly, an atomic model is defined by the following equation:

$$A = \langle X, Y, S, \delta_{\text{ext}}, \delta_{\text{int}}, \delta_{\text{con}}, \lambda, ta \rangle \tag{1}$$

where:

1. X is the set of inputs described in terms of pairs port-value: $\{p \in IPorts, v \in X_p\}$;
2. Y is the set of outputs, also described in terms of pairs port-value: $\{p \in OPorts, v \in Y_p\}$;
3. S is the set of sequential states;
4. $\delta_{\text{ext}} : Q \times X^b \to S$ is the external transition function. It is automatically executed when an external event arrives at one of the input ports, changing the current state if needed:
   - $Q = (s, e) s \in S, 0 \leqslant e \leqslant ta(s)$ is the total state set, where e is the time elapsed since the last transition;
   - X^b is the set of bags over elements in X;
5. $\delta_{\text{int}} : S \to S$ is the internal transition function. It is executed right after the output (λ) function and is used to change the state S;
6. $\delta_{\text{con}} : Q \times X^b \to S$ is the confluent function, subject to $\delta_{\text{con}}(s, ta(s), \emptyset) = \delta_{\text{int}}(s)$. This transition decides the next state in cases of collision between external and internal events, i.e., an external event is received and the elapsed time is equal to the time - advance. Typically, $\delta_{\text{con}}(s, ta(s), x) = \delta_{\text{ext}}(\delta_{\text{int}}(s), 0, x)$;
7. $\lambda : S \to Y^b$ is the output function. Y^b is the set of bags over elements in Y. When the time elapsed

since the last output function is equal to $ta(s)$, then λ is automatically executed;

8. $ta(s) : S \rightarrow \Re_0 + \cup \infty$ is the time advance function.

The formal definition of a coupled model is described as:

$$M = \langle X, Y, C, EIC, EOC, IC \rangle \quad (2)$$

where:

- X is the set of inputs described in terms of pairs port-value: $\{p \in IPorts, v \in X_p\}$;
- Y is the set of outputs, also described in terms of pairs port-value: $\{p \in OPorts, v \in Y_p\}$;
- C is a set of DEVS component models (atomic or coupled); note that C makes this definition recursive;
- EIC is the external input coupling relation, from external inputs of M to component inputs of C;
- EOC is the external output coupling relation, from component outputs of C to external outputs of M;
- IC is the internal coupling relation, from component outputs of $c_i \in C$ to component outputs of $c_j \in C$, provided that $i \neq j$.

Given the recursive definition of M, a coupled model can itself be a part of a component in a larger coupled model system giving rise to a hierarchical DEVS model construction.

2.2 DEVS simulation engines

In the last decade, many DEVS M&S engines have come into existence. All of them offer a programmer-friendly application programming interface (API) to define new models using a high level language. However, only a few of them provide a user-friendly graphical user interface (GUI) for model specification. In the following, we describe some of the most referenced DEVS M&S simulation frameworks.

2.2.1 DEVSJAVA. DEVSJAVA has been developed by Bernard P Zeigler (University of Arizona, USA) and Hessam Sarjoughian (Arizona State University, USA).[10] It is written in Java and supports virtual time, real time, and sequential and parallel execution. The definition of new models is performed through an API. Several M&S tools have been defined around DEVSJAVA (GUIs for results visualization, GUIs for models definition, etc.), as DEVSJAVA is one of the primary DEVS M&S reference simulators in the community.

2.2.2 DEVS-Suite and CoSMoS. DEVS-Suite is a simulator built based on the PDEVS formalism. This software provides a library of examples proving some experimental concepts. It also incorporates simulation visualization techniques consisting of displaying the static structure of models, animation of models, and run-time viewing of time-based trajectories.[11] CoSMoS (component-based system modeling and simulation) is a framework aimed at integrated visual model development, model configuration and automatic data collection simulation.[12] The CoSMoS environment supports component-based modeling with direct support for DEVS formalism and extensible markup language (XML) schemas. DEVS-Suite's core is largely DEVSJAVA. It is bundled within the CoSMoS distribution and thus enables both modeling and simulation of PDEVS models.

2.2.3 CD++. CD++ has been developed by Gabriel Wainer and his students (Carleton University, Canada; Universidad de Buenos Aires, Argentina). Written in C++, it allows the definition of DEVS and Cell-DEVS models graphically. These models are also defined using an API. CD++ supports virtual and real time, as well as sequential, parallel, and distributed simulations.[13]

2.2.4 PythonPDEVS. PythonPDEVS (PyPDEVS) implements both the classic DEVS and PDEVS in the Python language, with a matching simulator.[14] Models are defined through the provided API, allowing the execution of virtual time or real time simulations. The latest release of PyPDEVS is focused on improving the performance, mainly because Python is an interpreted language. To this end, several schedulers have been defined, obtaining good performance metrics.

2.2.5 aDEVS. aDEVS (a discrete event system simulator) is a C++ library for constructing discrete event simulations based on the PDEVS and dynamic DEVS (dynDEVS) formalisms.[15] Developed by Jim Nutaro, it allows the implementation of both sequential and parallel simulations using the provided C++ API. This tool has usually displayed the best performance.

2.2.6 JAMES-II. Developed at the University of Rostock, the Java-based multipurpose environment for simulation II (JAMES II) provides support for many different formalisms, including various variants of DEVS. Besides an API to define models, this framework also provides a GUI to configure experiments and check simulation results. This simulation engine supports sequential and parallel execution.[16]

2.2.7 DEVSim++:. Developed by Tag Gon Kim and his group at the Korea Advanced Institute of Technology

(KAIST),[17] this is a C++ based engine and is used extensively for large simulations focusing on wargaming and simulation interoperability.

2.2.8 xDEVS. xDEVS (cross-platform DEVS) engine is Java-based and is released under the GNU public license (GPL). This facilitates the rapid development of new components and extensions, and a wide adoption of the core engine. xDEVS provides the user with a set of base classes that can be used to develop new DEVS models, or to develop new DEVS simulation engines. It is based on the fundamental separation of the model and the underlying corresponding simulator,[2] and rightly so, provides, the modeling API and the simulation API.[18] It is made available as a standalone executable jar and as an Eclipse plugin.

2.2.9 Others. In addition to the above DEVS implementations used widely, there are others with selective adoption such as GALATEA (glider with autonomous, logic-based agents, temporal reasoning and abduction) for multi-agent systems (MAS),[22] SimStudio,[23] PowerDEVS for hybrid systems,[24] MS4Me based on DEVSJAVA,[25] and last but not the least, virtual laboratory environment (VLE),[26] that is based on C++ and is a multiparadigm environment based on several DEVS extensions.

We have selected five well-known DEVS simulation frameworks distributed among three implementation languages and compared their performance against a revisited DEVStone benchmark. The current diversity on the programming languages used is concentrated on C++, JAVA, and Python. As a consequence, we have selected two JAVA-based simulators (DEVSJAVA and xDEVS), two C++-based simulators (aDEVS and CD++) and PyPDEVS as the Python-based simulation engine. In the following, we describe the revisited DEVStone benchmark.

3 DEVStone

DEVStone is a synthetic benchmark that has been used in recent years to evaluate the performance of different DEVS simulators.[21,27,28] DEVStone can be used to automatically generate a vast variety of models with different shapes and sizes. These models can then be simulated to test different features with respect to the corresponding simulator. A DEVStone benchmark is defined with five parameters.

1. *Type*. Different structure and interconnection schemes between the components in the model.
2. *Width*. This parameter is based on the number of components in each intermediate coupled model.
3. *Depth*. The number of levels in the model hierarchy.
4. *Internal transition time*. The execution time spent by each internal transition function.
5. *External transition time*. The execution time spent by each external transition function.

According to the DEVStone specifications, both the internal and external transition function times are spent executing Dhrystones to keep the central processing unit (CPU) busy.[29]

In the work by Wainer et al.,[21] four different DEVStone benchmarks were presented; low level of interconnections (named LI), high input couplings (named HI), HI model with numerous outputs (named HO), and HOmod, deriving different equations to compute the number of external and internal transition functions.

In the following a formal definition of the DEVStone atomic model is introduced. Next, all five of the benchmarks considered in this work (LI, HI, HO, HOmod and the newly introduced HOmem) are presented. To simplify the computation of the total number of events triggered, it is assumed that:

1. The execution time spent by the external or internal transition function is equal to 0 s, i.e. the transition is instantaneous in a computational sense.
2. all of the events injected to the DEVStone benchmarks are separated in time by more than 0 s.

3.1 DEVStone atomic model

The atomic model of DEVStone can be defined as shown in Algorithm 1.

3.2 LI models

Figure 1 shows the general structure of an LI model. With d layers (depth), the first $d-1$ (with $d \geqslant 1$) layers have the structure of Figure 1(a). All of these layers have one coupled model and $w-1$ (with $w \geqslant 2$) atomic models (where w is the width). On the other hand, the d-th layer has the structure given in Figure 1(b), just with one atomic model. The arrows in the figure represent the connection between the input and output ports in the whole model.

As stated above, two metrics are measured: the execution time and the memory footprint (also known as memory high-water mark). Obviously, these two metrics depend on the number of atomic models, number of internal transitions, number of external transitions, and the total number of events internally generated. Additionally, the memory footprint depends on the concurrency of the model, that is, the number of pending events simultaneously waiting at the input ports.

Since the model structure is known, and the simplification $\Delta_{\text{int}} = \Delta_{\text{ext}} = 0$ is made, the theoretical execution time and the total number of events generated can be easily computed.

Algorithm 1 DEVStone atomic model.

```
Require: NUM_DELT_INTS, NUM_DELT_EXTS and
    NUM_OF_EVENTS are global variables, and store the total
    number of internal transition functions, external transition
    functions, and events triggered inside the whole model. $\Delta_{int}$
    and $\Delta_{ext}$ are the delays introduced in the internal and
    external transition functions, respectively.
  function [list,$phase$,$\sigma$] = init()
  list = [] {list is part of the state, and stores all the
  events received by this atomic model}
  $\sigma = \infty$
  function [list,$phase$,$\sigma$] = $\delta_{int}$(list,$phase$,$\sigma$)
  NUM_DELT_INTS = NUM_DELT_INTS + 1
  Dhrystone($\Delta_{int}$)
  list = []
  $\sigma = \infty$
  function [list,$phase$, $\sigma$] = $\delta_{ext}$(list,$phase$,$\sigma$,e, $X^b$)
  NUM_DELT_EXTS = NUM_DELT_EXTS + 1
  Dhrystone($\Delta_{ext}$)
  values = $X^b(in)${$X^b(in)$ is a list containing all the events
  waiting in the "in" input port}
  NUM_OF_EVENTS = NUM_OF_EVENTS + values.size()
  list = [list;values] {Concatenate both lists}
  $phase$ = "$active$"
  $\sigma = 0$
  function [list,$phase$,$\sigma$] = $\delta_{con}$(list,$phase$,$\sigma$,$ta(s)$,$X^b$)
  $\delta_{ext}$($\delta_{int}$(list,$phase$,$\sigma$),0,$X^b$)
  function $\lambda()$
  send("out", list) {sends the whole list by the "out" output
  port}
  function $\sigma$ = $ta$(list,$phase$,$\sigma$)
  $\sigma = \sigma$
```

Firstly, considering the model's $d - 1$ levels with $w - 1$ atomic models and one level with one atomic model, the total number of atomic models is as follows:

$$\#\text{Atomic} = (w - 1) \cdot (d - 1) + 1 \quad (3)$$

Secondly, LI models produce one external transition, output event and internal transition for each atomic model and external events injected. Thus, in LI models, the number of transitions and events generated is equal to the number of atomic models multiplied by the total number of external events injected N, as follows:

$$\#\delta_{\text{int}} = N \cdot ((w - 1) \cdot (d - 1) + 1) \quad (4)$$

$$\#\delta_{\text{ext}} = N \cdot ((w - 1) \cdot (d - 1) + 1) \quad (5)$$

$$\#\text{Events} = N \cdot ((w - 1) \cdot (d - 1) + 1) \quad (6)$$

In the following DEVStone benchmarks, we derive the equations for the number of transition functions and events internally generated given a single external event injected, i.e., for this benchmark this is given as follows:

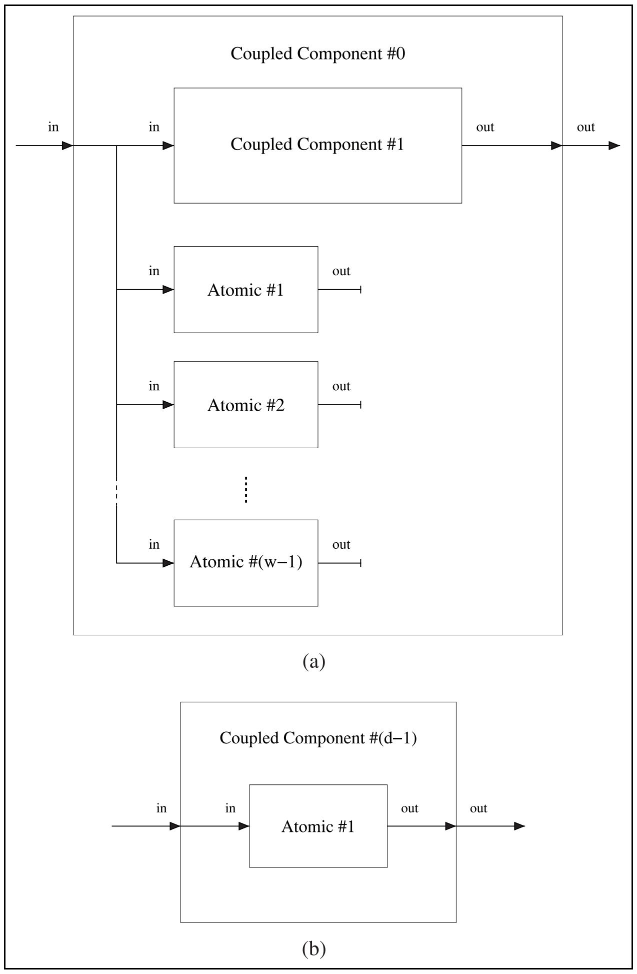

Figure 1. DEVStone LI components. (a) DEVStone LI regular coupled component and (b) DEVStone LI deepest coupled component.

$$\#\delta_{\text{int}} = (w - 1) \cdot (d - 1) + 1 \quad (7)$$

$$\#\delta_{\text{ext}} = (w - 1) \cdot (d - 1) + 1 \quad (8)$$

$$\#\text{Events} = (w - 1) \cdot (d - 1) + 1 \quad (9)$$

3.3 HI models

Figure 2 shows the general structure of a HI model. It is equal to the LI model, but where the output port of an atomic component i is connected to the input port of the next atomic component $i + 1$, as seen in Figure 2(a).

Therefore, the number of atomic models is equal to the LI model. However, the number of transition functions and events generated are quite different, because for each external input, the set of $w - 1$ atomic models acts as a shift register, generating one additional event for each external event. As a result, the number of atomic models, transition functions and events generated are computed as follows:

$$\#\text{Atomic} = (w-1)\cdot(d-1)+1 \tag{10}$$

$$\begin{aligned}\#\delta_{\text{int}} &= \left((w-1)+\sum_{i=1}^{w-2} i\right)\cdot(d-1)+1\\ &= \left(\frac{w^2-w}{2}\right)\cdot(d-1)+1\end{aligned} \tag{11}$$

$$\begin{aligned}\#\delta_{\text{ext}} &= \left((w-1)+\sum_{i=1}^{w-2} i\right)\cdot(d-1)+1\\ &= \left(\frac{w^2-w}{2}\right)\cdot(d-1)+1\end{aligned} \tag{12}$$

$$\begin{aligned}\#\text{Events} &= \left((w-1)+\sum_{i=1}^{w-2} i\right)\cdot(d-1)+1\\ &= \left(\frac{w^2-w}{2}\right)\cdot(d-1)+1\end{aligned} \tag{13}$$

3.4 HO models

Figure 3 shows the general structure of a HO model. The HO model has a more complex interconnection map with the same number of atomic and coupled components. For example, HO coupled components have two input and two output ports in each level. The main differences compared to the HI model are that the second input port of each coupled model is connected to the input of each atomic model. Additionally, the output of each atomic model is connected to the second output of its parent coupled model.

It is worthwhile to mention that the number of atomic models, transition functions, and events generated in HO models are exactly the same as in the HI model. However, the main difference is in both the execution time and the memory footprint, which are higher due to the additional external input connections. Thus, we have the following:

$$\#\text{Atomic} = (w-1)\cdot(d-1)+1 \tag{14}$$

$$\#\delta_{\text{int}} = \left((w-1)+\sum_{i=1}^{w-2} i\right)\cdot(d-1)+1 \tag{15}$$

$$\#\delta_{\text{ext}} = \left((w-1)+\sum_{i=1}^{w-2} i\right)\cdot(d-1)+1 \tag{16}$$

$$\#\text{Events} = \left((w-1)+\sum_{i=1}^{w-2} i\right)\cdot(d-1)+1 \tag{17}$$

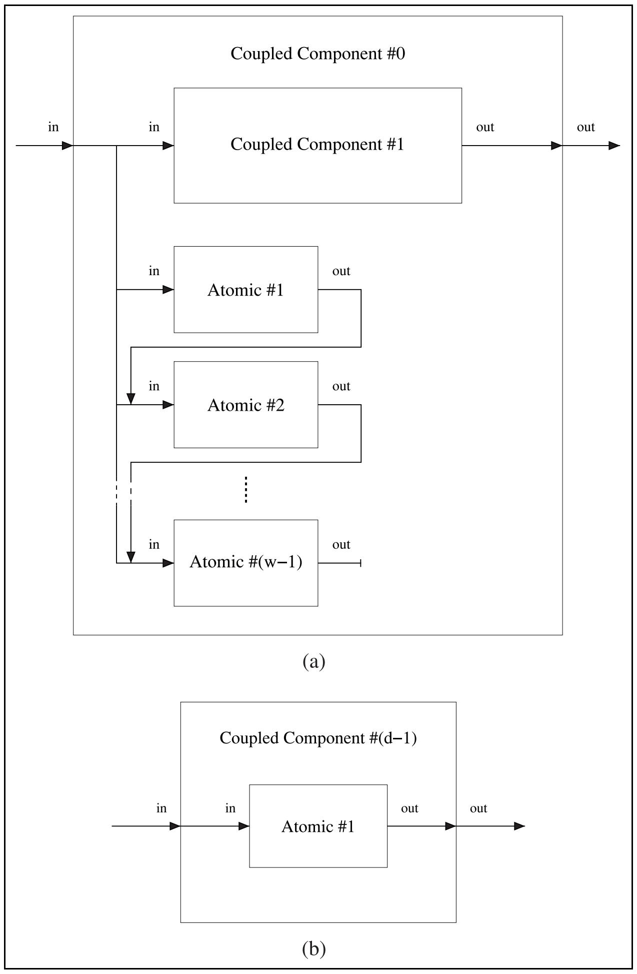

Figure 2. DEVStone HI components. (a) DEVStone HI regular coupled component and (b) DEVStone HI deepest coupled component.

3.5 HOmod models

Figure 4 depicts the structure of a HOmod DEVStone model. As usual, the deepest coupled model is formed by one single atomic model. The remaining coupled models are constituted of one coupled model, a chain of $w-1$ atomic models and a set of $k = 1\ldots w-1$ chains formed by $\sum_{i=1}^{k} i$ atomic models. The second external input port is connected to the whole first row and only to the first atomic component in the remaining rows. Additionally, all the atomic models in the second row are connected to the first row, which in turn sends the whole output directly to the coupled component. Finally, each remaining atomic component is connected to its upper component.

The computation of the number of atomic modes is quite straightforward, as follows:

$$\#\text{Atomic} = \left((w-1)+\sum_{i=1}^{w-1} i\right)\cdot(d-1)+1 \tag{18}$$

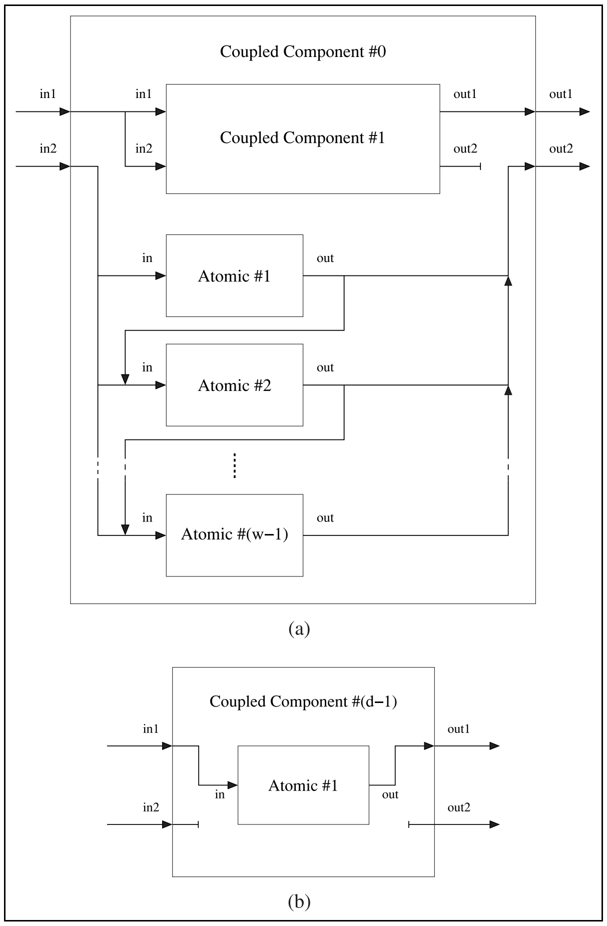

Figure 3. DEVStone HO components. (a) DEVStone HO regular coupled component and (b) DEVStone HO deepest coupled component.

However, the calculation of the number of transition functions is hard. After an exhaustive mathematical analysis we have determined that:

$$\#\text{Atomic} = \left((w-1) + \sum_{i=1}^{w-1} i\right) \cdot (d-1) + 1 \tag{19}$$

$$\begin{aligned}\#\delta_{\text{int}} = {} & (d-1)\cdot(w-1)^2 + \\ & + \left((d-1) + (w-1)\cdot\sum_{i=1}^{d-2} i\right) \\ & \times \left((w-1) + \sum_{i=1}^{w-1} i\right) + 1\end{aligned} \tag{20}$$

$$\begin{aligned}\#\delta_{\text{ext}} = {} & (d-1)\cdot(w-1)^2 + \left((d-1) + (w-1)\cdot\sum_{i=1}^{d-2} i\right) \\ & \times \left((w-1) + \sum_{i=1}^{w-1} i\right) + 1\end{aligned} \tag{21}$$

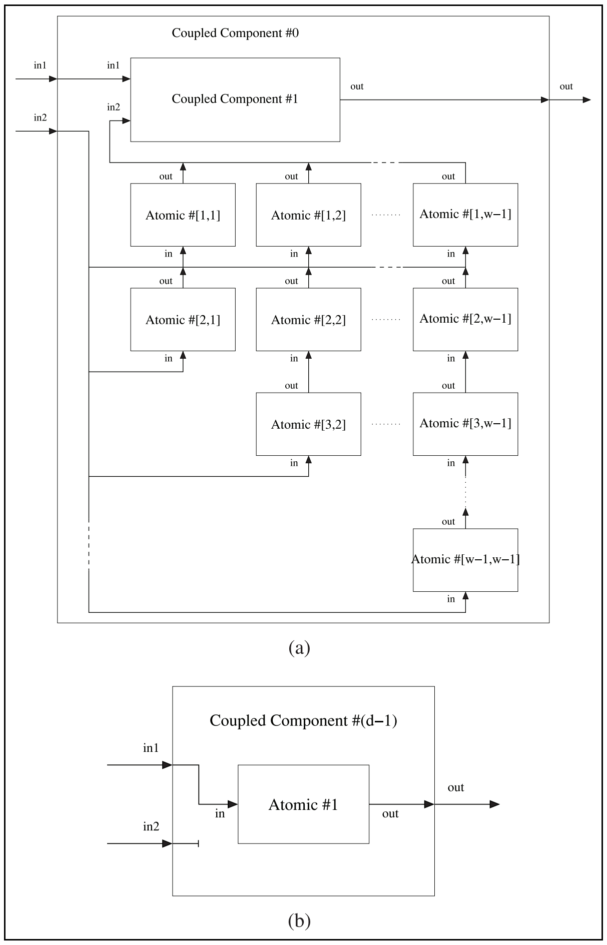

Figure 4. DEVStone HOmod components. (a) DEVStone HOmod regular coupled component and (b) DEVStone HOmod deepest coupled component.

Similarly, the computation of the number of events follows a recursive equation, which is defined as follows:

$$\begin{aligned}\#\text{Events} = {} & \sum_{l=1}^{d-1}\Big(\sum_{c=1}^{K_l+w-1}\Big(W_1 \times \sum_{i=1}^{w} {P_l}^{c-i+1} \\ & + \sum_{i=1}^{w} + \big(W_i \cdot {P_l}^{c-i+1}\big)\Big)\Big) + 1\end{aligned} \tag{22}$$

where

$$W_i = \begin{cases} w-i & \text{if } w-i \geqslant 0 \\ 0 & \text{otherwise}\end{cases} \tag{23}$$

$$K_l \quad = \quad \begin{cases} 1 & \text{if } l = 1 \\ K_{l-1} + W_1 & \text{if } l > 1\end{cases} \tag{24}$$

and

$$P_1^1 = 1 \tag{25}$$

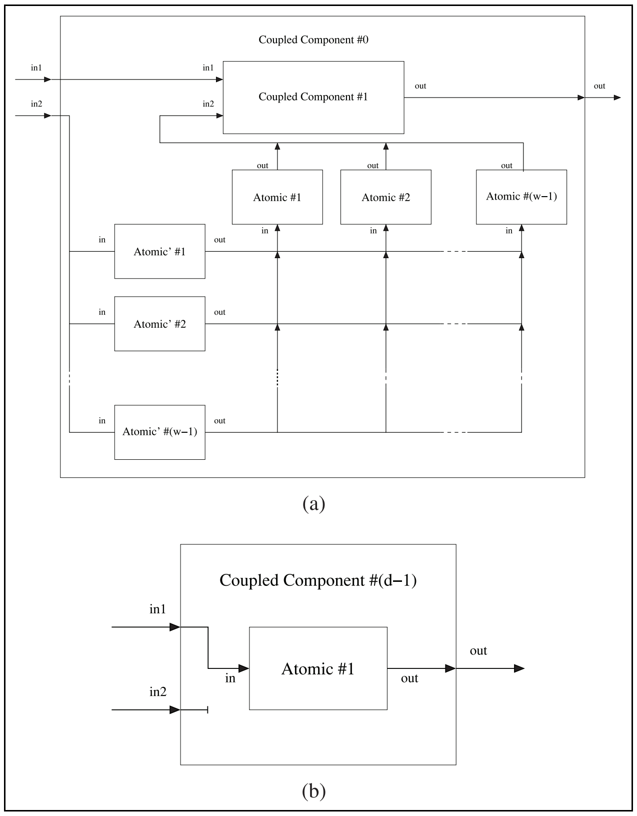

Figure 5. DEVStone HOmem components. (a) DEVStone HOmem regular coupled component and (b) DEVStone HOmem deepest coupled component.

$$P_l^j = 0 \text{ if } 1 > j > K_l \tag{26}$$

$$P_l^j = (w-1) \times \sum_{i=1}^{w} P_{l-1}^{j-i+1} \tag{27}$$

As can be seen, the complexity of the equations describing the metrics of the HOmod is high. The inclusion of these equations in a simulator is hard, and the theoretical analysis becomes prohibitive. For these reasons, we have defined a new DEVStone benchmark named HOmem that when providing the same computational effort as the HOmod into the different simulation frameworks, shows a straightforward mathematical formulation.

3.6 HOmem models

As stated above, we propose the inclusion of a new model in the DEVStone benchmark called HOmem. HOmem is proposed as a mechanism to increment the traffic of events with respect to the HO, equivalently to the HOmod, but with a simpler structure and mathematical description.

Figure 5 shows the structure of the HOmem DEVStone benchmark. As can be seen, the deepest coupled model is identical to the HOmod. As for the remaining coupled models, each one is formed by one coupled model and $2 \cdot (w-1)$ atomic models. The second $w-1$ chain receives the input through external input connections, and propagates these inputs to the first chain of $w-1$ atomic models. These, in turn, send all of the inputs received to the coupled model.

The number of transition functions are easy to compute, since it is equal to the number of atomic models. However, to calculate the number of events it must be taken into account that each single event is sent $w-1$ times to the whole second chain of atomic models. This grows exponentially with the depth of the model, in the following form:

$$\#\text{Atomic} = 2 \cdot (w-1) \cdot (d-1) + 1 \tag{28}$$

$$\#\delta_{\text{int}} = 2 \cdot (w-1) \cdot (d-1) + 1 \tag{29}$$

$$\#\delta_{\text{ext}} = 2 \cdot (w-1) \cdot (d-1) + 1 \tag{30}$$

$$\#\text{Events} = \sum_{l=1}^{d-1} \left((w-1)^{2 \cdot l} + (w-1)^{2 \cdot l - 1} \right) + 1 \tag{31}$$

Experimental results show that this straightforward specification leads to a similar execution time and memory footprint, when compared to the HOmod.

4 Experimental methodology

Once the DEVStone equations have been analytically derived, we compare the CPU execution time and memory footprint over a total of five well known simulation engines using all of the benchmarks presented above. Our aim is to show an exhaustive comparison and a standard procedure to evaluate the performance of any new discrete event simulator.

We first provide a detailed description of the experimental set-up used in this research.

All of the benchmarks presented above (LI, HI, HO, HOmod and HOmem) were executed using five simulation engines: aDEVS 2.8.1, CD++ 2.45 (a CD++ branch with support for PDEVS), DEVSJAVA 3.1, xDEVS 1.20151013 and PyPDEVS 2.2.4. Table 1 shows the programming language and the main data structures used in each simulation engine. As stated above, CD++ and aDEVS are C++ implementations, DEVSJAVA and xDEVS have been implemented using Java, whereas PyPDEVS is a Python simulation engine. As Table 1 shows, aDEVS and xDEVS use generic classes, whereas PyPDEVS uses duck typing. To store events, aDEVS and CD++ use standard C++ arrays. On the contrary, DEVSJAVA, xDEVS, and PyPDEVS use dynamic data structures, like linked lists or dictionaries. Finally,to store components and implement the simulation scheduler, all

Table 1. Main data structures used in the simulation engines.

	aDEVS	CD++	DEVSJAVA	xDEVS	PyPDEVS
Programming language	C++	C++	Java	Java	Python
Generics	Yes	No	No	Yes	Duck typing
Events container	array/port	array/port	Hashtable	LinkedList/port	dictionary
Components container	std::set	std::list	HashSet	LinkedList	list

Table 2. Configuration of the parameters.

Benchmark	Width			Depth			# Simulations
	Min.	Step	Max.	Min.	Step	Max.	
LI	2	100	1502	1	100	1501	25,600
HI	2	100	1102	1	100	1101	14,400
HO	2	100	1102	1	100	1101	14,400
HOmod	2	1	10	1	1	10	9000
HOmem	2	1	10	1	1	10	9000

of the frameworks use dynamic data structures such as sets or linked lists.

We tested all of these simulation engines in two different machines: a 48 GB AMD Opteron 6272 @ 2.1 GHz (abbreviated as AMD) and a 64 GB Intel Xeon 2670 @ 2.6GHz "Sandy Bridge" (abbreviated as Intel), in both cases under a GNU/Linux Debian 8 Operating System. aDEVS and CD++ were compiled using the `gcc -O3` optimization level.

In all of the test cases, only one external event was injected, generating the total number of transition functions and the total number of events given in the previous equations. As demonstrated in previous works,[21,27,28] the previous metric just scaled linearly with the number of external events.

Each benchmark type was generated for different values of width and depth. These values were defined for running different trials with all the five simulators. We looked for a good trade-off between the wall-clock simulation time and the memory footprint, since these are the metrics measured in all of the simulations. Table 2 shows these intervals, where each row represents a DEVStone benchmark type, in relation to the width and depth, each described by the minimum value, the step size, and the maximum value used to generate a full range for these parameters. For example, the smallest LI model is a 2×1 model, where width $= 2$ and depth $= 1$. The biggest model, on the other hand, is a 1502×1501 model.

Finally, each simulation is repeated 10 times for each simulator, benchmark, size, and hardware platforms. Simulation wall-clock times and the memory footprint are averaged over these 10 trials. Although no significant deviations were appreciated, we kept this number of trials to avoid spurious deviations. Table 2 shows in the last column the total number of simulations performed.

5 Results

5.1 CPU comparison

Table 3 shows a comparison between the execution time (in seconds, measured inside the simulator to avoid the loading time of the model) and the memory footprint (in GiB, measured using the general GNU time command) for the five simulation engines and the largest models of the DEVStone models tested in this work, i.e., LI 1502×1501, HI 1102×1101, and HO 1102×1101. HOmod 10×10 and HOmem 10×10 are not included because no simulator was able to finish them, at least during the 48 h in which we ran these tests. The same happened in all of the cases in Table 3 marked with ∞. As can be seen, only aDEVS and xDEVS were able to finish all of the models in Table 3, followed by CD++, which was not able to load the largest LI model. Regarding the memory footprint, there is not much difference between both servers. However, in terms of the execution time, the best server in almost all cases was the 64 GB Intel Xeon 2670 @ 2.6 GHz "Sandy Bridge", since between both servers, this one has the fastest processor and memory. Thus, simulation results are coherent with the server used, i.e., the faster the processor and the greater the memory size, the faster the simulation. The memory footprint is independent of the server, since it only depends on the internal structure of the DEVStone model.

As can be seen in Table 3, some simulators were not able to execute the model because the system was unable to handle the memory requirements. To tackle these issues

Table 3. Execution time (s) and memory footprint (GiB) of the larger models executed by the five simulation engines and in both the AMD and Intel servers.

Simulator	LI		HI		HO	
	AMD	Intel	AMD	Intel	AMD	Intel
aDEVS	2.5×10^0	$\mathbf{2.1 \times 10^0}$	1.0×10^3	1.0×10^3	1.2×10^3	1.2×10^3
	1.19	1.19	1.11	1.11	1.11	1.11
CD++	∞	∞	6.3×10^3	$\mathbf{5.1 \times 10^3}$	7.0×10^3	$\mathbf{4.5 \times 10^3}$
	∞	∞	3.46	3.46	3.69	3.69
DEVSJAVA	∞	∞	6.6×10^4	$\mathbf{4.0 \times 10^4}$	∞	∞
	∞	∞	4.23	4.21	∞	∞
xDEVS	3.8×10^0	$\mathbf{2.6 \times 10^0}$	9.3×10^2	$\mathbf{4.6 \times 10^2}$	1.0×10^3	$\mathbf{5.0 \times 10^2}$
	1.95	2.07	1.88	1.84	1.94	1.84
PyPDEVS	∞	∞	∞	∞	∞	∞
	∞	∞	∞	∞	∞	∞

in the remaining analysis, the wall clock execution time is limited to 1200 s and the memory footprint to 4 GiB, enough to perform more than 70,000 simulations in a reasonable amount of time, also obtaining significant values to compare. Thus, in the following, every experiment with time or memory greater than the aforementioned values is truncated to 1200 s or 4 GiB, respectively.

5.2 Execution time

Figure 6 shows the contour maps of the different execution times needed by all of the five simulators in both the LI and HI models. Blue regions mean low execution time, whereas red regions mean high execution time.

CD++, DEVSJAVA and PyPDEVS saturated the execution time of 1200 s multiple times in both models. aDEVS and xDEVS, on the contrary, reached the best results. Regarding the LI model, the ordered list of simulators, from best to worst contour maps is: aDEVS, xDEVS, CD++, DEVSJAVA ,and PyPDEVS. With respect to the HI model, the list is: xDEVS, aDEVS, CD++, DEVSJAVA, and PyPDEVS.

Continuing with this analysis, Figure 7 shows the same contour maps, this time in the HO and HOmem models.

Regarding the HO model, xDEVS obtained the best execution times, especially as the width and depth were increased. For low values of the width and depth, aDEVS was better than xDEVS. Once again, CD++, DEVAJAVA, and PyPDEVS saturated the execution time limit of 1200 s.

With respect to the HOmod and HOmem, all of the simulators reached the maximum execution time quite soon, with relatively small models. Moreover, in the case of the HOmod, only two simulators, aDEVS and xDEVS, were able to load all of the models in the memory, before the execution of the simulation. In fact, this is due to the intrinsic complexity of the HOmod benchmark, which includes many more atomic models than the HOmem. HOmem is simpler in structure than HOmod, and all of the simulation engines are able to load it. Once the simulation starts, the HOmod and HOmem offer similar execution times and memory footprints, as shown in Section 5.4.

We do not show a comparison between all of the simulators in the HOmod because only aDEVS and xDEVS were able to run a significant number of HOmod instances. These experiments are shown in the comparison between aDEVS and xDEVS.

5.3 Memory footprint

As mentioned before, the memory footprint is the memory high-water mark of a process. The comparison of all five of the simulators were performed constraining the execution time to 1200 s and the memory footprint to 4 GiB. The set of five simulators compared in this paper have been developed using different programming languages: aDEVS and CD++ in C++, DEVSJAVA and xDEVS in JAVA, and PyPDEVS in Python. Since JAVA and Python use their own virtual machines, it is expected that these simulators have a higher memory footprint. However, our experimental results showed some exceptions in this regard.

Figure 8 shows the memory footprint reached by the five simulators in the LI and HI models. As can be seen, DEVSJAVA and specially PyPDEVS reached the memory limit quite soon. aDEVS had by far the lowest memory usage. However, between CD++ and xDEVS, the latter obtained less memory footprint even when the Java virtual machine must be loaded into memory. This is because CD++ uses a complex structure to store the model, as is evident when CD++ is completely saturated once width and depth is greater than 1200.

Now, Figure 9 shows the memory footprint reached by the five simulators in the HO and HOmem models.

Regarding HO, the situation is almost identical to the HI model. aDEVS and xDEVS are still the two best simulators. With respect to HOmem, all five of the simulators reached the memory limit quite soon. As in the execution time analysis, DEVSJAVA was the first to leave the model, for a *width* greater than six. Surprisingly, PyPDEVS offered

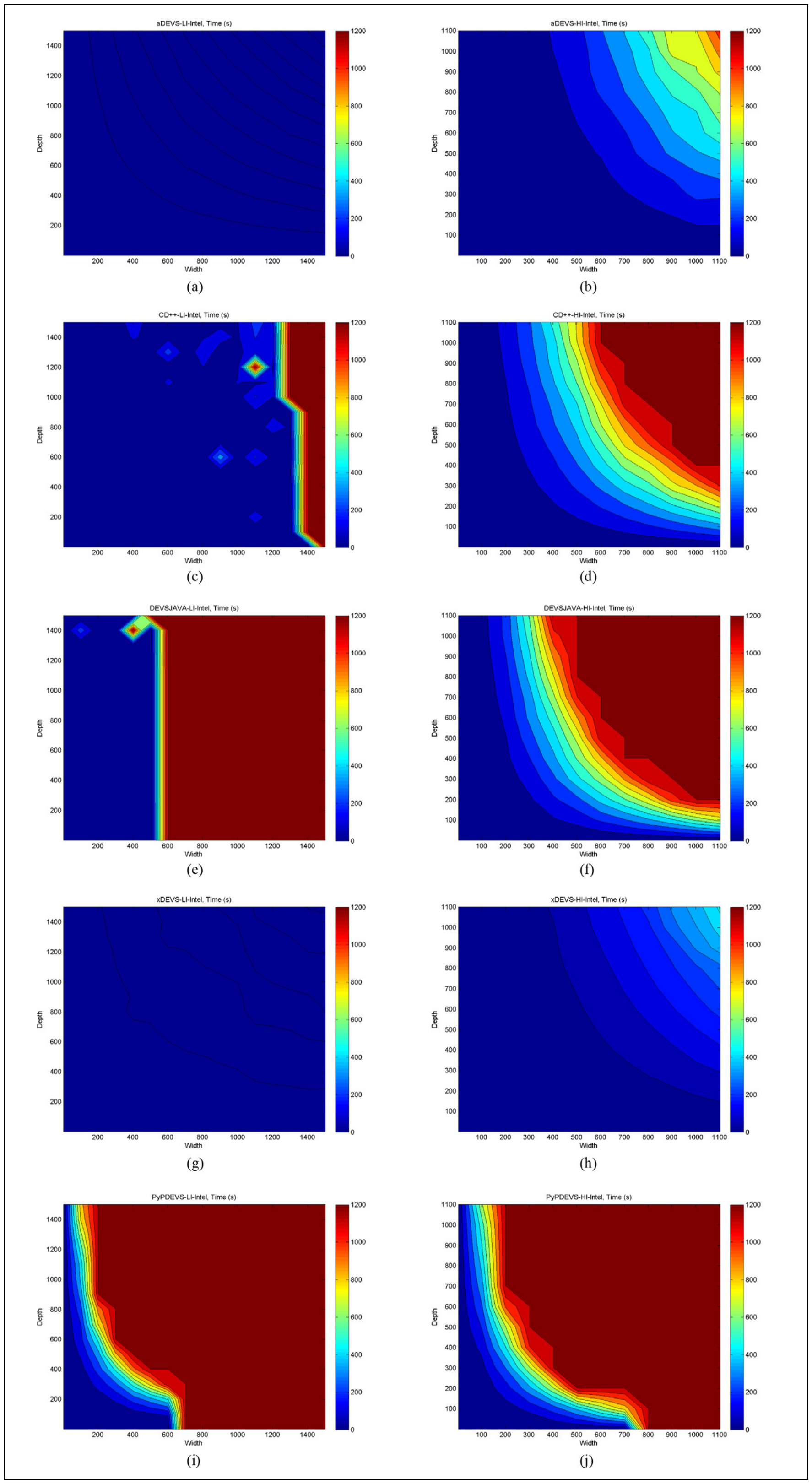

Figure 6. Execution time of LI and HI models. (a) aDEVS – LI, (b) aDEVS – HI, (c) CD++– LI, (d) CD++– HI, (e) DEVSJAVA – LI, (f) DEVSJAVA – HI, (g) xDEVS – LI, (h) xDEVS – HI, (i) PyPDEVS – LI, and (j) PyPDEVS – HI.

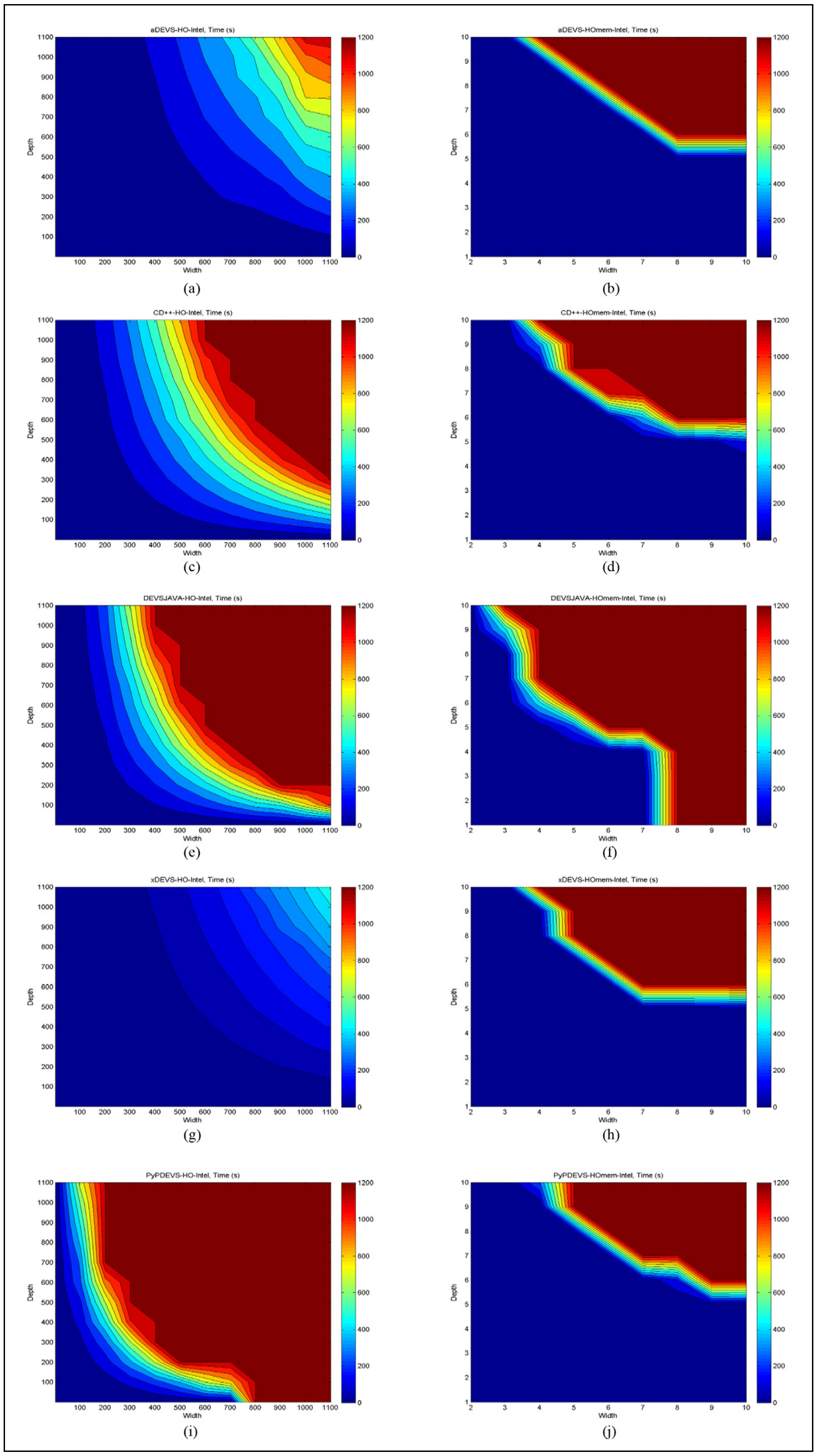

Figure 7. Execution time of HO and HOmem models. (a) aDEVS – HO, (b) aDEVS – HOmem, (c) CD++– HO, (d) CD++– HOmem, (e) DEVSJAVA – HO, (f) DEVSJAVA – HOmem, (g) xDEVS – HO, (h) xDEVS – HOmem, (i) PyPDEVS – HO, and (j) PyPDEVS – HOmem.

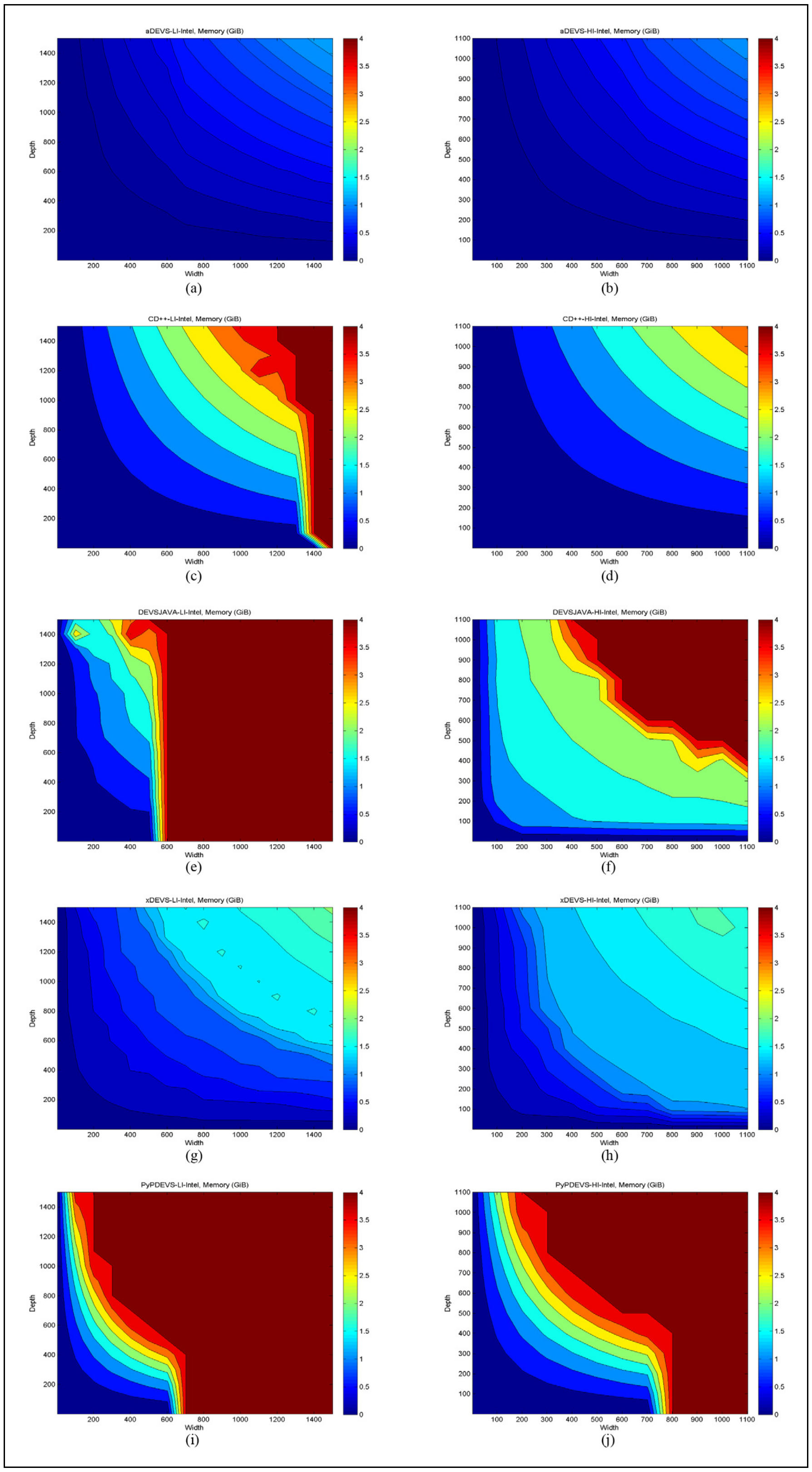

Figure 8. Memory footprint of LI and HI models. (a) aDEVS – LI, (b) aDEVS – HI, (c) CD++– LI, (d) CD++– HI, (e) DEVSJAVA – LI, (f) DEVSJAVA – HI, (g) xDEVS – LI, (h) xDEVS – HI, (i) PyPDEVS – LI, and (j) PyPDEVS – HI.

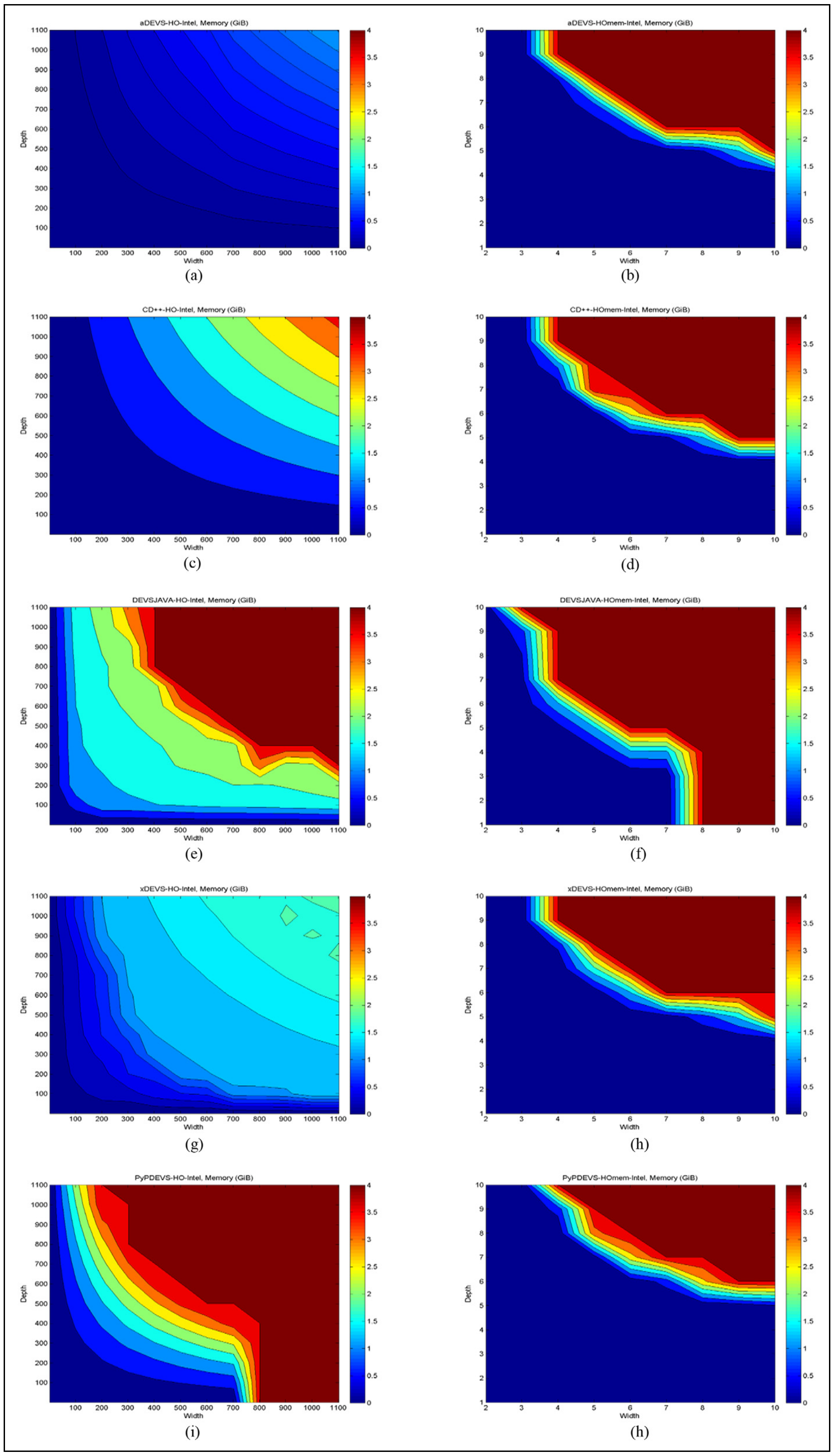

Figure 9. Memory footprint of the HO and HOmem models. (a) aDEVS – HO, (b) aDEVS – HOmem, (c) CD++– HO, (d) CD++– HOmem, (e) DEVSJAVA – HO, (f) DEVSJAVA – HOmem, (g) xDEVS – HO, (h) xDevs – HOmem, (i) PyPDEVS – HO, and (j) PyPDEVS – HOmem.

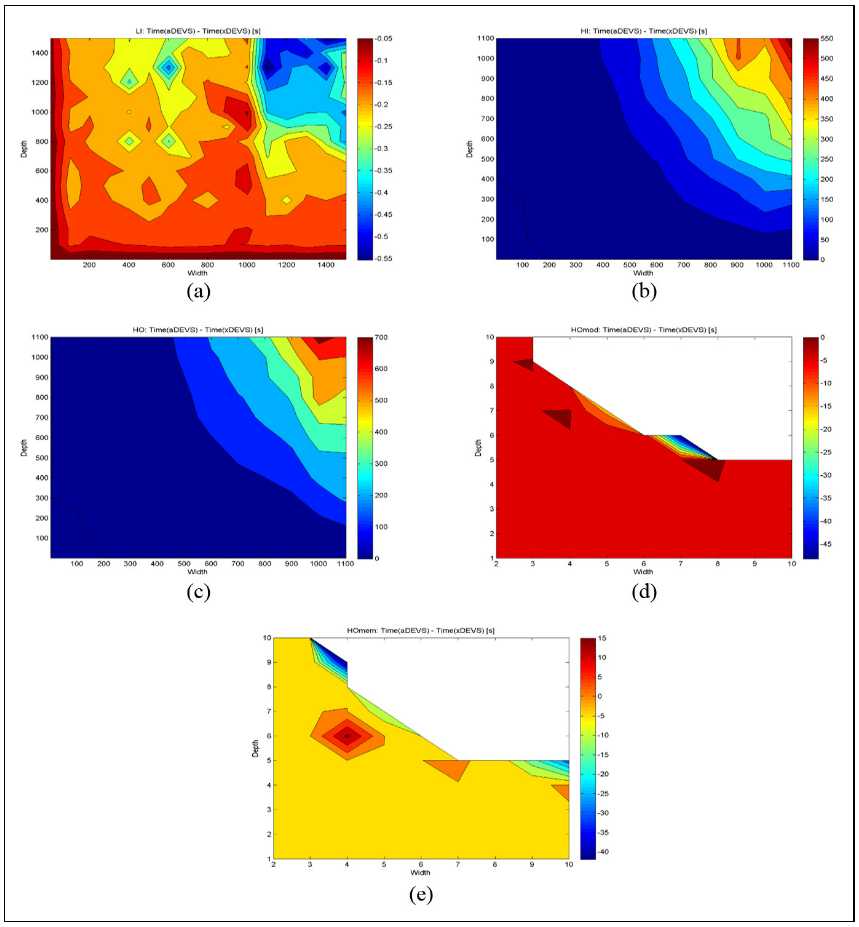

Figure 10. Execution time comparison of the LI, HI, HO, HOmod, and HOmem models computed as Time(xDEVS) – Time(aDEVS). (a) LI: xDEVS- - aDEVS, (b) HI: xDEVS – aDEVS, (c) HO: xDEVS – aDEVS, (d) HOmod: xDEVS – aDEVS, and (e) HOmem: xDEVS – aDEVS.

comparable results to aDEVS, CD++, and xDEVS in HOmem and HOmod, the more complex models.

As in the previous section, we do not show a comparison between all of the simulators in the HOmod because only aDEVS and xDEVS were able to run a significant number of HOmod instances.

As a conclusion, we may say that, regarding the memory footprint, aDEVS is by far the best DEVS simulator between those analyzed in this paper. In the case of the execution time, xDEVS is better as the complexity of the model increases, until the cases of HOmod and HOmem, where the complexity of both models cannot determine a classification with clarity. In the following, we investigate the performance of the aDEVS and xDEVS simulators in finer detail, as well as the similarities between the HOmod and HOmem.

5.4 Comparison between aDEVS and xDEVS

Firstly, we show the difference in the execution time and the memory footprint obtained by both simulators in the LI, HI, HO, HOmod, and HOmem models.

Figure 10 depicts five contour maps. Each one represents the difference in execution time of xDEVS minus aDEVS.

In the case of models with a lower complexity, like the LI model in Figure 10(a), the difference is small (2.5 s vs 2.1 s according to Table 3) and in favor of aDEVS. With respect to the HOmod and HOmem, the difference fundamentally varies from -10 s to 10 s, with more cases in favor of aDEVS. However, these two models remain indecisive since they show sparse maps.

The analysis of Figure 10(b) and (c) is much clearer. As the model complexity is increased, the difference is higher, in favor of xDEVS (up to 700 s faster in the case of HO).

We now compare both the simulator in the HOmod and the HOmem DEVStone model. aDEVS and xDEVS were the only two simulators that were able to simulate a significant number of HOmod models.

Figure 11 depicts both the execution time and the memory footprint reached by aDEVS and xDEVS in the HOmod and HOmem models. In both cases, contour maps are practically Yes/No maps, where, after a given *width* and *depth* both of the simulators immediately reach the

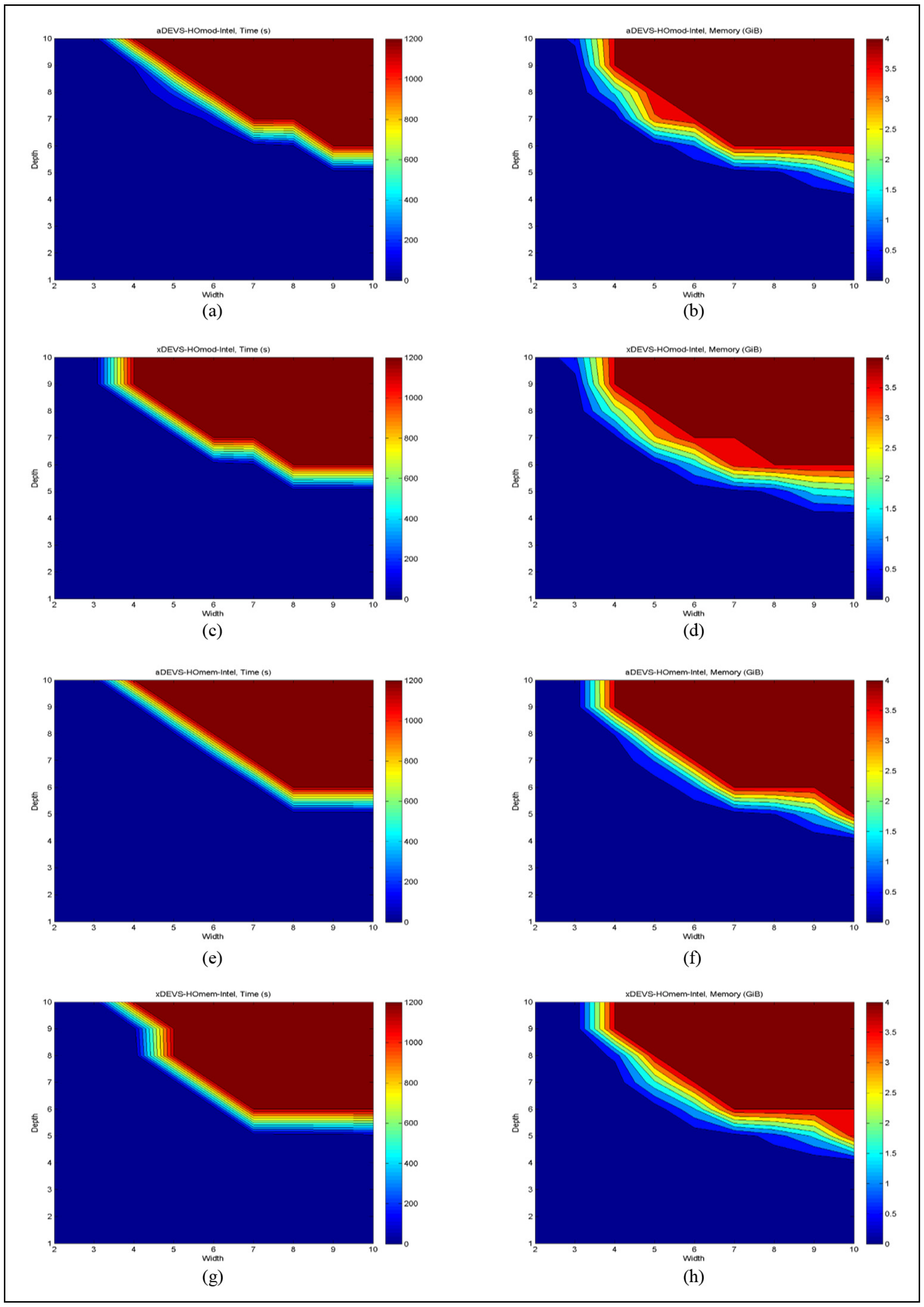

Figure 11. Execution times and memory footprints of HOmod and HOmem models given by aDEVS and xDEVS. (a) aDEVS – HOmod (time), (b) aDEVS – HOmod (memory), (c) xDEVS – HOmod (time), (d) xDEVS – HOmod (memory), (e) aDEVS – HOmem (time), (f) aDEVS – HOmem (memory), (g) xDEVS – HOmem (time), and (h) xDEVS – HOmem (memory).

limit in the execution time and memory footprint. These "saturation" values in the HOmod are reached "sooner" (in terms of w and d) than the corresponding values in the HOmem model. We prove here that the HOmem offers the same results compared to the HOmod with a more straightforward mathematical formulation, after a comparison of Equations (28) to (31) against Equations (18) to (27).

6 Conclusions

The DEVS formalism has been widely used to conceive, design, model, and develop a great variety of systems. DEVS has been implemented in various languages and platforms over the years. The DEVStone benchmark defines a set of models with varied structure and behavior, and was designed to evaluate the performance of DEVS-based simulators.

The key contributions of this work are the following. We have added a new model to the benchmark, called HOmem, which shows identical qualitative behavior to the HOmod but with a more manageable mathematical formulation. As with HOmod, HOmem is also intensive on both the execution time and memory usage. We have added the study of the memory footprint in DEVStone, deriving the equations needed to compute the number of events triggered inside the model and per each single injected external event. We have also recalculated the number of transition functions triggered in all of the DEVStone benchmarks. Finally, we have compared five simulation engines in two different hardware platforms, analyzing both the execution time and the memory footprint. To perform a fair comparison between simulation engines that allow and do not allow model flattening, we did not flattened the benchmark in any case.

These five DEVStone models are executed against five different DEVS simulators, implemented in different programming languages such as C++, JAVA, and Python.

Results show that all of the simulators were able to run the HOmem model for at least a significant range of *width* and *depth* values. Between all five of the simulators, aDEVS, which is based on C++, had the lowest memory footprint at least in the LI, HI, and HO models. With respect to the execution time, xDEVS was the fastest one, especially in the HI and HO models.

As future work, we propose the extension of this complete analysis to study the performance of the DEVStone parallel and distributed simulations.

Acknowledgements

The authors would like to thank Dr Gabriel Wainer and Dr Sixuan Wang for their assistance in the compilation of CD++ under GNU/Linux.

Funding

This work is supported by the Spanish Ministry of Economy and Competitivity (grant numbers TIN2014-54806-R and TIN2013-40968-P).

References

1. Zeigler BP. *Theory of modelling and simulation*. New York: John Wiley, 1976.
2. Zeigler BP, Praehofer H and Kim TG. *Theory of modeling and simulation. Integrating discrete event and continuous complex dynamic systems*. 2nd ed. San Diego, California, USA: Academic Press, 2000.
3. Wainer GA. Developing a software tool for urban traffic modeling. *Softw Pract Exp* 2007; 37: 1377–1404.
4. Byon E, Pérez E, Ding Y, et al. Simulation of wind farm operations and maintenance using discrete event system specification. *Simul T Soc Mod Sim* 2011; 87: 1093–1117.
5. Wainer GA, Daicz S and Troccoli A. Experiences in modeling and simulation of computer architectures using DEVS. *Simul T Soc Mod Sim* 2001; 18: 179–202.
6. Moreno A, Risco-Martín JL, Besada-Portas E, et al. Thermal analysis of the MIPS processor formulated within DEVS conventions. In: *Formal languages for computer simulation: Transdisciplinary models and applications*. IGI Global, Hershey, Pennsylvania, USA, 2013, pp.103–144.
7. Moallemi M and Wainer G. I-DEVS: Imprecise real-time and embedded DEVS modeling. In: *TMS-DEVS '11 proceedings of the 2011 symposium on theory of modeling & simulation: DEVS integrative M&S symposium*, Boston, MA, USA, 03–07 April, pp.95–102. San Diego, CA, USA: Society for Computer Simulation International.
8. Moreno A, Risco-Martín JL, Besada E, et al. DEVS/SOA: Towards DEVS interoperability in distributed M&S. In *13th IEEE/ACM international symposium on distributed simulation and real time applications*, 25–28 October 2009. Washington, D.C: IEEE Computer Society.
9. Perez E, Ntaimo L, Bailey C, et al. Modeling and simulation of nuclear medicine patient service management in DEVS. *Simulation* 2010; 86: 481–501.
10. DEVSJAVA, http://acims.asu.edu/software/devsjava (2015, accessed 1 July 2016).
11. DEVS-Suite, http://devs-suitesim.sourceforge.net (2015, accessed 1 July 2016).
12. CoSMoS. http://acims.asu.edu/software/cosmos (2015, accessed 1 July 2016).
13. CD++, http://cell-devs.sce.carleton.ca (2015, accessed 1 July 2016).
14. Tendeloo YV and Vangheluwe H. The modular architecture of the python(P)DEVS simulation kernel. In *Symposium on theory of modeling and simulation - DEVS integrative M&S symposium*, pp.1–6, Tampa, Florida, April 13–16, 2014. Society for Computer Simulation International San Diego, CA, USA.
15. aDEVS, http://web.ornl.gov/~1qn/adevs/ (2015, accessed 1 July 2016).
16. JAMES II http://wwwmosi.informatik.uni-rostock.de (2015, accessed 1 July 2016).
17. Kim TG, Sung CH, Hong SY, et al. DEVSim++ toolset for defense modeling and simulation and interoperation. *J Def Model Simul* 2011; 8: 129–142.
18. DUNIP Technologies, http://www.duniptechnologies.com (2015, accessed 1 July 2016).
19. Glinsky E and Wainer G. DEVStone: A benchmarking technique for studying performance of DEVS modeling simulation environments. In: *Ninth IEEE international symposium on distributed simulation and real-time applications*, Montreal, Canada, 10–11 October 2005, pp.265–272. Washington, DC, USA: IEEE Computer Society.
20. Gutierrez-Alcaraz M and Wainer GA. Experiences with the DEVStone benchmark. In: *Proceedings of the 2008 spring simulation multiconference*, Ottawa, Canada, 14–17 April 2008, pp.447–455. San Diego, CA, USA: Society for Computer Simulation International.

21. Wainer G, Glinsky E and Gutierrez-Alcaraz M. Studying performance of DEVS modeling and simulation environments using the DEVStone benchmark. *Simul T Soc Mod Sim* 2011; 87(7): 555–580.
22. Davila J and Uzcategui MY. GALATEA: A multi-agent, simulation platform. In: *International conference on modeling, simulation and neural networks (MSNN 2000)*, Mérida, Venezuela, 1–5 April 2000, pp.52–67 Berlin: AMSE.
23. Traoré MK. SimStudio: A next generation modeling simulation framework. In: *International ICST conference on simulation tools and techniques for communications, networks and systems*, Marseille, France, 03–07 March 2008, Brussels: ICST.
24. PowerDEVS, http://powerdevs.sourceforge.net (2015, accessed 1 July 2016).
25. MS4Me, http://www.ms4systems.com/pages/main.php (2015, accessed 1 July 2016).
26. VLE: The virtual laboratory environment, http://www.vle-project.org/wiki (2015, accessed 1 July 2016).
27. Van Tendeloo Y and Vangheluwe H. The modular architecture of the Python(P)DEVS simulation kernel: Work in progress paper. In: *Proceedings of the symposium on theory of modeling & simulation - DEVS integrative*, Tampa, Florida, 13–16 April 2014, pp.1–6. San Diego, CA, USA: Society for Computer Simulation International.
28. Vicino D, Niyonkuru D, Wainer G, et al. Sequential PDEVS architecture. In: *Symposium on theory of modeling and simulation* (TMS'15), Alexandria, Virginia, 12–15 April 2015, San Diego, CA, USA: Society for Computer Simulation International.
29. Weicker RP. Dhrystone: A synthetic systems programming benchmark. *Commun ACM* 1984; 27: 1013–1030.

Author Biographies

José L. Risco-Martín is Associate Professor at the Computer Architecture and Automation Department of Complutense University of Madrid (UCM), Spain. His research interests focus on design methodologies for integrated systems and high-performance embedded systems, including new modeling frameworks to explore thermal management techniques for Multi-Processor System-on-Chip, novel architectures for logic and memories in forthcoming nano-scale electronics, dynamic memory management and memory hierarchy optimizations for embedded systems, Networks-on-Chip interconnection design, and low-power design of embedded systems. jlrisco@ucm.es

Saurabh Mittal is the founder and president of Dunip Technologies, LLC, Virginia, USA. He is also affiliated with the Society of Computer Simulation (SCS) International, MITRE Corporation, and EABOK Consortium. He obtained his PhD in Electrical and Computer Engineering from the University of Arizona in 2007. His current research interests include modeling- and simulation-based systems engineering, netcentric complex adaptive systems, cyber physical system of systems, DEVS formalism and integrated M&S in Live, Virtual and Constructive (LVC) environments. Dr. Mittal is a recipient of US Department of Defense's highest civilian contractor recognition ''Golden Eagle'' award and SCS' Outstanding Service award. smittal@duniptech.com

Juan Carlos Fabero Jiménez received his Ph.D. degree from the Complutense University of Madrid, Spain, in 2005. He is currently Assistant Professor at the Department of Computer Architecture and Automation of the same university. His main research interests include networking and network simulations, routing protocols, IPv6 deployment and fault tolerance in reconfigurable hardware. jcfabero@ucm.es

Marina Zapater is Visiting Professor in the Computer Architecture Department at Universidad Complutense de Madrid, Spain. She received her Ph.D. degree in Electronic Engineering from Universidad Politécnica de Madrid in 2015, an M.Sc. in Telecommunication Engineering degree and a M.Sc. in Electronic Engineering degree, both from the Universitat PolitÒcnica de Catalunya, in 2010. She received a Ph.D. fellowship from the International Program for Attracting Talent (PICATA) of the Campus of International Excellence of Moncloa, Spain. Her research interests include proactive and reactive thermal and power optimization of complex heterogeneous systems, energy efficiency in data centers, ultra-low power architectures and embedded systems. In this area, she has co-authored over 25 publications in top-notch international conferences and journals, and she is currently involved in the H2020 project MANGO. She has served as TPC member of several conferences, including VLSI-SoC, MCSoC and SpringSim. She is a member of IEEE, CEDA and the HiPEAC network of excellence, from which she has received a HiPEAC collaboration grant. marina.zapater@ucm.es

Román Hermida Correa received a Ph.D. degree from the Complutense University of Madrid, Spain, in 1984. He is currently a Professor at the Department of Computer Architecture and Automation of the same university. His current research interests include design automation, computer architecture, reconfigurable computing, and embedded systems. Dr. Hermida has been actively involved in program committees of several international conferences, and served as the Program Chair of the International Symposium on System Synthesis in 2000, and General Chair of the same symposium in 2001. He received the 2002 IEEE VLSI Transactions best paper award. He has also served as Track Chair for CODES+ISSS and DATE. rhermida@ucm.es