Reconsidering Performance of DEVS Modeling and Simulation Environments Using the DEVStone Benchmark

José L. Risco-Martín[a,*], Saurabh Mittal[b], Juan Carlos Fabero Jiménez[a], Marina Zapater[a], Román Hermida Correa[a]

[a] *Department of Computer Architecture and Automation, Complutense University of Madrid, C/Prof. José García Santesmases 9, 28040 Madrid, Spain*
[b] *Dunip Technologies, LLC, Littleton, Colorado, USA*

Abstract

The Discrete Event System Specification formalism (DEVS), which supports hierarchical and modular model composition, has been widely used to understand, analyze and develop a variety of systems. DEVS 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 DEVS-based simulators. However, DEVStone is still in a preliminar 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, called HOmem, designed as an alternative version of HOmod, with similar CPU and memory requirements, but with an easier implementation and analytically more manageable. Finally, we compare both the performance and memory footprint of five different DEVS simulators in two different hardware platforms.

Keywords: DEVS, 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 generic dynamic systems theoretical concepts, 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

*Corresponding author
Email addresses: jlrisco@ucm.es (José L. Risco-Martín), smittal@duniptech.com (Saurabh Mittal), jcfabero@ucm.es (Juan Carlos Fabero Jiménez), marina.zapater@ucm.es (Marina Zapater), rhermida@ucm.es (Román Hermida Correa)

a *select* function provided in the coupled model specification. PDEVS is an extension of classic DEVS to allow all 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 to manage 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 chain [4], computer architecture [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 such a variety of applications.

In contrast to time-stepped discrete time simulation, DEVS advances time through the concept of minimum time to next event, thereby advancing time asynchronously and achieving 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, 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], xDEVS [18], to name but a few.

This variety of simulation engines has generated an extensive study in 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], [20], [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 [21] did not give the exact number of transitions, concretely equations (2), (3) and (4) in [21]. It is worthwhile to mention that these errors have not affected the reliability of previous paper results, since these models have been always used to compare wall-clock execution times. Finally, we define an additional benchmark, called HOmem, that demands the same computational effort than HOmod, the more complex model in DEVStone, but is analytically more manageable, as is demonstrated in the research work.

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 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 parallel DEVS (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 transition 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:

- X is the set of inputs described in terms of pairs port-value: $\left\{p \in IPorts, v \in X_p\right\}$.
- Y is the set of outputs, also described in terms of pairs port-value: $\left\{p \in OPorts, v \in Y_p\right\}$.
- S is the set of sequential states.

- $\delta_{\text{ext}} : Q \times X^b \rightarrow S$ is the external transition function. It is automatically executed when an external event arrives to one of the input ports, changing the current state if needed.
  - $Q = (s, e)s \in S, 0 \leq e \leq ta(s)$ 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.
- $\delta_{\text{int}} : S \rightarrow S$ is the internal transition function. It is executed right after the output (λ) function and is used to change the state S.
- $\delta_{\text{con}} : Q \times X^b \rightarrow S$ 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 elapsed time equals time-advance. Typically, $\delta_{\text{con}}(s, ta(s), x) = \delta_{\text{ext}}(\delta_{\text{int}}(s), 0, x)$.
- $\lambda : S \rightarrow Y^b$ 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.
- $ta(s) : S \rightarrow \mathfrak{R}_0^+ \cup \infty$ is the time advance function.

The formal definition of a coupled model is described as:

$$M = \langle X, Y, C, EIC, EOC, IC \rangle \tag{2}$$

where:

- X is the set of inputs described in terms of pairs port-value: $\left\{p \in IPorts, v \in X_p\right\}$.
- Y is the set of outputs, also described in terms of pairs port-value: $\left\{p \in OPorts, v \in Y_p\right\}$.
- 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, U.S.A.) and Hessam Sarjoughian (Arizona State University, U.S.A.) [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 Parallel DEVS formalism. This software provides a library of examples proving some experimental concepts. It also incorporates simulation visualization techniques consisting of displaying 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 XML Schema. DEVS-Suite's core is largely DEVSJAVA. It is bundled within the CoSMoS distribution and thus enables both modeling and simulation of Parallel DEVS 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. PyPDEVS

PythonPDEVS (a.k.a. PyPDEVS) implements both Classic and Parallel DEVS 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 Parallel DEVS 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 Korea Advanced Institute of Technology (KAIST) [17], this is a C++ based engine and used extensively for large simulations focusing on wargaming and simulation interoperability.

2.2.8. xDEVS

xDEVS engine is Java-based and is released under the GNU Public License (GPL). This facilitates the rapid development of new components and extensions, and 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 model and the underlying corresponding simulator [2] and rightly so, provides, the modeling Application Program Interface (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 [22] for Multi-Agent Systems (MAS), SimStudio [23], PowerDEVS [24] for hybrid systems, MS4Me based on DEVSJAVA [25] and last but not the least, Virtual Laboratory Environment (VLE) [26], that based on C++, 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 [21] 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:

Type: Different structure and interconnection schemes between the components in the model.

Width This parameter is based the number of components in each intermediate coupled model.

Depth The number of levels in the model hierarchy.

Internal transition time: The execution time spent by each internal transition function.

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 [29] to keep the CPU busy.

In [21], four different DEVStone benchmarks were presented (named LI, HI, 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 the five 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 seconds, i.e. the transition is instantaneous in a computational sense, and (2) all the events injected to the DEVStone benchmarks are separated in time by more than 0 seconds.

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. Δ_{int} and Δ_{ext} are the delays introduced in the internal and external transition functions, respectively.

function [list,*phase*,σ] = init()
list = [] {list is part of the state, and stores all the events received by this atomic model}
$\sigma = \infty$

function [list,*phase*,σ] = δ_{int}(list,*phase*,σ)
`NUM_DELT_INTS` = `NUM_DELT_INTS` + 1
Dhrystone(Δ_{int})
list = []
$\sigma = \infty$

function [list,*phase*,σ] = δ_{ext}(list,*phase*,σ,*e*,X^b)
`NUM_DELT_EXTS` = `NUM_DELT_EXTS` + 1
Dhrystone(Δ_{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*,σ] = δ_{con}(list,*phase*,σ,*ta*(*s*),X^b)
$\delta_{ext}(\delta_{int}$(list,*phase*,$\sigma$),0,$X^b$)

function λ()
send("out", list) {sends the whole list by the "out" output port}

function σ = *ta*(list,*phase*,σ)
$\sigma = \sigma$

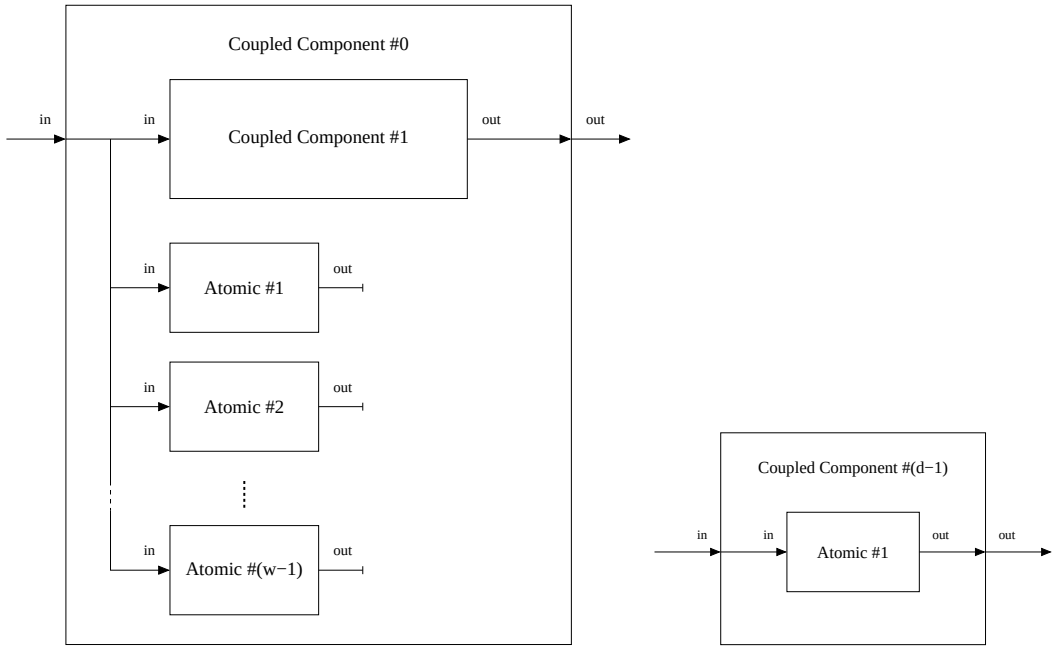
3.1. DEVStone atomic model

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

3.2. LI (Low level of Interconnections) models

Figure 1 shows the general structure of an LI model. With d layers (depth), the first $d - 1$ (with $d \geq 1$) layers have the structure of Figure 1(a). All these layers have one coupled model and $w - 1$ (with $w \geq 2$) 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: execution time and memory footprint (also known as memory high-water mark). Obviously, these two metrics depend on the number of atomic models, number of



(a) DEVStone LI regular coupled component

(b) DEVStone LI deepest coupled component

Figure 1: DEVStone LI components

internal transitions, number of external transitions, and the total number of events internally generated. Additionally, 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.

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

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

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:

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

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

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

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:

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

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

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

3.3. HI (High Input couplings) 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 is computed as follows:

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

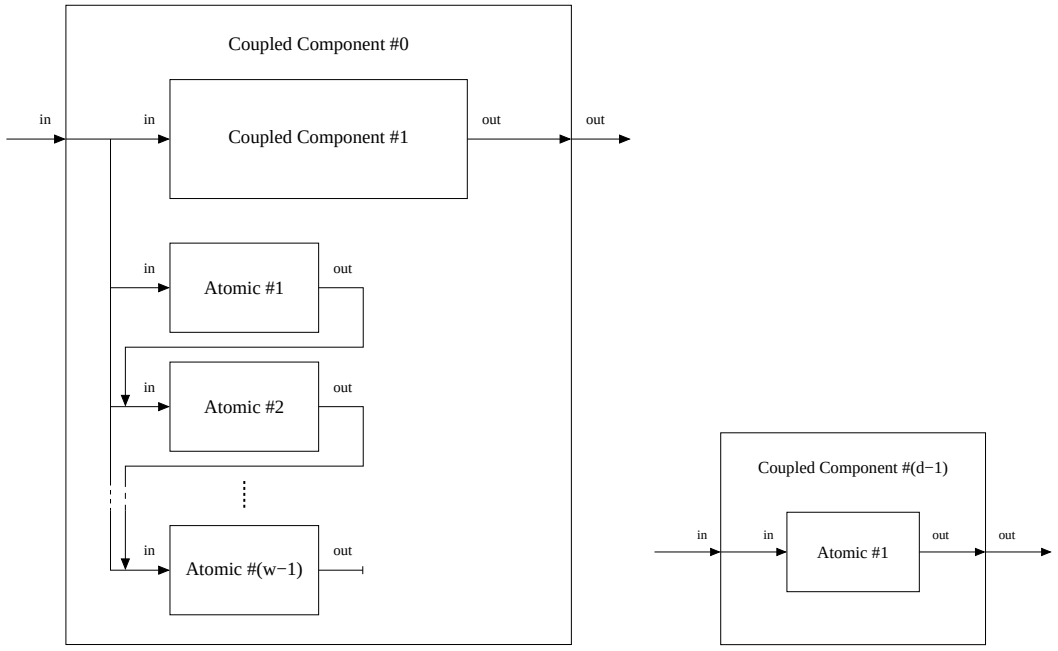
$$\#\delta_{\text{int}} = \left((w-1) + \sum_{i=1}^{w-2} i\right) \cdot (d-1) + 1 = \left(\frac{w^2 - w}{2}\right) \cdot (d-1) + 1 \quad (11)$$

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

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

3.4. HO (Hi model with numerous Outputs) models

Figure 3 shows the general structure of a HO model. HO has a more complex interconnection map with the same number of atomic and coupled components. For example, HO coupled components have two input



(a) DEVStone HI regular coupled component

(b) DEVStone HI deepest coupled component

Figure 2: DEVStone HI components

and two output ports in each level. The main differences with HI 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.

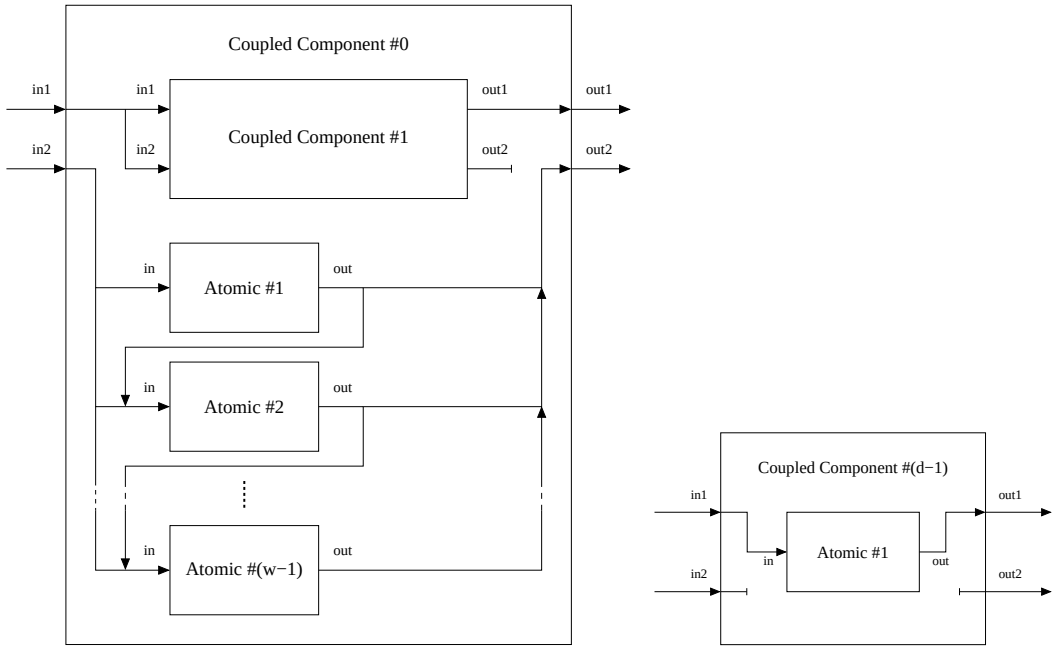
It is worthwhile to mention that the number of atomic models, transition functions and events generated in HO models are exactly the same as in the HI model. However, the main difference is in both the execution time and memory footprint, which are higher due to the additional external input connections.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

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

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

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

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



(a) DEVStone HO regular coupled component (b) DEVStone HO deepest coupled component

Figure 3: DEVStone HO components

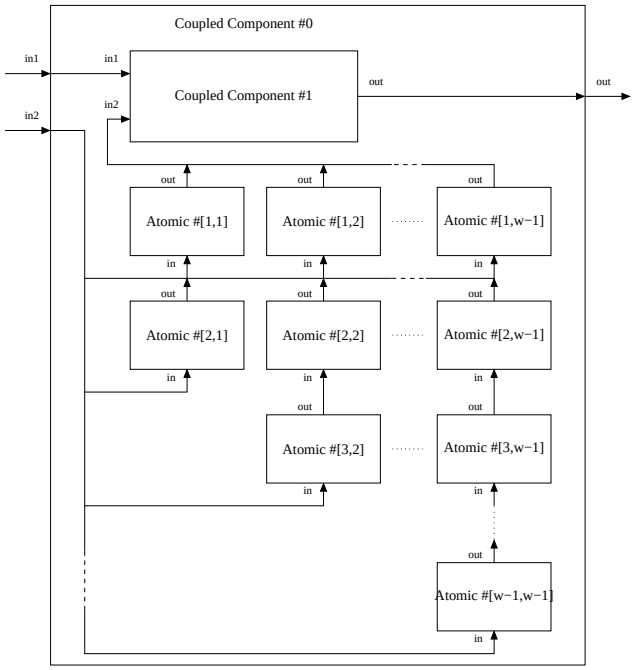
3.5. HOmod models

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 by 1 coupled model, a chain of $w - 1$ atomic models, and a set of $k = 1 \dots w - 1$ 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 send the whole output directly to the coupled component. Finally, each remaining atomic component is connected to its upper component.

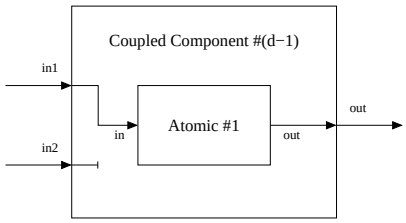
The computation of the number of atomic modes is quite straightforward.

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

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



(a) DEVStone HOmod regular coupled component



(b) DEVStone HOmod deepest coupled component

Figure 4: DEVStone HOmod components

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

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

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

Similarly, the computation of the number of events follows a recursive equation, defined below:

$$\#\text{Events} = \sum_{l=1}^{d-1}\left(\sum_{c=1}^{K_l+w-1}\left(W_1 \times \sum_{i=1}^{w} P_l^{c-i+1} + \sum_{i=1}^{w}\left(W_i \cdot P_l^{c-i+1}\right)\right)\right) + 1 \tag{22}$$

where:

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

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

and

$$P_1^1 = 1 \quad (25)$$

$$P_l^j = 0 \text{ if } 1 > j > K_l \quad (26)$$

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

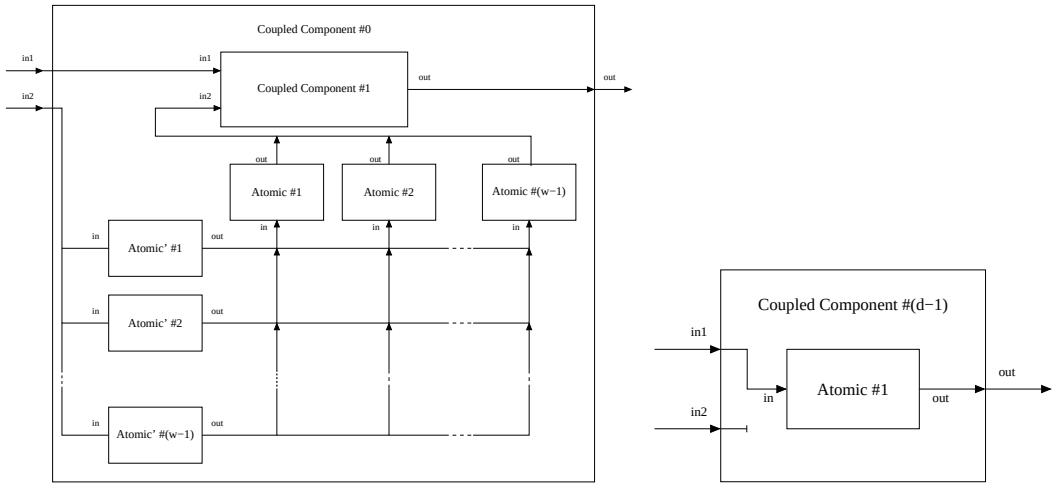
As can be seen, the complexity of the equations describing the metrics of HOmod is high. The inclusion of these equations in a simulator is hard, and the theoretical analysis becomes prohibitive. For these reasons, we have defined a new DEVStone benchmark named HOmem that, providing the same computational effort than 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 basically proposed as a mechanism to increment the traffic of events with respect to HO, equivalently to 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 HOmod. As for the remaining coupled models, each one is formed by 1 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 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:



(a) DEVStone HOmem regular coupled component

(b) DEVStone HOmem deepest coupled component

Figure 5: DEVStone HOmem components

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

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

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

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

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

4. Experimental Methodology

Once the DEVStone equations have been analytically derived, we compare CPU execution time and memory footprint over a total of five well known simulation engines using all 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 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 the frameworks use dynamic data structures such as sets or linked lists.

	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 1: Main data structures used in the simulation engines

We tested all 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 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 [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 wall-clock simulation time and memory footprint, since these are the metrics measured in all the simulations. Table 2 shows these intervals, where each row represent a DEVStone benchmark type, in relation with 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 time and 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.

	Width			Depth			
Benchmark	Min.	Step	Max.	Min.	Step	Max.	**# Simulations**
LI	2	100	1502	1	100	1501	25600
HI	2	100	1102	1	100	1101	14400
HO	2	100	1102	1	100	1101	14400
HOmod	2	1	10	1	1	10	9000
HOmem	2	1	10	1	1	10	9000

Table 2: Parameters configuration

5. Results

5.1. CPU comparison

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

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

Table 3 shows a comparison in execution time (in seconds, measured inside the simulator to avoid the loading time of the model) and 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 hours we run these tests. The same happended in all those cases in Table 3 marked with ∞. As can be seen, only aDEVS and xDEVS were able to finish all the models in Table 3, followed by CD++, which was not able to load the largest LI model. Regarding memory footprint, there is not much difference between both servers. However, in terms of execution time, the best server in almost all cases was the 64 GB Intel Xeon 2670 @ 2.6GHz "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. 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 in the remaining analysis, the wall clock execution time is limited to 1200 seconds and the memory footprint to 4 GiB, enough to perform our more than 70000 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 seconds or 4 GiB, respectively.

5.2. Execution time

Figure 6 shows the contour maps of the different execution times needed by all the five simulators in both 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 seconds 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 HO and HOmem models.

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

With respect to HOmod and HOmem, all the simulators reached the maximum execution time quite soon, with relatively small models. Moreover, in the case of HOmod, only two simulators, aDEVS and xDEVS, were able to load all the models in 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 HOmem. HOmem is simpler in structure than HOmod, and all the simulation engines are able to load it. Once the simulation starts, HOmod and HOmem offer similar execution time and memory footprint, as shown in Section 5.4.

We do not show a comparison between all the simulators in 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, memory footprint is the memory high-water mark of a process. The comparison of all the five simulators were performed constraining the execution time to 1200 seconds 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 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 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 the five simulators reached the memory limit quite soon. As in the execution time analysis, DEVSJAVA was the first to leave the model, for *width* greater than six. Surprisingly, PyPDEVS offered 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 the simulators in HOmod because only aDEVS and xDEVS were able to run a significant number of HOmod instances.

As a conclusion, we may say that, regarding memory footprint, aDEVS is by far the best DEVS simulator between those analyzed in this paper. In the case of 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 aDEVS and xDEVS simulators in finer details, as well as the similarities between HOmod and HOmem.

5.4. Comparison between aDEVS and xDEVS

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

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

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

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

We now compare both simulator in the HOmod and 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 memory footprint reached by aDEVS and xDEVS in HOmod and HOmem models. In both cases, contour maps are practically Yes/No maps, where, after a given *width* and *depth* both simulators immediately reach the limit in execution time and memory footprint. These "saturation" values in HOmod are reached "sooner" (in terms of w and d) than the corresponding values in the HOmem model. We prove here that HOmem offers the same results than HOmod with a more straightforward mathematical formulation, after a comparison of equations (28)-(31) against equations (18)-(27).

6. Conclusions

The Discrete Event System Specification formalism (DEVS) 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 than HOmod but with a more manageable mathematical formulation. As HOmod, HOmem is also intensive on both execution time and memory usage. We have added the study of 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 re-calculated the number of transition functions triggered in all the DEVStone benchmarks. Finally, we have compared five simulation engines in two different hardware platforms, analyzing both the execution time and 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 the simulators were able to run the HOmem model at least for a significant range of *width* and *depth* values. Between all the five simulators, aDEVS, which is based on C++, had the lowest memory footprint at least in LI, HI, and HO models. With respect to execution time, xDEVS was the fastest one, specially in the set of HI and HO more complex models.

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

Acknowledgment

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

This work is supported by the Spanish Ministry of Economy and Competitivity under research grants TIN2014-54806-R and TIN2013-40968-P.

[1] B. P. Zeigler, Theory of Modelling and Simulation, John Wiley, 1976.

[2] B. P. Zeigler, H. Praehofer, T. G. Kim, Theory of Modeling and Simulation. Integrating Discrete Event and Continuous Complex Dynamic Systems, 2nd Edition, Academic Press, 2000.

[3] G. A. Wainer, Developing a software tool for urban traffic modeling, Software, Practice and Experience 37 (13) (2007) 1377–1404.

[4] E. Byon, E. Pérez, Y. Ding, L. Ntaimo, Simulation of wind farm operations and maintenance using discrete event system specification, SIMULATION Transactions of The Society for Modeling and Simulation International 87 (12) (2011) 1093–1117.

[5] G. A. Wainer, S. Daicz, A. Troccoli, Experiences in modeling and simulation of computer architectures using DEVS, Transactions of the Society for Modeling and Simulation International 18 (4) (2001) 179–202.

[6] A. Moreno, J. L. Risco-Martín, E. Besada-Portas, L. de la Torre, J. Aranda, Formal Languages for Computer Simulation: Transdisciplinary Models and Applications, IGI Global, 2013, Ch. Thermal Analysis of the MIPS Processor Formulated within DEVS Conventions, pp. 103–144.

[7] M. Moallemi, G. Wainer, 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, Society for Computer Simulation International, 2011, pp. 95–102.
URL http://dl.acm.org/citation.cfm?id=2048476.2048488

[8] A. Moreno, J. L. Risco-Martín, E. Besada, S. Mittal, J. Aranda, DEVS/SOA: Towards DEVS Interoperability in Distributed M&S, in: 13th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications, IEEE, IEEE Computer Society, 2009, pp. 144–153. doi:10.1109/DS-RT.2009.18.
URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5361772

[9] E. Perez, L. Ntaimo, C. Bailey, P. McCormack, Modeling and Simulation of Nuclear Medicine Patient Service Management in DEVS, SIMULATION 86 (8-9) (2010) 481–501. doi:10.1177/0037549709358294.
URL http://dl.acm.org/citation.cfm?id=1841379.1841381

[10] DEVSJAVA (2015).
URL http://acims.asu.edu/software/devsjava

[11] DEVS-Suite,
http://devs-suitesim.sourceforge.net (2015).
URL http://devs-suitesim.sourceforge.net

[12] CoSMoS (2015).
URL http://acims.asu.edu/software/cosmos

[13] CD++ (2015).
URL http://cell-devs.sce.carleton.ca

[14] Y. V. Tendeloo, H. Vangheluwe, The modular architecture of the python(P)DEVS simulation kernel, in: Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium, 2014, pp. 1–6.

[15] aDEVS (2015).
URL http://web.ornl.gov/~1qn/adevs/

[16] JAMES II (2015).
URL http://wwwmosi.informatik.uni-rostock.de

[17] T. G. Kim, C. H. Sung, S.-Y. Hong, J. H. Hong, C. B. Choi, J. H. Kim, DEVSim++ Toolset for Defense Modeling and Simulation and Interoperation, The Journal of Defense Modeling & Simulation 8 (3) (2011) 129–142. doi:10.1177/1548512910389203.

[18] DUNIP Technologies (2015).
URL http://www.duniptechnologies.com

[19] E. Glinsky, G. Wainer, DEVStone: a Benchmarking Technique for Studying Performance of DEVS Modeling and Simulation Environments, in: Ninth IEEE International Symposium on Distributed Simulation and Real-Time Applications, IEEE, 2005, pp. 265–272. doi:10.1109/DISTRA.2005.18.
URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1530678

[20] M. Gutierrez-Alcaraz, G. A. Wainer, Experiences with the DEVStone benchmark, in: Proceedings of the 2008 Spring Simulation Multiconference, 2008, pp. 447–455.

[21] G. Wainer, E. Glinsky, M. Gutierrez-Alcaraz, Studying performance of DEVS modeling and simulation environments using the DEVStone benchmark, SIMULATION: Transactions of SCS 87 (7) (2011) 555–580.

[22] J. Davila, M. Y. Uzcategui, GALATEA: A multi-agent, simulation platform, in: International Conference on Modeling, Simulation and Neural Networks (MSNN 2000), 2000, pp. 52–67.

[23] M. K. Traoré, SimStudio: a Next Generation Modeling and Simulation Framework, in: International ICST Conference on Simulation Tools and Techniques for Communications, Networks and Systems, 2010.

[24] PowerDEVS (2015).
URL http://powerdevs.sourceforge.net

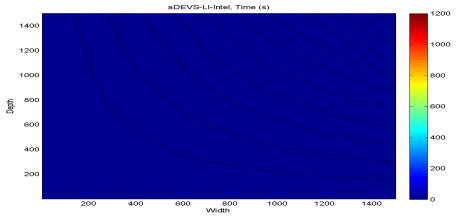
[25] MS4Me (2015).
URL http://www.ms4systems.com/pages/main.php

[26] VLE: The Virtual Laboratory Environment (2015).
URL http://www.vle-project.org/wiki

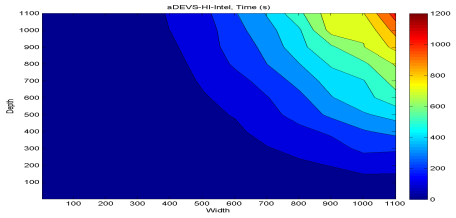
[27] Y. Van Tendeloo, H. Vangheluwe, The Modular Architecture of the Python(P)DEVS Simulation Kernel: Work In Progress paper, in: Proceedings of the Symposium on Theory of Modeling & Simulation - DEVS Integrative, Society for Computer Simulation International, 2014, pp. 1–6.
URL http://dl.acm.org/citation.cfm?id=2665008.2665022

[28] D. Vicino, D. Niyonkuru, G. Wainer, O. Dalle, Sequential PDEVS architecture, in: Symposium On Theory of Modeling and Simulation (TMS'15), 2015.

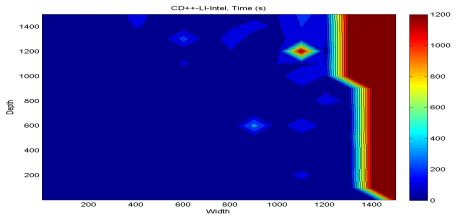
[29] R. P. Weicker, Dhrystone: a synthetic systems programming benchmark, Communications of the ACM 27 (10) (1984) 1013–1030. doi:10.1145/358274.358283.
URL http://dl.acm.org/citation.cfm?id=358274.358283



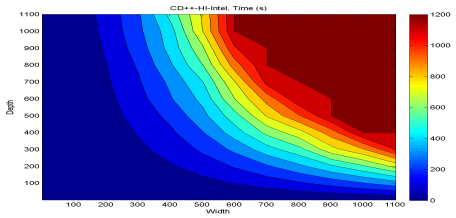
(a) aDEVS - LI



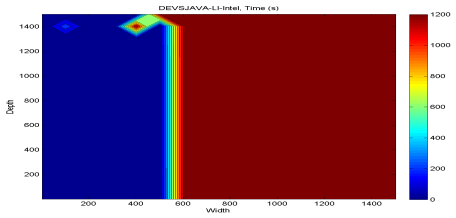
(b) aDEVS - HI



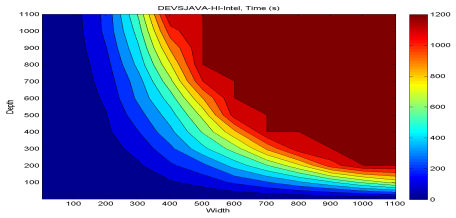
(c) CD++ - LI



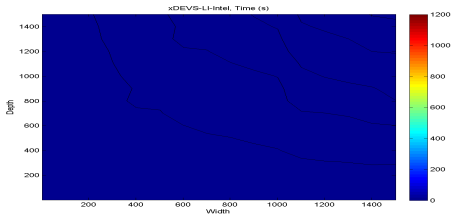
(d) CD++ - HI



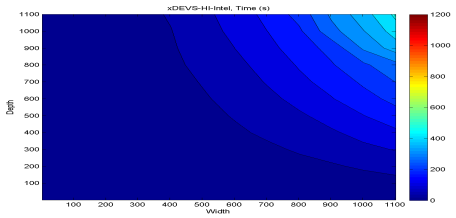
(e) DEVSJAVA - LI



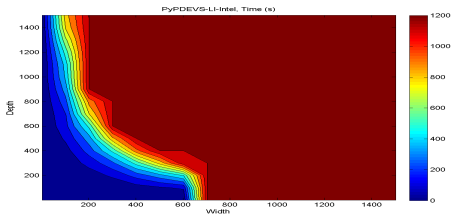
(f) DEVSJAVA - HI



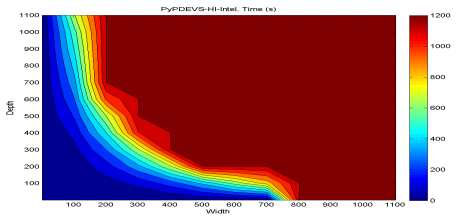
(g) xDEVS - LI



(h) xDEVS - HI

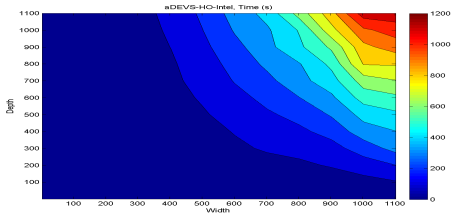


(i) PyPDEVS - LI

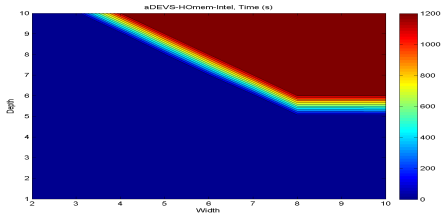


(j) PyPDEVS - HI

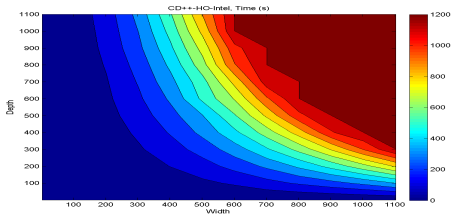
Figure 6: Execution time of LI and HI models



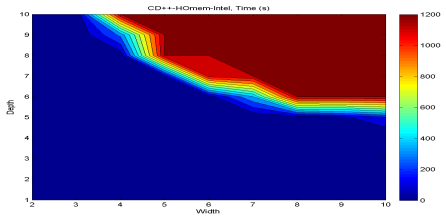
(a) aDEVS - HO



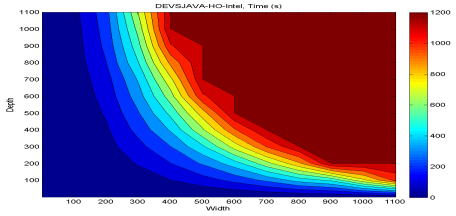
(b) aDEVS - HOmem



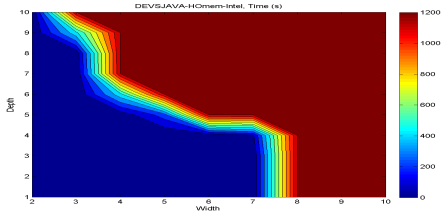
(c) CD++ - HO



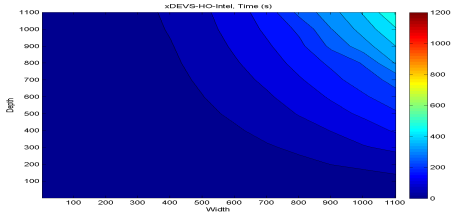
(d) CD++ - HOmem



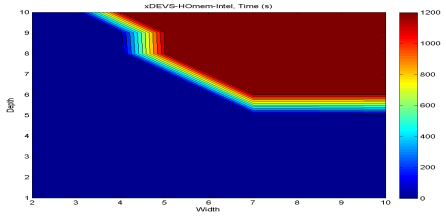
(e) DEVSJAVA - HO



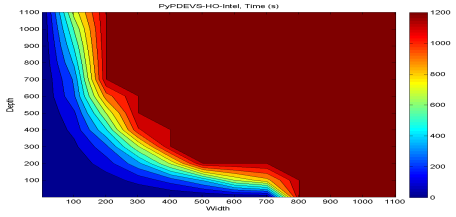
(f) DEVSJAVA - HOmem



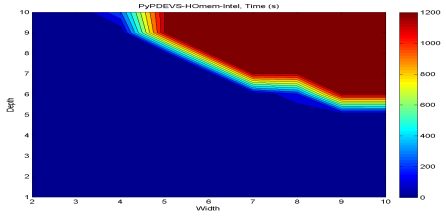
(g) xDEVS - HO



(h) xDEVS - HOmem

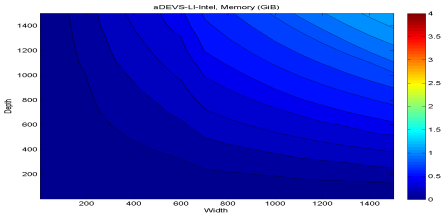


(i) PyPDEVS - HO

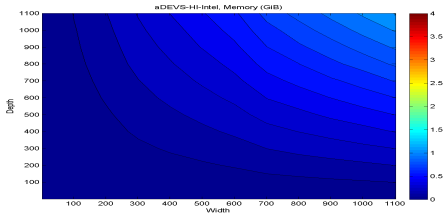


(j) PyPDEVS - HOmem

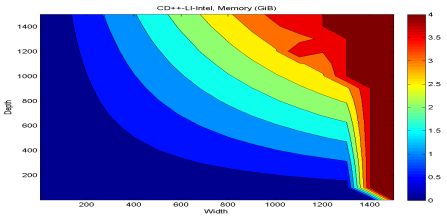
Figure 7: Execution time of HO and HOmem models



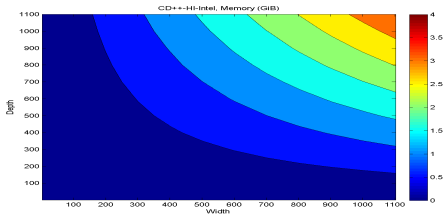
(a) aDEVS - LI



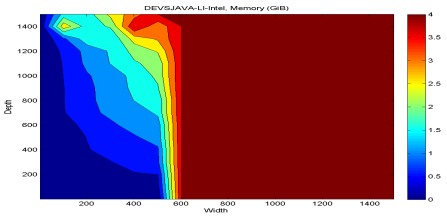
(b) aDEVS - HI



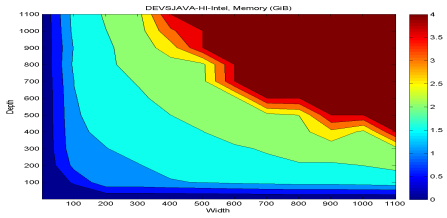
(c) CD++ - LI



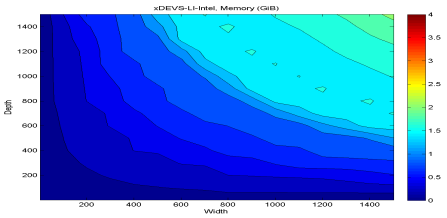
(d) CD++ - HI



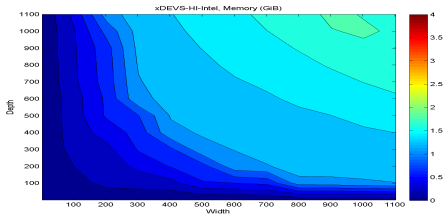
(e) DEVSJAVA - LI



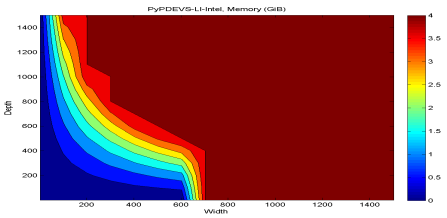
(f) DEVSJAVA - HI



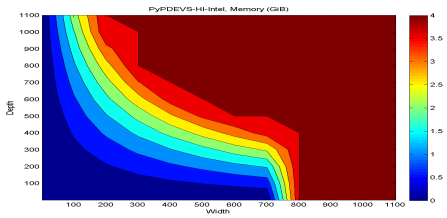
(g) xDEVS - LI



(h) xDEVS - HI

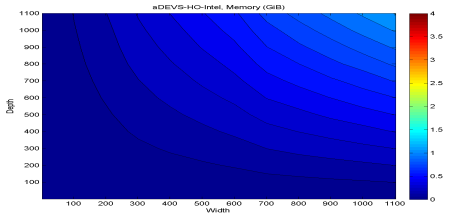


(i) PyPDEVS - LI

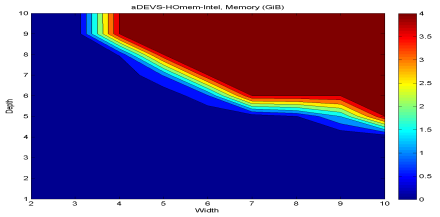


(j) PyPDEVS - HI

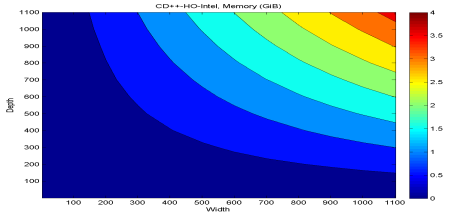
Figure 8: Memory footprint of LI and HI models



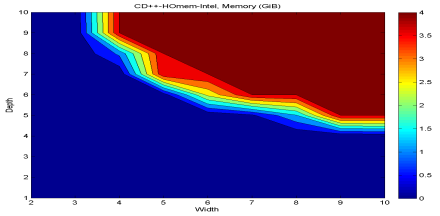
(a) aDEVS - HO



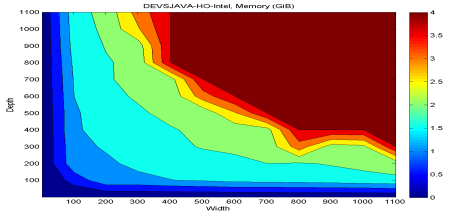
(b) aDEVS - HOmem



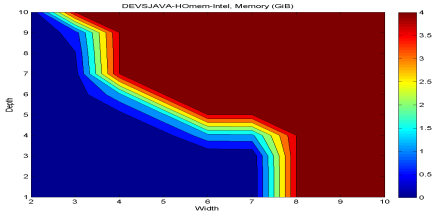
(c) CD++ - HO



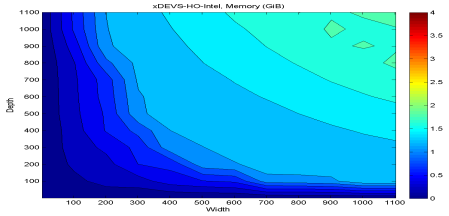
(d) CD++ - HOmem



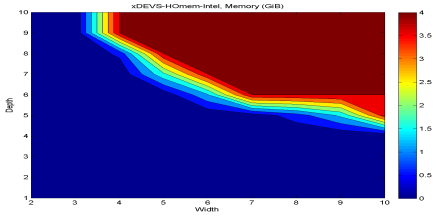
(e) DEVSJAVA - HO



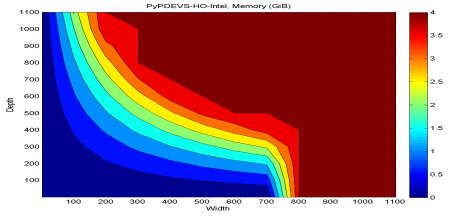
(f) DEVSJAVA - HOmem



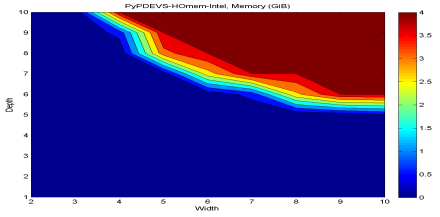
(g) xDEVS - HO



(h) xDevs - HOmem

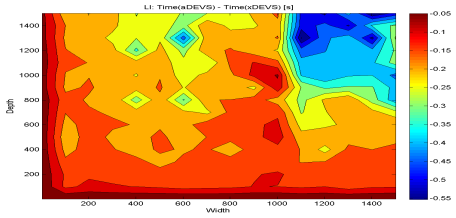


(i) PyPDEVS - HO

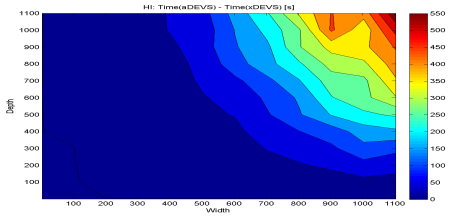


(j) PyPDEVS - HOmem

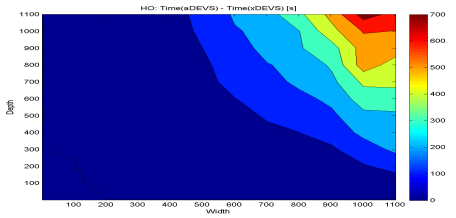
Figure 9: Memory footprint of HO and HOmem models



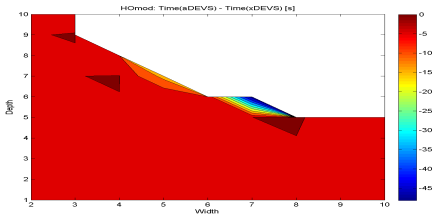
(a) LI: xDEVS - aDEVS



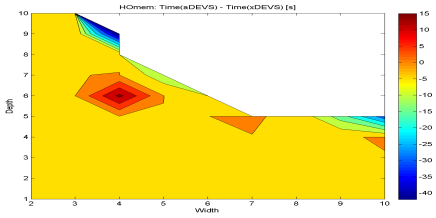
(b) HI: xDEVS - aDEVS



(c) HO: xDEVS - aDEVS

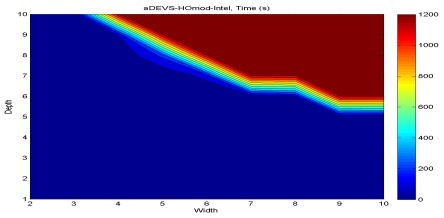


(d) HOmod: xDEVS - aDEVS

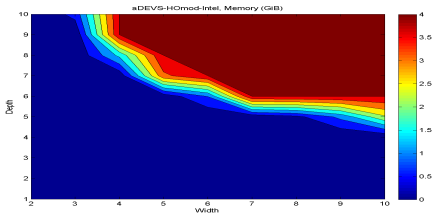


(e) HOmem: xDEVS - aDEVS

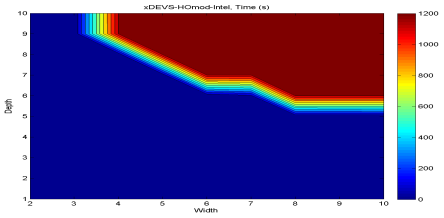
Figure 10: Execution time comparison of LI, HI, HO, HOmod and HOmem models computed as Time(xDEVS) - Time(aDEVS)



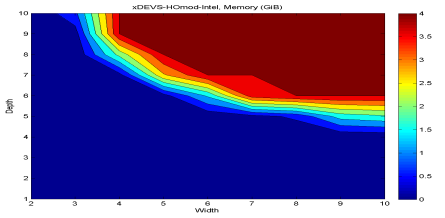
(a) aDEVS - HOmod (Time)



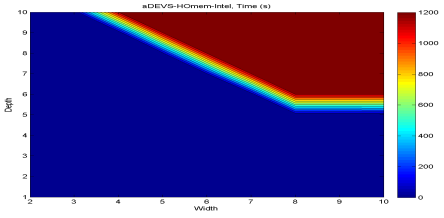
(b) aDEVS - HOmod (Memory)



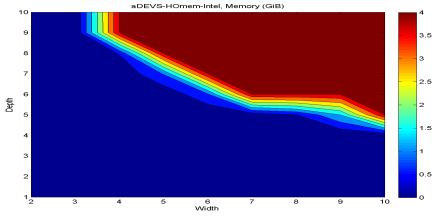
(c) xDEVS - HOmod (Time)



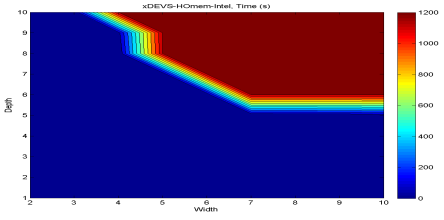
(d) xDEVS - HOmod (Memory)



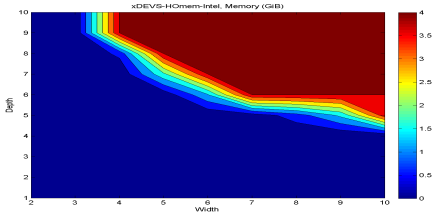
(e) aDEVS - HOmem (Time)



(f) aDEVS - HOmem (Memory)



(g) xDEVS - HOmem (Time)



(h) xDEVS - HOmem (Memory)

Figure 11: Execution time and memory footprint of HOmod and HOmem models given by aDEVS and xDEVS