Experiences with the DEVStone Benchmark

Marcelo Gutierrez-Alcaraz Gabriel Wainer

Dept. of Systems and Computer Engineering
Carleton University
4456 Mackenzie Building
1125 Colonel By Drive
Ottawa, ON. K1S 5B6. Canada.
gwainer@sce.carleton.ca

Keywords: DEVS, synthetic benchmark, M&S tool, simulator performance evaluation, CD++, ADEVS.

Abstract

DEVS is a formal modeling and simulation (M&S) framework that supports hierarchical, modular model composition. DEVS-based M&S environments have been used successfully to understand, analyze, and develop a variety of systems. As the systems under study become more large and complex, performance of the simulator becomes critical. Nevertheless, evaluating the operation of such simulators is a complex process. We present DEVStone, a synthetic benchmark devoted to automate the evaluation of DEVS-based simulators, which generates models with varied structure and behavior. DEVStone was used to study the efficiency of different simulation engines provided by the CD++ toolkit, this results were later compared with similar results on ADEVS with a demanding set of experiments, enabling thorough performance analysis. DEVStone facilitates performance analysis for successive versions (e.g., upgrades or fixes) of the same simulation engine, and provides a common metric to compare different M&S environments.

1. INTRODUCTION

In recent years, the DEVS (Discrete Event systems Specifications) formalism [1] has gained popularity. DEVS was successfully applied in a variety of applications due to the ease for model definition, improved composition and reuse. DEVS includes explicit specification of the model timing, and it uses a discrete event approach for simulation. Several tools have implemented DEVS theory, including: **CD++** [2,3,4,5], **ADEVS** [6], **DEVS-C++** [7] , **DEVS/HLA** [8] , **DEVSJAVA** [9] , **DEVSim++** [10] and others [11,12,13].

As the systems under study become larger and complex, performance of the simulator becomes critical; however, evaluating the operation of such simulators is a complex process. To provide uniform means for obtaining meaningful metrics, we introduce the **DEVStone** benchmark, a synthetic model generator that automatically creates models according to our goals. Its accuracy relies on the execution of a large pool of models to provide a robust test set for the study. DEVStone generates models with different size, complexity and behavior, resembling different kinds of applications. Hence, it is possible to analyze the efficiency of a simulation engine with relation to the characteristics of a category of models of interest. The tool can be used to assess the efficiency of several DEVS simulation engines, and provides a common metric to compare the results using different tools.

Initially, we used the synthetic generator to analyze the performance of different simulation techniques in CD++, which allowed us to show the feasibility of our approach [14]. Moreover, the performance results permitted us to characterize execution time of a standard DEVS simulator. A second step presented here, includes the execution of DEVStone in CD++ and ADEVS. DEVStone was used to study the efficiency of different simulation engines provided by the CD++ toolkit, and later compared with similar results in ADEVS with a demanding set of experiments, enabling thorough performance analysis. The present paper shows how DEVStone can be used to measure and improve DEVS simulation tools, by providing a unified mechanism for comparing different tools and environments.

2. DEVS AND THE DEVSTONE BENCHMARK

DEVS is a formal M&S framework based on generic dynamic systems concepts [1]. A real system modeled with DEVS can be described as a composite of submodels that can be behavioral (atomic) or structural (coupled). The framework supports the construction of models in a hierarchical, modular fashion, allowing component reuse; reducing development and testing time. An atomic DEVS model is formally described as:

$$M = < X, S, Y, \delta_{int}, \delta_{ext}, l, ta >$$

where X is the set of input events; S is the state set; Y is the set of output events; δ_{int} is the internal transition function; δ_{ext} is the external transition function; l is the output function; and ta is the time advance function.

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

$$CM = < X, Y, D, \{M_i\}, \{I_i\}, \{Z_{ij}\} >$$

where X is the set of input events; Y is the set of output events; D is an index for the components of the coupled model, and $\forall\ i \in D$, M_i is a basic DEVS (i.e., an atomic or coupled model), I_i is the set of influencees of model i (i.e., models that can be influenced by outputs of model i), and $\forall\ j \in I_i$, Z_{ij} is the i to j translation function. Coupled models are defined as a set of basic components (atomic or coupled) interconnected through the models' interfaces. The influ-

encees of a model define to which model outputs must be sent. The translation function converts the outputs of a model into inputs for other models. To do so, an index of influencees is created for each model (I_i). This index defines that the outputs of the model M_i are connected to inputs in the model M_j, where *j* is an element of I_i.

Analyzing the performance of a simulation tool is a complex task because the users can create a wide variety of models with different structures, levels of complexity and degrees of interaction. When analyzing previous studies on the performance of DEVS environments, we can see that they were only focused in performance results for a given tool, and were usually limited to a given type of models. For instance, in [15], the authors presented performance studies of a parallel simulation environment. Those studies provide an analysis of the execution of models with different characteristics, although DEVS tools permit running a wider variety of models. Such application lacks a common metric to compare results among different simulators.

We propose a method to compare not only different versions of a particular simulation engine but also different DEVS-based simulators. Varied model structures permit obtaining prototypes representative of those found in real world applications. The idea is to use a synthetic model generator (which we called DEVStone), which produces a variety of models with diverse structure and performing a mix of common operations. We focus in the aspects of the models that have impact on performance, namely size of the model and the workload carried out in the transition functions. A DEVStone generator produces models using the following parameters:

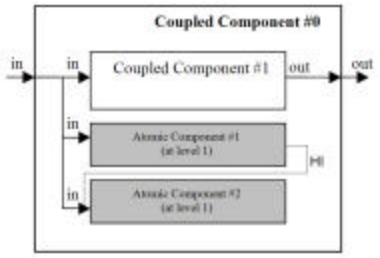
- type: different structure and interconnection schemes between the components.
- depth: the number of levels in the modeling hierarchy.
- width: the number of components in each intermediate coupled model.
- internal transition time: the execution time spent by internal transition functions.
- external transition time: the execution time spent by external transition functions.

In general, being *d* the depth and *w* the width, we build a coupled model with *d* coupled components in the hierarchy, all of which consist of *w–1* atomic models (in the lower level of the hierarchy, the coupled component consists of a single atomic model). Basically when an external event is received it triggers the external transition function, which runs Drhystones [16] for a certain time, right after it an internal transition function is scheduled which also runs Drhystones, this is replicated by all atomic blocks in the model. The use of integer arithmetic makes Dhrystone the best suitable choice for analyzing models like DEVS in which discrete state variables are used.

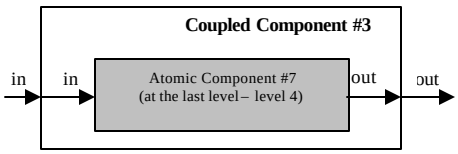
DEVStone uses four different types of models with variations in their internal and external structure:

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

In **LI** models, every coupled component includes only one input and one output port. The input port is connected to each component, and only one component generates an output through the output port in the external component. Figure 1 shows a sample LI model, in which we have four layers of coupled components, each containing three submodels (arrows: input/output ports; white boxes: coupled components; gray boxes: atomic components). *Coupled Component #0* in Figure 1 (a) consists of one coupled and two atomic components. Lower levels in the hierarchy (*Coupled Component #1/#2*) use the same internal structure. *Coupled Component #3* is a "leaf" model, which contains one atomic child (#7) connected to its output port in the lower layer.



(a)



(b)

Figure 1: Example of a LI (HI) model: (a) top level; (b) level 4.

Since the model structure is known and the time spent by each component in executing transition functions is known, we can compute its minimum theoretical execution time. First, we need to devise the number of atomic and coupled models in the structure. In LI models of depth *d* and width *w*, we have *d* coupled components with *w-1* atomic components each (except for the innermost one, with only one atomic component). Consequently, the total number of atomic components is:

$$\#\ Atomic\ Models = (width - 1) * (depth - 1) + 1$$

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

 # Internal Transitions = # Atomic Models (1)
 # External Transitions = # Atomic Models

HI models have the same number of atomic components, but more interconnections (Figure 1a). Each atomic component *k* connects its output to the input port of component *k+1* (with the exception of one last atomic component on each level, which does not have any output port). Therefore, there are more messages interchanged upon the reception of an external event. In a model with depth *d*, and width *w*, we have,

*# Atomic Models = (w – 1) * (d – 1) + 1*
*# Internal Transitions = ((w-1)+(w-2)) * (d – 1) + 1* (2)
*# External Transitions = ((w-1)+(w-2)) * (d – 1) + 1*

For each external event, each coupled model forwards the received message to its *w-1* atomic children and also to its coupled child. This process of forwarding messages is repeated in each coupled component except for the leaf component, which forwards the messages to its single atomic child.

HO type models have a more complex interconnection scheme with the same number of atomic and coupled components. HO coupled models have two input and two output ports in each level. The second input port in the coupled component is connected to its first atomic component. That atomic model connects its output to the second output of its parent. The increased number of interconnections results in the execution of more transition functions after the model issues its output. For this model type we have,

*# Atomic Models = (w – 1) * (d – 1) + 1*
*# Internal Transitions = ((w-1)+(w-2)) * (d – 1) + 1* (3)
 # External Transitions = ((w-1)+(w-2)) (d – 1) + 1*

HOmod models have a second set of *(w-1)* models where each one of the atomic components triggers the entire first set of *(w-1)* atomic models. These in turn have their outputs connected to the second input of the coupled model within the level. With such interconnections, the inner model receives an amount of events that has an exponential relationship between the width and the depth at each level. The equation that rules the behavior of the **HOmod** Model is given by:

$$\#\,\text{Internal Transitions} = n_{it} = (w-1)^{2*(d-1)} + \left(2*(w-1) + \sum_{1}^{w-2} i\right) * (d-1) + 1$$

$$\#\,\text{External Transitions} = n_{et} = (w-1)^{2*(d-1)} + \left(2*(w-1) + \sum_{1}^{w-2} i\right) * (d-1) + 1$$

(4)

External events are forwarded by each coupled component to its *w-1* atomic children and to its coupled child, and the process is repeated in each coupled module until the arrival to the leaf component.

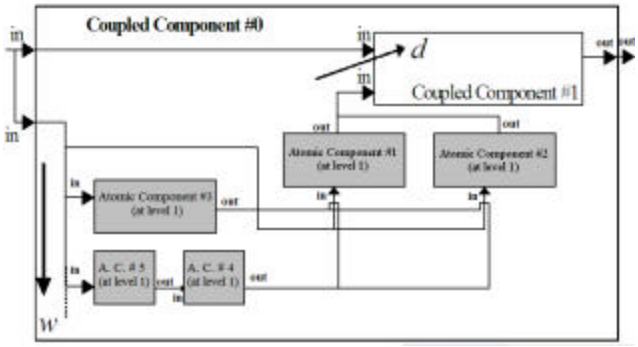


*Figure 2: DEVStone **HOmod** model (shown explicitly for w = 3)*

DEVStone can be used in any simulator with capabilities for defining and executing Dhrystone code.

ADEVS (A Discrete EVent System simulator) is a C++ library for constructing discrete event simulations based on the Parallel DEVS and Dynamic DEVS formalisms. The models are constructed based on a template of classes in C++ and then compiled and linked to the library to produce the simulation executable. Every atomic or coupled model in ADEVS is comprised of two files: a header file (.h); where the name of the model, input and output ports and local variables are defined for the particular atomic model, and a source code file (.cpp); where the actual model is built based on a template, common elements of the class include: constructor, internal transition function, external transition function, time advance function, output function, and destructor.

3. PERFORMANCE ANALYSIS OF DEVS SIMULATORS

Based on the DEVStone benchmark [16], we developed a tool to measure the performance of different versions of CD++ and ADEVS. Parameters of comparison needed to be defined to compare performance metrics among simulators; the *test environment* used for the benchmarking came with the following commercial grade characteristics: CPU: Pentium 4 Dual Core @ 3.2GHz (800MHz FSB, HT, 1MB L2), RAM: 2 GB (4 x DDR2-533), Hard Drive: 80 GB - 35 GB ex3 Linux partition (7,200 rpm, SATA), OS: Fedora Core 6

Running DEVStone, we found the following general limitations to the experimental setup:

- The depth of a model in ADEVS cannot be greater than 195 levels (due to the GCC compiler, which finds too many nested loops inside the executable of ADEVS). CD++ does not seem to have a limit on the depth.

- The minimum depth and minimum width for any model generated by DEVStone for CD++ or ADEVS is 2.
- In most extreme cases it is possible to initialize the simulation, but it is not possible to run the simulation to any time longer than 0. In CD++, an 'unexpected error' message is displayed or the process is killed by the Out-Of-Memory (OOM) kernel service; and in ADEVS, the simulation is killed by the OOM kernel service. It seems that whenever the simulator requests massive amounts of memory to the operating system, beyond the available physical memory and some of the virtual memory, the OS decides to terminate a potentially harmful process.

We started the benchmarking process by measuring the initialization time for CD++ and ADEVS. We selected a nominal value to start measure the initialization time: since the depth limit is set by ADEVS (195 levels) it is just natural to select this value; the width does not have an upper limit, however, we decided to select a width value that could fit a line in CD++ (nominally, 1839 components). For the LI model the outcome of the initialization test was:

(depth = 195, width = 1839, δ_{int} = 1.0 ms, δ_{ext} = 0.1 ms, T_{sim} = 0)

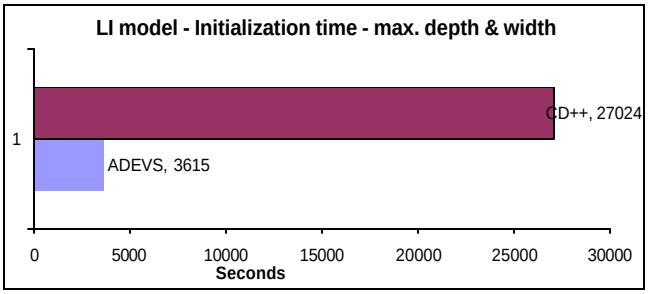


Figure 3: DEVStone Initialization time LI model

In Figure 3, we consider the initialization time for this purpose as the required time for the compilation of the source code and the initial execution of the executable to setup the model before simulating it, i.e. the simulation time to T_{sim} = 0. The compilation time of CD++ for big models is negligible; because the simulator and the model are two completely separated entities the compilation of the simulator takes only a couple of minutes and can be used for any model, in our case the simulator was compiled only once and reused for all the simulations in this report. On the other hand the compilation time in ADEVS is taken into account, only for initialization purposes, mainly because any change in the model involves a new compilation of the source code. Therefore, for small to medium sized models where a minor correction of the model was required the compilation time surpassed the simulation time. This affects the performance of the simulator whenever a change needs to be made. From Figure 3 it is obvious that ADEVS outperforms CD++ in terms of initialization time, even when the compilation time is considered. To find out the process that take up most of the time we used a profiler to analyze the execution of the simulation.

Table 1: Profiler output. Flat profile: for the simulation of LI models

% time	calls	Name
29.09	1427848	ProcessorAdmin::processor (basic_string<...> const &)
17.91	450903011	basic_string<...>::compare (basic_string<...> const &, unsigned int, unsigned int) const
15.86	162000611	basic_string<...>::rep(void)const
9.59	3307266338	basic_string<...>::length(void) const
7.91	989592590	basic_string<...>::data(void) const

According to the initial result of the profile in Table 1 most of the initialization time, 29.09 % of it, is spent somewhere inside the processor block. Other important source of delay during initialization seems to be the search and compare of symbols done by C++ libraries, which are performed while loading the model. A more exhaustive analysis is possible, using the profiler's option that indicates functions inside the program and the *relative* time that the computer takes running those functions. The caveat here is that the percentage of time spent in each function is not an *absolute* time, i.e. the sum of the percentages will not yield 100%. This time is relative to the total running time but also to the time spent inside the 'parent function'. An edited listing of the output is provided in Table 2.

Table 2: Call Graph of the CD++ Simulator

Granularity: each sample hit covers 4 byte(s) for 0.00% of 82897.48 seconds		
index	**% time**	**Function name**
[2]	98.5	MainSimulator::run(void) [2]
		MainSimulator::loadModels(istream &, bool) [3]
		…
[4]	85.4	Basic_string<...>::compare (basic_string<...> …) const [4]
		Basic_string<...>::length(void) const [7]
		Basic_string<...>::data(void) const [13]
		…
		Basic_string<...>::data(void) const [13]
		Basic_string<...>::length(void) const [7]
[5]	83.8	Basic_string<...>::rep(void) const [5]
		…
		Basic_string<...>::compare (basic_string<...> …) const [4]

[7]	64.1	Basic_string<...>::length(void) const [7]
		Basic_string<...>::rep(void) const [5]
...		

CD++ seems to spend most of the time looking and comparing the new incoming symbols –model names– in a linear fashion, this would explain the excessive time spent comparing and handling them. Most of the workload is spent in library functions that are specific to the compiler used.

Because ADEVS takes much more time compiling large depth models, another good piece of information would be to compare the performance of both simulators with a model of maximum depth and minimum width and see if CD++ behaves the same way. An event file was created that provides 10 external events evenly spaced every 0.250 (s), the same file was used in all subsequent simulations; the results of the first simulation with such file are shown in Figure 4.

		1		2		3	
Length	Width	δ_{int}	δ_{ext}	δ_{int}	δ_{ext}	δ_{int}	δ_{ext}
195	2	0.1	0.1	0.1	1	1	0.1

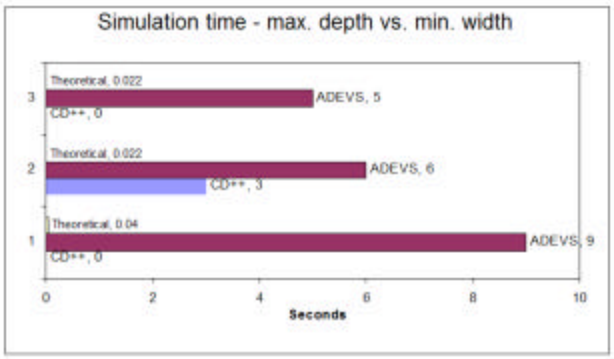


Figure 4: Minimum width and maximum depth of models. d_{int} and d_{ext} are measured in ms.

In this case CD++ proves to be faster than ADEVS, mainly because for CD++ the models are loaded in memory as they are needed, regardless of their nature and processed accordingly by the CD++ entity that controls each node, either atomic models or coupled models. One interesting note in this case is that for the first scenario where the internal transition function equals the external transition function ADEVS takes almost 50% more time, than the rest of the tests, to finish the simulation, this suggests that ADEVS loads the entire code of the model-simulator into memory, then performs the simulation and finally flushes all contents from RAM and terminates, the reading and writing of the whole file to and from the hard-drive would explain the increase in simulation time. In addition, a check to the memory usage in a second run demonstrated that while ADEVS is running, it takes up to 99% of the available memory right from the beginning while CD++ increases the memory usage incrementally. However, a detailed analysis of the memory usage was not deemed necessary for our purposes because we assume that all the resources will be given to the simulation during its execution, i.e. a dedicated system will be put in place for a simulation and it will only run simulations of a particular model.

Having provided a feasible methodology to analyze the performance of CD++, it is possible to show the flexibility of DEVStone in both different environments and at the same time extract a more reliable tendency of the performance of the simulators. With this idea a set of simulations were executed with different models of similar values for every model, except for the last HOmod model. For the rest of the tests, the simulators were compiled without any option that might slow down the performance.

We tested the performance of the simulators based on 4 main parameters:

- the model type,
- variations in the width of the model,
- variations in the length of the model.
- variations in the real-time running internal and external transition functions (in ms)

We simulated the same model with equal time spent in the transitions, the internal transition longer than the external transition and finally the external transition longer than the internal transition. By changing the width of the model the analysis can focus on the time that the simulator spends sending messages back and forth to atomic blocks within each level.

		1		2		3	
Length	Width	δ_{int}	δ_{ext}	δ_{int}	δ_{ext}	δ_{int}	δ_{ext}
10 5-10	100-600 100	0.1	0.1	0.1	1	1	0.1

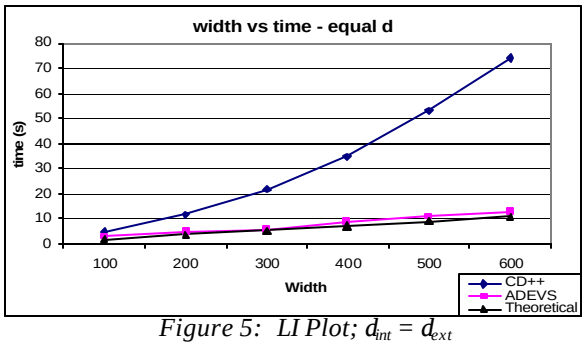


Figure 5: LI Plot; $d_{int} = d_{ext}$

In Figure 5, we can see that ADEVS outperforms CD++ by a significant margin for large models. The major difference between CD++ and ADEVS becomes wider when the internal and external transition times are equal; however the tendency is similar for the other variations of internal and external transition functions. One possible reason for this is that CD++ runs using a temporary file where intermediate results

are stored and read when necessary whereas ADEVS runs the simulation in the available memory. With a similar approach, we could see if this trend remains constant when running the simulation varying the depth variable, in this case by increasing the depth levels of the model we are analyzing the variation in the performance when the interchange messages travel among coupled blocks.

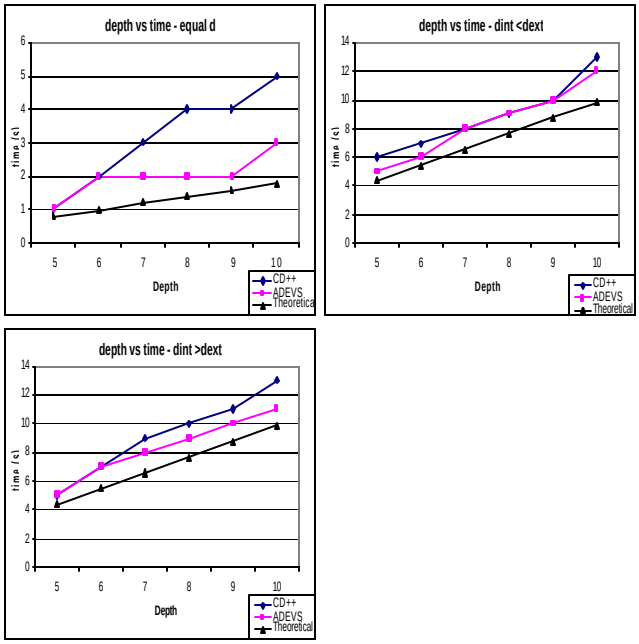


Figure 6: LI Plot; $d_{int} = d_{ext;}$ $d_{int} < d_{ext}$; $d_{int} > d_{ext}$

In Figure 6 we can see a change in the performance, the difference between ADEVS and CD++ is not so big for multiple levels of coupled models.

The HI type of model connects the atomic blocks in linear fashion but with a greater number of interconnections between the components within the coupled model. For the simulation-runs of HI, the same values used for the depth and width of the LI models were used.

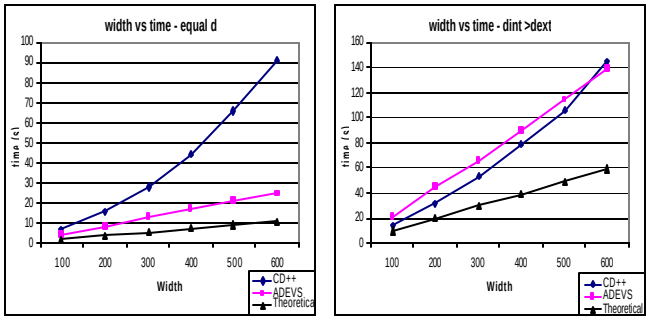


Figure 7: HI Plot; $d_{int}=d_{ext}$;$d_{int} > d_{ext}$

In Figure 7 we can see that the performance of the simulators have changed, for $\delta_{int}<\delta_{ext}$ the curves are similar to $\delta_{int}=\delta_{ext}$ but with a shorter gap between them. Although CD++ still lags behind ADEVS when the transition functions are equal, the performance when the internal transition function is less than the external transition is not as big as the one we could expect based on the results of the simulation of LI models. Even more unexpected is the result when the external transition is greater, where for large models CD++ outperforms ADEVS.

The same experiment was done by varying the depth of the levels and keeping the width constant. The same three different sets of internal and external transition functions were used. For variable depth HI models with constant width the simulation parameters were:

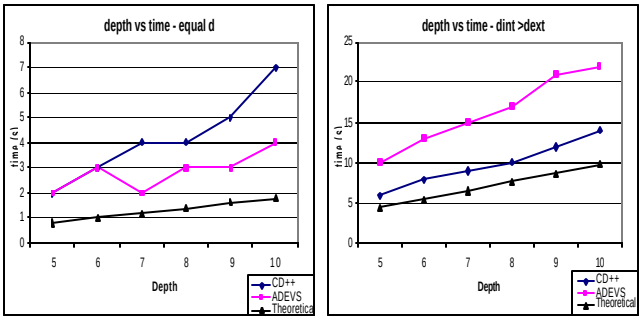


Figure 8: HI Plot; $d_{int} = d_{ext}$; $d_{int} > d_{ext}$

In figure 8 we can see the results of varying the depth in the HI model. When the transition functions are equal ADEVS behaves somewhat better than CD++, for $\delta_{int}<\delta_{ext}$ both simulators had the exact same performance. It seems that, for this type of models, the way transitions are triggered and processed in CD++ contribute to the overall performance of the simulator.

In HO models, the model structure is similar to HI models but there are more messages coming through the coupled models, via the second input. We used the same values for the width, depth and the transition functions.

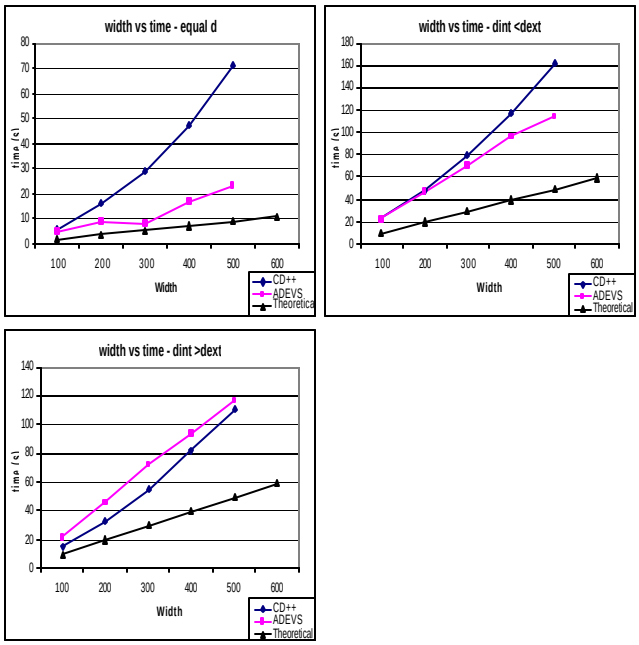


Figure 9: HO Plot; $d_{int} = d_{ext;}\ d_{int} < d_{ext}$; $d_{int} > d_{ext}$

In figure 9, neither simulator could complete the last test for 600 atomic components; both were terminated by the OOM Linux service. In the first graph, with equal transition functions, we can see a drastic difference in the performance of the simulators; CD++ is clearly being outperformed by ADEVS by as much as 4 times the theoretical value, just like the simulation results for the LI models. A possible answer to this CD++ behaviour would be in the way models are loaded into virtual memory for processing but only accessed when they are needed, therefore the time to load an incumbent block to the memory and then writing the results to virtual memory lags the performance of CD++ in respect to ADEVS. We can analyze these last results when the depth varies and keeping the rest of the values constant.

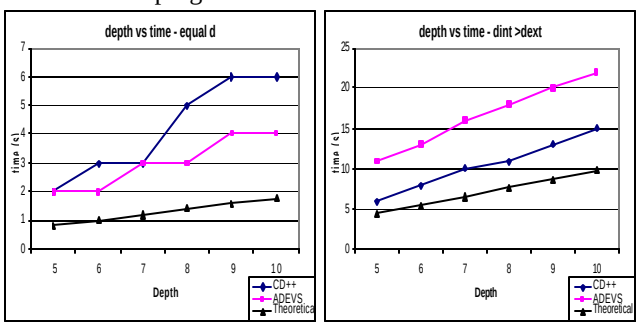


Figure 10: HO Plot; $d_{int} = d_{ext;}\ d_{int} > d_{ext}$

From the graphs, it can be seen that the overall tendency is kept, when $\delta_{int} = \delta_{ext}$ ADEVS behaves better than CD++, and when $\delta_{int} < \delta_{ext}$ both simulators have similar performance and lastly when $\delta_{int} > \delta_{ext}$ CD++ outperforms ADEVS. For HOmod models the parameters need to be changed, due to the exponential growth of messages between coupled components, however the time given to each transition function is kept equal. But even then, CD++ had some problems with the last simulation, being terminated by the OOM Linux service. In this case the number of messages passing between components is many times greater than the number of components. The simulation parameters used for the width test were:

		1		2		3	
Length	Width	δ_{int}	δ_{ext}	δ_{int}	δ_{ext}	δ_{int}	δ_{ext}
5	5-8	0.1	0.1	0.1	1	1	0.1
3-6	5						

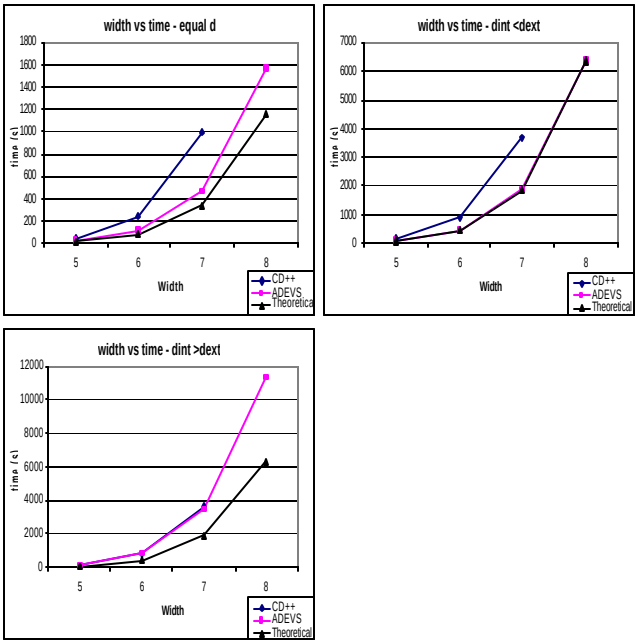


Figure 11: HOmod Plot; $d_{int} = d_{ext;}\ d_{int} < d_{ext}$; $d_{int} > d_{ext}$

From the graphs it can be seen that the general overall tendency is kept constant, ADEVS behaves better than CD++ for the first and second cases. It is important to note that CD++ could not provide a result for w=8 and was terminated by the OOM service, most likely by the use of symbol look-up library of CD++. In the second case where $\delta_{int} < \delta_{ext}$ ADEVS seems to perform better than expected, very close to the theoretical value. The behaviour captured so far suggests that ADEVS performs very close to the theoretical curve, whereas CD++ is almost 0-20% slower than ADEVS, which is a different behaviour compared to the one seen using previous models of DEVStone.

For the last case both simulators have similar behaviour or very little difference in the performance. When $\delta_{int} > \delta_{ext}$, although inconclusive, CD++ and ADEVS seem to match each other performance for larger values of w.

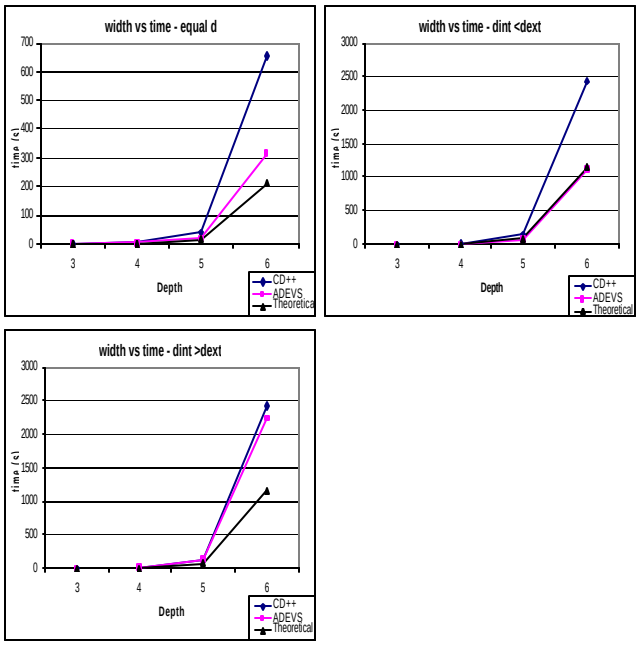


Figure 12: HOmod Plot; $d_{int} = d_{ext}$; $d_{int} < d_{ext}$; $d_{int} > d_{ext}$

This last test confirms the results provided by the test analyzed before. Where CD++ and ADEVS present a very close performance for models where the internal transition function is greater than the external transition function. For the rest of the cases ADEVS performs better than CD++, although depending on the model and complexity of it. Along with this trend, CD++ offers equivalent performance for models that consume equal time in the transition functions as well as models that have bigger loads in the external transition function.

4. CONCLUSIONS

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

DEVStone makes it possible to: (i) create models with different sizes, shapes, and behavior; (ii) generate an arbitrarily large number of such models; and (iii) execute those models using the simulator(s) under study. A precise performance characterization of a simulator allows modelers to consider the actual overhead of the tool based on solid results, and then analyze the feasibility of executing timed models with specific timing constraints. Our framework provides a common metric to compare the results that were obtained using the different simulation tools, although we restricted our case study to the existing CD++ and ADEVS simulation engines, the same ideas may hold for other DEVS-based simulators. There is no doubt that the particular implementations of the DEVStone benchmark for both CD++ and ADEVS can be improved for code efficiency or for simulation performance depending on what the end-user requires.

REFERENCES

1. Zeigler, B.; Kim, T.; Praehofer, H. Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems. Academic Press. 2000.
2. Wainer, G. “CD++: a toolkit to develop DEVS models”. Software - Practice and Experience. vol. 32, pp. 1261-1306. 2002.
3. Rodriguez, D.; Wainer, G. “New extensions to the CD++ tool”. Proceedings of the SCS Summer Computer Simulation Conference. Chicago, USA. 1999.
4. Troccoli, A.; Wainer, G. “Implementing Parallel Cell-DEVS”. Proceedings of 36th IEEE/SCS Annual Simulation Symposium. Orlando, USA. 2003.
5. Glinsky, E.; Wainer, G. “Definition of Real-Time simulation in the CD++ toolkit”. Proceedings of the SCS Summer Computer Simulation Conference. San Diego, USA. 2002.
6. Nutaro, J. ADEVS website. Available via http://www.ece.arizona.edu/~nutaro/. Accessed on May 27, 2003.
7. Zeigler, B.; Moon, Y.; Kim, D. “DEVS-C++: A High Performance Modeling and Simulation Environment”. 29th Hawaii International Conference on System Sciences (HICSS'96) Volume 1: Software Technology and Architecture. Hawaii, USA. 1996.
8. Zeigler, B.P.; H.S. Sarjoughian, “Support for Hierarchical Modular Component-based Model Construction in DEVS/HLA”. Simulator Interoperability Workshop, 99S-SIW-066.
9. Sarjoughian, H.S.; Zeigler, B.P. “DEVSJAVA: Basis for a DEVS-based collaborative M&S environment”. Proceedings of the SCS International Conference on Web-Based Modeling and Simulation, vol. 5, pp. 29-36. San Diego, USA. 1998.
10. Kim, T.G. “DEVSim++: C++ based Simulation with Hierarchical Modular DEVS Models”. User’s Manual CORE Lab, EE Dept, KAIST, Taejon, Korea. 1994.
11. Filippi, J-B.; Bernardi, F.; Delhom, M. “The JDEVS environmental modeling and simulation environment” Proceedings of the the IEMSS’02 Conference on Integrated Assessment and Decision Support. Lugano, Switzerland. 2002.
12. de Lara, J.; Vangheluwe, H. “ATOM3: A Tool for Multi-Formalism Modeling and Meta-Modeling". European Joint Conferences on Theory And Practice of Software. Grenoble, France 2002.

13. Praehofer, H.; Sametinger, J.; Stritzinger, A. “Discrete Event Simulation using the JavaBeans Component Model”. Proceedings of International Conference On Web-Based Modeling & Simulation. California. 1999.
14. DEVSTONE: a Benchmarking Technique for Studying Performance of DEVS Modeling and Simulation Environments”. E. Glinsky, G. Wainer. In *Procedings of IEEE in IEEE/DS-RT*. Montréal, QC. 2005.
15. Troccoli, A.; Wainer, G. “Performance Analysis of Cellular Models with Parallel Cell-DEVS”. Proceedings of the SCS Summer Computer Simulation Conference. Florida. 2001.
16. Weicker, R. P. “Dhrystone: A synthetic systems programming benchmark”. Communications of the ACM, volume 27, pages 1013-1030, 1984.