DEVSTONE: a Benchmarking Technique for Studying Performance of DEVS Modeling and Simulation Environments

Ezequiel Glinsky 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

Abstract

DEVS is a sound, 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 wide variety of systems. As the systems under study become larger and more complex, the performance of the simulator becomes critical. Nevertheless, evaluating the performance of such simulators is a complex process that requires the execution of large numbers of models with different characteristics. We present DEVStone, a synthetic benchmark devoted to automate the evaluation of DEVS-based simulation approaches, which generates models similar to those existing in the real world. DEVStone was used to study the efficiency of five simulation engines provided by CD++ (one of the existing DEVS-based M&S tools) with a very 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

DEVS (Discrete EVents systems Specification) [1] is a sound formal framework for discrete-event modeling based on generic dynamic systems concepts, which supports provably correct, efficient, event-based, distributed simulation. A real system modeled with DEVS is 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 and reducing development and testing time. There is a common preconception that the hierarchical nature of DEVS models may degrade the efficiency of model execution. These arguments are based primarily on intuition, as there are no studies providing a means to compare the efficiency of DEVS simulators, nor comparing different versions of a particular simulation engine. We decided to attack this open discussion by doing a thorough analysis of the performance of DEVS simulators. Instead of limiting our effort solely to testing individual models, we developed a synthetic benchmark to aid not only this but also future initiatives in the area. To do so, we introduced 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 and flexible 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 it provides a common metric to compare the results using different tools.

Our particular implementation of the DEVStone benchmark was used to test the performance of CD++ [2], a tool that implements DEVS theory. CD++ was revised and extended several times, and it supports stand-alone [2], parallel [3] and RT simulation [4]. CD++ and has also been successfully used to model a variety of applications: urban traffic [5, 6, 7]; complex physical systems [8, 9]; computer architectures [10]; and different ecological, artificial, and biological systems [9].

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. Moreover, the performance results permitted us to characterize execution time of a standard DEVS simulator. The benchmark can be used to determine which directions and decisions should be taken when updating the tool's simulation techniques. Furthermore, DEVStone can be used to aid the measurement and improvement of other existing simulation tools.

2. THE DEVS FORMALISM

DEVS is a mathematical formalism that has well-defined concepts of hierarchical and modular model construction, coupling of components, and support for discrete event model approximation of continuous systems. The hierarchical nature of DEVS allows coupling existing models in a modular fashion in order to build larger systems. Since the formalism is closed under coupling, a coupled model can be treated as a basic DEVS component, enabling reuse of previously tested components. An atomic DEVS model is formally described as follows:

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

At any time, the system is in some state $s \in S$. In the absence of external events, the system stays in state s for the time specified by ta(s). When that time is consumed, the system outputs the value $\lambda(s)$ and changes immediately to the state $s' = \delta_{int}(s)$. If an external event $x \in X$ is received before the expiration time ta(s), the new state s' of the system is determined by $\delta_{ext}(s, e, x)$, where e is the time elapsed since the last transition. 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 $\boldsymbol{X}$ is the set of input events; $\boldsymbol{Y}$ is the set of output events; $\boldsymbol{D}$ is an index for the components of the coupled model, and $\forall\ i \in D$, $\boldsymbol{M_i}$ is a basic DEVS (*i.e.*, an atomic or coupled model), $\boldsymbol{I_i}$ is the set of influencees of model i. The influencees of a model define to which model outputs must be sent. and $\forall\ j \in I_i$, $\boldsymbol{Z_{ij}}$ is the i to j translation function, which converts the outputs of a model into inputs for other models.

We carried out our experiments in the CD++ environment. This tool has been revised, upgraded and extended several times since its release. It currently has a fairly large community of users, some of which work in updating the simulation engines. The toolkit provides a specification language that allows describing model coupling, setting initial values, and indicating external input events. Atomic DEVS models are developed in C++, providing flexibility to the modeler. CD++ was built as a class hierarchy that corresponds to simulation entities, as defined in [1].

3. DEVSTONE: A SYNTHETIC MODEL GENERATOR

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 on performance results for a given tool, and were usually limited to a given type of models of interest. In [11], the authors presented performance studies of Cell-DEVS models in a parallel simulation environment. In [12], the authors focused on a watershed model to show performance improvements in parallel and distributed architectures. DEVS was shown to be more efficient than the continuous counterparts when simulating natural [13], and artificial systems, such as a photovoltaic system [14]. However, those studies do not provide a thorough analysis for the execution of models with different characteristics, neither do they give a common metric to compare results among different DEVS simulators.

We propose a method to compare not only different versions of a particular simulation engine but also different DEVS-based software. Varied model structures permit obtaining prototypes representative of those found in real world applications. In order to provide a complete and thorough evaluation, we created a synthetic model generator, called **DEVStone**, which produces a variety of models with diverse structure and behavior performing a mix of common operations. We focus in the aspects that impact performance: the 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), and *external transition time* (the execution time spent by external transition functions).

In general, being *d* the depth and *w* the width, we build an artificial coupled model with *d* coupled components in the hierarchy, all of which consist of *w-1* atomic models, with the exception of the lower level of the hierarchy, in which a coupled component is composed of a single atomic model. In addition, internal and external transition functions are programmed to execute a fixed amount of time, as specified by the user. In both transition functions we consume CPU clocks by running Dhrystones [15]. Dhrystone code consists of a mix of instructions using integer arithmetics, therefore it is a good choice for analyzing models like DEVS in which state variables have discrete values.

DEVStone uses three 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, and
- **HO** models with high level of coupling and numerous 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.

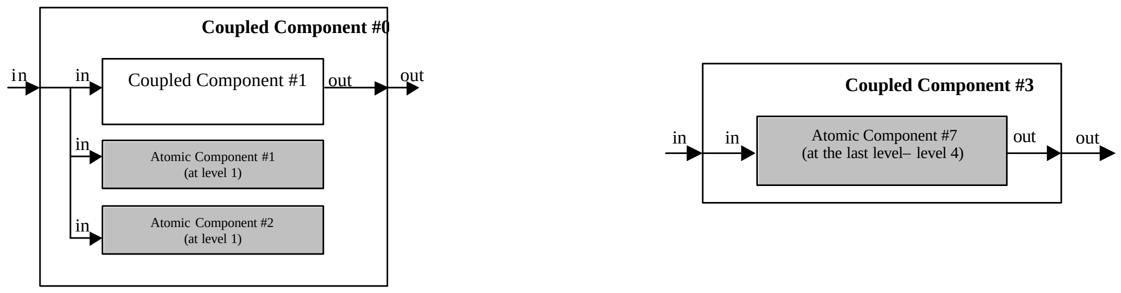

Coupled Component #0
in
in
Coupled Component #1
out
out
in
Atomic Component #1 (at level 1)
in
Atomic Component #2 (at level 1)
Coupled Component #3
in
in
Atomic Component #7 (at the last level– level 4)
out
out

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

As we know the model structure and the time spent by each component in executing transition functions, we can compute the theoretical execution time for the model. First, we devise the number of atomic and coupled models in the structure, which 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. Thus, the number of internal and external transition functions to be triggered equals the number of atomic components in the model, as shown below.

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

HI models have the same number of atomic components, but more interconnections. 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,

$$\begin{aligned} \#\,Atomic\ Models &= (w - 1) * (d - 1) + 1\\ \#\,Internal\ Transitions &= S_{(i=1\,..\,w-1)}\ i * (d - 1) + 1 \qquad (2)\\ \#\,External\ Transitions &= S_{(i=1\,..\,w-1)}\ i * (d - 1) + 1 \end{aligned}$$

HO type models have a more complex interconnection scheme (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, and consequently generates more overhead. For this model type we have,

$$\begin{aligned} \#\,Atomic\ Models &= (w - 1) * (d - 1) + 1\\ \#\,Internal\ Transitions &= S_{(i=1\,..\,w-1)}\ i * (d - 1) + 1 \qquad (3)\\ \#\,External\ Transitions &= S_{(i=1\,..\,w-1)}\ i * (d - 1) + 1 \end{aligned}$$

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

4. PERFORMANCE ANALYSIS OF VIRTUAL-TIME SIMULATORS

CD++ has been provided with different simulation techniques. A stand-alone simulator is used in single-processor simulations, while the parallel version (based on [16]) was built on top of Warped [17], which provides different optimistic synchronization algorithms and a non-synchronized protocol (called *NoTime*). We will present the results obtained when DEVStone was applied to characterize the overhead of these simulators. The first tests are devoted to compare the overhead of three CD++ simulators: (i) original, (ii) parallel with unsynchronized kernel, and (iii) parallel with optimistic kernel. We compared the execution results with the theoretical execution time for each type, computed as in equation (4).

Simulation	**Model Type**	**Depth**	**Width**	d_{int}	d_{ext}
E	HI	3	6	50 ms	50 ms
F	HI	6	3	50 ms	50 ms
G	HI	5	5	50 ms	50 ms
H	HI	6	6	50 ms	50 ms
I	HO	3	6	100 ms	0 ms
J	HO	6	3	0 ms	100 ms
K	HO	5	5	50 ms	50 ms
L	HO	6	6	50 ms	50 ms

Table 1: Simulation parameters

These models were executed using 10 external events at a constant rate, each of them triggering a known number of external and internal transition functions defined by equations (1), (2), and (3). We use the Dhrystone code to compute the time spent on each of these functions, which is used to calculate the time required to process a single event. Table 1 shows the parameters we used for different tests, including model type, structure and time spent on transition functions. The results presented in the following figures summarize the execution of a batch with hundreds of models. We report the worst execution case from the group (variation for the best execution cases was less than 1%, mainly due to overhead of the OS). The experiments were executed in a single processor, allowing us to measure the pure overhead incurred by the parallel simulator. As expected, the original version provided the best execution time, whereas the parallel unsynchronized kernel (NoTime) version always outperformed the parallel optimistic (TimeWarp) version.

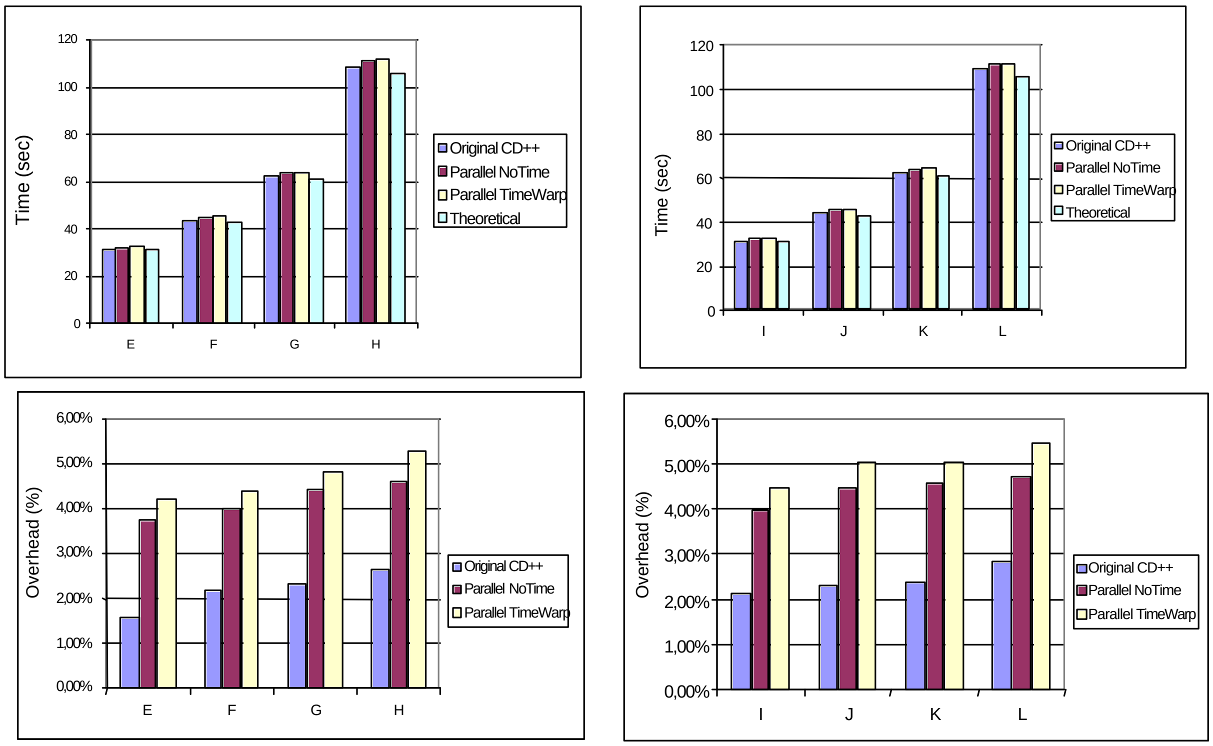


Figure 2: Execution times with (a) HI, (b) HO; Overhead (c)HI (d) HO.

The amount of overhead for the original version is only between 1% and 3% in every case. Likewise, the percentage of overhead of the parallel versions is below 5.5% for the most complex problems (a promising result, as the amount of speedup time achievable by these simulators is considerable). If we analyze the overhead spent in simulating models with identical number of components and transition functions, but different structure, we can determine the impact of model complexity on the overall performance.

When we analyze HI models E (3x6) and F (6x3), we can see the greater impact over efficiency when executing models with larger number of levels. Here, both models execute the same number of transition functions (Model F creates a larger number of intermediate coordinators). Model E has an overhead of 1.59%, and F 2.18%, showing the impact of structure changes. As seen in Figure 2 (c), spending 100 ms in the internal and none in the external transition (model I, 3x6), or vice versa (model J, 6x3) does not seem to affect the performance. Although the simulation of the latter incurs approximately 10% more overhead, this is a result of using a deeper structure. Model K (HO) has 17 components, whereas model L has 26 components (approximately 50% more models). However, simulating model L incurs only 21% more overhead (2.89%) than model K

(2.37%) using the original version (approximately 10% more if we consider the parallel alternatives).

Despite these favorable results, we obtained worse overhead ratios for models with a very large number of components (approximately 10,000), due to the number of messages interchanged. CD++ used the abstract simulation algorithms presented in [11], which create a one-to-one correspondence between DEVS models and execution engines called *processors*: *simulators* execute atomic models, and *coordinators* execute coupled models, as shown in .

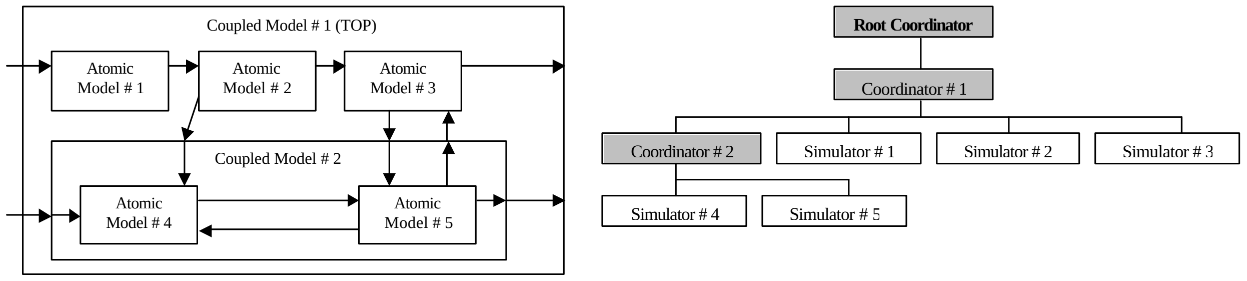


Figure 3: (a) Sample model structure, (b) Associated processor hierarchy

The number of these intermediate coordinators can be arbitrarily large depending on the model under study. Table 2 shows the number of simulators and coordinators created for large sample models, and the number of messages involved in the processing of a single external event. Models R and S have identical structure; they only differ in the interconnections of their inner components, resulting in a remarkable difference in the number of messages involved. The same occurs when models T and U are compared.

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

Table 2: Simulated models using the hierarchical simulation approach

The overhead obtained when these models were executed is now approximately 10%. Observing the table, we see that model U has 12.50% more components than S, although the overhead incurred by CD++ on executing the former is lower. The same happens with models T and R. These results provided a hint to optimize the simulation technique: reducing intermediate coordinators would improve time spent routing messages. A new simulation technique flattens the simulation hierarchy as suggested in [18], reducing the number of messages exchanged. The idea is to create only two processors: one root coordinator and one flat coordinator in charge of the tasks of simulators and coordinators. The flat coordinator executes the δ_{int}, δ_{ext} and $\lambda(s)$ functions for each atomic component, transforming the hierarchical structure of the model into a flat structure by mapping the ports for all atomic and coupled components in the hierarchy.

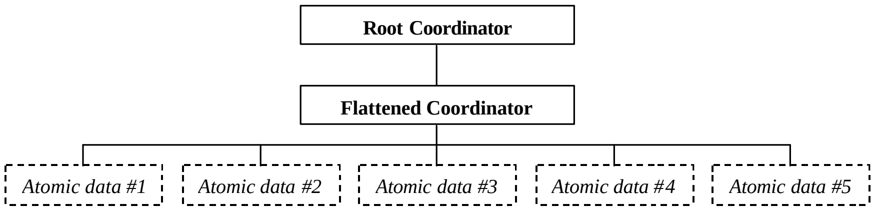


Figure 4: Flat simulator approach for Figure 5

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

	Model			
Simulation parameter	R	S	T	U
Number of components per level	100	100	150	150
Number of levels in the hierarchy	100	100	75	75
Model Type	LI	HO	LI	HO
Number of atomic components	9,802	9,802	11,027	11,027
Number of simulators	0	0	0	0
Number of coupled components	99	99	74	74
Number of coordinators	0	0	0	0
Number of root-coordinators	1	1	1	1
Number of flat coordinators	1	1	1	1
Number of messages exchanged to process a single external event	4	9804	77	11029
Execution overhead	4.51%	5.64%	4.38%	5.21%

Table 3: Simulated models using the flat simulation approach

Figures 5 (a) and (b) contrast the performance of both simulators. The figures show the differences between actual and theoretical execution times when running different types of models with variable depth (fixed width 6), and with variable width (fixed depth 7) respectively. Each of its components has a workload of 50 ms for the transition functions, and models receive 100 events at a fixed frequency.

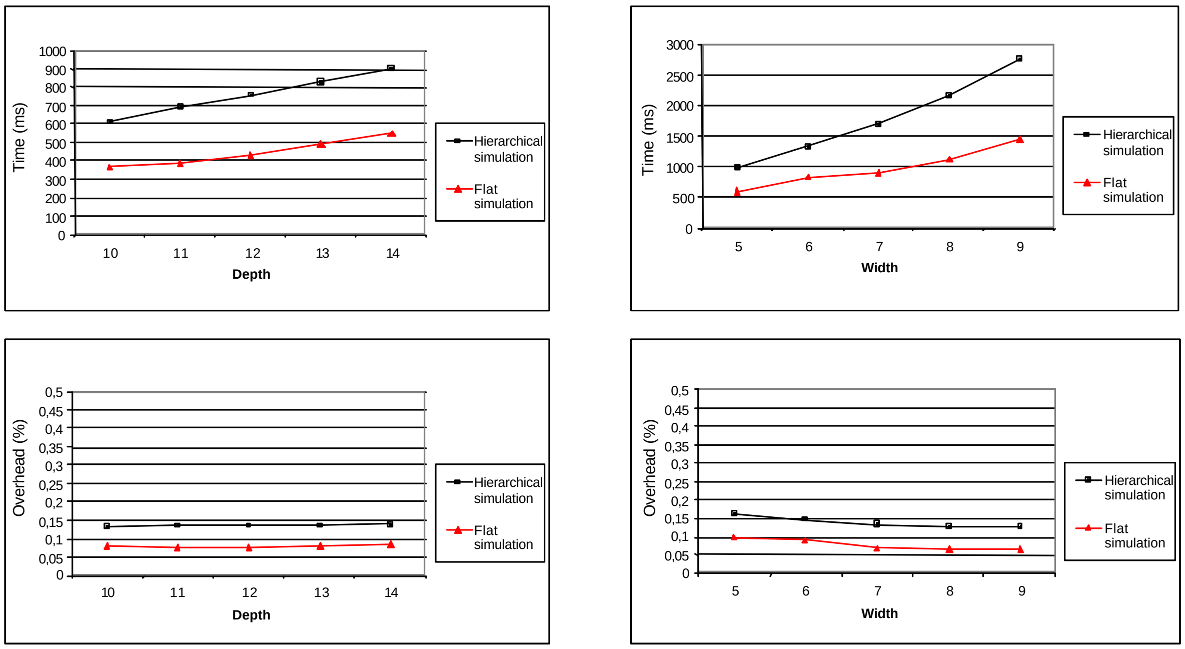
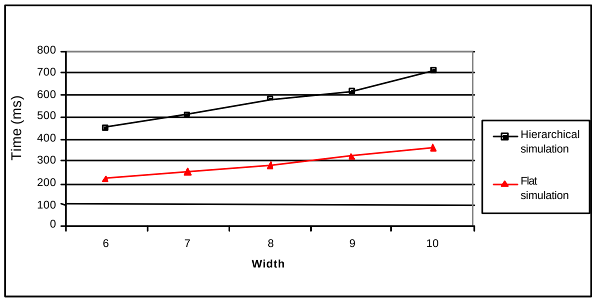


Figure 5: Difference between theoretical/ and execution times: (a) LI, (b) HO; Overhead comparison of hierarchical and flat simulators: (c) LI, (d) HO

Regardless of the type of model (LI or HO) and the structure (deeper or wider), we can see a clear improvement in execution times when using the flat approach. When executing a small LI model (10x6) the difference between theoretical and execution time is reduced in 38.3%. For larger LI models the reduction remains stable or even rises marginally (*e.g.*, for the a 14x6 model, it is 39.6%). Similar results can be seen in the execution of these HO models, which are more complex and have approximately the same size (*e.g.*, 39.4% for a 7x5 HO model, 47.7% for a 7x9 HO model). In general, the reduction obtained for these HO models is in the range of 39.4%-47.7%. The overhead is always less than 0.16% for the hierarchical simulator, and 0.1% for the flat simulator. Even though models in Figure 8 (a) are larger than those in Figure 8 (b), the latter ones have more complex structures that compensate the differences in size, resulting in similar percentages of overhead. The flat simulator reduces the overhead in 40%-56% depending on the depth, width, and complexity of the model, although it is important to note that the overhead is stable for both techniques.

In order to analyze the performance degradation purely due to overhead in the simulation engine, we executed several examples with empty transition functions (total execution time is solely depending on the message exchange).



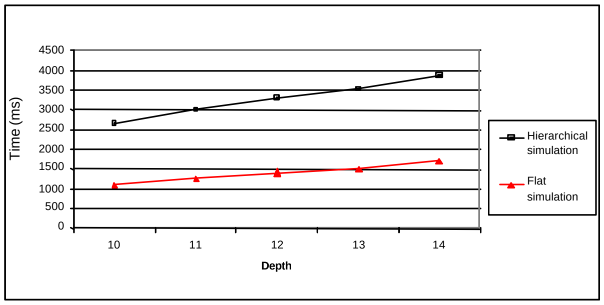


Figure 6: Execution time for hierarchical/flat simulators: (a) LI (b) HO

Figures 9 (a) and (b) show the execution of several LI models with variable width and fixed depth of 8, and HO models with variable depth and fixed width of 8 (*i.e.*, models that have between 36 and 92 components). Figure 6 (a) shows that regardless of the model's width, the flat simulator reduces the execution time in 52.4%-54.7%. Figure 9 (b) shows that for HO models the improvement in performance becomes more noticeable when the depth of the model increases. The impact of the intermediate coordinators that are eliminated from the hierarchy results in fewer messages being exchanged, and more efficient simulation. The obtained results are similar to those presented before; the flat approach provides a reduction of 52%-58% in the total execution time depending on the size and type of the model.

5. PERFORMANCE ANALYSIS OF RT SIMULATORS

CD++ was extended to allow RT execution [4]. In RT systems, correctness depends not only on computation results, but also on the time at which the results are produced [**Error! Reference source not found.**]. A correct answer after its deadline is regarded as an unsuccessful response. The RT CD++ extension allows the execution of events triggered by the RT clock, enabling interaction between the models and their environments.

In order to analyze the performance of the new simulator, we used DEVStone to study the performance of different models in RT. We started comparing the virtual-time and RT techniques, in order to assess the efficiency of the latter. Our approach consisted in executing virtual-time and RT simulations using DEVStone, and comparing the time required to process a single event in virtual time against the ir worst-case response in real time. The worst-case response time represents a meaningful measure since RT systems deal with predictable responses in critical applications.

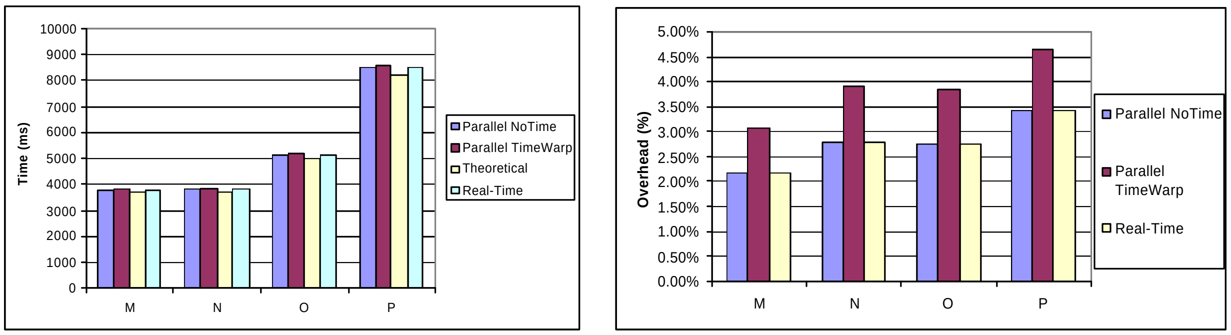


Figure 7: Comparing RT and virtual-time simulators: (a) Execution times (b) overhead

Figure 7 shows a comparison for sample LI models M (5x10), N (10x5), O (8x8), and P (10x10), with internal and external functions executing 50 ms of Dhrystone code. In all cases, models received 100 events at a constant rate, and we measured the worst-case response time. Since the new functionality that deals with deadlines adds a small amount of additional overhead in the RT version, it was expected to observe some degradation in performance. Nevertheless, the experiments showed that the added overhead is actually imperceptible. In general, we see that the RT results are in the same level of those obtained with the parallel unsynchronized version (incurring overheads in the range of 2%-3.5%) and outperforming the optimistic TimeWarp simulator (with overheads in the range of 3%-4.6%).

We used DEVStone to measure the ability of CD++ to execute models with different structure under very demanding situations. Simulations received a fixed number of external events generated at a constant rate. Each model was expected to deliver responses for each event before a given deadline, and the percentage of success and worst-case response time were obtained as follows,

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

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

where r_i is the response time for the *i-th* event, and *N* is the number of events.

The models used in the following experiments had different sizes ranging from 10 to 35 components in total (width and depth are in the range of 4 to 11). There is no Dhrystone code generated in the internal or external functions (all the time spent is overhead). The results are grouped in four categories: (1) LI models with variable depth, (2) LI models with variable width, (3) HO models with variable depth, and (4) HO models with variable width.

In the first set of experiments, each simulation receives 100 events with inter-event periods of 100 ms, and associated deadlines were set at 100 ms. The results are shown in the next figure.

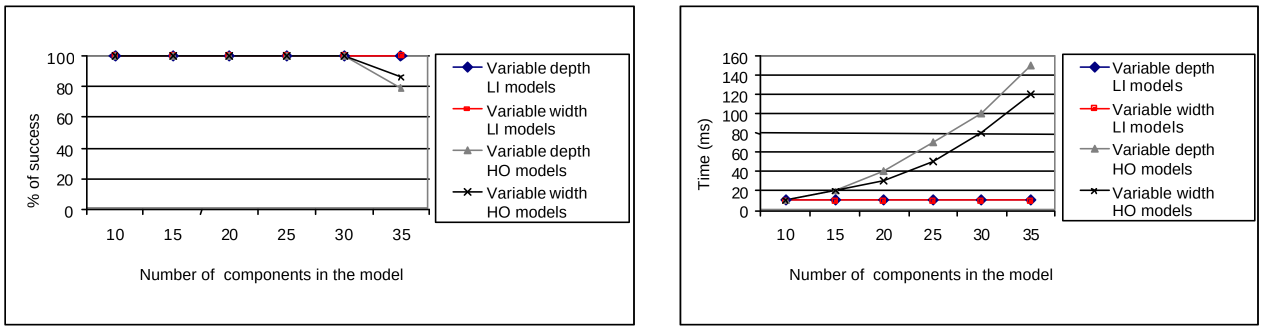


Figure 8: Comparing variable depth/width (a) Percentage of success, (b) WCET

Figure 8 (a) shows the percentage of success for LI and HO models when depth is variable and width is fixed, and also when width is variable and depth is fixed, whereas Figure 8 (b) illustrates the worst-case response time for each case. We observe that the RT simulator was able to deliver all the outputs on time for models with 30 or fewer components, regardless the model type. However, when the size of the model is 35, HO models started missing a few deadlines. The pace of external events and the complexity of the model prevent a timely response to some of the external events. For a HO model with 35 components, the success was 79% and 86%. In contrast, LI models always met all their deadlines under the same circumstances (a result of the greater complexity of HO models). Figure 8 (b) illustrates that the worst-case response time is gradually deteriorated for HO models as a result of the increased number of messages exchanged. LI models always had a worst-case response time of 10 ms in these conditions, despite the size of the model, showing that their interconnection pattern is easier to handle. These results provide a reference about the conditions in which the engine is capable of meeting the deadlines, focusing on a particular scenario (inter-event periods and associated deadlines) and the characteristics of the model (size and model type). In the next set of experiments, we executed larger LI and HO models (with between 25 to 50 components). We focused on the performance of the RT simulator under a more stressful scenario, with external events arriving at higher frequencies with very strict deadlines. This test focuses on the cases where the simulator is unable to meet its deadlines due to highly overloaded conditions. External events (e_i) arrive every 30 milliseconds and deadlines (d_i) are set at 60 ms after their arrival, as shown in Figure 11.

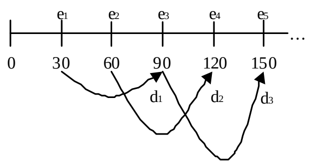


Figure 9: Event arrival and deadlines

An alternative analysis can be performed focusing on the surrounding environment; for instance, studying the effect of different inter-event periods (*i.e.*, frequency of event arrivals) on the execution performance. In the following set of experiments, events arrive at different pace (20 to 180 ms), and we analyze the behavior of the simulator under such circumstances. The charts show the results for 8x8 HO models receiving 100 events with deadlines set at 1000 ms.

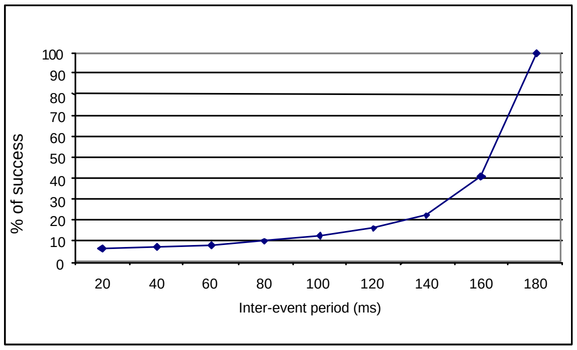
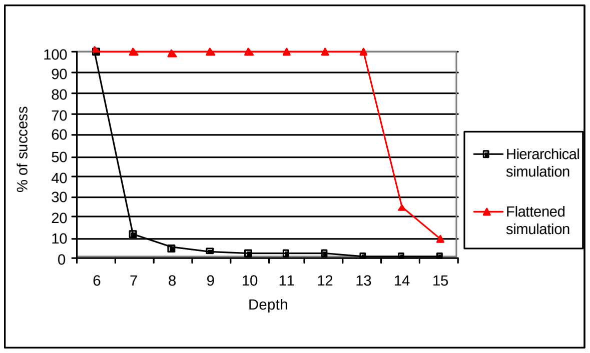


Figure 10: Worst-case execution time for HO models with variable inter-event period:

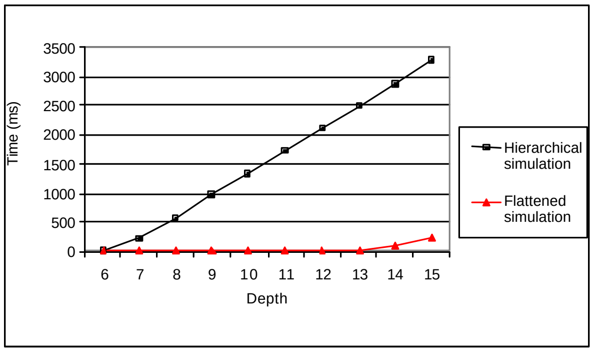
Figure 10 shows that larger inter-event periods result on greater percentages of success. When the intervals between events become greater than 180 milliseconds, the simulator meets all the associated deadlines for the execution of this 8x8 model.

In the last set of experiments, we contrasted the RT performance of the flat simulator against that of the hierarchical simulator. We analyzed both simulators executing HO models with variable depth and width, focusing on complex models and examining their performance under demanding conditions. The next chart shows the results for models with fixed width of 9 and variable depth (from 6 to 15 levels, *i.e.*, between 41 and 113 atomic components in total) that received 100 events at a fixed arrival rate.

In this scenario, the flat approach allowed the simulation of a 13x9 HO model (97 components) with minimum worst-case response times, while meeting all the deadlines. When using the flat approach, performance degradation is first noticed in the execution of a 14x9 model (105 components). In contrast, even when simulating a smaller 7x9 model (49 components) the hierarchical approach had an inferior percentage of success (87%) and showed worse response times. Figure 15 (b) shows that in the larger models the difference between the hierarchical and flat simulators becomes more noticeable in terms of the worst-case response times, as a result of the greater overhead incurred in the simulation of deeper models. Our analysis showed similar results when models with variable width were executed. Similarly to what was illustrated before, the flat coordinator simulated wider models more effectively.



(a)



(b)

Figure 11: Comparison of hierarchical and flat approaches (variable depth): (a) Percentage of success, (b) Worst-case response time

In general, the flat technique outperforms the hierarchical one, reducing the incurred overhead up to 50% and therefore providing improved response times and better percentage of success in the execution. Thus, the use of the non-hierarchical approach allows the simulation of larger models with better performance results. These results are a consequence of the reduced number of messages exchanged in the flat simulation mechanism.

6. CONCLUSIONS

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

We developed DEVStone, a synthetic model generator that supports the process of evaluating the performance of simulation engines. DEVStone produces models that are similar to those existing in the real world, thus making 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.

DEVStone relies on executing a collection of models with different characteristics. In order to emulate several degrees of complexity in their structures, we identified three types of models that correspond to three interconnection patterns. In addition, each atomic component usually executes code in its transition functions; we proposed the use of Dhrystone code to resemble the task to be performed by these components.

As a result, we have a systematic way to assess the performance of a simulation engine, reducing the time required to measure its efficiency. It is possible to analyze the efficiency of any DEVS simulator with relation to the size, the behavior and the structure of the model under execution. 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, and also enables an analysis of the efficiency of successive versions of the same simulator, such as upgrades or fixes. We used the CD++ toolkit to show how to apply the proposed benchmark. These experiments allowed us to test the usefulness of the benchmark, and to thoroughly test CD++ (which is the first systematic effort to characterize the performance of DEVS modeling and simulation environments). Although we restricted our case study to the existing CD++ simulation engines, the same ideas may hold for other DEVS-based simulators. Using DEVStone, we showed that hierarchical simulation techniques are capable to simulate models with low overhead, even for models with complex structure. By means of the proposed framework, the performance of both virtual-time and RT hierarchical simulators was shown to be satisfactory. Moreover, the results demonstrated that the flat simulation technique could improve the efficiency in some cases, especially when model structure is particularly large or complex. Regardless of the size and complexity of the models, the flat simulator outperformed the hierarchical one. In general, the charts illustrate that the overhead incurred by the flat simulator is reduced up to about 55% of the overhead incurred by the hierarchical approach.

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. Troccoli, A.; Wainer, G. “Implementing Parallel Cell-DEVS”. Proceedings of 36th IEEE/SCS Annual Simulation Symposium. Orlando, USA. 2003.

4. Glinsky, E.; Wainer, G. “Definition of RT simulation in the CD++ toolkit”. Proceedings of the SCS Summer Computer Simulation Conference. San Diego, USA. 2002.
5. Davidson, A.; Wainer, G. “Specifying truck movement in traffic models using Cell-DEVS”. Proceedings of the 33rd IEEE/SCS Annual Symposium on Computer Simulation. Washington, DC. 2000.
6. Lo Tartaro, M.; Torres, C.; Wainer, G. “Defining Models of Urban Traffic using the TSC Tool”. Proceedings of the SCS Winter Simulation Conference. Washington, USA. 2001.
7. Díaz, A.; Vázquez, V.; Wainer, G. “Application of the ATLAS language in models of urban traffic”. Proceedings of the Annual Simulation Symposium. Seattle, USA. 2001.
8. Ameghino, J.; Troccoli, A.; Wainer, G. “Models of complex physical systems using Cell-DEVS”. Proceedings of the 34rd Annual Simulation Symposium. Seattle, WA. 2001.
9. Ameghino, J.; Wainer, G.; Glinsky, E. “Applying Cell-DEVS in Models of Complex Systems”. Proceedings of the SCS Summer Computer Simulation Conference. Montreal, Canada. 2003.
10. Wainer, G.; S. Daicz, S.; De Simoni, L.; Wasserman, D. “Using the ALFA-1 simulated processor for educational purposes”. ACM Journal on Educational Resources in Computing. vol. 1, no. 4. pp. 111-151. December 2001.
11. Troccoli, A.; Wainer, G. “Performance Analysis of Cellular Models with Parallel Cell-DEVS”. Proceedings of the SCS Summer Computer Simulation Conference. Florida. 2001.
12. 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.
13. Zeigler, B.; Moon, Y.; Kim, D.; Ball, G. “The DEVS Environment for High-Performance Modeling and Simulation” IEEE Computational Science and Engineering, vol. 4 (3), pp. 61 - 71. 1997.
14. Filippi, J-B.; Chiari, F.; Bisgambiglia, P. “Using JDEVS for the modeling and simulation of natural complex systems”. Proceedings of the AI, Simulation and Planning Conference. Lisbon, Portugal. 2002.
15. Weicker, R. P. “Dhrystone: A synthetic systems programming benchmark”. Communications of the ACM, volume 27, pages 1013-1030, 1984.
16. Chow, A.; Ziegler, B. “Parallel DEVS: A parallel, hierarchical, modular modeling formalism”. Proceedings of the SCS Winter Simulation Conference. Orlando, USA. 1994.

17. Martin, D.; McBrayer, T.; Radhakrishan, R.; Wilsey, P. "Time Warp Parallel Discrete Event Simulator". Technical report. Computer Architecture Design Laboratory. University of Cincinnati. USA. 1997.
18. Kim, K.; Kang W.; Sagong, B.; Seo, H. "Efficient Distributed Simulation of Hierarchical DEVS Models: Transforming Model Structure into a Non-Hierarchical One". Proceedings of the 33rd Annual Simulation Symposium. Washington DC, USA. 2000.
19. Liu, J. RT Systems. Prentice Hall. 2000.