PERFORMANCE ANALYSIS OF DEVS ENVIRONMENTS

Ezequiel Glinsky

Departamento de Computación
FCEN – Universidad de Buenos Aires
Planta Baja. Pabellón I.
Ciudad Universitaria (1428)
Buenos Aires. Argentina.

Gabriel Wainer

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

ABSTRACT

The CD++ toolkit was developed in order to implement the theoretical concepts specified by the DEVS formalism. CD++ has been enhanced lately to support both parallel and real-time simulation. The abstract simulation algorithms associated with the hierarchical modelling techniques have a cost in terms of processing time overhead. Besides this, using the tool for real-time modelling involve task models that usually do not take overhead into account. Our goal is to characterize the overhead of stand-alone, parallel and real-time simulators available in the toolkit. To do so, we used a synthetic benchmarking tool that can be applied to DEVS environments. Different types of models are tested automatically, making the performance analysis easy. The results here presented show that we are able to develop DEVS models paying a small cost in terms of processing overhead.

1 INTRODUCTION

The **DEVS** (Discrete EVents Systems specifications) formalism [ZPK00] provides a framework for the construction of hierarchical models in a modular fashion, allowing model reuse, and reducing development and testing time. DEVS models can be executed using abstract simulation mechanisms independent of the model itself. Models are built using a set of basic models called **atomic**, which can be combined to form **coupled** ones. A DEVS atomic model is described as:

$$M = < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, D >$$

Here, ***X*** is the input events set, **S** is the state set, and ***Y*** is the output events set. There are also several functions: δ_{int} manages internal transitions, δ_{ext} external transitions, λ the outputs, and **D** the elapsed time.

A DEVS coupled model is defined as:

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

Here, ***X*** is the set of input events, and ***Y*** is the set of output events. **D** is an index of components, and for each i $\in$ D, $\mathbf{M_i}$ is a basic DEVS model, where $M_i = < X_i, S_i, Y_i, \delta_{inti}, \delta_{exti}, ta_i >$. $\mathbf{I_i}$ is the set of influencees of model i. For each j $\in I_i$, $\mathbf{Z_{ij}}$ is the i to j translation function.

The **CD++** toolkit [RW99, WCD01] implements DEVS theory. A specification language allows the creation of coupled models, the initial configuration for the atomic models, and the creation of external events to be used during the simulation. Lately the CD++ tool has been enhanced to support both parallel [TW01] and real-time simulation [GW02].

DEVS provides the advantages of a discrete event approach in terms of execution performance. Discrete event models evolve in continuous time. Events are instantaneous and can occur asynchronously at unpredictable times. DEVS simulators can be seen as hierarchical schedulers of events that activate the corresponding submodels. The schedules allow skipping periods of inactivity in the simulation. Nevertheless, explicit synchronization of the components is required, which involves a certain amount of overhead to be paid. We want to study the overhead involved in models of different complexity, as a first step towards execution in Real-Time environments.

Real-time scheduling techniques try to guarantee the timely execution of a set of subtasks. Most existing schedulers are based in theoretical task models, which usually avoid analyzing the overhead of the algorithm implementations. Our goal here was to be able to analyze the overhead of DEVS tools in each of these cases. To improve the performance testing, we generated a synthetic experimental frame, which can run different types of models automatically. These models have been used for testing the different simulation techniques available, which include stand-alone, parallel and real-time simulators. The tool can be easily applied to any DEVS environment, in order to analyze their overhead.

2 A SYNTHETIC DEVS MODEL GENERATOR

In order to be able to study system overheads in detail, the synthetic model generator can produce a variety of models. The produced models try to mimic the structure of those used in real applications. The shape and behavior of these models can be defined using the following parameters:

1. *model_type*: the amount of messages involved in the simulation is related to the number and type of links between the components. Therefore, this parameter allows us to choose among different predefined interconnections between the components of the model.
2. *depth*: determines the number of levels of the modeling hierarchy.
3. *width*: determines the number of children of each intermediate coupled component. Along with *depth*, it lets us establish the size of the model.
4. *#intdhrystones*: indicates the execution time to be consumed in the *internal transition function*, which allows us to simulate code to be executed.
5. *#extdhrystones*: indicates the execution time to be consumed in the *external transition function*, which allows us to simulate code to be executed.

We used the Dhrystone benchmark [WEI84] to generate different workload in both internal and external transition functions. Dhrystone code is a synthetic benchmark intended to be representative for system (integer) programming. The *#intdhrystones* and *#extdhrystones* parameters allow us to execute time-consuming code inside the internal and external transition, according to a number of milliseconds specified.

2.1 Model type 1

This model type has a small number of interconnections between components. The structure of this model is shown with the following example. If we use a *width* of 3 and a *height* of 4, the *top* model generated will have the structure described in the Figure 1. The arrows indicate the existing input and output ports in each depicted model. Boxes denote the different subcomponents. White-solid boxes represent coupled components and shaded boxes represent atomic components.

The *top* model (*Coupled Component #0*) consists of one coupled component (#1) and two atomic ones (#1 and #2) as shown above. *Coupled Component #1* has the same internal structure as the top model (*Coupled Component #0*). Therefore, it contains one coupled model (*Coupled Component #2*) and two atomic ones (#3 and #4). Likewise, *Coupled Component #2* has the same structure, and accordingly contains *Coupled Component* #3 and *Atomic Components* #5 and #6.

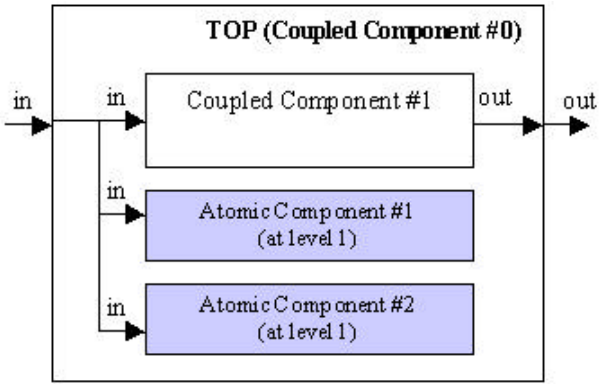

Figure 1: Top Model (type 1)

Finally, *Coupled Component #3* only contains one atomic child (#7) connected to its output port, regardless of the specified width. The structure of this model is shown below.

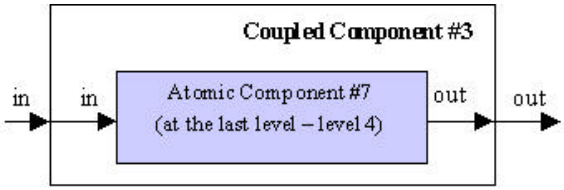

Figure 2: Bottom Coupled Component (type 1)

Given a specified depth *d* and width *w*, we end up having *k* coupled components with *w-1* atomic components inside each model (except for the last coupled model, which will only include one atomic component).

2.2 Model type 2

Model type 2 has more interconnections between the components of each coupled model. The inner atomic components are interconnected; therefore, there is a greater number of messages interchanged in these kinds of models and the overhead grows accordingly.

The following example uses *depth* = 4 and *width* = 4 (as explained before, we have in this case four components per level). The top model (*Coupled Component #0*) is composed by one coupled model and three atomic ones.

Coupled Component #1 is also formed by three atomic components and one coupled model (*Coupled Component #2)*. The same structure can be found in *Coupled Component #2* that is composed by *Coupled Component #3*. Finally, *Coupled Component* #3 is quite simple and identical to the model described in Figure 2 as it is the last coupled component in the produced hierarchy. This outermost model is shown in the following figure:

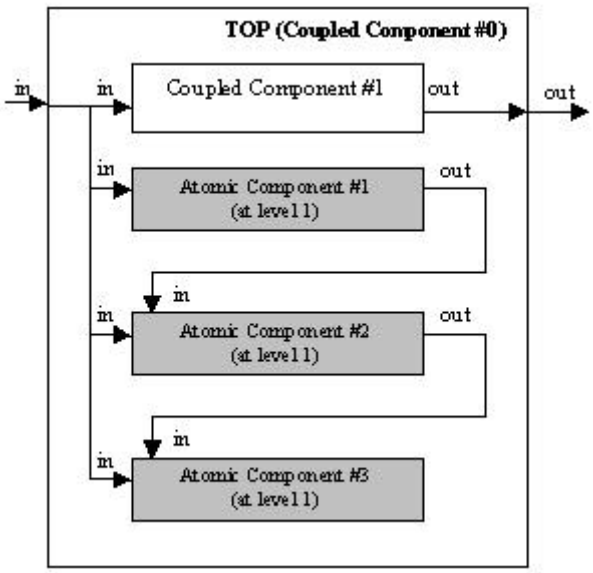

Figure 3: Top Model (type 2)

2.3 Model type 3

Model type 3 is comparable to the type 2, but it also connects the outputs of the inner components to the output of its parent model, thus generating even more overhead in the simulation.

Coupled Component #1 is also formed by three atomic components and one coupled model (*Coupled Component #2)*. The same structure can be found in *Coupled Component #2* that is composed by *Coupled Component #3*. Finally, *Coupled Component* #3 is quite simple and identical to the model described in Figure 2 as it is the last coupled component in the produced hierarchy.

The topmost component of a model with *depth* = 4 and *width* = 4 is depicted below:

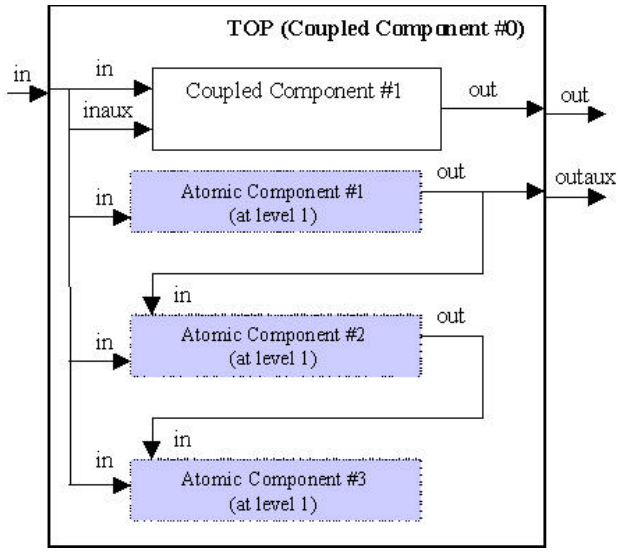

Figure 4: Top Model (type 3)

The figure shows that even more interconnections are added in this type of model. Thus, the overhead incurred due to message passing is increased. The rest of the components are analogous to those described before, and will not be detailed here.

3 PERFORMANCE COMPARISONS

This section describes some of the tests we developed to analyze CD++ performance. Different shapes and sizes of models have been generated in order to simulate different model characteristics.

We studied different versions of the tool, which use different simulation techniques:

- Original CD++ simulator
- Parallel CD++ simulator
- Parallel CD++ simulator with TimeWarp kernel
- Real-time simulator

The parallel version was developed on top of a middleware layer, which ensures synchronization of the models being executed. We are currently using Warped [MMRW97], which provides an API for running parallel simulations. Two simulation kernels are provided: TimeWarp and a non-synchronized kernel (NoTime).

We have executed the synthetic benchmark, and tested performance of the different techniques in a stand-alone computer. The tool lets the user to choose different sizes (by varying the parameters *width* and *height*) and different workloads (by varying the parameters *#intdhrystones* and *#extdhrystones*).

The results were compared with the total theoretical time involved in simulating a model with the same structure. This value is determined by the number of atomic components in the model, the amount of interconnections between them, and the number of internal and external functions being executed. Once we know the number of transition functions, we must multiply these values by the amount of time spent in each transition function. Theoretical execution time does not include any overhead at all and it is the sum of the time spent in transitions. This value is shown in the charts and compared with the obtained execution times.

To measure the total theoretical time involved in a complete simulation, we have to take into account the number of internal and external functions being executed by atomic components. This value can be computed as follows. First, we count the number of atomic models as:

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

The number of atomic components can be used to count the number of external transition functions to be executed upon the reception of an external event. We must multiply these values by the amount of time spent in

each transition function to obtain the *theoretical time* needed to process a single incoming event. Lastly, we multiply this result by the *number of incoming events* received, in order to measure the *total theoretical time*, thus:

$$TotalTheoreticalTime = [(\#AtomicModels * TimeInExternalTrans) + (\#AtomicModels * TimeInInternalTrans)] * NumberOfEvents$$

This number represents the sum of the time spent in transitions. This value is shown in the charts and compared with the obtained execution times that include different execution overheads for each technique. The experiments have been performed running a cycle of 10 external events per simulation.

Here, we show different results obtained according to the class of structure employed. We show examples in different categories, in order to exemplify the results obtained. Table 1 presents the parameters used in the simulations and their associated values.

Simulation	Depth	Width	Internal Transition	External Transition
A	3	10	50 ms	50 ms
B	10	3	50 ms	50 ms
C	5	5	50 ms	50 ms
D	10	10	50 ms	50 ms

Table 1: Simulation's parameters

The following graphs show the results obtained after the execution of these simulations. The first graph shows the execution times for each technique, as well as the corresponding theoretical execution time in order to compare the results.

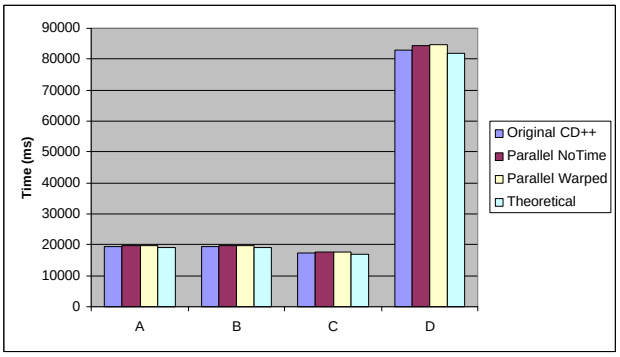

Figure 5: Execution time for different models

The next graph illustrates the difference between the given execution times and the theoretical time in each case.

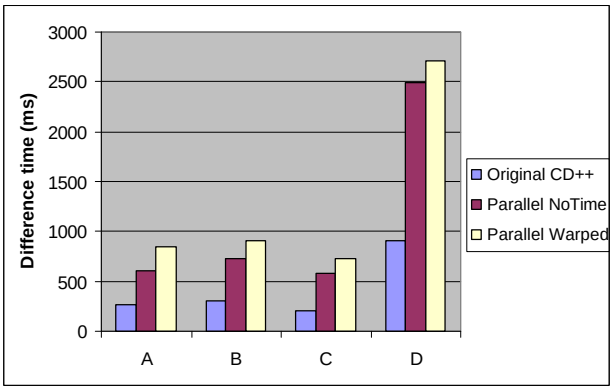

Figure 6: Difference between experiments and theoretical execution times

Finally, Figure 7 presents the overhead incurred by each abstract simulator. The amount of overhead is obtained by subtracting the theoretical time from the execution time and dividing that by the execution time itself, that is:

$$\text{Overhead (\%)} = \frac{(executionTime - theoreticalTime)}{executionTime}$$

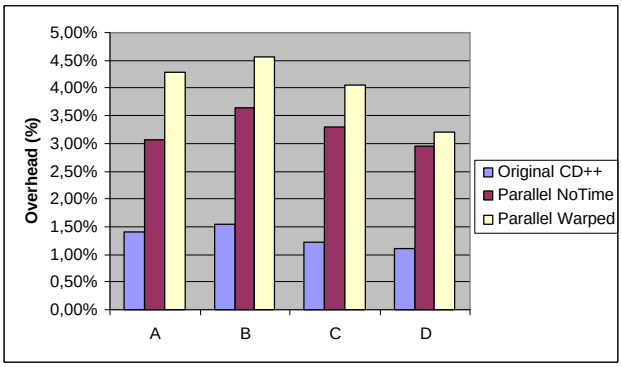

Figure 7: Overhead incurred by the abstract simulators

As we can see, each technique induces a different overhead to the simulation. The charts show that the original CD++ technique is the one that executes with minimum overhead. When executing the parallel abstract simulator, the NoTime kernel adds fewer overheads than the TimeWarp kernel.

4 TESTING REAL-TIME PERFORMANCE

As mentioned earlier, we want to characterize the tool performance for real-time execution. We developed a real-time simulator, and we want to be able to guarantee deadlines considering the overhead of the simulator. The benchmarking tool generates a set of external events with fixed frequencies. Each incoming external event has an associated deadline. The real-time simulator tests have been performed using coupled models with different sizes using the tool described in Section 2 within the real-time simulator.

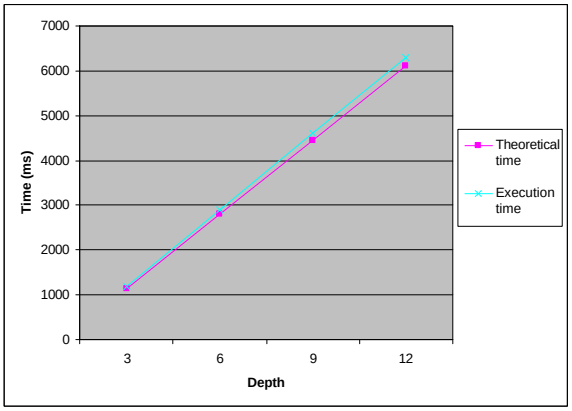

Figure 8: Worst-case execution times in a real-time simulation using different depths

In this case, we store the worst-case response time and the number of missed deadlines for further analysis. For instance, figure 8 shows the theoretical and execution time for a model of type 1, with fixed width of 12 models per level, and 50 milliseconds in both the internal and external transitions. The x-axis represents the different depths used in each simulation.

The following figures show the difference between the real and theoretical execution times, and the associated overheads. This information shows how much time is spent on carrying out the simulation by the abstract real-time simulator and the execution of other tasks involved (such as logging some information on disk, for example).

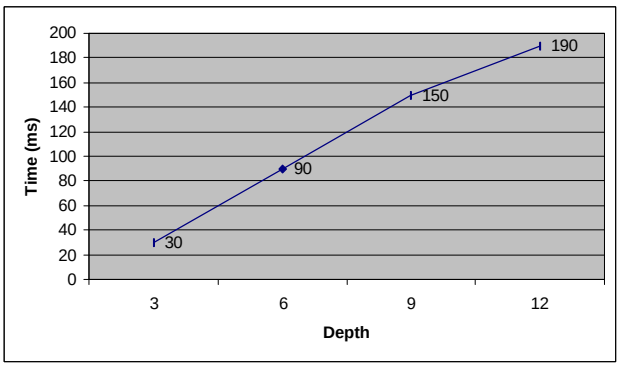

Figure 9: Difference between real & theoretical time using different depths

As long as the model depth increases, the execution time augments linearly. However, the corresponding percentage of overhead remains nearly stable as we can observe in Figure 10.

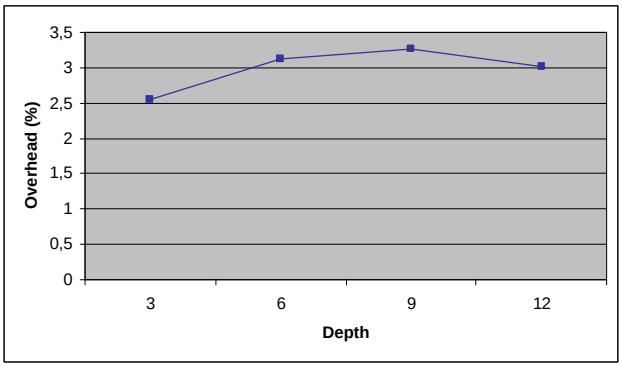

Figure 10: Associated overhead (percentage) using different depths

We will now analyze the results when different widths are used with a fixed depth. The results shown in these graphs are analogous to the ones that were obtained in the previous case. When the size increases due to the increasing width of the models, the difference between the real and theoretical execution time becomes bigger, nevertheless the percentage of overhead remains nearly stable.

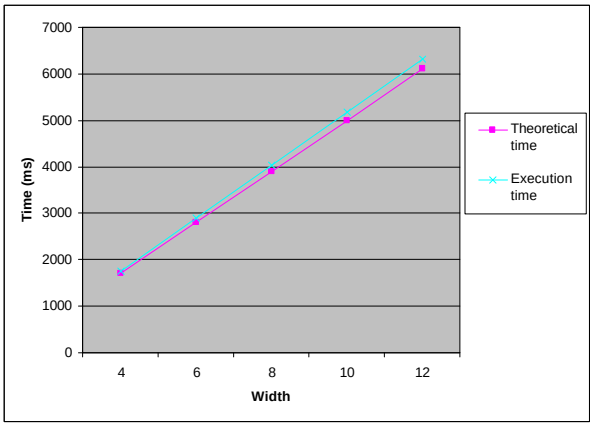

Figure 11: Worst-case execution times in a real-time simulation using different depths

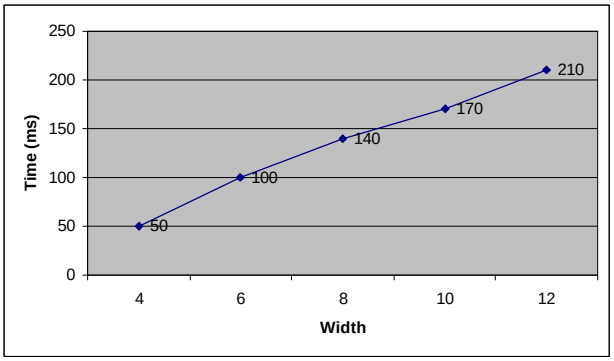

Figure 12: Difference between real & theoretical time using different widths

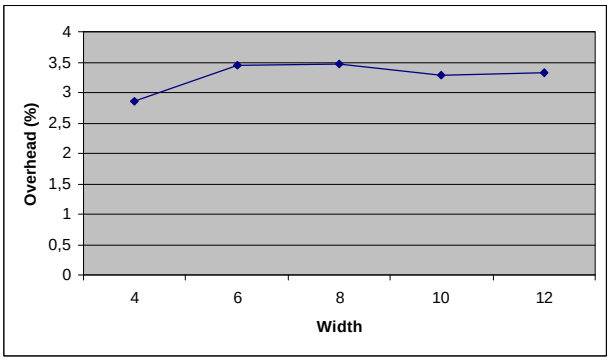

Figure 13: Associated overhead (percentage) using different widths

5 CONCLUSION

The CD++ toolkit enables us to define DEVS models. The tool now supports parallel and real-time simulation. We used a synthetic benchmarking tool that can be applied to DEVS environments. Different types of models were tested automatically, showing that we can execute models paying a small cost in terms of processing overhead.

As we could see, each simulation technique has an associated overhead that depends on the size, shape and behavior of the simulated model. We found that even with medium and large-scale models, the simulation can be carried out properly and the obtained overhead is of a manageable size and stable. The original CD++ tool executes with minimum overhead and therefore it is the appropriate tool when stand-alone execution is adequate. The NoTime kernel outperforms the TimeWarp kernel when using the parallel simulator.

The real-time performance tests show that deadlines are more likely to be missed when the model grows significantly. Nevertheless, it is important to point out that overhead remains roughly stable and consequently simulations can be carried out satisfactorily.

REFERENCES

[GW02] Glinsky, E. ; Wainer, G. "Definition of RT simulation in the CD++ toolkit". *Internal report, Computer Science Department.* Universidad de Buenos Aires. Submitted for publication. 2002.

[MMRW97] Martin, D.; McBayer, T.; Radhakrishnan, R.; Wilsey, P. "Time Warp Parallel Discrete Event Simulator". *Technical Report*, University of Cincinnati. 1997.

[RW99] Rodriguez, D.; Wainer, G. "New Extensions to the CD++ tool". In *Proceedings of SCS Summer Computer Simulation Conference*, Chicago, IL. 1999.

[TW01] Troccoli, A.; Wainer, G. "CD++, a tool for simulation Parallel DEVS and Parallel Cell DEVS models". In *Proceedings of SCS Summer Computer Simulation Conference*, Orlando, FL. 2001.

[WCD01] Wainer, G.; Christen, G.; Dobniewski, A. "Defining DEVS models with the CD++ toolkit". In *Proceedings of the European Simulation Symposium*, Marseilles, France. 2001.

[WEI84] Weicker, R. "Dhrystone: A synthetic systems programming benchmark". In *Communications of the ACM*, volume 27, pages 1013--1030, 1984.

[ZKP00] Zeigler, B.; Kim, T.; Praehofer, H. "Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems". *Academic Press*. 2000.

EZEQUIEL GLINSKY is a M. Sc. student in the Computer Sciences Department of the Universidad de Buenos Aires, Argentina. He is a Research and Teaching Assistant in the same department, and a member of the PARDEVS lab. He worked in the IT industry in Argentina for the past 7 years.Currently, he is a senior developer at Bumeran International. Part of this work was developed while he was a visiting research scholar at Carleton University.

GABRIEL WAINER received the M.Sc. (1993) and Ph.D. degrees (1998, with highest honors) of the Universidad de Buenos Aires, Argentina, and Université d'Aix-Marseille III, France. He is Assistant Professor at the Systems and Computer Engineering, Carleton University (Ottawa, Canada). He was Assistant Professor at the Computer Sciences Dept. of the Universidad de Buenos Aires, Argentina. He has been the PI of several research projects, and participated in different international research programs. Prof. Wainer is a member of the Board of Directors of The Society for Computer Simulation International (SCS). He is also a Co-associate Director of the Ottawa Center of The McLeod Institute of Simulation Sciences.