Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

INFORME TÉCNICO

A General Application DEVS environment

Amir Barylko, Jorge Beyoglonián, Gabriel Wainer

Report n.: 98-005

Pabellón 1 - Planta Baja - Ciudad Universitaria
(1428) Buenos Aires
Argentina

http://www.dc.uba.ar

Title: **A General Application DEVS environment**

Authors: Amir Barylko, Jorge Beyoglonián, Gabriel Wainer.

E-mail: {abarylko, jbeyoglo, gabrielwl}@dc.uba.ar

Report n.: 98-005

Key-words: Discrete Events simulation; Object-Oriented simulation; Modeling tools.

Abstract: In this work a tool to develop simulators is presented. The tool is a general application framework following the DEVS formalism. The main features of the tool are presented, and its use is shown through an application example. The employment of this formalism allowed to develop safe and cost-effective simulators through using a formal specification paradigm.

A General Application DEVS environment

Amir Barylko Jorge Beyoglonián Gabriel Wainer

{abarylko, jbeyoglo, gabrielw}@dc.uba.ar

Departamento de Computación.
Facultad de Ciencias Exactas y Naturales.
Universidad de Buenos Aires.
(1428) Pabellón I. Ciudad Universitaria.
Buenos Aires. Argentina.

Abstract

In this work a tool to develop simulators is presented. The tool is a general application framework following the DEVS formalism. The main features of the tool are presented, and its use is shown through an application example. The employment of this formalism allowed to develop safe and cost-effective simulators through using a formal specification paradigm.

Keywords: Discrete Events simulation; Object-Oriented simulation; Modeling tools.

INTRODUCTION.

Modeling and simulation of complex dynamic systems has improved because a wide variety of modeling paradigms has been developed to specify them. The use of a formal modeling paradigm can improve the development of the executable models. If the model behavior is formally specified, it can be validated against that of the real system. The executable model responses can be verified against the conceptual specification. In the last 20 years several efforts have been made to define formalisms to specify Discrete Events Dynamic Systems (i.e., production plants, computer networks, VLSI circuits, etc.).

The special characteristics of these real systems make their modeling different of continuous variables' systems, usually described by differential equations (a continuous time formalism). The existing paradigms also differ of those used in discrete variable models. This happens because DEDS trajectories are piecewise constant and event driven [Ho89]. Each segment sequence represents the state value and its length represents the time duration for the state. A DEDS modeling formalism should use discrete variables and continuous time.

The use of a continuous time paradigm allows accurate timing representation. This fact allows to improve the precision of the conceptual models and to reduce the CPU time requirements. Higher timing precision can be obtained without using small discrete time segments (that increases the number of simulation cycles). A formalism should provide decomposition mechanisms, because usually the phenomena to model have a hierarchical nature. A paradigm also should capture the system dynamics to model the dynamic comportment of the system. A modeling paradigm also should supply facilities to translate the formal specifications into executable models.
Zeigler [Zei76] proposed a modeling formalism for DEDS that meets these goals. It is a continuous time formalism known as DEVS (Discrete events systems specification) that allows modular model description that can be integrated using a hierarchical approach. Object Oriented programming concepts have been incorporated to the theory [Zei90, Zei97].

This work is devoted to analyze the characteristics of a general application tool used to build and simulate DEVS models. The paper is organized as follows. Section 0 is devoted to recall the main features of the DEVS formalism. Section 0 presents the main characteristics of the tool. Finally, the results of an application example are shown.

DEVS FORMALISM.

A real system modeled using the DEVS paradigm can be described as composed of several submodels. Each model can be behavioral (atomic) or structural (coupled).

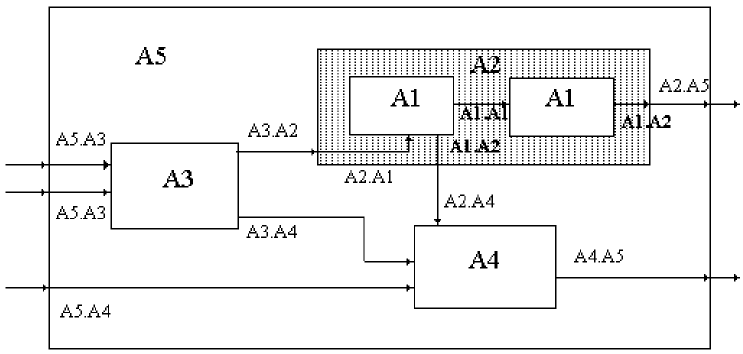

Figure 1. Coupling of DEVS models (A1, A3, A4: atomic models) [WG97].

Each basic model consists of a time base, inputs, states, outputs and functions to compute the next states and outputs. As the formalism is closed under closure, coupled models can be integrated into a model hierarchy. The use of this hierarchical modeling strategy allows the reuse of created and tested models, enhancing the security of the simulations, reducing the testing time and improving productivity.

Atomic Models.

A DEVS atomic model can be formally described as:

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

where

X: input events set;

S: state set;

Y: output events set;

δ_{int}: S → S, internal transition function;

δ_{ext}: Q x X → S, external transition function; where

$$Q = \{ (s, e) / s \in S, \text{ and } e \in [0, D(s)] \};$$

λ: S→Y, output function; and

D: S → $\boldsymbol{R}_0^+ \cup \infty$, ellapsed time function.

Each model is seen as having input and output ports to communicate with other models. The input and output events will determine the values to appear in those ports. The input external events are received in an input port, and the model specification should define the behavior under such inputs. The internal events produce state changes, whose results are spread through the output ports. The ports influences will determine if these values should be sent to other models.

Coupled Models.

A basic model can be integrated with other DEVS basic models to build a structural model. These models are called coupled, and are formally defined as:

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

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

$\forall i \in D$, $\mathbf{M_i}$ 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, and $\forall$ j $\in$ I_i, and

$\mathbf{Z_{ij}}$: $Y_i \rightarrow X_j$ is the i to j translation function.

Finally, **select** is the tie-break selector.

Simulation Mechanism.

A main advantage of the DEVS paradigm is that the models can be specified independently of the simulation mechanism. Zeigler also suggested an abstract simulation mechanism that will be briefly introduced in this section, as the tool here presented uses it.

The simulation process begins by initializing all the component models. The state of each basic model is defined and the next internal transition for each of them is computed. The abstract simulator analyzes the external events and the scheduled internal transitions, and chooses the first model to be activated (called the **imminent** model). In the simulated time t, each component M_i has a state s_i, and have ellapsed time e_i. The next event in the system will be the one with lower scheduled time. If there is more than one component with that time, the select function will be used to select the imminent.

Once chosen, the imminent model is activated. If a basic model receives an external event e $\in$ X, the model executes the external transition function δ_{ext}. As a result, the next internal event can be re-scheduled.

When the time for an internal even arrives, the imminent model must execute its internal transition function. The first step is to execute the output function λ and to generate an output event y $\in$ Y. Each output is sent to the influencees as a translated input, using the Z_{ij} translation function. After, the internal transition function δ_{int} will execute, resulting in a state change and the scheduling of a new internal transition function. The behavior for each of the internal and external transition functions is dependent of the basic model behavior.

GAD.

GAD is a tool built to implement the theoretical concepts specified in the previous section [BBW98]. It was built using a C++, and a modeling class hierarchy was defined. Atomic models should be programmed and incorported to the class hierarchy. A specification laguage allows to define model coupling, initial values and external events. In this section, the main features of the tool will be described briefly.

Simulation class hierarchy.

As stated, the tool is based on the DEVS formalism, and provides an environment to build discrete events models. The system architecture was built using the abstract simulator concepts described in [Zei90], as it can be seen in Figure 2.

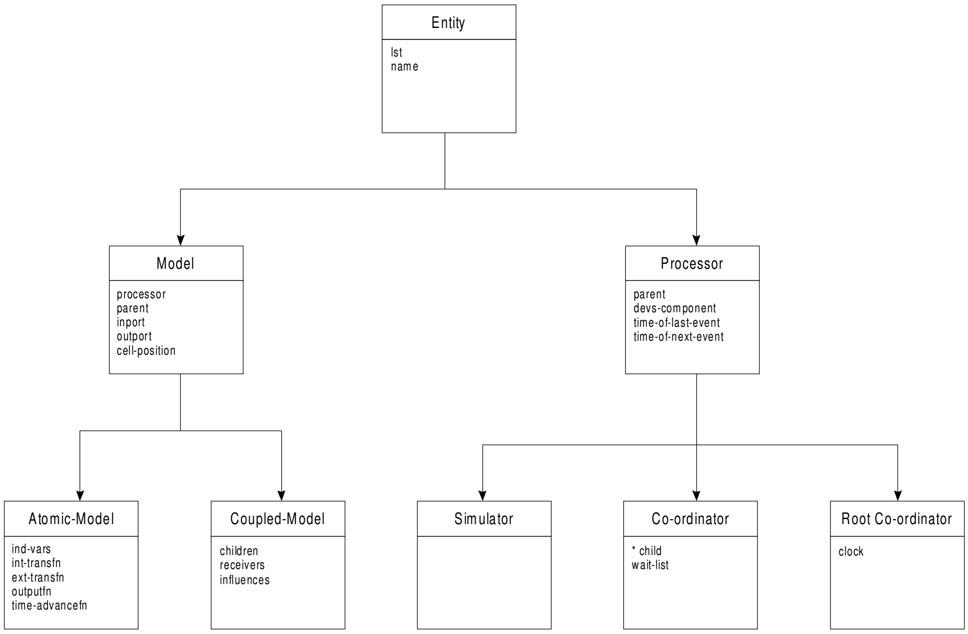

Figure 2. Basic class hierarchy.

There are two basic classes: Models and Processors. The Models classes are devoted to define the conceptual models, and the Processors implement the simulation mechanism. There are different kind of simulation Processors: Simulators, Coordinators and a Root-Coordinator. These processors are related with the models in pairs: a Simulator is associated with an Atomic model, and a Coordinator with a Coupled model. Each existing model will have a unique associated processor.

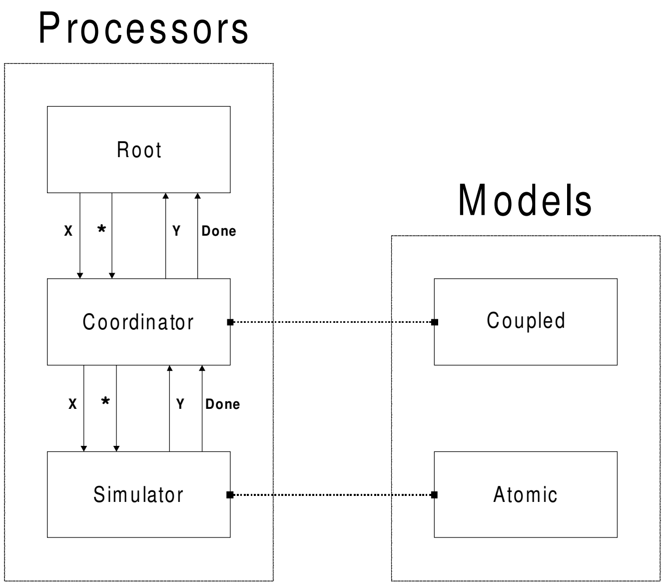

Figure 3. Relationship between Models and Processors.

Model has instance variables *processor* (to identify its associated processor), *parent* (link to the coupled model containing this model), *inport* and *outport* (to specify intermodel interaction).

The *Atomic* class is used to represent the atomic basic models. The methods *int-transfn*, *ext-transfn*, *outputfn* and *time-advancefn* represent the internal transition, external transition, output and time advancement functions respectively. The unctions will be overloaded by the modeler to define the desired behavior depending on the system to be modeled.

Coupled-Model implement the hierarchical constructions defined by the modelling formalism. A coupled model is defined by specifying its components (*children*), and the coupling relationships. The coupling is specified by the *receivers* and *influences* instance variables, that allow the definition of the Z_{ij} function.

The *Processors* are built to execute the abstract simulation procedures by implementing the DEVS theorethical concepts. Following these concepts, *Simulators* and *Coordinators* are built to manage atomic and coupled models, as stated earlier. The *Root-Coordinator* drives the simulation in its global aspects. It maintains the global time, and it is in charge of the start and finish of the simulation. It also collects the output results. It is related with the highest level coupled model and its corresponding coordinator.

The coupling relationship is recorded in the instance variables *devs-component* and *processor* of the Processor and Model respectively. The *parent* variable is used to indicate the parent processor in the simulators hierarchy. The times of the last event and the next event are recorded to identify the imminent children, as to verify correctness in the message simulated times.

The simulation process is carried out by data transfer carried out through message passing. The messages include information related with the message's origin, the time of the related event, and a content consiting of a port and a value.

There are four different messages:
*: used to signal a state change due to an internal event;
X: used when an external event arrives (including its port and value);
Y: model's output; and
done: indicating that a model has finished with its task.

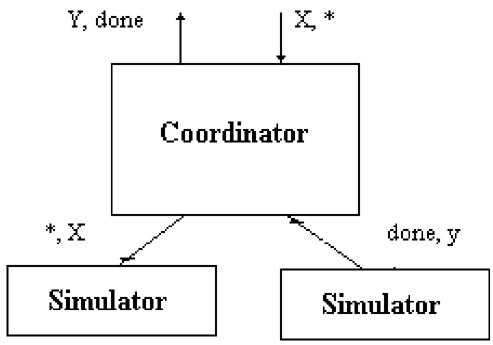

Figure 4. Processor's reaction to different messages.

Simulation advances through message passing between the Processors. When the imminent model is selected, a *-message is sent to its simulator, passing through the middle level coordinators.

When an external message arrives, a X-message is consumed and the external transition function executed. The simulators return done-messages and Y-messages that are converted to new *-messages and X-messages, respectively. The messages are translated using the cell coupling previously defined.

Message management class hierarchy.

The *MessageAdm* class is devoted to receive the message invocation between modules, and to manage their communication. It provides a single method used to send and receive messages. This centralized approach allows that the message passing policy gets encapsulated into the message manager. At present, the chosen policy is FIFO, but the manager can be modified to implement other message distribution policy (for instance, a parallel optimistic approach [Fuj90]).

Message is a base class to define the messages interchanged by the models. Each message carries data about the model that generated the value and its event time.

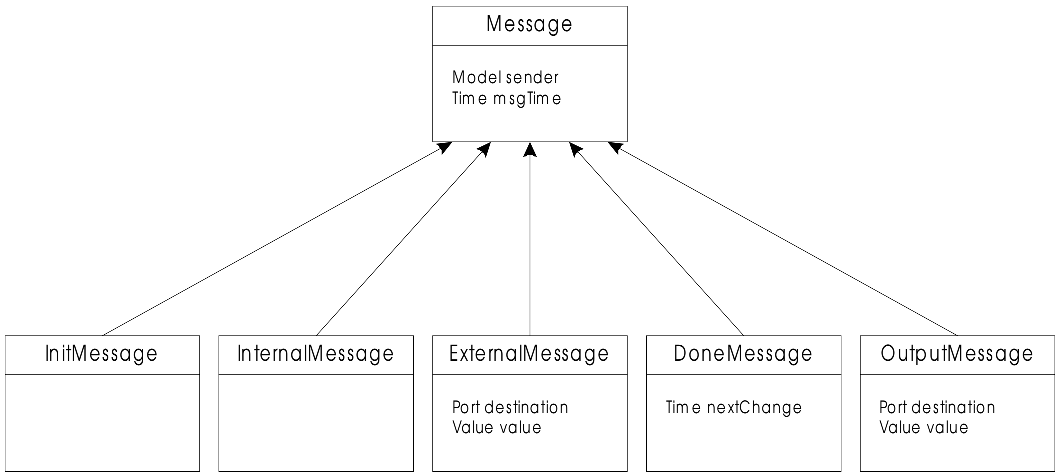

Figure 5. Messages' class hierarchy.

The *ModelAdm* class must manage the created models. Its main functions are:

- Creation of new models: it creates a model’s instance and assigns it a unique identifier. This is the only class that can create new models;
- Association of model’s identifiers with existing models: all the existing models are contained into a list maintained by the model’s manager. The manager is in charge of referencing a model by using a unique key.

To start the simulation, the model’s specification (and its coupling), the external input events, the time for the end of simulation, and input/output files must be specified. The model specification is defined by using a language developed with that purpose. The description for each model includes the input/output ports and the coupling with other models.

These services are carried out by the *SimLoader* class, that provides an interface to load the simulator configuration. There are two possible procedures to start the simulation. The first one is by using the class *StandAloneLoader*, that is responsible to load the parameters by using the shell's commnad line. The *NetworkLoader* class is responsible to get the same parameters by using TCP/IP services. In this way, the simulator can be executed as a simulation server, and the parameters can be loaded remotely, getting the results in a remote fashion. The simulator will wait in a port the simulation specification, will execute it, and will return the results using the same way.

Finally, the *Simulator* class is responsible of the creation of the model tree and to establish the links between ports using the specification. To do so, the *IniFile* class is used to parse the model's specification. The root coordinator is in charge of the model's loading. Once the model's hierarchy is built, the simulation can begin . To do so, the external events must be added, an event list created and the stop time initialized.

AN APPLICATION EXAMPLE.

To test the tool, several models were built. In this section on of them is presented. The goal of the experience is to analyze the performance of a processor by using the GAD environment.

Let us consider the modelling and simulation of a computer processor. The environment to be modeled will include a user providing tasks to be executed, a task scheduler with a certain scheduling policy, and a processor [Zei90]. When a new task arrives, the task scheduler will decide how long will it wait to begin its processing. When the task starts, it will execute during a fixed amount of time. The scheduler will be considered as non premptive (the task executes up to its end without being interrupted) with a FIFO scheduling policy.

Model definition.

In this case, the simulation will execute a model composed of four atomic models, each representing a different task of the processing environment. The first one (called Generator) is devoted to provide an experimental framework to generate new tasks. The second model (Queue) is devoted to simulate the task scheduler. The third one (Processor) models the processor executing the system tasks. Finally, the Transducer model is devoted to record the measures generated by the simulation.

The behavior of each of the atomic models is the following:

- **Generator**: this model is devoted to generate new tasks in the simulation. It is used as part of the experimental framework. It has an output port (**out**) used to output the tasks to be processed. Each generated value represents the task identifier (unique during the simulation process). The creation process interval was generated using random numbers with Poisson and χ^2 probability distributions (chosen during the configuration process).

- **Processor**: this model is responsible to simulate the tasks' execution. An arriving task is received through the **in** input port. When a task has finished, it must output the process identifier through the **out** output port. When a task is executing, the processor remains busy until the processing is finished. The processing time was generated using random numbers with an exponential distribution.

- **Queue:** as the processor executes one task at a time, a queue must be used to receive the new tasks and to store them while the processor is busy. The queue was implemented using a non-preemptive FIFO policy.

- **Transducer**: this model is in charge to take the measures of the simulation and to compute statistics. Two variables have been taken into account: throughput (measured as tasks executed in each time unit) and CPU usage (computed as an average of the tasks waiting in the ready queue).

The models are coupled to form a multicomponent model, whose interconnection can be seen in Figure *6*.

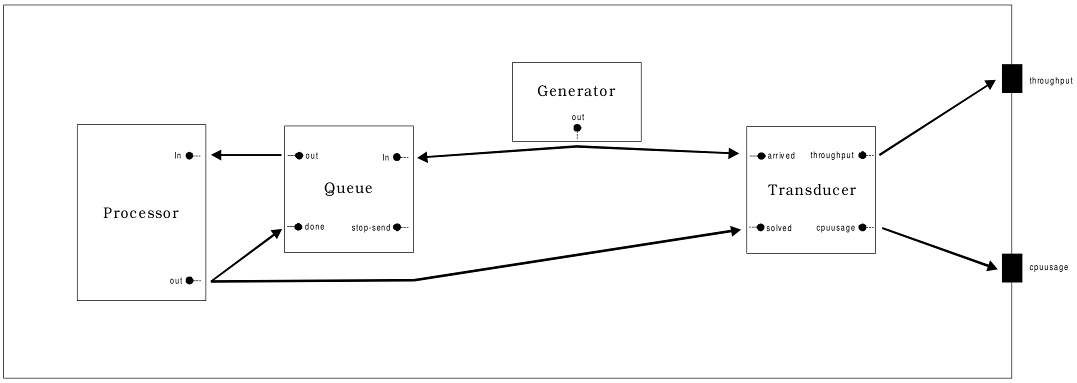

Figure 6. Model interconnection.

The Generator output is connected with the ready queue (to record the new task) and to the Transducer (that records the time of the new process to compute statistics). Each time a task finishes, it is output using the **out** port, and input into the Processor model to be executed. Once a task has finished, the Processor outputs its number through the out port, that will be sent to the Queue and the Transducer. The Transducer records information about the processes, and sends the results using the output ports **Throughput** and **Cpuusage**.

Simulation results.

Several tests were made by combining different probability distributions with diferent parameters. The results obtained allowed to see that processor performance (measured in response times and throughput) improved when the average execution time was lower than tasks' generation time.

Figure 7 shows the results obtained using a Generator with Poisson probability distribution (10 time units average), and χ^2 distribution for the processing time (30 time units average). As a result the tasks get queued waiting for the processor. The throughput value to 2 tasks per minute (the average processing time is 30 seconds and the period considered is one minute). Analogous tests were carried out changing the distribution probability and parameters, obtaining similar results.

Figure 8 results were obtained using a Generator with Poisson probability distribution (30 seconds average) and a processor time with exponential distribution (10 time units average). In this case, the throughput also tends to 2 tasks/minute, but the processor is free most of the time. This happens because the works are consumed much faster than the generation of new tasks.

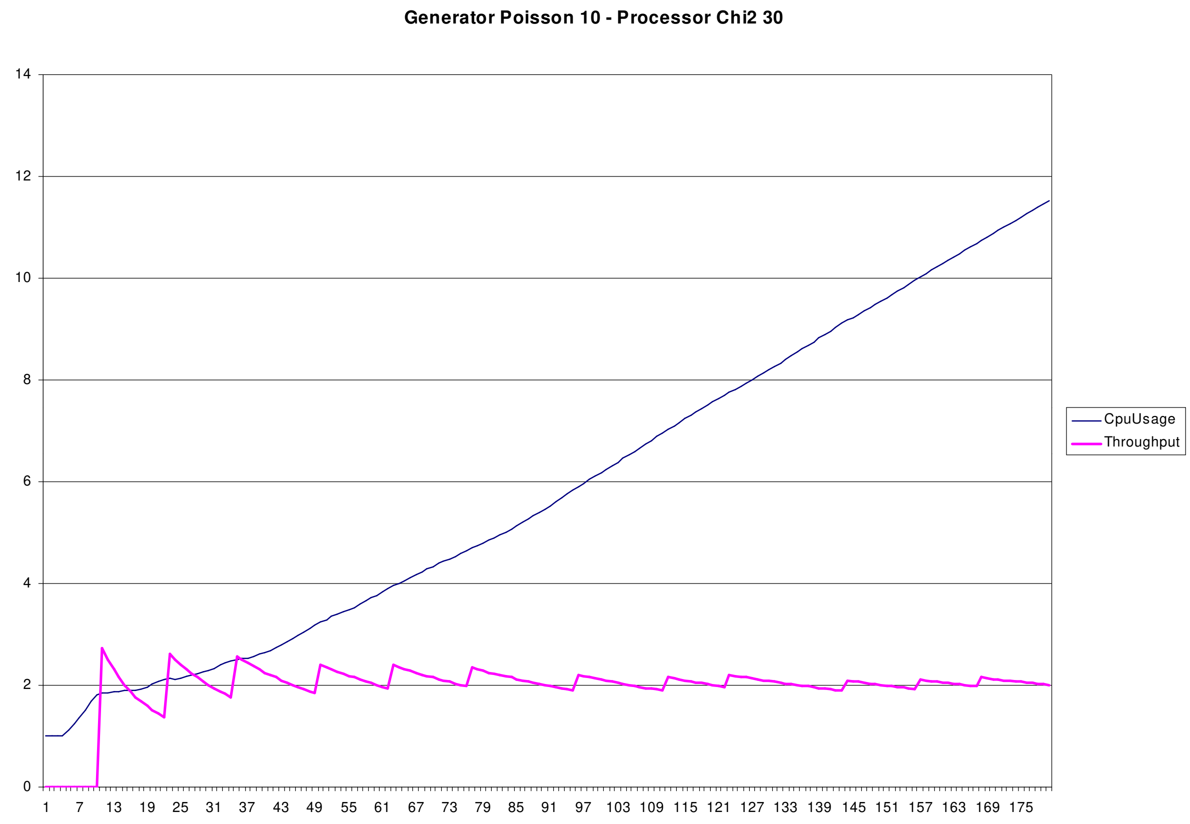

Figure 7. Simulation Results.

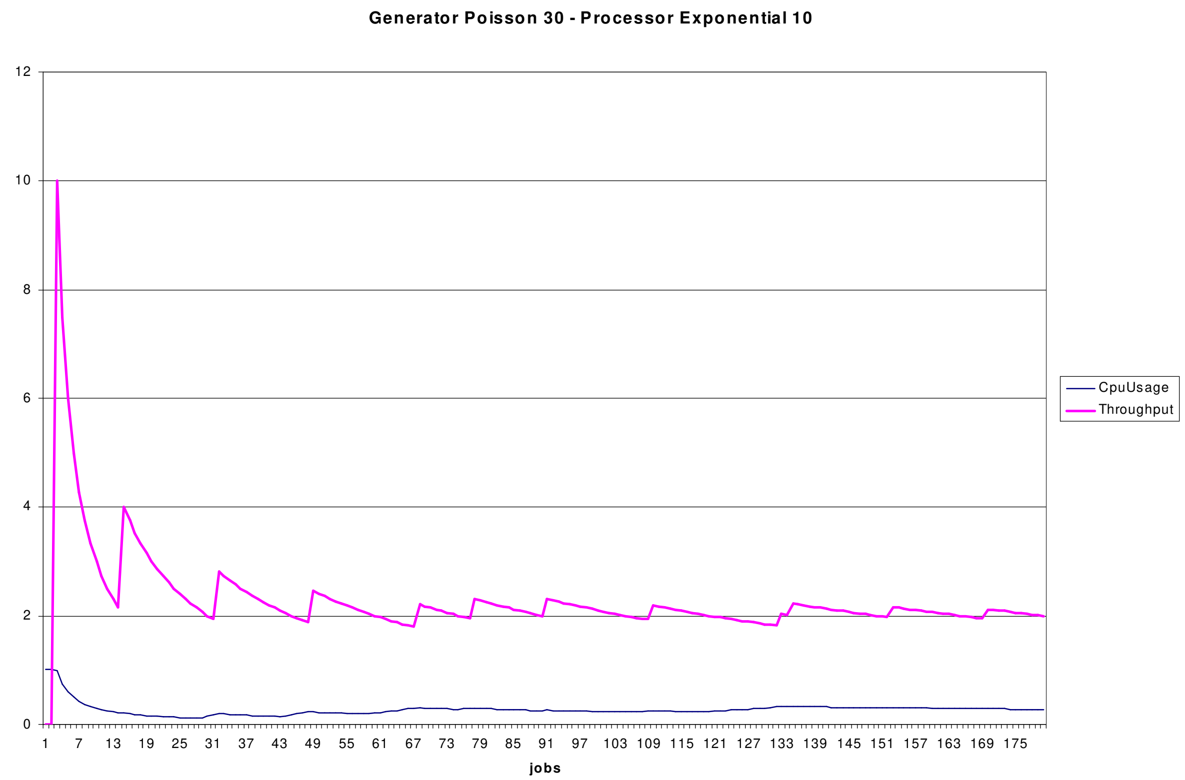

Figure 8. Simulation results.

CONCLUSION.

In this work, the main features of GAD, a tool for General Application DEVS modeling and simulation was presented. The tool was built using a formal modeling paradigm, improving the safety and development times of the simulations. The tool executes in a stand-alone mode, or as a simulation server that can be executed remotely.

Several tests were executed, proving the usefulness of the tool. A data base of models can be created, enhancing the development process. The tool is being used with educational purposes, and the models presented are being used to test multiprocessor configurations.

The tool is public domain and can be obtained at "http://www.dc.uba.ar/people/materias/sim". At present, a new modeling paradigm, called Time Cell-DEVS ([WFG97], [WG98]) is being implemented as an extensiion of the tool. The formalism is based on the DEVS and Asynchronous Cellular Automata paradigms. The concepts of transport or inertial delays used in the circuit modeling domain have been combined, allowing simple specification of accurate timed models. The specifications have been defined for binary or three-states systems. The formalisms allow automatic definition of the spaces, and eases verification of the models, allowing efficient and cost-effective development of simulators.

REFERENCES.

[BBW97] BARYLKO, A.; BEYOGLONIAN, J.; WAINER, G. "GAD: a General Application DEVS environment". Technical Report No. 98-001. Departamento de Computación. Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires. 1997.

[Fuj90] FUJIMOTO, R. "Parallel Simulation of Discrete Events". Communications of the ACM. Vol. 33. No 10. 30-53. 1990.

[Ho89] Ho, Y. "Special issue on discrete event dynamic systems", Proceedings of the IEEE, 77 (1), 1989.

[WFG97] G. A. Wainer, N. Giambiasi, "CELL-DEVS models with transport and inertial delays". Proceedings of the SCS European Simulation Simposium on Industrial Simulation, Passau, Germany. 1997.

[WG98] G. A. Wainer, N. Giambiasi, "Specification, modelling and simulation of Timed Cell-DEVS spaces". Technical Report No. 97-006, Universidad de Buenos Aires. Departamento de Computación. Submitted to the Journal of the Society of Computer Simulation. 1998.

[Zei76] Zeigler, B. “Theory of Modelling and Simulation”. Wiley, N.Y.

[Zei90] Zeigler, B. “Object Oriented Simulation with Hierarchical Modular Models”. Academic Press. 1990.

[Zei97] ZEIGLER, B. "Objects and systems: principled design with Java/C++ Implementation". Springer-Verlag. New-York, 1997.