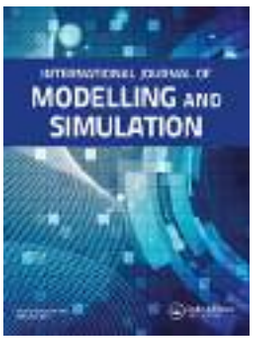

International Journal of Modelling and Simulation

ISSN: 0228-6203 (Print) 1925-7082 (Online) Journal homepage: https://www.tandfonline.com/loi/tjms20

Experiences with Devs Modelling and Simulation

G. Wainer, A. Barylko & J. Beyoglonián

To cite this article: G. Wainer, A. Barylko & J. Beyoglonián (2001) Experiences with Devs Modelling and Simulation, International Journal of Modelling and Simulation, 21:2, 138-147, DOI: 10.1080/02286203.2001.11442196

To link to this article: https://doi.org/10.1080/02286203.2001.11442196

Published online: 15 Jul 2015.

Submit your article to this journal

Article views: 7

View related articles

Full Terms & Conditions of access and use can be found at
https://www.tandfonline.com/action/journalInformation?journalCode=tjms20

International Journal of Modelling and Simulation, Vol. 21, No. 2, 2001

EXPERIENCES WITH DEVS MODELLING AND SIMULATION

G. Wainer,* A. Barylko,** and J. Beyoglonián**

Abstract

This paper presents the results obtained with a tool used to model and simulate discrete event systems, based on Discrete Event systems Specification (DEVS) formalism. Its main features are presented and its use shown through application examples. The use of this formal approach allowed development of safe and reusable simulations. A simulated processor was built to study the different levels of a computer system. The goal was to help the full comprehension of the computer behaviour used in computer organization courses. The environment helped the students understand these complex systems and also allowed them to make empirical comparisons and performance studies for educational purposes.

Key Words

Discrete event simulation, modelling methodologies, simulation tools, computer organization, computer system levels.

1. Introduction

In recent years, new modelling paradigms allowed the simulation of complex dynamic systems to improve. The use of a formal modelling paradigm allows improvement in the development of executable models by validating their behaviour against that of the real system.

Several efforts have focused on the specifications of Discrete Event Dynamic Systems (DEDS) (e.g., production plants, computer networks, Very Large Scale Integration (VLSI) circuits, etc.). These real systems have special features that make their modelling different from those with continuous variables. DEDS trajectories are piecewise constant and event driven, hence the modelling formalisms should use continuous time and discrete variables. Continuous time allows accurate timing representation, improving the precision of conceptual models, and reducing the processing requirements. Higher timing precision can be obtained without using small discrete time segments (which increase the number of simulation cycles).

Decomposition mechanisms should be provided to reflect the characteristics of the phenomena to be modelled (usually of a hierarchical nature). System dynamics should be captured, supplying facilities to translate the formal specifications into executable models. In [1], a modelling formalism for DEDS with these goals was proposed. It is a continuous time formalism known as DEVS that allows modular descriptions of models that can be integrated using a hierarchical approach.

This work analyzes the characteristics of a general application tool used to build and simulate DEVS models. The main goal is to show the application of the formal approach. The article is organized as follows. Section 2 recalls the main features of the DEVS formalism. Section 3 presents the main characteristics of the tool. Use of the tool is then presented using several examples. Finally, Section 6 presents the design of a simulated computer for educational purposes.

2. DEVS Formalism

A real system modelled, using the DEVS paradigm, can be described as being composed of several submodels. Each model can either be behavioural (atomic) or structural (coupled). Each basic model consists of a time base, inputs, states, outputs, and functions used to compute the next states and outputs. As the formalism is closed under closure, coupled models can be integrated into a model by coupling. The use of this hierarchical modelling strategy allows reuse of created and tested models, enhancing security of the simulations, reducing testing time, and improving productivity.

2.1 Atomic Models

A DEVS atomic model can be formally described as:

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

where:

I is the model's interface
X is the input events set
S is the state set

* Systems and Computer Engineering Dept., Carleton University, 3105 Mackenzie Bldg., 1125 Colonel By Drive, Ottawa, ON K1S 5B6 Canada; e-mail: gwainer@sce.carleton.ca
** Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires (1428) Pabellón I Ciudad Universitaria, Buenos Aires, Argentina; e-mail: {abarylko,jbeyo}@dc.uba.ar

Recommended by Dr. Peter Nejjema
(paper no. 1998-219)

Y is the output events set
δ_{int} is the internal transition function
δ_{ext} is the external transition function
λ is the output function
D is the elapsed time function

Each model is seen as having an interface consisting of input and output ports used to communicate with other models. The input external events (events received from other models) are received in input ports and the model specification should define the behaviour of the external transition function under such inputs. The internal transition function is activated after consumption of the elapsed time, with the goal of producing internal state changes. The desired results are spread through the output ports and sent by the output function, which executes before the internal transition.

2.2 Coupled Models

A basic model can be integrated with other DEVS basic models to build a structural model (see Fig. 1). These models are called coupled, and formally defined as:

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

where:

- I is the model's interface
- 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
- $\forall i \in D$, M_i is a basic DEVS model (that is, an atomic or coupled model), defined by:

$$M_i = < I_i, X_i, S_i, Y_i, \delta_{int_i}, \delta_{ext_i}, \lambda_i, ta_i >$$

- I_i is the set of influencees of model i (that is, the 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.

Finally, select is the tie-breaking selector.

The basic idea is that, each coupled model consists of a set of basic models (atomic or coupled), connected through the input/output ports. The influencees of the model will determine which output values should be sent

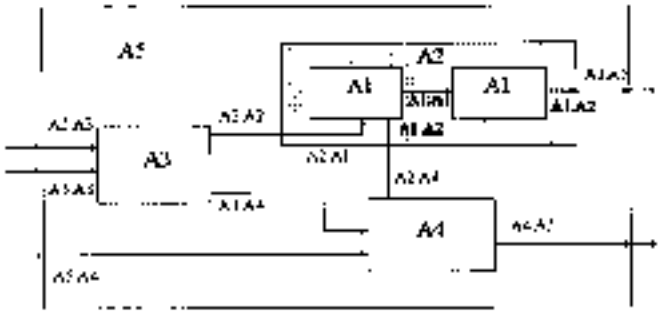

Figure 1. Coupling of DEVS models (A1, A3, A4: atomic models).

models. The translation function is in charge of translating outputs of a model into inputs for the other models. To do this, an index of influencees is created for each model (I_i). This index defines that the outputs of model M_i are connected to inputs in model M_j, where j is an element of I_i. Finally, if several models are activated simultaneously, the select function defines which models must be executed first.

2.3 Simulation Mechanism

One main advantage of the DEVS paradigm is that the models can be specified independently of the simulation mechanism. '1] also suggested an abstract simulation mechanism, that will be briefly introduced in this section, as the tool presented here is based on it.

The simulation process begins by initializing all of the component models. The state of each basic model is defined and the next internal transition for each is then computed. The abstract simulator analyzes the external events and 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 elapsed time e_i. The next event in the system will be the lowest-scheduled time one. If there is more than one component with that time, the *select* function will be used to choose the imminent model.

Once chosen, the imminent model is then activated. If a basic model receives an external event $x \in X$, the model executes the external transition function δ_{ext}. Consequently, the next internal event (that is, those produced by the consumption of time in the model) is re-scheduled. When the time for an internal event arrives, the imminent model executes its internal transition function. The first step is to execute the output function λ and generate an output event $y \in Y$. Each output is sent to the influencees as a translated input, using the Z_{ij} translation function. The internal transition function δ_{int} then executes, resulting in a state change and scheduling of a new internal transition. The behaviour of internal and external transition functions depends on the model's behaviour.

3. GAD

GAD is a tool for General Application DEVS modelling and simulation. It was built to implement the theoretical concepts specified in the previous section 2. Atomic models can be programmed and incorporated into a basic class hierarchy, programmed in C++. A specification language allows definition of the model's coupling, including initial values and external events. In this section, the main features of the tool will briefly be described.

As stated, GAD is based on the DEVS formalism and provides an environment for building discrete event models. The system architecture was built using the abstract simulator concepts described in 3, as seen in Fig. 2.

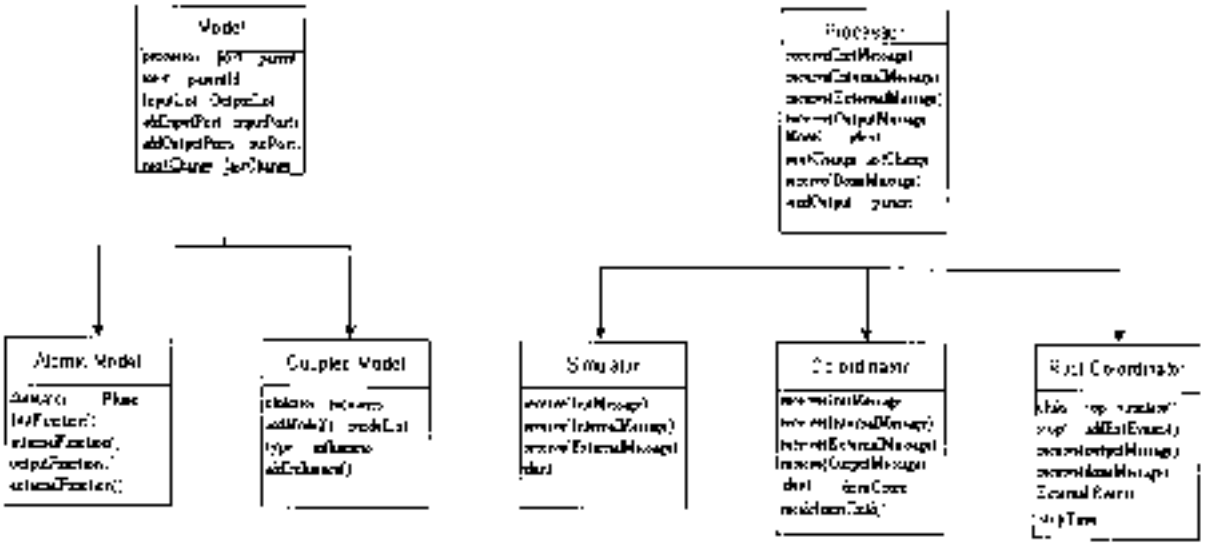

Figure 2 Basic class hierarchy.

There are two basic classes: *Models* and *Processors*. The *Models* class is devoted to defining conceptual models and the *Processors* class to implementing the simulation mechanism. Different simulation processors are used: *Simulators*, *Coordinators*, and *Root-Coordinators* related with different models: *Simulators* are associated with *Atomic* models and *Coordinators* with *Coupled* models.

Model has instance variables *processor* (to identify its associated processor), *parent* (linked to the coupled model containing this model), and *inport* and *outport* (to specify model 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 functions must be overloaded by the programmer in order to define the desired behaviour, depending on the system to be modelled. *Coupled-Model* implements 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 *influencers* instance variables, which allows definition of the Z_i function.

The *Processors* are built to execute the abstract simulation procedures explained earlier. *Simulators* and *Coordinators* are built to manage atomic and coupled models. The *Root-Coordinator* drives the simulation in its global aspects. It keeps the global time and it is in charge of the simulation's start and finish. 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 *data-component* and *processor* of the *Processor* and *Model*, respectively. The *parent* variable indicates the parent processor in the simulators' hierarchy. The times of the last event and the event are recorded in order to identify the imminent children and verify correctness in the message's simulated times.

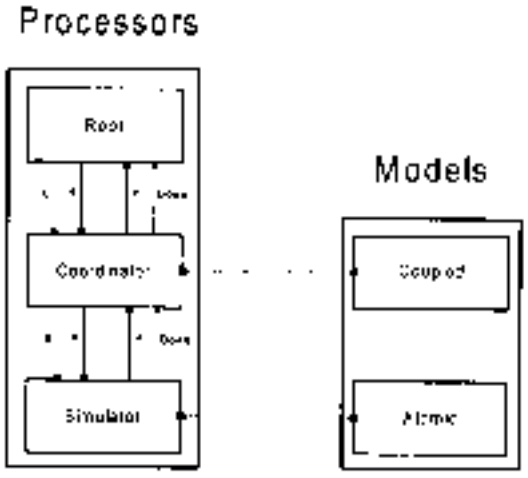

Figure 3 Models/ Processors relationship.

The simulation process is carried out by data transfers through message passing. The messages include information related to the message's origin, time of the related event, and a content, consisting of a port and a value. There are four messages: * (used to signal a state change, due to an internal event), *X* (used when an external event arrives), *Y* (the model's output), and done (indicating a model has finished with its task). The 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, an 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 *MessageAdm* class in Fig. 4 is devoted to receiving the message invocations between modules and manage their communication. *Message* is the base class used to define the message's interchange. Each message carries data of the model generating the value and its event time.

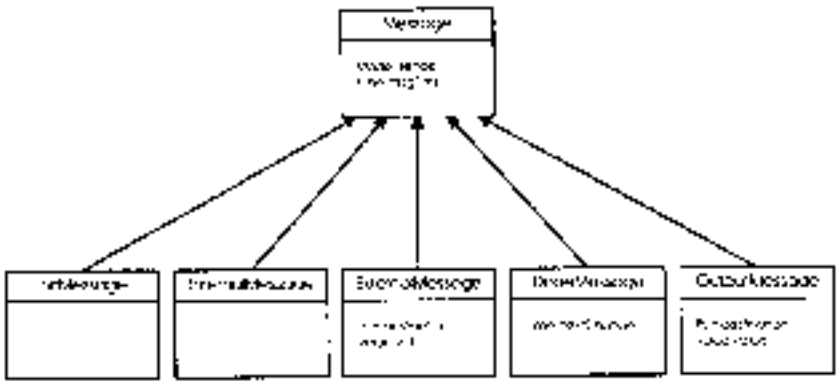

Figure 4. Messages' class hierarchy.

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

- Creation of new models: creates an instance of a model and assigns it a unique identifier. This is the only class that can create new models and
- association of identifiers with models: all existing models are included in a list, which is kept by the model manager.

The modeller must define the model's specification (and coupling), external input events, and finish time of the simulation. The model is specified using a language developed for that purpose. The *SimLoader* class is in charge of these functions, providing an interface to load the simulator configuration. There are two possible procedures used to start the simulation. The first one uses the *StandAloneLoader* class, responsible for loading the parameters by using the shell's command line. The *NetworkLoader* class is responsible for getting the same parameters, using TCP/IP services. In this way, the simulator can be executed as a simulation server and the parameters loaded remotely, getting the results in a remote fashion.

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

4. Model Definition Using GAD

As stated in the previous section, the atomic models and their coupling must be specified. The coupled models are defined using a specification language, which is developed for that purpose. The description for each model includes the input/output ports and the coupling with other models. Instead, atomic models must be incorporated in the class hierarchy as subclasses of the *Atomic Model* class. The following sections will explain how to incorporate the atomic and coupled models to be simulated.

4.1 Atomic Models

A new atomic model is generated by designing a new class, derived from the *Atomic* class. First, the model must be registered using the *MainSimulator::registerNewAtomics()* method. Then the following methods should be overloaded:

- *initFunction*. This method is invoked at the beginning of the simulation. It allows definition of initial values and execution of the initial functions for the model. When this method is executed, the value of *sigma* (next scheduled event) is set to infinity and the model phase to *passive*.
- *externalFunction*: This method is invoked when an external event arrives from an input port.
- *internalFunction*: This method is invoked when the value of *sigma* is zero, since an internal event has occurred.
- *outputFunction*: This method executes before the internal function, allowing outputs for the model to be provided.

These methods have been built by following the formal specifications of DEVS models, defined in Section 2.1. In addition, several primitives have been defined, allowing interaction with the abstract simulator:

- *holdIn(state, time)*: It is used to define that model as remaining in *state* during *time*. When this time is consumed (*sigma* = 0), the model executes an internal transition. This function is used to implement the **D** (lifetime) function of the DEVS formal specification.
- *passivate()*: The model enters in passive mode and will be reactivated by an external event.
- *sendOutput(time, port, value)*: It sends an output message through the given port.
- *state()*: It returns the present model phase.
- *getParameter(modelName, parameterName)*: It allows access to the model state variables.

4.2 Coupled Models

Coupled models are defined using a specification language, specially defined for this purpose. This specification language also follows the formal definitions for DEVS coupled models. Therefore, each of the components defined in Section 2.2, are included. Each coupled model is composed using a set of definitions. Optionally, configuration values for the atomic models may be included. Each set indicates the name of the model and its attributes. The [top] model defines the coupled model at the top level.

Four properties must be configured: components (using the clause "*components*"), output ports (clause "*out*"), input ports (clause "*in*") and links between models (clause "*link*"). The syntax is the following:

- *Components*: It describes the models composing the coupled model. The syntax is *model_name@class_name*. The name of the model is needed because we can use more than one instance of the same model. The class's name can reference either atomic or coupled models. The last ones should be defined in the same configuration file as a new group. The order used when the models are set defines the priority for the select function (that is, the execution order under simultaneous events).
- *Out*: It defines the names of output ports.
- *In*: It defines the names of input ports.
- *Link*: It describes the internal and external coupling scheme. The syntax is *source_port@model destination_port@model*. The name of the model is optional because if not indicated, the coupled model being defined will be used.

5. Experimental Framework for Single Processor Execution

The tool was tested by building several models, including examples of computer Local Area Networks (LANs), Personal Communication Systems (PCS), routing in Wide Area Networks (WANs), plane flow in an airport, etc. This section shows implementation of the simplest models. We do not provide an exhaustive analysis of the problem because we intend to show the use of the main features and applications of the tool.

Let us consider the modelling and simulation of a computer processor. The environment to be modelled includes a group of users providing tasks to be executed, a task scheduler with a certain scheduling policy and a processor [3]. When a new task arrives, the task scheduler faces a delay before beginning its processing. When the task starts, it executes during a fixed amount of time. The scheduler is non-preemptive (the tasks execute without being interrupted) with a first-in-first-out (FIFO) scheduling policy.

5.1 Model Definition

In this case, the model is composed of four atomic models, each representing a different function of the processing environment. The first one (called Generator) provides an experimental framework to generate new tasks. The second model (Queue) simulates the FIFO task scheduler. The third one (Processor) models the processor executing the system tasks. Finally, the Transducer model records the metrics generated by the simulation.

The behaviour of each atomic model is the following:

- *Generator*: It generates new tasks, transmitted through an output port. The output value represents a task identifier (unique during the simulation process). The period used to create a new process is generated using random numbers with probability distributions chosen during the configuration process.
- *Processor*: This model simulates the tasks' execution. A new task is received through an input port and the processor remains busy until processing is finished. Then it sends the process identifier through an output port. The processing time is generated using random numbers with exponential distribution.
- *Queue*: This queue receives new tasks and stores them while the processor is busy. The queue was implemented using a non preemptive FIFO policy.
- *Transducer*: This model records metrics and computes statistics of the simulation. Two measures are considered: throughput (tasks executed per time unit) and CPU usage (average of tasks waiting in the ready queue).

The functionality of each of these models is coded in the tool using the definitions provided in the previous section. As stated earlier, these functions follow the formal specification for DEVS. For instance, the Queue model can be formally described as:

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

where

$$X \subset \mathbf{N} \cup \{\text{stop}\} \cup \{\text{done}\}$$

$$S = \{\text{preparationTime}, \text{timeLeft} \in \mathbf{R}^-\} \cup \{\text{elements} \in \{\mathbf{N}\}^*\}$$

$$Y \subset \mathbf{N}$$

These sets and the transition functions are described, as explained in the previous section, in the Fig. 5.

After each model is defined as was outlined in Section 5.1, the models are then coupled to form a multicomponent model. This is shown in Fig. 6.

The Generator output is connected to the ready queue (to record the new task) and the Transducer (to record the length of each process). The task is kept for at least a preparation time. This time is used to represent the overhead of the task scheduler. Next, its identifier is sent through the out port and is received by the Processor model to be executed. Once a task has finished, the Processor outputs its number through the out port, which 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*. For instance, the top model in this hierarchy is formally des-

```
class Queue : public Atomic {
public:
   Queue();
protected:
   Model &initialFunction();
   Model &externalFunction( const ExternalMessage & );
   Model &internalFunction( const InternalMessage & );
   Model &outputFunction( const InternalMessage & );
private
   const Port &in, &stop, &done;
   Port &out;
   Time preparationTime, timeLeft;
   List<Value> elements;
};


Model &Queue::externalFunction( const ExternalMessage &msg )
{
   if ( msg.port() == in )                          // A new value arrived
      elements.push_back( msg.value() );            // Add it to the queue
      if ( elements.size() == 1 )                   // The queue was empty
         state.position( active, preparationTime );// Then, the first job must be prepared
   }
   if ( msg.port() == done )                        // A job has finished
      elements.pop_front();                 // Delete it from the queue
      if ( elements.empty() )                       // Take the next element in the queue
         this->holdIn( active, preparationTime );// This job must be prepared to output

   if ( msg.port() == stop                          // Stop the transmission. Buffer overflow
     || (this->value() == passive && msg.value() == 0) )  // The queue was active
      timeLeft = state.sigma( state.getChange() );  // Record the time left
      this->passivate();                    // Deactivate the queue
   }
   else                                             // Reactivate the queue
      if ( this->value() == passive && msg.value() )
         this->holdIn( active, timeLeft );          // Simulate the time left
   return *this;
}

Model &Queue::outputFunction( const InternalMessage &msg )
{
   this->sendOutput( msg.time(), out, elements.front() );
   return *this;                    // Transmit the value of the first element in the queue
}

Model &Queue::internalFunction( const InternalMessage & )
{
   this->passivate();
   return *this;
}
```

Figure 5. Definition example: Queue model's header.

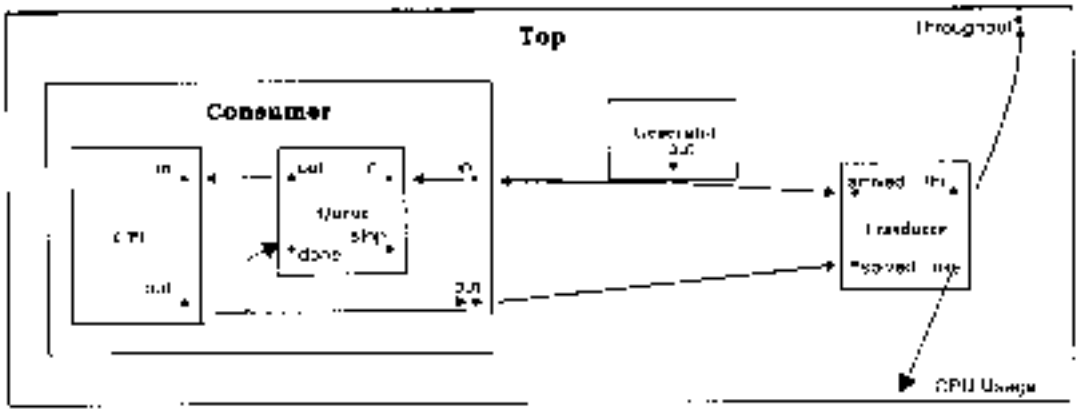

Figure 6. Model interconnection.

cribed by:

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

where

$$\mathbf{X} \quad \{\emptyset\}$$

$$\mathbf{Y} \quad \{\text{Throughput}, \text{CpuUsage} \;/\; \text{Throughput} : \text{CpuUsage} \in \mathbf{R}-\}$$

$$D = \{\text{Transducer}, \text{Generator}, \text{Consumer}\}$$

$$I_{\text{Generator}} = \{\text{Transducer}, \text{Consumer}\}$$

$$I_{\text{Consumer}} = \{\text{Transducer}\}$$

$$I_{\text{Transducer}} = \{\text{Self}\}$$

```
[top]
components = transducer@Transducer
components = generator@Generator
components = Consumer
Out = CPU-Usage
Out = Throughput
Link = out@generator arrive@transducer
Link = out@generator in@Consumer
Link = out@Consumer solved@transducer
Link = use@transducer CPU-Usage
Link = thr@transducer Throughput

[Consumer]
components = Queue@Queue cpu@CPU
in = in
out = out
Link = in in@Queue
Link = out@Queue in@cpu
Link = out@cpu done@Queue
Link = out@cpu out
```

Figure 7. Coupled Model's definition.

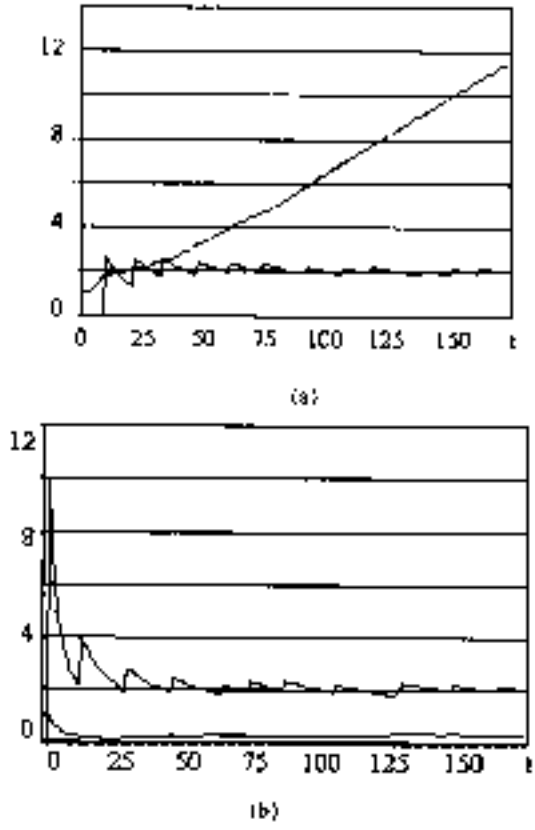

Figure 8. Simulation results

Fig. 7 shows the definition of this formal description using the coupling specification language of the tool.

5.2 Simulation Results

Several tests were made by combining different probability distributions with different parameters. This procedure was done by simply changing a parameter in the coupled model specification. The main goal was to test the validity of the models and correct use of the tool. Fig. 8(a) shows the results obtained, generating jobs every ten time units (average) and processing them in 30 time units (average). Consequently, the tasks are queued waiting for the processor (the growing curve) and the throughput is around two tasks per minute. Fig. 8(b) shows the results obtained using a generator with 30 sec. of activation and a processing time of ten time units. In this case, the throughput also tends to two tasks per minute, but most of the time, the processor is free because the works are consumed much faster than the generation of new tasks.

6. ALPHA-0: A Simulated Computer

The tool has been used to study the multiple levels of the organization of a computer. Theoretical study in this field usually gives students an incomplete and sometime erroneous view of how a computer system works. The lack of practical experience can make that the underlying complexity of the subsystems and their interaction may not be understand completely. The main problems are related to the existence of several abstraction levels (assembly language, instruction set, microprogramming, and digital logic). The introduction of higher levels (programming languages, operating systems) makes the task even more complex.

At present, there are several simulators (for instance, [1,7,]) devoted to analyzing architecture properties but most of them are devoted to the study of architecture performance. They allow for building of the main architecture blocks and defining their interaction, but none are devoted to meeting educational purposes. Moreover, several are commercial applications unavailable for public domain or massive use in computer organization courses. As they are devoted to analyzing architectural properties, several levels needed to study computer organization (for instance, the digital logic level or assembly language level) are not supported. In addition, no changes can be done (for instance, to implement logical gates level using the composing circuits).

Alpha-0 [8] is a simulated computer, built for academic purposes. It allows one to understand the behaviour of a computer system from the architectural point of view. It also permits one to make performance analyses of the subsystems. Each of the system's levels are simulated individually. At present, an extension using the DEVS formalism allowed to build components as atomic models could be coupled and reused. They could be tested separately and lately, integrated to complete construction of the computer. The following sections will explain the design of this computer.

6.1 Digital Logic Level

The lower level specified considered each model as a basic circuit built using Digital Logic [9]. Complex circuits are built as a set of primitive components: the logical gates AND, NOT, OR, NOR and XOR (the last two were derived from the first ones). Using the basic logical gates, higher level circuits can be built as coupled models. The following are included:

- *Comparator*: It simulates a circuit comparing two inputs, determining which is different from the other

(including also inequality comparators).

- *Multiplexor*: Input lines are detected and one is chosen. This value is transmitted, ignoring the other values.
- *Decoder*: It activates one output line (there are 2^n outputs) corresponding to a number composed by the input values (n input lines).
- *D-latch*: These circuits simulate the storage of information into the processor. The d-latch stores one bit and is driven by the poise of a clock.
- *Shifter*: It shifts a set of bits one bit to the left or to the right, filling the empty places with zeros.
- *Adder*: It adds two bits by considering a third input representing the carry bit. The result of the addition and propagation of the carry are returned.
- *Register*: It is built by connecting several d-latches.
- *One-bit ALU*: The unit has only four one-bit operations: ADD, AND, OR and NOT. It was built using the decoder and adder units, showing the use of simple circuits used to build complex ones (see Fig. 10).
- *N-bits ALU*: It was built by connecting several one-bit ALUs. A row of one-bit ALUs should be connected, linking each carry-out with the next carry in.

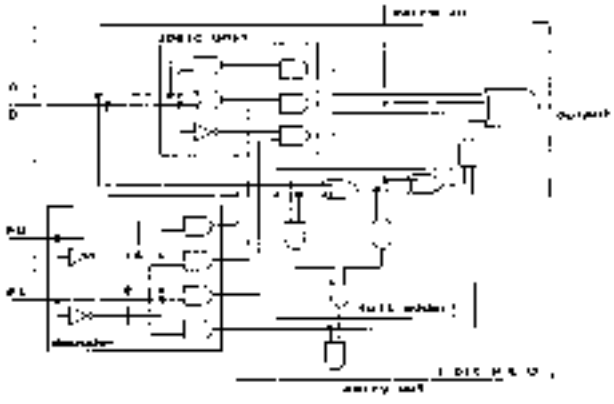

Figure 10. One-bit ALU.

The size of the circuits are dynamic and a graphical interface allows one to see the circuits' basic schemes (the previous figures were generated using the library). The an

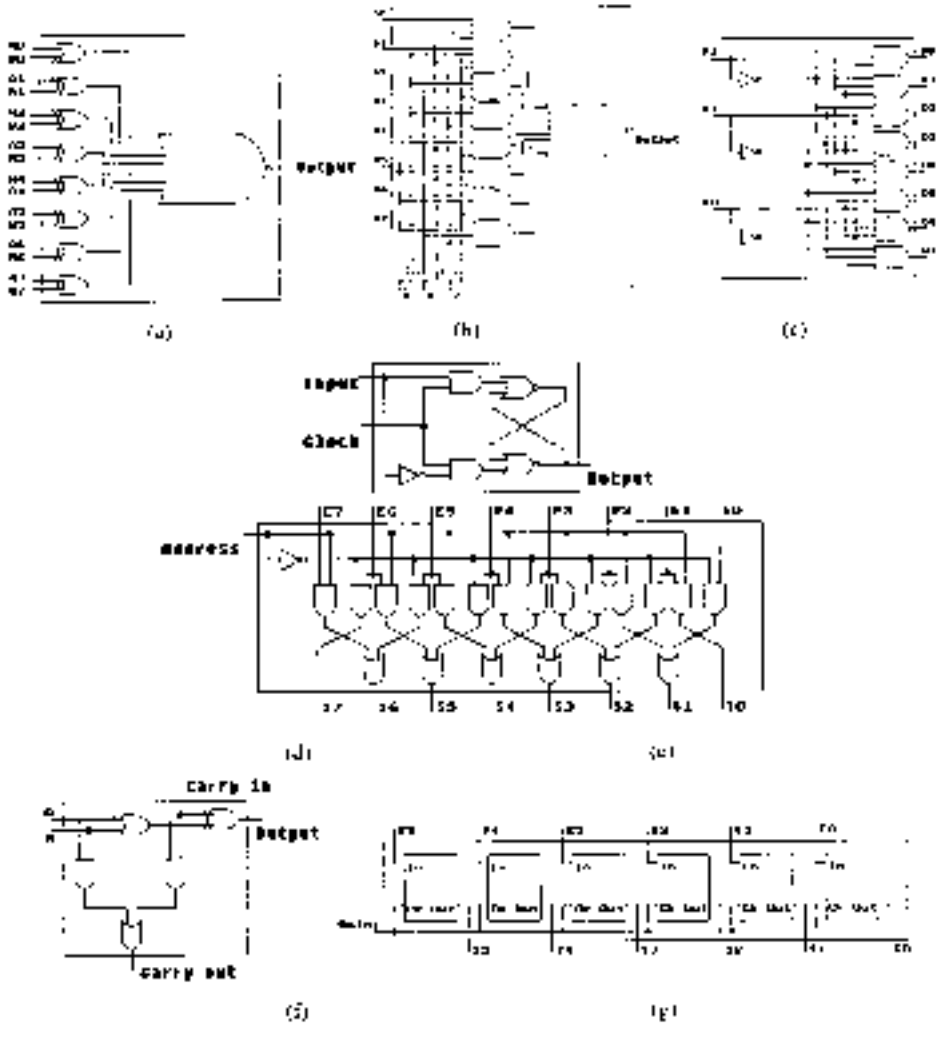

Figure 9. Modelled circuits (a) comparator (b) multiplexor (c) decoder (d) D latch (e) shifter (f) adder (g) register.

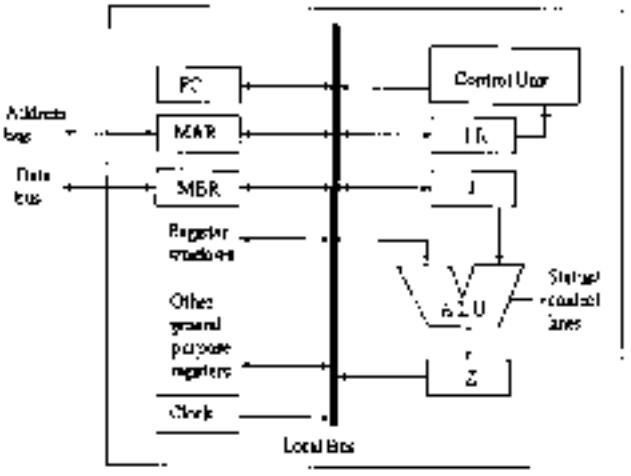

Figure 11. Structure of the simulated microarchitecture.

terface also shows the changes in the input lines' values, allowing one to study the detailed behaviour of each circuit.

6.2 Microarchitecture Level

As a second step, the circuits are used to simulate the execution of a microprogrammed processor. The microarchitecture components are supposed to be connected by a single local bus. The Control Unit executes a microprogram for each instruction, using a language that allows defining input/output flow between the processor's components. Each component is defined as a coupled models using the Digital Logic level, and when other models were needed, new atomic models were built. The structure of the microarchitecture is defined in Fig. 11.

It is supposed that the memory, processor, and input/output subsystems are connected by synchronic buses. The delays for each microoperation were also specified, therefore, the total execution time for each instruction can be computed. Each macroinstruction can be traced, showing the status of the local bus and registers, and the data path.

A cache memory with 64 bytes' cache was also simulated [10]. It has 32 words divided into eight blocks of eight bytes each. Several algorithms were tested, including Direct, Associative (FIFO, Least Frequently Used (LFU), Random, and Least Recently Used (LRU)), and Set Associative Mappings. Various tests were executed, comparing execution time of the microcode operations using the original simulator and the ones with cache memory. The results obtained can be seen Fig. 12.

The Instruction Level set is encapsulated into the Control Unit behaviour. The SPARC architecture was chosen as a reference to build the model, allowing use of a RISC (Reduced Instruction Set Computer) platform in low cost processors. The complexity of this level was reduced by restraining the complexity of the processor [11].

7. Conclusion

This work introduced the main features of CAD, a tool for General Application DEVS modelling and simulation. The

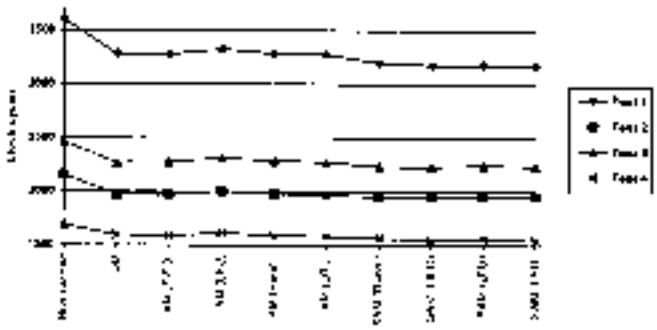

Figure 12. Test results of caching with different policies. DM: Direct Mapping, AM: Associative, SAM: Set Associative.

tool was built using a formal modelling 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 carried out, proving the usefulness of the tool. A data base of models can be created, enhancing the development process. The tool is being used for educational purposes and the models presented used to test multiprocessor configurations.

A complete set of models was used to simulate a simple computer. The resulting environment can be used in computer organization courses to analyse and understand the basic behaviour of the different levels of a computer system. Interaction between the levels can be studied and an experimental evaluation of the system can be done.

The tools are public domain and can be obtained at "http://www.dc.uba.ar/people/proyinv/celldevs". A new modelling paradigm called Timed Cell DEVS, was also implemented. The formalism is based on the DEVS and Asynchronous Cellular Automata paradigms. The concepts of transport or inertial delays used in the circuit modelling domain have been combined, allowing simple specification of accurate timed models. The specifications have been defined for binary or three-state systems. The formalisms allow automatic definition of the spaces and cases verification of the models, allowing efficient and cost-effective development of simulators.

Acknowledgements

This work is an extension of those previously presented at IASTED AMS'98. It was partially supported by USENIX Foundation, ANPCYT Project 11-04460 and UBACYT Projects JW16 and TX-004.

References

1. B. Zeigler, *Theory of modeling and simulation* (New York: Wiley, 1976).
2. A. Beyelke, J. Beyaloman, & G. Wainer, GAD: A general application DEVS environment, *Proc. IASTED Applied Modelling and Simulation AMS'98*, 1998.
3. B. Zeigler, *Object oriented simulation with hierarchical modular models* (Academic Press, 1990).
4. M. Zagar & D. Basch, Microprocessor architecture design with ATLAS, *IEEE Design and Test of Computers, 1993*, 1997.

[5] J. Edmondson, & M. Reilly, Performance simulation of an Alpha microprocessor, *IEEE Computer*, 1998.
[6] M. Rosenblum, E. Bugnion, S. Devine & S. Herrod, Using the SimOS machine simulator to study complex computer systems, *ACM Trans. Modelling and Computer Simulation*, 1997
[7] K. Shanmugan, V. Frost & W. La Rue, A block-oriented network simulator (BONeS), *Simulation*, 1992
[8] G. Wainer, Alpha 0: a simulated computer as a tool for Computer Organization courses, *Proc. IASTED Applied Modelling and Simulation AMS'98*, 1998
[9] A. Ferrari S. Romano, & G. Wainer, *Implementation of a digital logic library*, Internal Report. Departamento de Computación. Universidad de Buenos Aires. 1998
[10] R. Rogers & G. Wainer, *Implementation of a simulated cache memory for the Alpha-0 processor*, Internal Report. Departamento de Computación. Universidad de Buenos Aires. 1998
[11] A. Tolcenli & G. Wainer, *CRAYS, A simulator for the SPARC processor*, Internal Report. Departamento de Computación, Universidad de Buenos Aires. 1998

Biographies

Amir G. Barylko received the M.Sc. degree in 1998 from the Universidad de Buenos Aires, Argentina. He is a Ph.D. candidate at the same university and a teaching assistant in the same department. He has published several articles in the field of discrete-events simulation and participated in a research project in the area. He has been a free-lance consultant since 1996.

Jorge Beyoglonian received the M.Sc. degree in 1998 from the Universidad de Buenos Aires, Argentina and he is a Ph.D. candidate at the same university. He has published several articles in the field of discrete events simulation, and participated in a research project in this area. He has been a free-lance consultant since 1996.

Gabriel A. Wainer received his Licentiate degree (M.Sc., 1993) and his Ph.D. degree (1995, with honours) from the Universidad de Buenos Aires, Argentina, and DEA/DEA/PIM, Université d'Aix-Marseille III, France, respectively. He is an Assistant Professor at the SCE Dept., Carleton University (Ottawa, Canada). He was an Assistant Professor at the Computer Sciences Dept. of the Universidad de Buenos Aires, Argentina, and has been a teaching and research assistant in that department since 1988, and a Visiting Research Scholar at the University of Arizona, Tucson, AZ. He has published more than 50 articles in the field of operating systems, real-time systems and discrete-event simulation including two books. He is a member of the Board of Directors of the Society for Computer Simulation International, and a member of a group on standarization of DEVS modelling tools. His current research interests include modelling methodologies and tools, modelling and simulation of cellular models, parallel execution of models and real time simulators.