# DEFINING DEVS MODELS WITH THE CD++ TOOLKIT

Gabriel Wainer Gastón Christen Alejandro Dobniewski

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

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

E-mail: gwainer@sce.carleton.ca

**KEYWORDS**

Discrete Event modelling and simulation; Modelling methodologies; Modelling and Simulation tools.

**ABSTRACT**

We introduce the features of a toolkit for modeling and simulation based on the DEVS formalism. It is built as a set of independent software pieces running in different platforms. We show the main features of the environment and how to use it through application examples for a variety of problems. Many models can be defined in an automated fashion, simplifying the construction of new models, and easing the verification of structural models. The use of this formal approach allowed developing safe and cost-effective simulations, reducing significantly the development times.

**INTRODUCTION**

In recent years, several efforts have been devoted to define modelling paradigms, allowing improving the analysis of complex dynamic systems through simulation of these models. DEVS (Discrete Event systems Specification) allows modular description of models that can be integrated using a hierarchical approach (Zeigler et al. 2000).

We have built a toolkit with the goal of develop models and simulate them based on the DEVS and Cell-DEVS paradigms. The core of the toolkit is the **CD++** environment (Wainer *et al.* 2001), which implements the DEVS and Cell-DEVS theory. The toolkit has been built as a set of independent software pieces, each of them independent of the operating environment chosen. There are versions running under Windows 95/NT, Linux, AIX, IRIX, HP-UX and Solaris. Graphical interfaces are built as independent front-ends. This approach lets the user, for instance, to debug the models in a workstation, execute them in a high performance environment, and visualize the results in a personal computer (locally or remotely).

The goal of this article is to analyze the characteristics of the toolkit, and its use to develop simulation models based on the DEVS paradigm. We focus in the development process of simulated models using a graphical interface that allows defining DEVS models.

**THE DEVS FORMALISM.**

DEVS was originally defined in the '70s as a discrete-event modelling specification mechanism. It is a systems theoretical approach that allows the definitions of hierarchical modular models that can be easily reused (Zeigler et al. 2000). A real system modeled with DEVS is described as a composite of submodels, each of them being behavioral (atomic) or structural (coupled). A DEVS atomic model is formally described by:

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

where $\boldsymbol{X}$ is the input events set; **S** is the state set; $\boldsymbol{Y}$ is the output events set; $\boldsymbol{\delta_{int}}$ is the internal transition function; $\boldsymbol{\delta_{ext}}$ is the external transition function; $\lambda$ is the output function; and **D** is the duration function.

A DEVS coupled model is composed of several atomic or coupled submodels. They are formally defined 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; **D** is an index for the components of the coupled model, and $\forall\, i \in D$, $\mathbf{M_i}$ is a basic DEVS (that is, an atomic or coupled model), $\mathbf{I_i}$ is the set of influencees of model i (that is, the models that can be influenced by outputs of model i), and $\forall\, j \in I_i$, $\mathbf{Z_{ij}}$ is the i to j translation function. We can see that coupled models are defined as a set of basic components (atomic or coupled) interconnected through the model's interfaces. The translation function is in charge of converting the outputs of a model into inputs for the others. To do so, an index of influencees is created for each model ($I_i$). This index defines that the outputs of the model $M_i$ are connected to inputs in the model $M_j$, where j is an element of $I_i$.

**DEVS MODEL DEFINITION IN CD++**

CD++ implements the DEVS theory. It allows defining models according to the specifications introduced in the previous section (Wainer *et al*. 2001, Rodríguez and Wainer 1999). A set of independent applications related with the tool allows the user to have a complete toolkit to be applied in the development of simulation models.

The tool is built as a hierarchy of models, each of them related with a simulation entity. Atomic models can be programmed and incorporated onto a basic class hierarchy programmed in C++. A specification language allows defining the model's coupling, including the initial values and external events.

Model definition in C++ allow the user great flexibility to define behavior. Nevertheless, a non-experienced user can have difficulties in defining models using this approach. The provision of graphical notations can provide the modeler with a powerful tool to define models. Graph-based notations have the advantage of allowing the user to think about the problem in a more abstract way. Therefore, we have used an extended graphical notation to allow the user define atomic models behavior (Zeigler et al. 1996). Each graph defines the state changes according to internal and external transition functions, and each is translated into an analytical definition.

A unique identifier that will be used subsequently defines a DEVS model. Each model can include a graph-based specification representing state changes for an atomic model.

States are represented by bubbles including an identifier and the state lifetime. This specification allows to define the pair (state, duration) associated with internal transition functions. When the lifetime is consumed, the model will change of state by executing an internal transition function. For instance, the Figure 1 shows a state called "Start", whose duration is 15 time units.


Figure 1: State Graphical Notation: Identifier, Time Length

The syntax for the state analytical specification is:

```
state : stateId …
stateId : lifetime
```

As explained earlier, each model includes an interface with input/output ports. Ports are described by including their name and a type, based on the formal specification for DEVS models. They are specified as:

```
in  : portId:type  portId:type …
out : portId:type  portId:type …
```

Internal transition functions are represented by arrows connecting two states. Each of them can be associated to pairs of ports with values (p,v) corresponding to the output function. The syntax for the output function values is **p!v.** For instance, this figure represents that the model will change from state A to state B after 2 time units. First, the output function will send the value 8 through the port q1; 4 through the port q2, and 12 through the port q3.

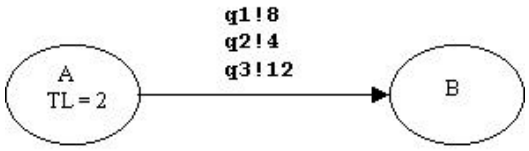


Figure 2: Definition of an Internal Transition Function

The syntax for the internal transition function construction is the following:

```
int : startState endState [outPort!value]+
```

Here we indicate the origin and destination states, and a port list with the corresponding values. For instance, the model in the previous figure can be described as:

```
int : A B q1!8 q2!4 q3!12
```

External transition functions are represented graphically by a dashed arrow connecting two states. The notation used to represent ports and expected values is similar to the one used for external transition, but replacing the exclamation mark by a question mark: **p?v [$t_i$..$t_f$].** Here, ti...tf represent the initial and final expected simulated times for the external transitions. These values allow to validate the timing of the models, rising an error if an external transition comes out of time. The syntax for this construction is the following:

```
ext : startState endState inPort value timeRange
```

It describes the origin and destination states, an input port and a time range counted since the instant arriving to the start state.

All these constructions can be combined to define the behavior of atomic models. For instance, the following figure represents a simple model using all the constructions:

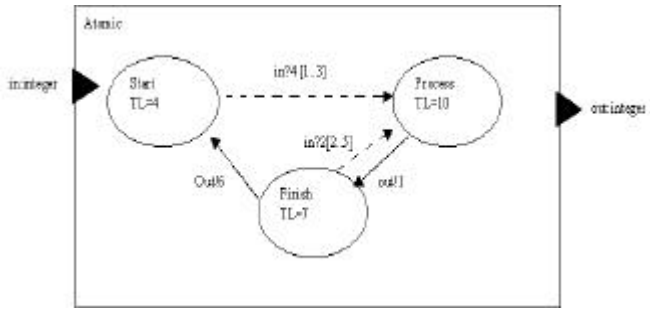


Figure 3: Definition of an Atomic Model

The tool is provided with a GUI entitling the definition of these constructions. The previous graphical specifications are used to generate an analytical specification, which is used by CD++. We can see the definition of this example in Figure 4. This model can be formally specified as:

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

**I** = $<P^X, P^Y>$ where $P^X$ = { ("in", integer) }; $P^Y$ = { ("out", integer) };

**X** = **Y** = ***Z*** ;
**S** = { Start, Process, Finish };

$\delta_{ext}$(s,e,x):

```
case port (in) {
        4:      if (e < 1 or e > 3) error();
                phase = Process;
                σ = 10;
        2:      if (e < 2 or e > 5) error();
                  if (phase != finish)
                        phase = Process;
                        σ = 10;                 }
```

λ(s):

```
case (phase) {
        Finish:   send(out, 6);
        Process: send(out, 1);        }
```

$\delta_{int}$(s):

```
case (phase):
        Finish: passivate();
        Process: hold_in(Finish, 7);
```

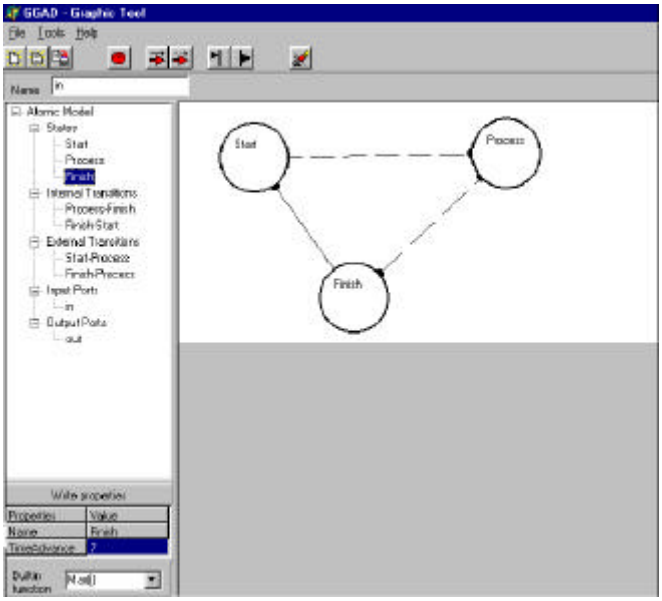


Figure 4: Graphical Specification of the Example

This description is translated into the following analytical form, which is equivalent to the previous specification:

```
[exampleGG]
in: in
out: out
state: Start Process Finish
int: Process Finish out!1
int: Finish Start out!6
ext: Start Process  in 10
ext: Finish Process  in 2
Start:0
Process:10
Finish:7
```

Figure 5: Analytical Definition of the Example

## COUPLED MODELS DEFINITION

After each atomic model is defined, they can be combined into a multicomponent model. Coupled models are defined using a specification language specially defined with this purpose. The language was built following the formal definitions for DEVS coupled models. Therefore, each of the components defined in section 2 for coupled models can be included. Optionally, configuration values for the atomic models can be included.

The **[top]** model always defines the coupled model at the top level. As showed in formal specifications presented in section 2, four properties must be configured: components, output ports, input ports and links between models. The following syntax is used:

- **`Components`**: it describes the models integrating a coupled model. The syntax is: `model_name@class_name`, allowing more than one instance of the same model using different names. The class name reference to either atomic or coupled models (which should be defined in the same configuration file).
- **`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 and, if it is not indicated, the coupled model being defined is used.

```
[top]
components: DeparturesQ@StoppableQueue Track@Track
LandingQ@StoppableQueue  ControlTower@ControlTower
Hangar
in : In
out : Out
link : out@DeparturesQ In_d@ControlTower
link : out@LandingQ In_a@ControlTower
link : In in@LandingQ
link : Out_a@Track In@Hangar
link : Out_d@Track Out
link : Done_a@ControlTower in@LandingQ
link : Stop_a@ControlTower stop@LandingQ
link : Departing@ControlTower Departing@Track
link : Landing@ControlTower Landing@Track
link : Done_d@ControlTower in@DeparturesQ
link : Stop_d@ControlTower stop@DeparturesQ
link : Out@Hangar in@DeparturesQ

[Hangar]
components   :   selector@selector   deposit1@queue
deposit4@queue    deposit3@queue    deposit2@queue
Chosen@DeMux
in : In
out : Out
link : out1@selector in@deposit1
link : out2@selector in@deposit2
link : out3@selector in@deposit3
link : out4@selector in@deposit4
link : out@deposit1 in1@Chosen
link : out@deposit4 in4@Chosen
link : out@deposit3 in3@Chosen
link : out@deposit2 in2@Chosen
link : In in@selector
link : out@Chosen Out
```

Figure 6: Coupled Model Definition

Let us consider the specification in the previous figure, which represents a small Airport. The control tower is connected to two queues: one for departures, and the other for arrivals. These queues are used to model the time employed by planes to enter or leave the airport area. The control tower is also connected to a model representing the track. Every time a plane is authorized to depart or land, the track

model is activated. Finally, all landed planes go to a Hangar for maintenance. A plane can only leave after service. The hangar can be defined as an another atomic model, or as a coupled model with different service stations for the planes. Figure 7 in the Appendix shows the definition of this formal description using the coupling specification language of the tool. The tool uses this information to generate instances of previously defined atomic models, or creates new instances of coupled models that can be later reused to form other multicomponent models. The analytical specification for the example is described following:

## SIMULATING MODELS

Once a model has been generated and its description is included in the modelling hierarchy, it can be simulated. As explained earlier, the model interaction is carried out through message passing. The ultimate goal of each model is to receive inputs through the input ports, and generate outputs in the output ports according to the definitions of the transition functions. The tool provides a way of registering every input and output of individual models. Nevertheless, fully detailed interaction between the models can be registered by analyzing a log output file. The values in the log file can be used to provide a generic graphical output. The following figure shows its use when executing the Control Tower model presented in an earlier section (at present, a prototype – all the state variable values have been added by hand to make clearer the description of the model execution).

In Figure 8 we show the execution of the control tower when three different requests are demanded. The model is initially in a *passive* state (with no scheduled internal events, that is, *sigma = infinite*). In simulated time 3, an input request arrives through the port *in_d*. Checking the model specification, we see that the flight information is stored, and an internal event is scheduled. In this case, we need a preparation time of 7 time units. During this time, the model remains in the *prep_landing* phase. When the time is consumed (*sigma* = 0), an internal function is executed. The output function executes first, sending the STOP signal to other models (represented by a short arrow in the figure). Then, the internal transition function is executed, queuing the plane, choosing it (as there is no other plane queued), and putting the model in the *landing* state during 7 time units. When this time is consumed, the GO signal is sent to the output ports, and the flight #1 is sent through the corresponding port (in this case, *departing*). A second plane arrives, and the procedure is repeated. When 3 time units have been consumed, a new external event occurs, indicating an emergency plane (#4). Then, the emergency signal is sent to the landing port, the previous plane is dequeued, letting flight #4 to land. When it finishes landing, flight #2 is authorized to use the track, and lands (the values related with this flight are kept in the control tower queue).

## CONCLUSION

We have introduced several features CD++, a toolkit for DEVS modelling and simulation. The tool was built using the DEVS 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. It executes in different platforms, from low priced personal computers up to high performance multiprocessors or distributed systems (keeping the semantics for predefined models).

The tool was used to develop several kinds of application examples, which allowed us to show the flexibility of the toolkit. Several types of models can be integrated in an efficient fashion, allowing multiple points of view to be analyzed using the same model. The formalism allows improving the security and cost in the development of the simulations. Experimental results of application showed improvements for expert developers. The main gains have been reported in the testing and maintenance phases, the more expensive for these systems (Wainer and Giambiasi 2001). The tools are public domain and can be obtained at "http://www.sce.carleton.ca/faculty/wainer/celldevs". The developed models are available, and will be incorporated to a web-based environment that can be applied to the development of DEVS models.

## REFERENCES


Rodríguez, D.; Wainer, G. "New Extensions to the CD++ tool". In *Proceedings of SCS Summer Computer Simulation Conference*. Chicago, IL. U.S.A. 1999.

Wainer, G.; Barylko, A.; Beyoglonián, J. "Experiences with DEVS modelling and simulation". In *IASTED Journal on Modelling and Simulation*. March 2001.

Wainer, G.; Giambiasi, N. "Application of the Cell-DEVS paradigm for cell spaces modelling and simulation.". *Simulation.* January 2000.

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

**APPENDIX**

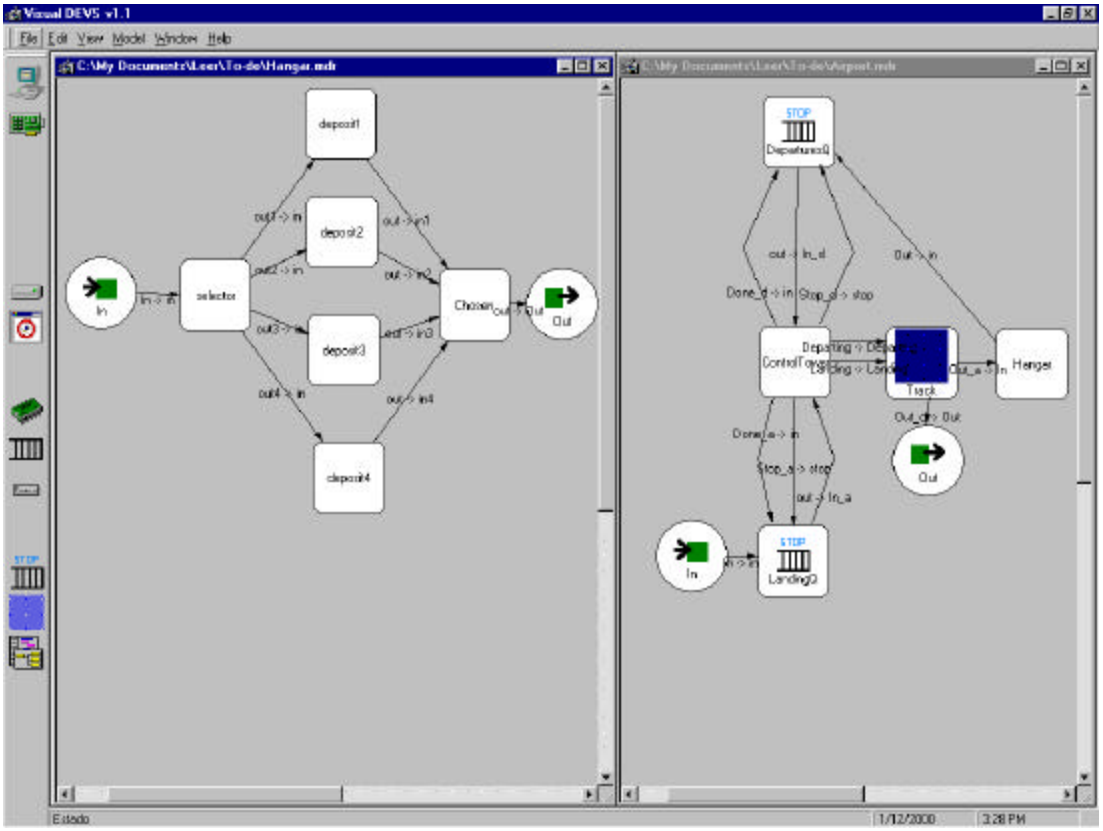


Figure 7: Graphical Specification for the Airport Coupled Model

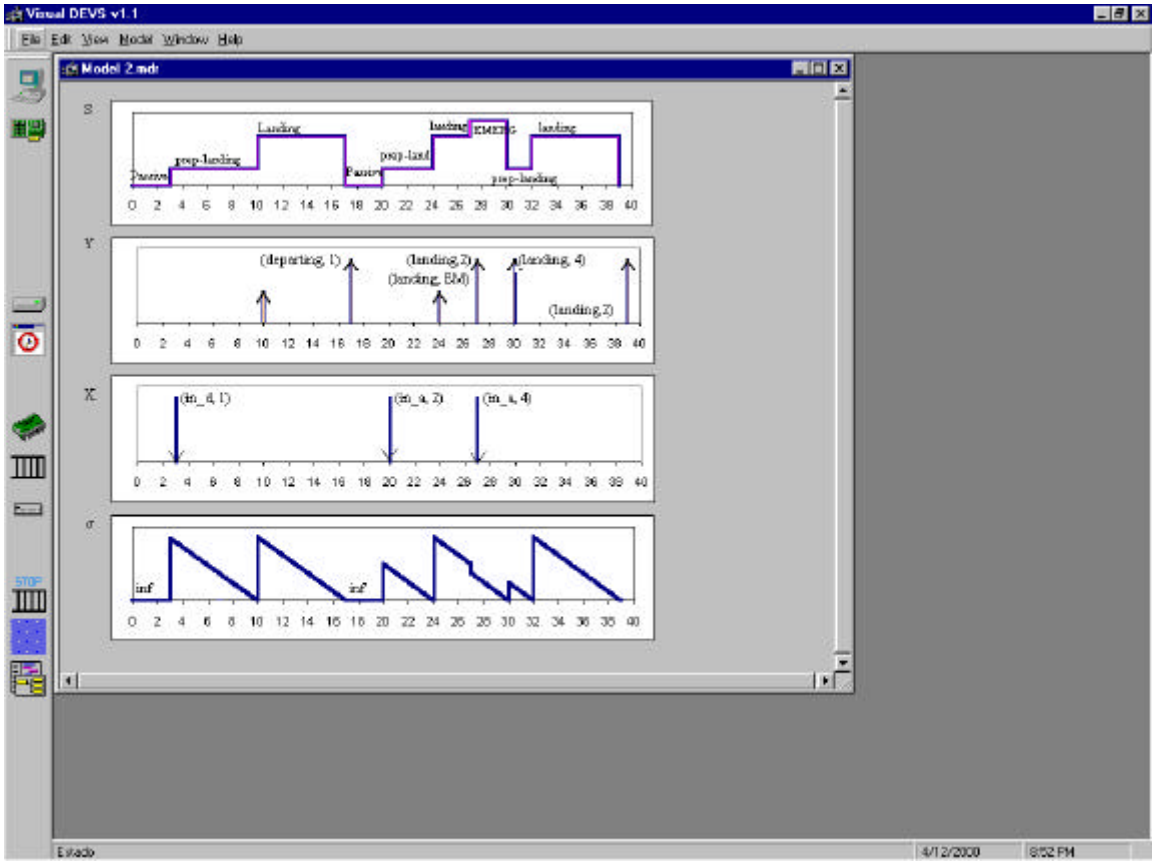


Figure 8: Execution Results for the Airport Example