CD++: A TOOLKIT TO DEVELOP DEVS MODELS

Gabriel Wainer

Department of Systems and Computer Engineering.
Carleton University
4456 Mackenzie Building
1125 Colonel By Drive
Ottawa, ON. K1S 5B6
gwainer@sce.carleton.ca

Summary: the features of a toolkit for modeling and simulation based on the DEVS formalism are presented. The tool is built as a set of independent software pieces running in different platforms. Not only the main characteristics of the environment are presented, but also focus in its use is considered by inclusion of application examples for a variety of problems. Many models can be defined in an automated fashion, simplifying the construction of new models, and easing their verification. The use of this formal approach allowed developing safe and cost-effective simulations, reducing significantly development time.

Keywords: Discrete Event modeling and simulation; Modeling methodologies; DEVS formalism; Cell-DEVS; Modeling and Simulation tools.

INTRODUCTION

In recent years, a number of modeling techniques were introduced in order to improve the definition and analysis of complex dynamic systems. The development of simulation tools has often been closely related to the execution of these models. A considerable effort has been put in the development of formal modeling techniques, which showed to be useful thanks to their ability to define executable models. Several of these methodologies were created with the purpose of analyzing discrete-event systems, that is, systems that can be represented using continuous time and discrete state variables [1].

The use of a continuous time base enables accurate timing definitions, which improves model precision. Continuous time representations avoid small discrete time segments, thus reducing processing requirements. A discrete-event formalism that gained popularity in recent years is called DEVS (Discrete Event systems Specification). It allows modular description of models that can be integrated using a hierarchical approach [2, 3]. It was developed as a theory for discrete event models, but recent extensions allow the inclusion of continuous variable systems [4, 5, 6]. Using these approaches, Quantized or Generalized DEVS can be applied to define arbitrary ordinary differential equations. The Cell-DEVS formalism [7] is a combination of DEVS and Cellular Automata [8] formalisms with timing delays. The idea is to permit describing complex physical systems as a space composed by cells, in which each element in the grid is defined as a DEVS model using explicit timing.

We have built a toolkit to develop models based on DEVS and Cell-DEVS. The core of the toolkit is the **CD++** environment [9], which implements DEVS and Cell-DEVS theory (including quantized systems). The toolkit has been built as a set of independent software pieces. Each tool runs in different operating environments (Windows 95/NT, Linux, AIX, IRIX, HP-UX, and Solaris). Graphical interfaces were built as independent front-ends, and current versions use Java and VRML to insure platform-independent execution. 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. The visualization of the execution results can be done locally or remotely, as we have included facilities to do web-based simulation. We will present the main features of the toolkit and its use to develop simulation models, focusing in the development process. After reviewing the basic concepts related with DEVS and Cell-DEVS theory, we will introduce the main features of the toolkit and show how to define new models. Then, we will see how to execute models, and will present some simulation results.

THE DEVS FORMALISM.

DEVS was originally defined in the '70s as a discrete-event modeling specification mechanism. It was derived from systems theory, and it allows one to define hierarchical modular models that can be easily reused. A real system modeled with DEVS is described as a composite of submodels, each of them being behavioral (atomic) or structural (coupled). Closure under coupling allows coupled models to be integrated to a model hierarchy. Consequently, the security of the simulations is enhanced, testing time reduced, and productivity improved.

Each model is defined by a time base, inputs, states, outputs, and functions to compute the next states and outputs. A DEVS atomic model is formally described by:

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

X is the input events set;

S is the state set;

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; and

ta is the time advance function.

Each model is seen as having input and output ports to communicate with other models. The input and output events determine the values to appear in those ports. The input external events are received in input ports, and the specification of the external transition function defines the behavior under such inputs. The internal transition function is activated after the lifetime of the present state has been consumed, which is defined by the time advance function. Its goal is to produce an internal event, which lead to a state change. The desired results are spread through output ports by the output function, which executes before the internal transition.

A DEVS coupled model is composed of several atomic or coupled submodels. They are formally defined as:

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

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 (that is, an atomic or coupled model),

$\mathbf{I_i}$ is the set of influencees of model i (that is, 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), which are interconnected. The influencees of a model define to which model outputs must be sent. 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.

Cell-DEVS [10] is an extension to the DEVS formalism, which allows the implementation of cellular models with timing delays. A cellular model can be defined as an infinite n-dimensional lattice with cells whose values are updated according to a local rule. This is done using the present cell state and those of a finite set of nearby cells (called its neighborhood). The goal of Cell-DEVS is to improve execution performance of cellular models by using a discrete-event approach, and to enhance timing definition by making it more expressive. Here, each cell is defined as an atomic model using timing delays, and it can be later integrated to a coupled model representing the cell space. Cell-DEVS atomic models can be specified as:

$$TDC=< X, Y, S, N, delay, d, \delta_{int}, \delta_{ext}, \tau, \lambda, ta >$$

X defines external input events,
Y is the set of external output events,
S is the set of sequential states for the cell,
N is the set of input events;
delay defines the kind of delay for the cell,
d defines the delay's length;
d_{int} is the internal transition function,
d_{ext} the external transition function,
t is a local computing function,
l the output function, and
ta is the time advance function.

Each cell uses N inputs to compute its next state. These inputs, which are received through the model interface, activate the local computing function. A delay can be associated with each cell, allowing the deferral of the computed result to be transmitted to other models. **Transport** delays model a variable commuting time. Instead, **inertial** delays have preemptive semantics (some scheduled events can be avoided). The model advances through the activation of the internal, external, output and state's duration functions, as in other DEVS models.

Once the cell behavior is defined, they can be put together to form a coupled model:

$$GCC=< Xlist, Ylist, X, Y, n,\{t1,..,tn\}, N, C, B, Z >$$

Xlist is an input coupling list,
Ylist is an output coupling list,
X is the set of external input events,
Y is the set of external output events,
n defines the dimension of the cell space,

{t1,...,tn} is the number of cells in each dimension,

N is the neighborhood set,

C is the cell space state set,

B is the set of border cells, and

Z the translation function.

This specification defines a coupled model for an n-dimensional array of atomic cells. Each of them is connected to its neighborhood, whose shape must be defined. In order to define the cell space within finite boundaries, the border cells should be provided with a different behavior than the rest of the space. Otherwise, it is considered that the cells in a border are connected with those in the opposite one. Finally, the Z function defines the external couplings using the Xlist and the Ylist and internal coupling using the neighborhood definition.

As explained in the Introduction, both formalisms provide the advantages of being discrete-event approaches in terms of execution performance. Discrete event models evolve in continuous time, represented by the occurrence of instantaneous events that can occur asynchronously at unpredictable times. The hierarchical and modular organization allows describing multiple layers of a given application. DEVS models are closed under coupling, therefore, a coupled model is equivalent to an atomic one, improving reuse. This organization makes easier the definition of submodels, which makes easy the definition of different levels of abstraction. The existence of an internal transition function eases the definition of certain properties. Internal state changes can be captured, describing complex internal interactions in a simple and natural way.

Both DEVS and Cell-DEVS provides the advantages of being a formal approach. Formal specification mechanisms are useful to improve the security and development costs of a simulation. DEVS supplies facilities to translate the formal specifications into executable models. In this way, the behavior of a conceptual model can be validated against the real system, and the response of the executable model can be verified against the conceptual specification. The formal specification of the models makes easier the verification of the simulators, as they should mimic the behavior of the models in a homomorphic fashion. In [23], a mechanism for achieving this goal was proposed, and we have used a simulation approach based on this strategy. Simulation engines are independent from the models, and a modeler only has to focus in defining the model correctly following the previous specifications.

DEVS also has shown to be a general formalism such that several other existing ones can be expressed as DEVS models (including Petri Nets, FSM, Cellular Automata, VHDL, or timed graphs). Consequently, a modeler can express different properties in an adequate formalism, and use DEVS hierarchical coupling as integration mechanism.

The hierarchical nature of the formalism makes easy the reuse of previously defined models. Therefore, model databases can be created, and models included in these repositories can be integrated to new hierarchies if their input/output trajectories have equivalent semantics. A related advantage is that the learning curve for DEVS tools is short enough to enable new modelers to learn the basics of the formalism without going into details about the underlying theories. In the following sections, we will exemplify these issues, showing how to develop new models and analyzing several examples focusing the mentioned advantages.

DEVS MODEL DEFINITION IN CD++

CD++ [11,12] allows defining of models following the specifications introduced in the previous section. 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++.

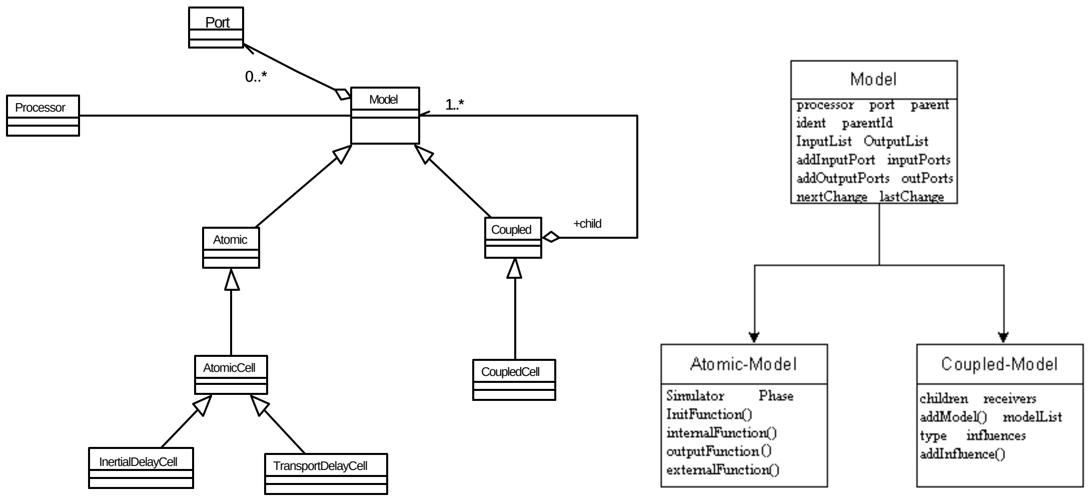


Figure 1. Cell-DEVS Models and Processors.

This class hierarchy implements the model theoretical definition presented in the previous section. New atomic models must be incorporated to the class hierarchy as subclasses of the Atomic Model class. The hierarchical nature of the formalism made the implementation of the tool in an Object-Oriented language is straightforward. We can see the tool as a modeling environment that complements the programming capabilities of C++.

The use of C++ also enabled us to obtain good performance in terms of model execution. Our models executed one order of magnitude faster than current existing Java environments [13]. The use of an Object-Oriented implementation reduced the time spent in testing, as reported in [14]. The formal definition permits focusing in the model to develop, whereas the Object-Oriented approach makes much easier to find related errors. A specification language allows defining coupled and Cell-DEVS models. This language provides a textual representation independent from any tool and development environment chosen.

The abstract simulation mechanism provided enables the modeler to focus in the definition of the models. The only relationship between the models and the simulation engine is defined by the manipulation of a variable containing the time of the next scheduled event, called *sigma*. This variable is used to implement the time advance function: it stores the time remaining until the next scheduled. The internal transition function is activated when *sigma=0*, and *sigma* must be recomputed every time a model is activated, as each state has an associated lifetime. Every model also includes a "phase" variable (whose basic states are active/passive), which can be used to verify correctness of the functions defined. For instance, a model in passive phase cannot have an internally scheduled event. Likewise, an active model cannot have an infinite value for *sigma*. The expression of these constraints is straightforward. Following, we explain how to incorporate atomic and coupled models to be simulated.

Atomic models definition

A new atomic model is created by including a new class derived from *Atomic*. In doing so, the following methods may be overloaded:

- ***initFunction***: this method is invoked when the simulation starts. It allows to define initial values and to execute setup functions for the model.
- ***externalFunction***: this method is invoked when an external event arrives from an input port.
- ***internalFunction***: this method is started when an internal event occurs (that is, the value of *sigma* is zero).
- ***outputFunction***: this method executes before the internal function, in order to generate outputs for the model.

These functions are equivalent to those defined in the formal specifications for atomic models. Consequently, different properties may be verified: inconsistent states (i.e., an internal transition function is activated before the scheduled internal events, external functions are activated before/after the proper timeslots, inconsistent states or valued are received via input ports, etc.). The newly defined models can be incorporated to the modeling class hierarchy. The following primitives can be used when defining an Atomic model:

- ***holdIn***(state, time): a model executing this sentence remain in *state* during *time* (*sigma*= *time*). When the time is consumed (*sigma* = 0), the model executes the internal transition. This macro was included to make easy the definition of the duration function.
- ***passivate***(): the model enters in passive mode (*phase = passive*; *sigma = infinite*) and it is only 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.

Let us consider the modeling of the activities in a small airport. The environment to be modeled includes several planes arriving and leaving the airport, a control tower, the runway, and a hangar where planes needing service can be sent to. For instance, we can define an Atomic Model for the Control Tower as follows:

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

X $\in$ { in_d $\in N$, in_a $\in N$ };

S $\in$ { rnwy-use_time, preparation_time $\in$ ***R+*** } $\cup$ { pl_queue $\in$ {fl_number $\in N$ x arr_time $\in$ ***R+*** x rnwy-use_time $\in$ ***R+*** x port_name $\in$ I }* } $\cup$ { which $\in$ ***N*** } $\cup$ { phase $\in$ passive, rnwy-use, prep-rnwy-use };

Y $\in$ { rnwy-use / rnwy-use $\in N$ } $\cup$ { departing / departing $\in N$ } $\cup$ { done_a, done_d $\in$ boolean } $\cup$ { stop_a, stop_d $\in$ boolean } ;

A formal specification for the transition functions can be found in [15]. Here, we show how to implement the transition functions using the tool. This model uses two input ports (X) and four output ports (Y). The *rnwy-use_time* variable stores the landing or departure time for the following plane. The *preparation_time* is related with the delay of the control tower controller to decide the next plane to be authorized. We use a plane queue, containing instances of (*flight number*, *arrival time*, *rnwy-use time*, *use of the runway*), which includes the basic information of each flight received by the control tower. Two different port names are used in order to recognize if it is a landing or departing flight. The *which* variable stores a pointer to the plane chosen to use the runway.

```
control::control( const string &name ) : Atomic( name ),
in_a(this->addInputPort("in_a")), in_d(this->addInputPort("in_d")),
done_a(this->addOutputPort("done_a")), done_d(this->addOutputPort("done_d")),
stop_a(this->addOutputPort("stop_a")), stop_d(this->addOutputPort("stop_d")),
rnwy-use(this->addOutputPort("rnwy-use")), departing(this->addOutputPort("departing")) {
}

Model &control::initFunction() {
        Preparation_time = control_tower_preparation();
        fl_number=0 ;
        return *this ;
}

Model &control::externalFunction( const ExternalMessage &msg ) {
        fl_number = msg.value();
        rnwy-use_time = query_time();
        if(port() == in_a )  // The chosen plane lands
                runwy-use = "landing";
        if(port() == in_d )  // The chosen plane departs
                runwy-use = "departing";

        // Only emergency planes are accepted while the runway is being used
        if (state() == rnwy-use) {
               // If the previous plane is finishing rnwy-use, continue
                if (elapsed_time < rnwy-use_time/2) { // Otherwise, empty the runway
                        queue(fl_number, msg.time(), rnwy-use_time, "emergency");
                        holdIn(emergency, preparation_time);
                        return (*this);
                }
        // A new plane wants to use the runway. Queue its information.
        queue(fl_number, msg.time(), rnwy-use_time, runwy-use);
        if (state() != rnwy-use)
               this->holdIn( prep_rnwy-use, preparation_time );
        return *this;
}

Model &control::internalFunction( const InternalMessage & ) {
    if (state() == rnwy-use or state()==emergency)

        if (queue == EMPTY)
               this->passivate();
        else
                this->holdIn( prep_rnwy-use, preparation_time );

    if (state() == prep_rnwy-use) {
                // Choose the first plane to be landed in the queue.
                which = schedule(queue);
                tail(queue);  // Delete the first element in the queue, that was chosen
                this->holdIn(rnwy-use, rnwy-use_time );
     }
     return *this ;
}

Model &control::outputFunction( const InternalMessage &msg ) {
        if (state() == emergency) {
                this->sendOutput( msg.time(), rnwy-use , RNWY-USE_CANCEL);

        if (state() == rnwy-use) {
               this->sendOutput( msg.time(), done_a , GO) ;
               this->sendOutput( msg.time(), done_d , GO) ;

        if( use == "landing")   { // The chosen plane lands
                        // Record the landing
                this->sendOutput( msg.time(), "landing" , fl_number) ;
                }
        if( use == "departing" )  {  // The chosen plane departs
```

```
                        // Record the departure
                        this->sendOutput( msg.time(), "departing" , fl_number) ;
                }
        }
        if (state() == prep_rnwy-use) {
               // While using the runway stop receiving new requests
                this->sendOutput( msg.time(), stop_a , STOP);
               this->sendOutput( msg.time(), stop_d , STOP);
        }
    return *this ;
}
```

Figure 2. Control Tower transition functions

The transition functions for this model define the activities carried out by the control tower. New requests for landing and departures are received at any moment, thus, they are implemented in the external transition function. Whenever a new plane needs to use the runway, it sends a request to the control tower, and the request is queued. We use two input ports (landing or departing) to define the type of use for the runway. We also detect emergencies here. If the model phase is *rnwy-use*, this means that the runway is being used. During this time, the planes do not send new requests to the control tower, unless they are in an emergency state. In this case, we use the *elapsed time* variable to decide how to react. If half of the time needed to depart/land has been elapsed, the control tower lets the present plane to finish using the runway. Otherwise, the plane is preempted and queued again, letting the emergency plane to land first.

The operator needs some time to do the related work (*preparation_time*). During this time, the control tower remains in the *prep_rnwy-use* state. Then, a plane must be chosen. When this has been decided, the control tower remains in the state *rnwy-use* during the use of the runway. During the use, other waiting planes are informed, and they do not request to use the runway. The *query_time* function analyzes the flight number and computes the time it takes to the plane to depart/land, according to the kind of aircraft, weather conditions, etc. When this time is consumed, the output function is executed. If the model was in *prep_rnwy-use* phase, this means that the control tower is prepared to let a new plane to use the runway. Hence, we send a "*STOP*" signal in order to avoid them to send new planes. Instead, if the present phase is *rnwy-use*, this means that a plane finished using the runway and the model can receive new planes. Therefore, a "*GO*" signal is sent to other models. When an airplane departs, its number is sent to other models through the "departing" port. When the plane is landing, its information is sent through the "landing" port. This information could be used to collect statistics, or to interact with other models.

As we can see, the formal specification provides a clear separation between input, output, and internal functions. It also facilitates verifying the model behavior. The occurrence of different states is automatically informed. For instance, if a phase different than those programmed is detected, an error is raised. If a message arrives in the wrong port or its associated time is wrong, the tool automatically raises an error. This procedure reduces the possibility of errors and makes easy the testing phase, as the user only has to focus in the behavior of the corresponding function. Besides this, the user only has to focus in the modeling activities (as we can see, there is no relationship with the simulation code at all).

These facilities enabled us to build an independent tool that facilitates model verification [16]. Once an atomic model has been built and incorporated to the modeling hierarchy, we can control if the model being verified returns the expected results at a given time. An Experimental Framework composed by a Generator and an Acceptor can be coupled to a base model to be verified. The Generator recognizes the input ports of the base model, and connects its outputs to the inputs of the model. The Acceptor recognizes output ports to be analyzed.

The Generator verification data is provided by the modeler, who should write an entry table indicating the testing values, and their corresponding correct results. The data corresponding to external transitions is sent by the Generator to the base model in order to be processed. Data corresponding to internal transitions is sent to the Acceptor, who stores this information and later compares it with the real values issued by the model. Consequently, output error messages are issued. The Results file allows checking the differences between the expected data and the outputs issued by the model. In addition, differences in timing can be analyzed.

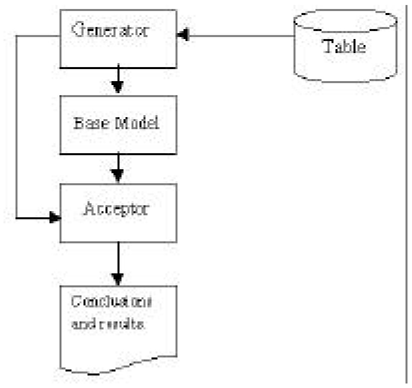


Figure 3. Format for the verification framework.

For instance, the Runway model definition was tested using the data shown in Figure 4. We have injected different types of inputs, to show the possible errors that can be obtained. Transition Type might be *I* or *E*, indicating if it refers to an internal or external transition respectively. For an *E* type transition, data in the table should be interpreted as "*Value X enters the model at time T*". Taking as reference, line number 1 in the example, we should interpret "*Value 2 enters the model at 00:01:08:000*". Likewise, for an *I* type transition, data in the table should be interpreted as *"Model must output value Y at time T*". Line 4 of the example should be read "*Model must output value 15 at 00:15:13:00".*

E 00:01:08:000 2	Output file:
I 00:05:11:000 2	
E 00:10:10:000 15	00:05:08:003 out 2
I 00:15:13:000 15	00:15:13:000 out 15
E 00:20:12:000 40	00:25:17:000 out 20
I 00:25:17:000 40	00:35:18:000 out 60
E 00:30:15:000 60	00:45:19:000 out 70
I 00:35:18:000 60	
E 00:40:19:000 70	
I 00:45:22:000 70	

Figure 4. Input/Output data for the Runway model.

The behavior of the *Runway* model is to receive planes. Every time a plane leaving or departing is received, it uses the runway for a period, after which, an output (representing the flight number) is generated. We run the verification frame in this model, we obtain the results presented in Figure 5. We first inject the flight number 2, and expected the runway to be ready at 5:11:000. Nevertheless, the runway outputs the plane 2 only 4 minutes and 0.003 seconds after the input. This execution results in an output error. Then, we inject flight 15, whose output is expected at 15:13:000. As we can see, no errors were raised in this case. The following error shows that, after injecting a value 20 at 20:12:000, the model returns a 20 at 25:17:000, which is an unexpected value according to the input definition. The final error shows another timing problem according to the inputs specified.

```
Verification results file:
Invalid result:
   Expecting: 2.000000 At: 00:05:11:000
   Getting: 2.000000 At: 00:05:08:003

Invalid result:
   Expecting: 40.000000 At: 00:20:17:000
   Getting: 20.000000 At: 00:20:17:000

Invalid result:  Expecting: 70.000000 At: 00:45:22:000 Getting: 70.000000 At: 00:45:19:000
```

Figure 5. Output data for the Runway model.

The verification mechanism highly enhances the testing procedures. The user can provide a set of test cases, whose definition can be done simultaneously with the model specification.

Graphical definition of Atomic models

Coding in C++ allow the user great flexibility to define model behavior. Nevertheless, non-experienced users can have difficulties in defining models using this approach. The provision of a graphical notation to specify the model behavior 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 graphs to allow the user define atomic models behavior [17].

Each graph defines the state changes according to internal and external transition functions, and each is translated into a textual definition. In this way, the users do not need to compile the new added models, which are interpreted by the modeling tool. The graphical notation of a model is just a frame with the model name. The corresponding textual notation has the following syntax: `[idModel]`. This identifier declares a DEVS model that can be used subsequently.

The graph-based specification for atomic models represents their state changes. Each state is represented by a bubble including an identifier and the duration for the state. This allow one to define the pair (state, duration) associated with internal transition functions. The following figure shows a state called "Start", whose duration is 15 time units.

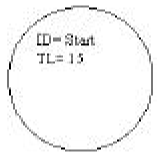


Figure 6. Graphical notation for a state: identifier and time length.

The text representation for states uses the following syntax:

`state :` `stateId1 … stateIdn`

`stateIdj : lifetimej`

We first enumerate all the states present in the model, and associate each of them with their corresponding lifetime. When the lifetime is consumed, the model changes of state by executing an internal transition function.

As explained earlier, each model includes an interface with input/output ports. We represent them as arrowheads associated to a model definition. The text specification for the ports include their name and a type, based on the formal specification for DEVS models, as follows:

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

For instance, the model in the following figure is specified as:

```
in  : p2:integer , p1;  /* integer default values */
out : q1:float , q2: integer;
```

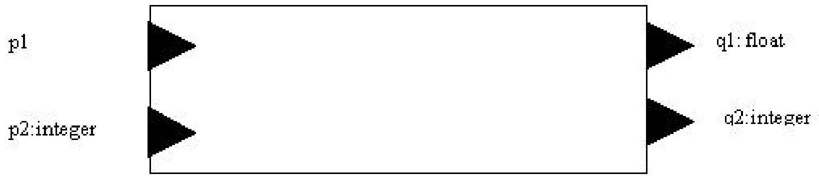


Figure 7. Graphical notation for input/output ports.

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 changes from state A to state B after 2 time units. First, the output function sends the value 8 through the port q1; 4 through the port q2, and 12 through the port q3.

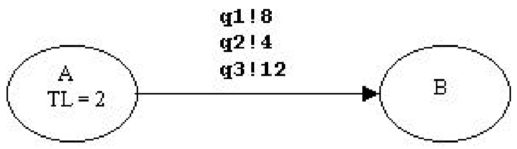


Figure 8. Definition of an internal transition function.

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 verifying 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 repres ents a simple model using all the constructions:

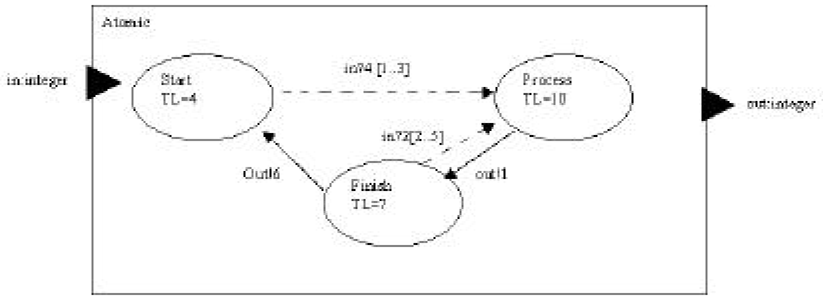


Figure 9. Definition of an atomic model

This model can be formally specified as:

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

X = { ("in", integer) }; **Y** = { ("out", integer) };

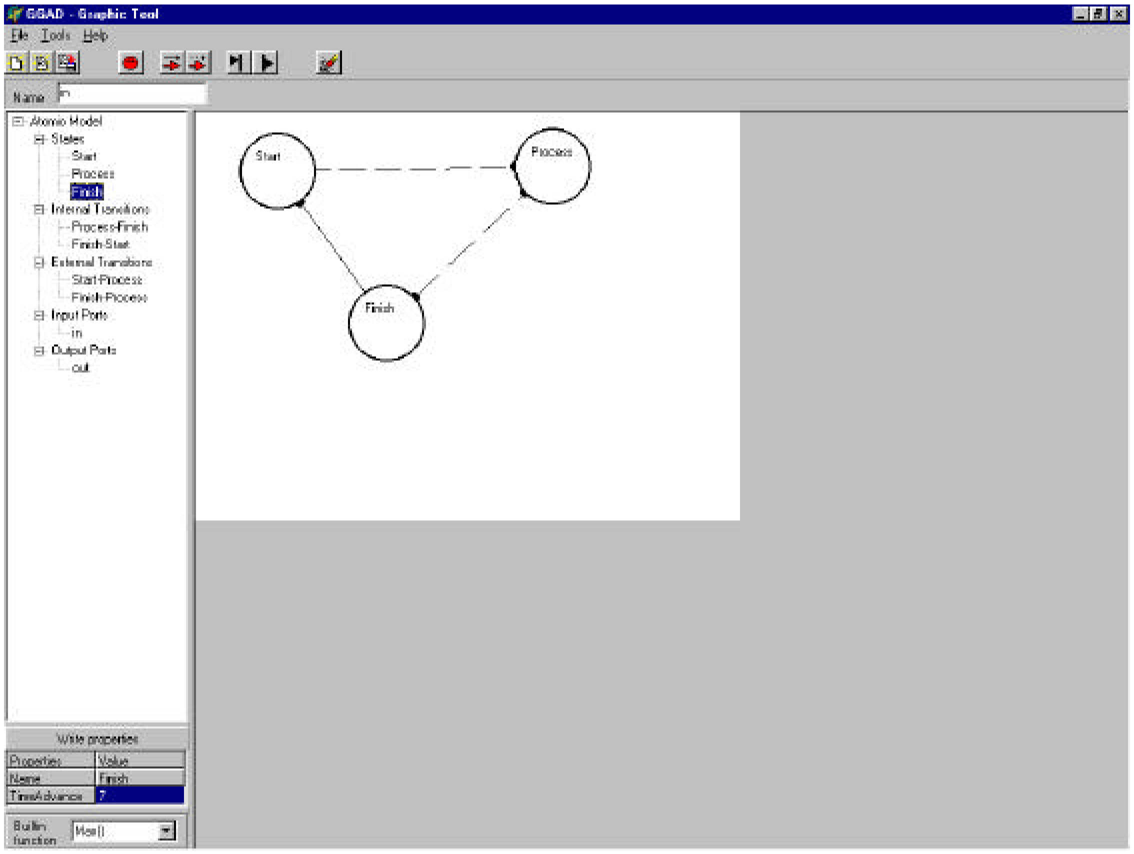
S = { Start, Process, Finish };

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

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

Following, we show the definition of this model using the tool, and the intermediate code generated.



Name: in

Properties	Value
Name	Finish
TimeAdvance	7

Figure 10. Graphical specification of a simple problem.

The tool uses the previous defined graphical specification to generate the text specification that is used by CD++. The behavior generated when this specification is read is that explained previously.

```
[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 11. Textual definition of the problem.

This notation enables a clear and clean graphical representation for DEVS atomic models, enabling non-expert users to define models using the toolkit. It also enables a faster learning curve. The core of the tool remains unchanged; therefore, the verification mechanisms mentioned earlier are still valid. It is even easier to detect ambiguous states or timing errors thanks to the reduce expressivity of the associated language. This simple notation enables the user to develop complex applications thanks to the hierarchical nature of the formalism. If the model states are too complex to be defined in a certain level of abstraction, several simpler submodels can be created and combined together. Nevertheless, in many cases, state machines are not powerful enough to solve complex problems, in which one cannot simply analyze state changes (as in FSM or Petri Nets). To attack these cases we have allowed the transition functions to invoke user-defined routines written in C++. They can be associated to the links representing the transition functions. Besides, if models that are more complex are needed, the user can directly define them in C++ as explained earlier. This variety of ways to express atomic models, combined with libraries of existing models let each user to attack the problem solving tasks at the adequate level of complexity. For instance, graph-based notations can be used for educational purposes, or for problems well suited for state-based approaches (such as communication protocols), whereas C++ coding can be used for defining complex hybrid systems.

Coupled models definition

After defining the atomic models for a given application, 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 formally for DEVS 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 the formal specifications presented earlier, 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.

Let us consider the following coupled model representing the airport example of the previous section:

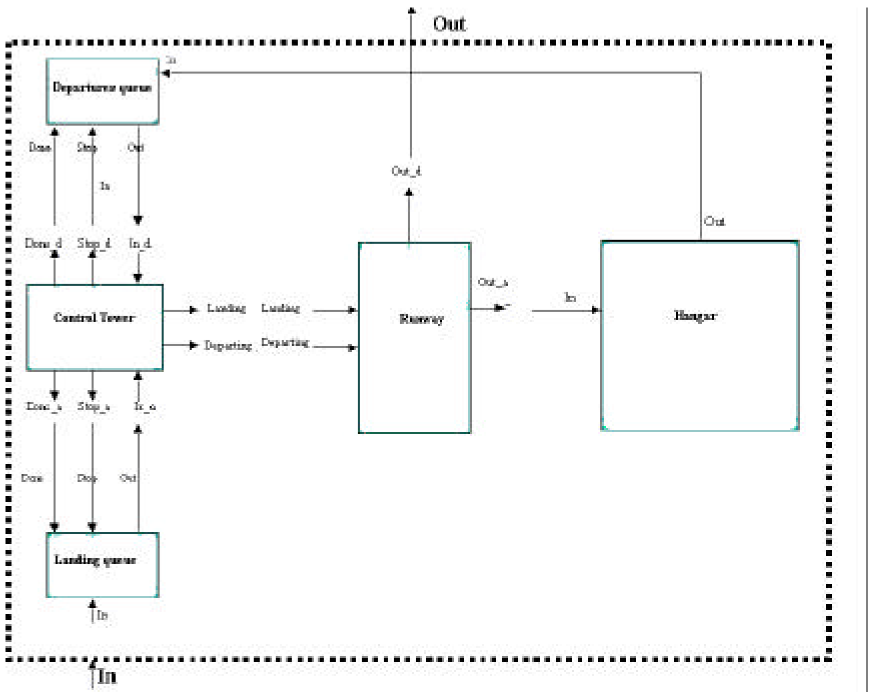


Figure 12. Model interconnection for the Airport coupled model

We can see that 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 runway. Every time a plane is authorized to depart or land, the runway 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 a another atomic model, or as a coupled model with different service stations for the planes. The airport coupled model can be formally specified as:

Airport = < X, Y, D, {M_i}, {I_i}, {Z_{ij}}>

X = { ("In", unsigned int) }; ***Y*** = { ("Out", unsigned int) };

D = { DeparturesQ, Control Tower, RunwayQ, Runway, Hangar };

$\mathbf{I}_{\mathbf{Self}}$ = { RunwayQ };

$\mathbf{I}_{\mathbf{RunwayQ}}$ = { Control Tower};

$\mathbf{I}_{\mathbf{Control\ Tower}}$ = { RunwayQ, DeparturesQ, Runway };

$\mathbf{I}_{\mathbf{Runway}}$ = { Self, Hangar };

$\mathbf{I}_{\mathbf{Hangar}}$ = { DeparturesQ };

The following figure shows the definition of this formal description using the coupling specification language of the tool. In this case, the Hangar also was defined as a coupled model, which is included in the specification.

```
[top]
components: DeparturesQ@StoppableQueue Rnwy-useQ@StoppableQueue Runway@Runway Hangar
components: ControlTower@ControlTower
in : In                out : Out
link : out@DeparturesQ In_d@ControlTower
link : out@RunwayQ In_a@ControlTower
link : In in@RunwayQ
link : Out_a@Runway In@Hangar
link : Out_d@Runway Out
link : Done_a@ControlTower in@RunwayQ
link : Stop_a@ControlTower stop@RunwayQ
link : Departing@ControlTower Departing@Runway
link : Rnwy-use@ControlTower Runway@Runway
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 Chosen@DeMux
components : deposit2@queue
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 13. Coupled Model definition.

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.

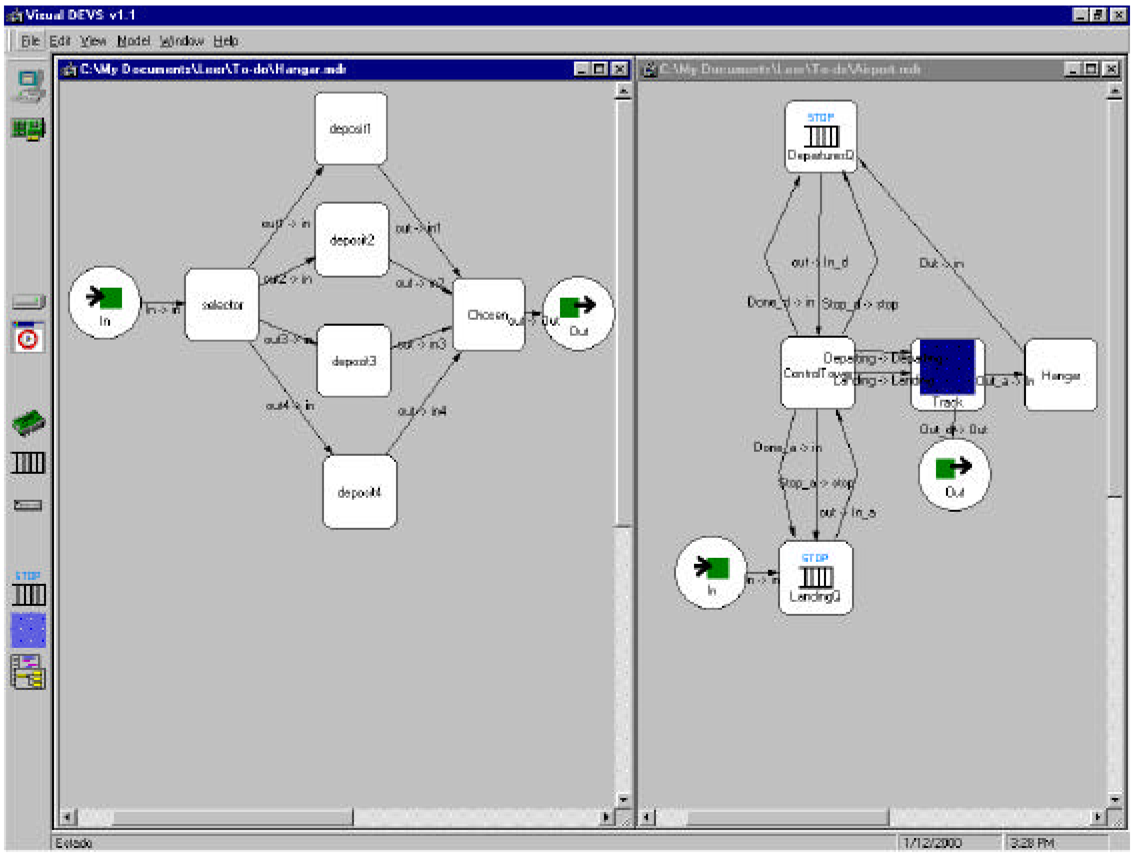


Figure 14. Graphical specification for the Airport coupled model.

Coupled models can be defined using a graphical specification that maps the representation for coupled models explained earlier. In this case, squares epresent submodels (atomic or coupled), and circles the model's input/output ports. Internal links are represented using arrows connecting the components. The previous specification can be graphically defined as is shown in figure 14. The tool translates each graphical construct into the corresponding syntax element (for instance, the previous text specification was generated using the tool). Both ways of specifying coupled models are straightforward and easy to be learned by new users. The modeler does not need to learn a programming language to define new coupled models.

Model reuse is straightforward thanks to the hierarchical construction of the models. In the previous example, for instance, several of the models were defined as instances of others previously created. We reused two kinds of queues: non stoppable and stoppable. The hangar was implemented using a selector and a demultiplexor previously defined for a model of a digital computer, because the model semantics for both cases is similar. The only new models specifically defined for this example are the control tower and the runway (which is also a variation of the Queue model).

The formal specification of DEVS models makes easy the verification of coupled models. First, it is easy to recognize not existing submodels or model using wrong names as coupled models are constructed using the textual specifications, which implement coupled model formal specifications. We have also included verification tools in charge of authenticate ports and their links to atomic DEVS models.

We first create a list of influencees associated to each output port. This list holds all the input ports linked to the current output port, and it can be analyzed to find if there is any unlinked port. First, we check every output port, analyzing their influencee lists for every model. An empty list means that the output port is not linked to any input port in the simulation. In addition, a global influencee's list is built using with all the input ports that are linked to any output port. This list must contain all the input ports defined in the coupled model. If a port does not belong to it, no output port is linked to it.

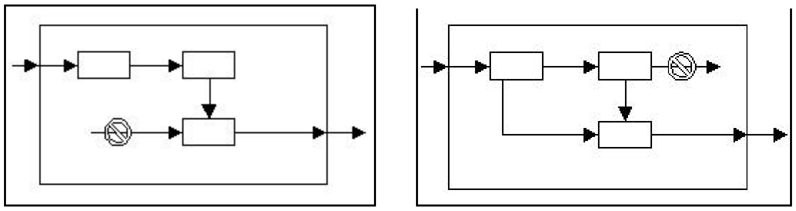


Figure 15. Unlinked input/output ports.

In both cases, CD++ raises an error message, enabling the analysis of the problem. In some cases, the modeler does not want to use an existing port, but in others, this can result in undesirable errors complex to be discovered. No other couplings are verified, as this is not needed. We have followed the DEVS specifications for couplings, and the tool use this constructive approach. Therefore, we know that the links built from the specifications are correct. The tool includes type validation, ensuring compatibility of the data shared through input/output ports.

A coupled model that was verified using the services of the tool represents a switching station for mobile phones.

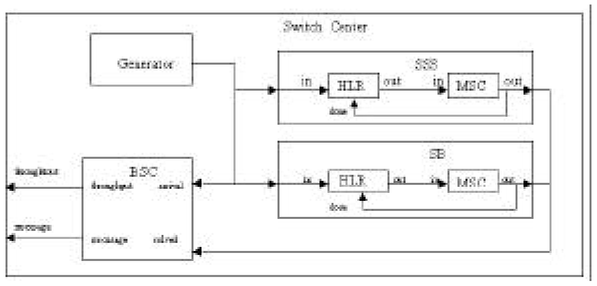


Figure 16. Switching station coupled model.

HLR model is in charge of managing the call flow. When it receives a request, queues it, and advances the switching time associated to the call. When the call is sent, it receives an ACK signal. When the coupled model was verified, we found the following errors:

```
CD++
----------------------------------------------
Version 2.0-R.43

Starting simulation. Stop at time: Infinity

Exception thrown!
Model Name: HLR1
Input Port Name: stop, is not influenced.
Description: Input Ports without Influences!

Model Name: HLR2
Input Port Name: stop, is not influenced.
Description: Input Ports without Influences!

Aborting simulation...
```

Figure 17. Error detected: Unlinked ports.

In this case, the ***HLR*** class includes an input port named ***stop***, therefore the user is informed about this situation enabling to solve the problem, if needed.

Cell-DEVS models definition

The tool includes a specification language allowing the description of Cell-DEVS models. These definitions are based on the formal specifications defined earlier, and can be completed by considering a few parameters: size, influencees, neighborhood and borders. These are used to generate the complete cell space. The behavior of the local computing function is defined using a set of rules with the form: VALUE DELAY { CONDITION }. These indicate that when the *CONDITION* is satisfied, the state of the cell changes to the designated *VALUE*, and it is *DELAY*ed for the specified time. If the condition is *false*, the next rule in the list is evaluated until a rule is satisfied or there are no more rules. In the latter case, an error is raised, indicating that the model specification is incomplete. The existence of two or more rules with same condition but with different state value or delay is also detected, avoiding the creation of ambiguous models. In these situations, the simulation is aborted.

The following figure shows the rules for the "Life" Game [18]. This model represents a cell space with entities that evolve according to the neighbors' states. A cell remains active when the number of active neighbors is 3 or 4 (*truecount* indicates the number of active cells) using a transport delay of 10 ms.. If the cell is inactive and the neighborhood has 3 active cells, the cell activ ates. In every other case, the cell remains inactive (*t* indicates that whenever the rule is evaluated, a *True* value is returned).

```
Rule: 1 10 { (0,0) = 1 and ( truecount = 3 or truecount = 4 ) }
Rule: 1 10 { (0,0) = 0 and truecount = 3 }
Rule: 0 10 { t }
```

Figure 18. Rule definition for the Life game.

Several useful operations are included: boolean (*AND*, *OR*, *NOT*, *XOR*, *IMP* and *EQV*), comparison (=, !=, <, >, <= and >=), and arithmetic (+, -, * and /). In addition, different types of fu nctions are available: trigonometric, roots, power, rounding and truncation, module, logarithm, absolute value, minimum, maximum, G.C.D. and L.C.M. Other existing functions allow to check if a number is integer, even, odd, or prime. Some functions allow to query the cell state of the neighborhood: *truecount*, *falsecount*, *undefcount*, and *statecount*(*n*).

The *Time* function returns the model simulated time. Functions *RadToDeg* and *DegToRad* are used for angle conversion. There are also conversion for polar and rectangular coordinates and temperatures in Celsius, Fahrenheit or Kelvin degrees. Other functions allow evaluating a certain condition. The function *IF*(*c*, *t*, *f*) returns *t* if *c* evaluates to *True*, and *f* otherwise. On the other hand, *IFU*(*c*, *t*, *f*, *u*) evaluates *c*, and if it is *True*, it returns the *t* value. If it is *False*, it return *f*, and *u* if it is *undefined*.

Some common constants are predefined: *pi*, *e*, gravitational constant, light speed, Planck constant, etc. Finally, pseudorandom numbers generation is inclu ded. Different probability distributions are considered, including Uniform, Chi Square, Beta, Exponential, Φ, Gamma, Gaussian, Binomial and Poisson.

The cells can use integer or real values. Undefined values can be used, allowing to apply three-valued logic. The undefined value is also useful with other purposes. For example, in the rule:

10 100 { random >= 0.4 }

the condition is evaluated to *True* in some cases, and to *False* in others. Therefore, a model using this rule could return all the rules evaluated as *False*, which occurs when a model is incomplete. Nevertheless, in this case the model is well specified but the use of a random number produces this result for the present state. The tool automatically identifies these cases, and assigns the *Undefined* value to the cell, informing this situation and continuing with the simulation.

The language permits to manipulate n-dimensional references. Likewise, a neighborhood can be composed of non-adjacent cells, and the neighborhood's dimension can be similar or inferior to the dimension of the model. N-dimensional space *zones*, defined by a cell range, can be associated with a set of rules different from the rest of the cell space.

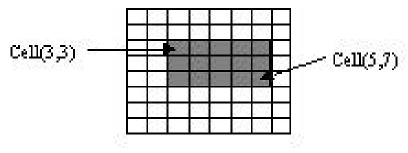


Figure 19. A Zone defined by the range {(3,3)..(5,7)}

Several applications of the formalism can be found in [12, 19, 20]. We show the definition of a well-known model for fire propagation in forest fires [21]. This model uses environmental and vegetation conditions to infer the fire spread ratio and the intensity. Three parameter groups determine the fire spread ratio: vegetation type, fuel properties (vegetation classified according to its size), and environmental parameters (wind speed, fuel humidity and field slope).

```
Wind direction = 45.000000 (bearing)   Wind speed = 8.045000 [kph]
NFFL model = 1                          Cell Width = 15.240000 [m] (E-W)
Cell Height  = 15.240000 [m] (N-S)
Max. Spread = 17.967136 [mpm]( in the direction of the wind)
0° Spread = 5.106976 [mpm]              Distance = 15.240000 [m]
45° Spread = 17.967136 [mpm]            Distance = 21.552615 [m]
90° Spread = 5.106976 [mpm]             Distance = 15.240000 [m]
135° Spread = 1.872060 [mpm]            Distance = 21.552615 [m]
180° Spread = 1.146091 [mpm]            Distance = 15.240000 [m]
225° Spread = 0.987474 [mpm]            Distance = 21.552615 [m]
270° Spread = 1.146091 [mpm]            Distance = 15.240000 [m]
315° Spread = 1.872060 [mpm]            Distance = 21.552615 [m]
```

Figure 20. Parameter definition for a Rothermel fire model.

The fuel model, the speed and direction of the wind, terrain topology and dimensions of a cell are used to get the spread ratio in each direction. For instance, the previous figure shows the values obtained for a fuel model group number 9, a SE wind of 24.135 km per hour and a cell size of 15.24x15.24 meters.

In CD++, this model can be defined as a 20 by 20 cellular model representing the terrain and vegetation. A cell value of 0 indicates the absence of fire and other values indicate the time the fire has started on that cell. We use a non-wrapped border, a 3x3 neighborhood, and inertial delays.

```
[ForestFire]
type : cell                                 dim : (20,20)
delay : inertial                            border : nowrapped
neighbors : (-1,-1) (-1,0) (-1,1) (0,-1)  (0,0)  (0,1) (1,-1)  (1,0)  (1,1)
localtransition : FireBehavior

[FireBehavior]
rule : {(1,-1)+(21.552615/17.967136)} {(21.552615/17.967136)*60000} {(0,0)=0 and 0<(1,-1)}
rule : {(1,0)+(15.24/5.106976)} {(15.24/5.106976)*60000} {(0,0)=0 and 0<(1,0)}
rule : {(0,-1)+(15.24/5.106976)} {(15.24/5.106976)*60000} {(0,0)=0 and 0<(0,-1)}
rule : {(-1,-1)+(21.552615/1.872060)} {(21.552615/1.872060)*60000} {(0,0)=0  and 0<(-1,-1)}
rule : {(1,1)+(21.552615/1.872060)} {(21.552615/1.872060)*60000} {(0,0)=0 and 0<(1,1)}
rule : {(-1,0)+(15.24/1.146091)} {(15.24/1.146091)*60000} {(0,0)=0 and 0<(-1,0)}
rule : {(0,1)+(15.24/1.146091)} {(15.24/1.146091)*60000} {(0,0)=0 and 0<(0,1)}
rule : {(-1,1)+(21.552615/0.987474)} {(21.552615/0.987474)*60000} {(0,0)=0 and 0<(-1,1)}
rule : {(0,0)} 0 { t }
```

Figure 21. Definition of a fire forest model.

The local computing function is called *FireBehavior*, and the rules are devoted to detect the presence of fire in the eight neighboring cells. If there is fire in one of the them, the cell can burn. For instance, the first rule checks if the current cell is not burning (*(0,0) = 0*) or the SW neighbor has started to burn (*0 < (1,-1)*). If this condition is met, the value for the cell is set to the result of the expression (1,-1) + (21.552615/17.967136), which is the time the fire starts in the cell, using the values previously computed. As the spread ratio for the fire in the NE direction is 17.967136 meters/min (mpm) and a cell has a diagonal of 21.552615 meters, it takes (21.552615/17.967136) minutes for the fire to reach the a cell once it has started in its SW neighbor. The delay component of the rule says this value is set after (21.552615/17.967136) * 60000 milliseconds. The remaining rules represent the behavior associated to the remaining neighbors.

We could use inertial delays to model other behaviors. For instance, if rain is considered, wet cells do not burn and if any of the cells scheduled to start burning gets wet before the fire starts, it do not burn either. Inertial delays preempt any scheduled change if it receives an event from a neighbor cell before the scheduled time, causing the present state to acquire a different value .

As we can see, the burn time for each cell depends on the spread ratio in the direction of the burning cell. It is important to notice that the cells are updated at different times, as set by a rule's delay component. This is a clear departure from the classical approach to cellular automata where all active cells are updated at the same time. A non-burning cell in the direction of the fire spread is updated in a shorter period than a non-burning cell in the opposite direction. Another advantage is that expressing a timing delay is done in a natural fashion (compared, for instance, with the timing control defined in the airport model). In this way, the modeler can reduce the development time related with timing control programming.

We also have included verification facilities that can be applied when executing Cell-DEVS models. The simplest ones include checks on the number of cells in each dimension, initialization of each of the cells, position of the border cells and zones and positions of the cells representing the *Xlist* and *Ylist*. The most complex verification aids are related with the definition of the local computing function rules. These aids allow to detect some inconsistencies in the model definition:

- Ambiguous models: a cell with the same precondition (state and neighbors) can produce different results;
- Incomplete models: no result exists for a certain precondition;
- Non-deterministic models: different preconditions are valid simultaneously. If they produce the same result, the simulation can continue, but the modeler must be notified. Instead, if different results are found, the simulation should stop because the future state of the cell cannot be determined.

For instance, the following figure shows a modification of the Life game model (using 0, 1 or 2 as cell values), in which the rules are not completely defined. Here, we can find cases in which all the preconditions are False (i.e., if the cell being evaluated has a value of 2).

```
[new-life-rule]
rule : 2 100 { (0,0) = 1 }
rule : 1 100 { (0,0) = 0 }
```

Figure 22. Redefinition of the Life game.

In these situations, an error is raised, as shown in the following figure. The message describes the event occurred, and shows the state values for the neighboring cells.

```
CD++
-----------------------------------------------
Version 2.0-R.43

Starting simulation. Stop at time: 00:00:05:000

Exception thrown!
Description: None of the rules evaluates TRUE!
  Model used is: new-life-rule
  The state of the Neighbors is:
  +-----------------------------------------+
  |      0.00000       1.00000       2.00000 |
  |      1.00000       2.00000       1.00000 |
  |      0.00000       1.00000       2.00000 |
  +-----------------------------------------+
Aborting simulation...
```

Figure 23. Error detected: no valid rule.

The error describes that the rules are not complete (in the absence of this verification, the simulation would crash, as there are no rules to be executed). In this case, the origin cell has a value of 2, and there is no rule whose precondition is valid for this case.

The tool includes rule definition using three-valued logic. Therefore, when a set of rules is being defined, the values *True*, *False*, or *Undefined* can be obtained. If any of the rules results in an *Undefined* value, the cell state is *Undefined*. In this case, a warning is issued. For instance, the following figure shows a redefinition of the previous example.

```
[new-life-rule2]
rule : 2 100 { (0,0) = 1 and (0,1) = ? }
rule : 1 100 { (0,0) = 0 }
rule : 0 100 { (0,0) = 2 }
```

Figure 24. Life game with Undefined values.

When the state value for the cell is 1, and the neighbor (0,1) is not *Undefined* (?), the first rule results in an *Undefined* state. In that case, when we evaluate `(0,1) = ?`, the result is ?, and the result of the AND operation is *Undefined*. When the rest of the rules are evaluated, no valid precondition is found. In this case, the value of the cell is set to Undefined, and the following warning message is issued:

```
CD++
----------------------------------------------
Version 2.0-R.43

Starting simulation. Stop at time: 00:00:05:000

Warning! – None of the rules evaluate to True, but any evaluates to undefined
...
```

Figure 25. Warning: undefined state for a cell.

If there are two or more rules whose condition evaluate to True, and their postconditions or delays are different, an error is raised. In this case, the model is ambiguous, and the simulation is aborted, avoiding the execution of models running in a non-deterministic fashion. In the following figure, we show a set of ambiguous rules for the Life game.

```
[new-life-rule3]
rule : 2 100 { (0,0) = 1 }
rule : 1 100 { (0,0) = 0 or (0,0) = 1 }
rule : 0 100 { (0,0) = 2 }
```

Figure 26. Life game with ambiguous rules.

In this case, if a cell has a value of 1, the first and second rules are valid, but the results are different. The following figure shows the execution when these rules are evaluated. Instead, when two different rules are valid, but their results are the same, a warning is raised, but the simulation continues. In this case, although two rules are valid simultaneously, the simulation results would be the same if any of them is executed. The warning enables the modeler to check possible ambiguities.

```
CD++
----------------------------------------------
Version 2.0-R.43

Starting simulation. Stop at time: 00:00:05:000

Exception thrown!
```

```
Description: Two rules evaluate to TRUE and the result is different!
  Model used is: new-life-rule3
  The state of the Neighbors is:
+----------------------------------------+
|      1.00000      0.00000      0.00000 |
|      1.00000      0.00000      1.00000 |
|      1.00000      0.00000      0.00000 |
+----------------------------------------+
Aborting simulation...
```

Figure 27. Error detected: ambiguous rules.

There are a few special cases to consider: if a stochastic model is used, it might happen either that multiple rules are be valid or that none of them is. For the first case, the first valid rule is considered. For the second case, the cell have an undefined value (?), and the delay time is the default d elay time specified for the model. In any case, the simulator notifies this situation to the user, showing a warning message on standard output, but the simulation is not aborted.

As we can see, Cell-DEVS formal specifications were directly mapped into CD++. These specifications allowed us to define formal verification mechanisms, and to prove properties of the model. Likewise, they make easier the definition of a cellular model. These models can be defined using very simple rules and a few parameters. The delay functions enable simple definition of timing relations. Therefore, a very simple set of procedures is needed to define complex models, thus improving the development process. The following section shows other extensions that make easier the definition of complex models (including n-dimensional models or alternative topologies for the meshes). Important improvements in the development times can be obtained, moreover when expert users of the tool are compared with experienced programmers in other environments.

N-dimensional models definition

Cell-shaped real systems are usually modeled by using two-dimensional representations. Nevertheless, several theoretical problems can be defined as cellular models with three or more dimensions. Therefore, provisions for the inclusion of n-dimensional models were included in the tool. Two -dimensional models are stored in an array of $t_1 . t_2$ cells, where the element (x_1, x_2), $x_i \in [0, t_i -1]$, is in the position $x_1 + x_2 . t_1$. Likewise, we use an array of $\prod_{i=1...n} t_i$ cells to store the states for the cellular automata with dimension $(t_1, t_2, ..., t_n)$. In this case $(x_1, x_2, ..., x_n)$ occupies the position $\sum_{i=1...n} x_i . (\prod_{k=1...i-1} t_k)$.

We show the use of multidimensional models presenting a simple variation of the '*Life*' game. We consider a population of active cells represented by '1' values, distributed in an area of 7x7x3 cells. The neighborhood has 3x3x3 cells. Inactive cells have a '0' value. An inactive cell activates when it has over 10 living neighbors. Besides, a cell remains active when the neighborhood contains 8 or 10 active neighbors. Otherwise, the cell is deactivated. The following figure shows a description for this model using the cell description language:

```
[3dl]
type : cell                          dim : (7,7,3)
delay : transport                    border : wrapped
neighbors: (-1,-1,-1)   (-1,0,-1) (-1,1,-1) (0,-1,-1) (0,0,-1) (0,1,-1)
neighbors: (1,-1,-1)    (1,0,-1)  (1,1,-1)  (-1,-1,0) (-1,0,0) (-1,1,0) (0,1,1)
neighbors: (0,-1,0)     (0,0,0)   (0,1,0)   (1,-1,0)  (1,0,0)  (1,1,0)  (0,0,1)
neighbors: (-1,-1,1)    (-1,0,1)  (-1,1,1)  (0,-1,1)  (1,-1,1) (1,0,1)  (1,1,1)
```

```
localtransition: 3dl-rule

[3dl-rule]
rule : 1 100 { (0,0,0) = 1 and (truecount = 8 or truecount = 10) }
rule : 1 100 { (0,0,0) = 0 and truecount >= 10 }
rule : 0 100 { t }
```

Figure 28. Definition of the 3D Life Game.

The first lines define the dimension parameters of the cell space. The kind of delay and the shape of the neighborhood are also included. Finally, we include the local computing function. Models of higher dimensions are defined in the same way.

Coupling DEVS and Cell-DEVS models

Cellular models can be coupled with other DEVS and be integrated to the model hierarchy using a standard interface. The *portInTransition* directive is used by a cell to query the value of a message arrived in an input port different of those connected to the neighbors. On the other hand, output ports are activated using the *send* function.

The following figure [12] presents a simple cellular model to show the use of these directives. It is a 2x2 Cell-DEVS receiving inputs in the cell (1,1). It also uses the values of cell (1,1) to send outputs to other models using two output ports.

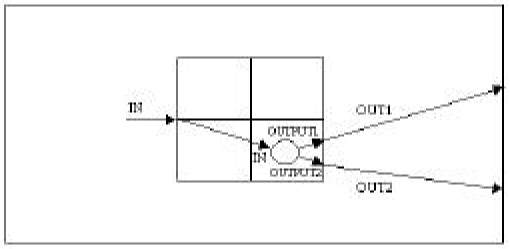


Figure 29. Coupling scheme of the previous example.

```
[EX]
type : cell                            dim = (2, 2)
delay : transport                      border : wrapped
neighbors : (-1,-1) (-1,0) (-1,1) (0,-1)  (0,0)  (0,1) (1,-1)  (1,0)  (1,1)
in : in
out : out1 out2
link : in in@(1,1)
link : output1@l(1,1) out1
link : output2@(1,1) out2
portInTransition : in@(1,1)  specialRule
localtransition : nothing-rule
zone : generateOut { (1,1) }

[nothing-rule]
rule : { (0,0) } 1 { t }

[specialRule]
rule: { portValue(thisPort) } 1 { t }

[generateOut]
rule:{(0,0)+send(output1,9.9999)} 1 {(0,0)>=10}
rule :{(0,0)+send(output2,3.3333)} 1 {(0,0)<10}
```

Figure 30. Example of use of the new features.

First, we define the Cell-DEVS coupled model values. In this case, it is a 2x2 cell space with transport delays. The borders are wrapped, and all the adjacent cells conform the neighborhood. We use the *Link* directive to define the internal and external coupling. In this case, the coupled model input port "*in*" is connected to the "*in*" port in the cell (1,1). Then we can see that two output ports are defined for cell (1,1) (output1 and output2). They are linked to the coupled model output ports. After, the *portInTransition* directive allows to define the name of the function to be used when a message arrives through the specified port. In this case, the *specialRule* is executed when an input arrives to the *in* port in cell (1,1). When this rule is activated, the function *portValue*(*x*) gets the value of the last message arrived through the port *x* of the cell (*thisPort* returns the name of the port where the message has arrived).

The local computing function, called *nothingRule* just keeps the present value of the cell. The cell (1,1) is also defined as a special zone (*generateOut*) with different behavior. In this case, when the cell is activated it checks the present cell value. If it is smaller than 10, the value 3.3333 is sent through the port output2. Otherwise, the value 9.9999 is sent through the port output1.

Cell-DEVS Lattice Translator

Most existing cellular models use grids with *square* geometry. They are simple to define and visualize, but have poor *isotropy* capacities (the capacity of a phenomenon to be expanded in every direction in an homogeneous fashion). Certain phenomena cannot even be modeled accurately if square grids are used. The provision of other topologies can solve these problems. For instance, *triangular* grids use a reduced number of neighbors and reduce the number of states to be evaluated. *Hexagonal* cells have the highest isotropy, making the models more natural to be developed. The difficulties with any of these geometry are related with the model visualization.

CD++ was developed considering square geometry. Therefore, to include other geometrical shapes it was necessary to extend the original definitions. In [22], some mappings between different geometry have been proposed. Some of these functions are easy to implement, but the visualization of results is difficult. Neighborhood definition is also complex. The following figure shows a possible mapping of hexagonal meshes in to square lattices.

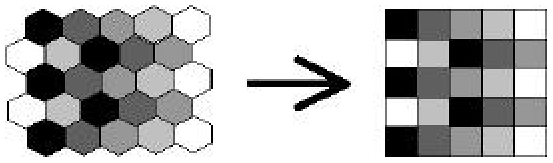


Figure 31. Mapping a hexagonal geometry in a square lattice.

This transformation is carried out analyzing the parity of the row where a cell is located. If we consider that a cell is in the position (x,y), where x is the row and y the column, the position is translated as shown in the following figure.

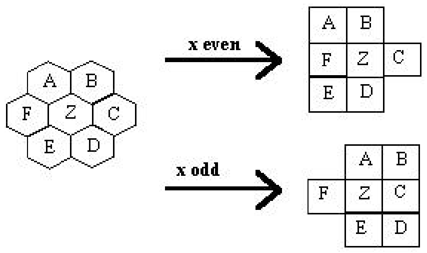


Figure 32. Distribution of close neighbors in the mapping.

The mapping results in a correspondence that differs according to the original row position. Therefore, the translated cellular model is inhomogeneous. The case for triangular meshes is similar. Here, each cell also is provided with a different orientation than those of the neighboring cells. These ideas can be seen in the following figure.

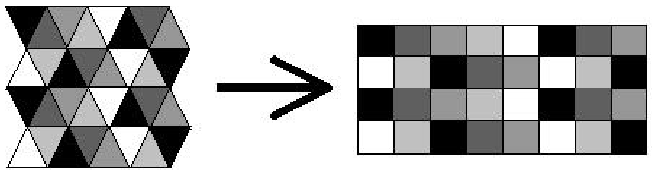


Figure 33. Mapping a hexagonal geometry in a square lattice.

Using this function, each row of triangles is mapped into a row of squares, providing a non-homogeneous set of rules. In this case, the position depends on the parity of x+y, that is, the parity of adding row and column positions, as shown in the following figure.

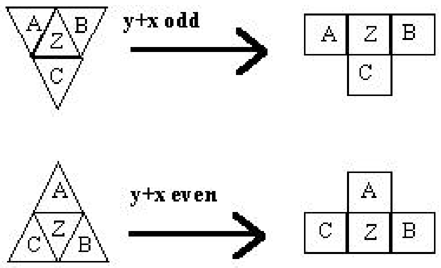


Figure 34. Distribution of close neighbors in the mapping.

These procedures were used to build a lattice translator. The translator allows the user to define rules using triangular or hexagonal meshes. Then, it translates them to the original syntax using square rules that can be executed by CD++. The specification language was extended to provide a different way of referencing the neighboring cells. The new specification syntax is depicted in the following figure.

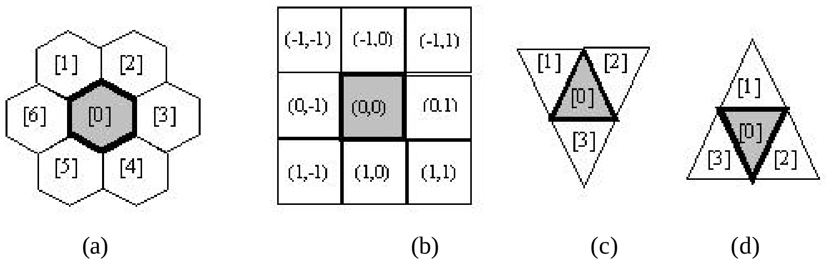


Figure 35. Relative position of the neighbors in the different geometry.

Each rule in a hexagonal or triangular lattice is translated into two different square rules, each of them considering the parity of rows and columns, as explained earlier. The translation procedures must know which row we are translating. The *cellpos* function returns the value of the i-eth position of a tuple referenced by the cell where the rule is being executed. The following figure shows the translation of a simple rule into a hexagonal geometry.

```
Rule: 1 10 {[3]=1 and statecount(9.99)=3}

Rule: 1 10 {(((0,1)=1) and (if((statecount(9.99)-(if((-1,1)=9.99,1,0))-
(if((1,1)=9.99,1,0)))<0,0,(statecount(9.99)-(if((-1,1)=9.99,1,0))-(if((1,1)=9.99,1,0))))=3)) and
even(cellpos(0))}

Rule: 1 10 {(((0,1)=1) and (if((statecount(9.99)-(if((-1,-1)=9.99,1,0))-(if((1,-
1)=9.99,1,0)))<0,0,(statecount(9.99)-(if((-1,-1)=9.99,1,0))-(if((1,-1)=9.99,1,0))))=3)) and
odd(cellpos(0))}
```

Figure 36. Example of a rule in an hexagonal geometry with the corresponding translation.

This example shows a rule saying that, if the value in the neighbor called [3] is 1, and there are three cells in the neighborhood whose value is 9.99, the cell has a future value of 1. This value is sent to the neighboring cells using a transport delay after 10 time units. The following two rules represent the translation of that rule into a square mesh, using the definition showed in the previous figures.

In a square lattice, we use 8 neighbors, against 6 used in the hexagonal mesh. Therefore, when we translate the rules from one topology to the other, we should ask not to count those neighbors that should not be included. The function *statecount(n)*, which returns the number of cells whose value is *n*, is used with this purpose. We can see that we compute *statecount(9.99)* like in the previous case, and then, in the even rows, we subtract the values of cells ((-1,1) and (1,1)). The following rule is evaluated only in odd rows. The function *cellpos(n)*, which returns the n-eth value of the tuple referencing a cell is used to check the cell row used.

The following figure shows the result obtained when the same procedure is applied to a triangular geometry.

```
Rule: 1 10 {[3]=1 and statecount(9.99)=3}

Rule: 1 10 {(((0,-1)=1) and (if((statecount(9.99)-(if((-1,-1)=9.99,1,0))-(if((-1,0)=9.99,1,0))-
(if((-1,1)=9.99,1,0))-(if((1,-1)=9.99,1,0))-(if((1,1)=9.99,1,0)))<0,0,(statecount(9.99)-(if((-1,-
1)=9.99,1,0))-(if((-1,0)=9.99,1,0))-(if((-1,1)=9.99,1,0))-(if((1,-1)=9.99,1,0))-
(if((1,1)=9.99,1,0))))=3)) and even(cellpos(0)+cellpos(1))}

rule: 1 10 {(((1,0)=1) and (if((statecount(9.99)-(if((-1,-1)=9.99,1,0))-(if((-1,1)=9.99,1,0))-
(if((1,-1)=9.99,1,0))-(if((1,0)=9.99,1,0))-(if((1,1)=9.99,1,0)))<0,0,(statecount(9.99)-(if((-1,-
1)=9.99,1,0))-(if((-1,1)=9.99,1,0))-(if((1,-1)=9.99,1,0))-(if((1,0)=9.99,1,0))-
(if((1,1)=9.99,1,0))))=3)) and odd(cellpos(0) + cellpos(1))}
```

Figure 37. Example of a rule in an triangular geometry with the corresponding translation.

The main difference in this rule is that we have more neighbors. Therefore, we have to control the values corresponding to even and odd rows/columns.

SIMULATION MODELS

As we could see in the previous sections, a main advantage of the DEVS formalism is that the models can be specified independently of the simulation mechanism. At present, we have implemented a number of different simulation techniques, namely: Hierarchical [12], Flat [25], Real-Time [26] and Parallel [27]. In every case, the model specifications remained untouched, and they could be reused, even after changing a simulation technique completely. CD++ was defined using the abstract simulation mechanisms presented in [23]. 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.

The basic idea is to provide to independent basic abstract classes: *Models* and *Processors*. The first is used to represent the behavior of the atomic and coupled models, and they were presented in the previous section. The second implements the simulation mechanisms associated with the models. The following figure shows a detailed class hierarchy including all the modeling and simulation entities.

Each modeling entity is associated with a Processor that is in charge to activate the model, which is in charge to define the behavior to be implemented. Each kind of model is associated with a different processor. *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 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.

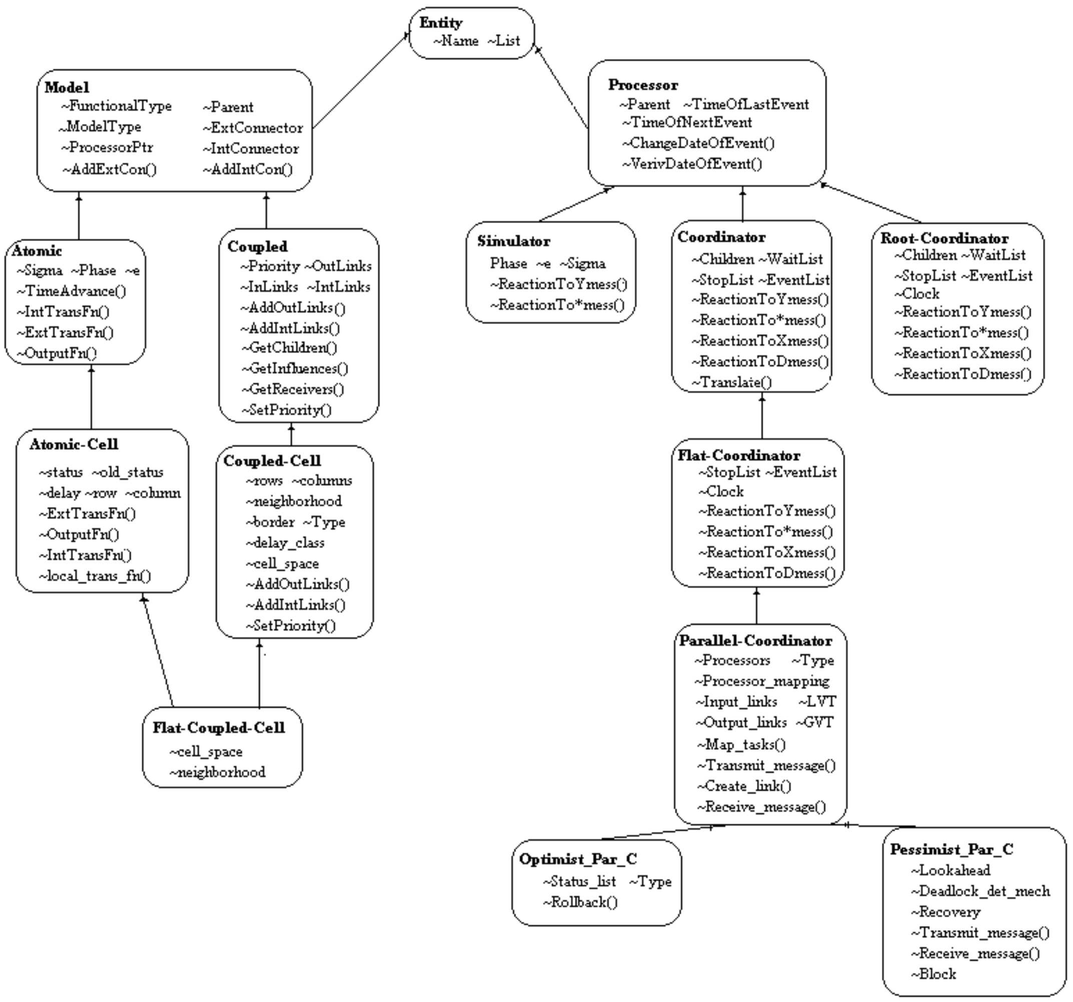

Entity
~Name ~List
Model
~FunctionalType ~ModelType ~ProcessorPtr ~AddExtCon()
~Parent ~ExtConnector ~IntConnector ~AddIntCon()
Processor
~Parent ~TimeOfLastEvent ~TimeOfNextEvent ~ChangeDateOfEvent() ~VerivDateOfEvent()
Atomic
~Sigma ~Phase ~e ~TimeAdvance() ~IntTransFn() ~ExtTransFn() ~OutputFn()
Coupled
~Priority ~OutLinks ~InLinks ~IntLinks ~AddOutLinks() ~AddIntLinks() ~GetChildren() ~GetInfluences() ~GetReceivers() ~SetPriority()
Simulator
Phase ~e ~Sigma ~ReactionToYmess() ~ReactionTo*mess()
Coordinator
~Children ~WaitList ~StopList ~EventList ~ReactionToYmess() ~ReactionTo*mess() ~ReactionToXmess() ~ReactionToDmess() ~Translate()
Root-Coordinator
~Children ~WaitList ~StopList ~EventList ~Clock ~ReactionToYmess() ~ReactionTo*mess() ~ReactionToXmess() ~ReactionToDmess()
Atomic-Cell
~status ~old_status ~delay ~row ~column ~ExtTransFn() ~OutputFn() ~IntTransFn() ~local_trans_fn()
Coupled-Cell
~rows ~columns ~neighborhood ~border ~Type ~delay_class ~cell_space ~AddOutLinks() ~AddIntLinks() ~SetPriority()
Flat-Coordinator
~StopList ~EventList ~Clock ~ReactionToYmess() ~ReactionTo*mess() ~ReactionToXmess() ~ReactionToDmess()
Parallel-Coordinator
~Processors ~Type ~Processor_mapping ~Input_links ~LVT ~Output_links ~GVT ~Map_tasks() ~Transmit_message() ~Create_link() ~Receive_message()
Flat-Coupled-Cell
~cell_space ~neighborhood
Optimist_Par_C
~Status_list ~Type ~Rollback()
Pessimist_Par_C
~Lookahead ~Deadlock_det_mech ~Recovery ~Transmit_message() ~Receive_message() ~Block

Figure 38. Basic class hierarchy.

Hierarchical Processors

As previously explained, different simulation Processors are used: Simulators, Coordinators, and Root-Coordinator. They are related with different models: Simulators are associated with Atomic models, and Coordinators with Coupled ones.

The coupling relationship is recorded in the instance variables *devs-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 next event are recorded to identify the imminent children, and to verify correctness in the message's simulated times.

The simulation process is driven by message passing between the Processors. The messages include information related with the message's origin, the time of the related event, and a content consisting of a port and a value. There are four types of 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 that a model has finished with its task).

The submodel with the smaller scheduled internal event in the hierarchy is called the **imminent**. When this model must execute, a *-message is sent to its simulator. The message passes through the middle level coordinators, provided with a list of imminent children with this purpose. 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. Y-messages carry the results to be transmitted to other models. Done-messages are used to inform that each model has finished with the task given by its higher level coordinator.

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 rescheduled. X-messages are also transmitted to the lower levels in the hierarchy using the coupling scheme, up to the moment where it arrives to an atomic model.

The abstract simulator analyses the external events and the scheduled internal transitions for all its children, and chooses the imminent child. These procedures are repeated up to the moment when a simulator is chosen. This is the simulator related with the imminent model, which must be activated. The imminent model starts by executing the output function λ and to generate an output event $y \in Y$. Each output is sent to the parent coordinator, which checks the influencees and translates it into new inputs, using the Z_{ij} translation function. After, the internal transition function δ_{int} executes, resulting in a state change and the scheduling of a new internal transition. Every activation of a model finishes with a done message used to update the future activation time of the model. The done message with the earlier future time is transmitted to the upper level coordinator, allowing to maintain the local time of the particular coupled models.

The *SimLoader* class is in charge parsing the model specification, providing an interface to load the simulator configuration. Once the models are loaded, there are two possible procedures to start the simulation. The first one is by using the class *StandAloneLoader*, responsible to load the parameters. 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. Finally, the *Simulator* class is responsible of the creation of the model tree and to establish the links between ports using the specification. Once the hierarchy of the model is built, the simulation can begin. To do so, the external events are added, an event list created and the stop time initialized.

Flat Processors

The overhead that results from the exchange of messages between processors could be minimized if the hierarchy is properly flattened. Therefore, the number of messages can be reduced accordingly [Kim et al., 2000]. It is important to conserve the usual model definition, execution, and the separation between *models* and *processors*. We extended the flat simulators to introduce flat simulation in CD++. This new processor is unique all across the processors' hierarchy and replaces all the usual coordinators and simulators existing in a hierarchical approach.

The processor hierarchy corresponding to a hierarchical simulator is shown in the following figure. Whenever the *root coordinator* has to schedule an event to lowermost simulators (*Simulators #4* and *#5*), the overhead incurred by message passing can be considerable. The same phenomenon is produced if the *Simulator #5* sends an output through a port connected to *Simulator #3*. The number of intermediate coordinators can be arbitrarily high depending on the studied model.

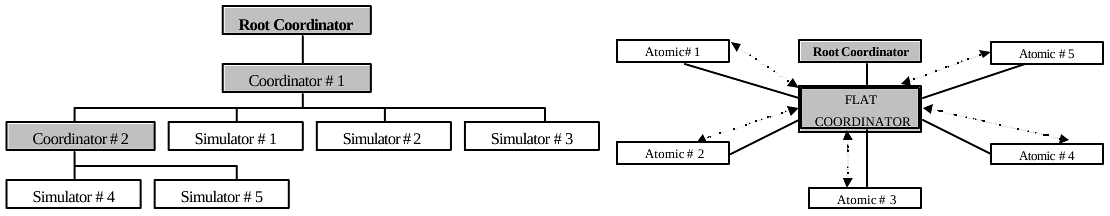


Figure 39. Processors' hierarchy (hierarchical and flat approaches)

If we simulate the model shown using a flat simulator, the resulting hierarchy is remarkably simplified, and the overhead incurred by message passing is significantly reduced. The flat coordinator stores information concerning the atomic models he handles. Information about ports, links, time of next event, time of the last event processed as well as the queue of pending events must be saved.

Real-time Processors

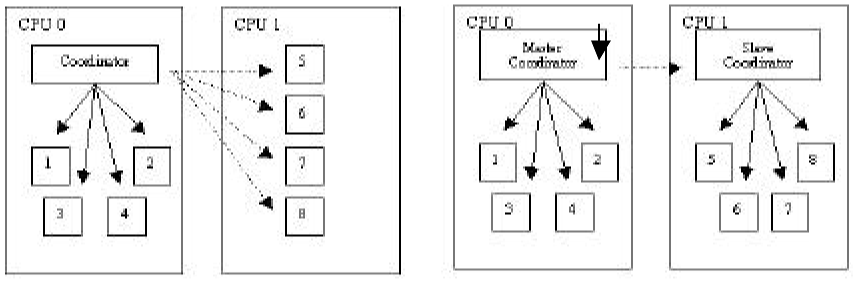
We developed several modifications to the original simulation engines in order to provide real-time responses. A real-time simulator must handle events in a timely fashion, where time constraints can be stated and validated. These new features would allow interaction between the simulator and the surrounding environment. Therefore, inputs could be received by ports connected to real input or even data collected from human interaction. Similarly, outputs could be sent through output ports connected to specialized devices.

In this case, the *root coordinator* manages the advance of time along the simulation. This coordinator must wait until the physical time reaches the next event time to initiate the new cycle. Periods of inactivity are not skipped; instead, the *root-coordinator* expects the scheduled time to be reached and only then starts the new simulation cycle.

Parallel Processors

When a simulation is run in distributed fashion, each machine run one logical process which hosts one or more DEVS processors. Under these assumptions, a coordinator's children need not be executing on the same logical process. If the correspondence between models and DEVS processors is one to one, then every coupled model is associated to only one coordinator. Then every message sent to child processors running on a different CPU requires inter-process communication. The following figure illustrates this case. It shows a coordinator sending a message to its 8 children distributed on two CPUs. Four inter-process messages are required for the four children running on processor 1.

To reduce inter-process messages, coupled models use a coordinator on each logical process where a child processor is running. Children processors send messages to the local coordinator, which decides how to handle the received messages. Upon receiving a message from a child, a coordinator could forward this message to all the coordinators for the model. This organization would require all coordinators to know about each other. For instance, if coupled model *A* is a child of coupled model *B*, then *B´s* coordinators would have to interact with *A*´s coordinators. If handled not carefully, this communication can turn out producing a big number of inter-process messages. In such a scenario, a way of keeping the number of inter-process messages to a minimum is to have only one of the coordinators to receive messages from or route messages to the parent model coordinator. This specialized coordinator is known as a **master coordinator,** and all other coordinators are **slaves**. The master coordinator for model *A* is the only one that can receive or send messages to *B*´s local coordinator.



(a) (b)

Figure 40. (a) Coordinator sending messages (b) master- slave pair (dashed lines = inter-process messages).

When master and slave coordinators are used, DEVS processors are organized in a hierarchy, which does not have a one to one correspondence with the model hierarchy . Therefore a parent child -relationship that takes into account the existence of master and slave coordinators must be defined. This relationship is defined as follows:

a. for each *simulator*, the parent coordinator is the parent's model local processor (it is guaranteed that this one exists)
b. for each *slave coordinator*, the parent coordinator is the master coordinator of the model.

c. for each *master coordinator*, the parent coordinator is the parent's model local processor; just as if it were a *simulator*.

The simulation advances as a result of the exchange of messages between parent and child DEVS processors. Every message is a pair of the form *(type, time)* and can belong to one of two categories: synchronization messages and content messages. The synchronization messages are (@ , *t*), (*, *t*), and (*done*, *t*) and the contents messages are (*y*, *t*) and (*q* , *t*).

The synchronization messages (@ , *t*), (*, *t*) are sent from a parent DEVS processor to its imminent children. A (@ , *t*) is used to tell the children to send their outputs and (*, *t*) tells the children to invoke their transition function (whether it corresponds to execute an external, internal or confluent transition). All outputs produced by a model are translated to (*y*, *t*) messages between a child DEVS processor and its parent. Finally, those external messages that arrive from outside the system or that are generated as a result of an output message being sent to an influencee are sent as (*q* , *t*) messages.

Implementation of the Cell-DEVS specification language

The rules defining the models coupling and those related to the behavior of a cell should be translated into an executable definition. We show how rules for cellular models are evaluated (those used for coupled models and analytical specifications for atomic models are handled similarly). The rules specifications are associated with a function's identifier, which is registered by each cell. When the models to be simulated are loaded, if the definition of a transition function is not is registered, then it is added to a table. In this one, the function name acts as an identifier, and each one of the rules is represented with a tuple (value, delay, condition). Each element of the tuple is represented by a tree. The following class hierarchy is devoted to build such rule definition tree.

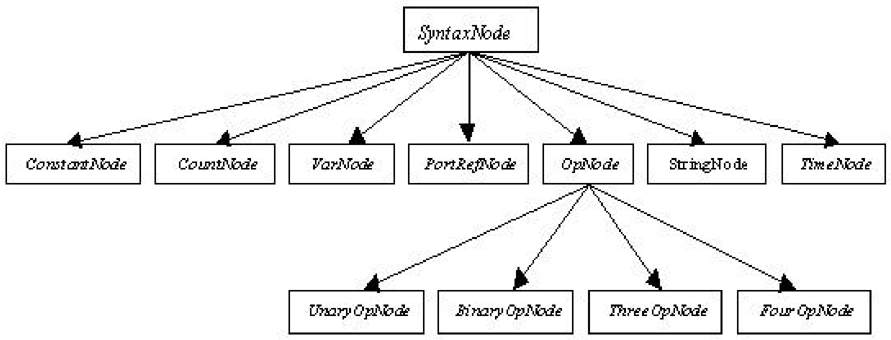


Figure 41. Subclasses of SyntaxNode

The *SyntaxNode* class is an abstract class that allows describing a node in the rule evaluation tree. It is composed by:

- *ConstantNode*: it stores a constant value with domain in the real numbers, integers, or three-valued logic.
- *CountNode:* it allows defining the functions: *TrueCount*, *FalseCount* and *UndefCount*.
- *VarNode*: it stores a reference to a neighbor cell, defined as an offset from the present cell.
- *PortDefNode*: it defines a reference to an input port of the cell.

- *StringNode*: it contains a string of characters repres enting the name of an input port of the cell. It is used to evaluate the *PortValue* function.
- *TimeNode*: it allows defining the *Time* function, which returns the present simulated time.
- *OpNode*: it is an abstract class representing functions with one or more parameters. It has the subclasses: *UnaryOpNode*, *BinaryOpNode*, *ThreeOpNode* and *FourOpNode*, which represent to the functions with one, two, three or four parameters respectively.

Each rule defining the behavior of a cell can be represented by a tree structure. For example, the representation of the first rule of the *Life Game* presented in Figure 18 can be represented as follows.

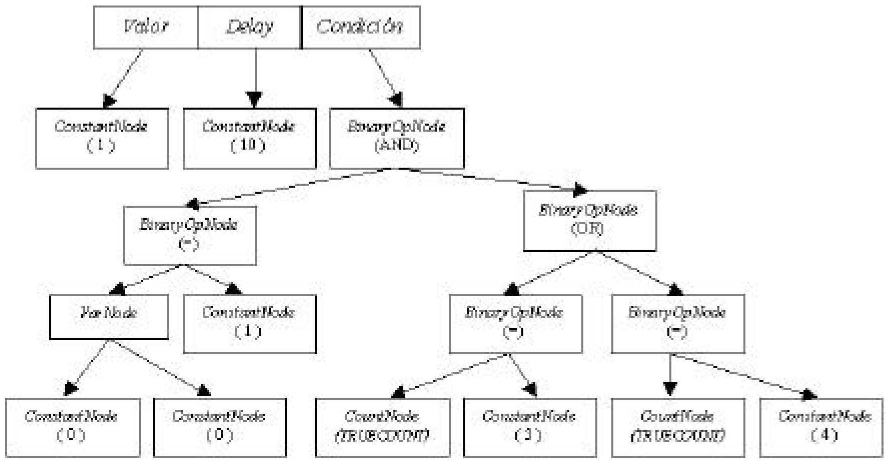


Figure 42. Tree structure used to represent the fist rule of the Life Game.

Therefore, to evaluate a rule, it evaluates in recursive form the tree that represents the condition. If the result of the evaluation is *True*, it proceeded to evaluate the trees corresponding to the value and the delay, and the result of these evaluations are the values used by the cell.

Cell-DEVS quantized models

Recently, a theory of quantized DEVS models was developed [4, 5]. The theory has been verified when applied to predictive quantization of arbitrary ordinary differential equation models. In this way, continuous variable models can be defined as DEVS. A curve is represented by the crossings of an equal spaced set of boundaries, separated by a *quantum* size. A *quantizer* checks for boundary crossings whenever a change in a model takes place. Only when such a crossing occurs, a new value is sent to the receiver. This operation reduces substantially the frequency of message updates, while potentially incurring into error.

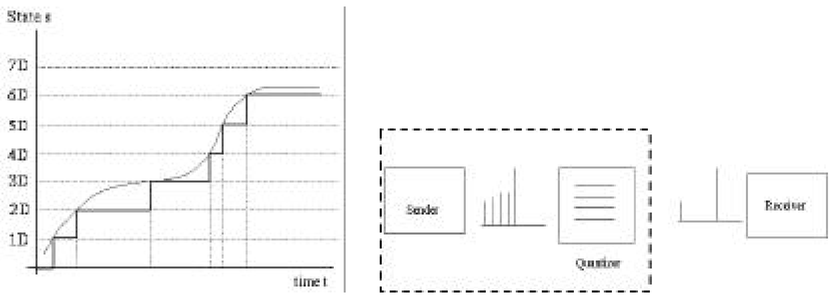
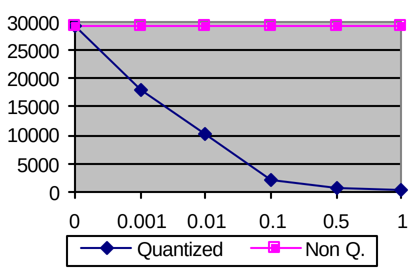
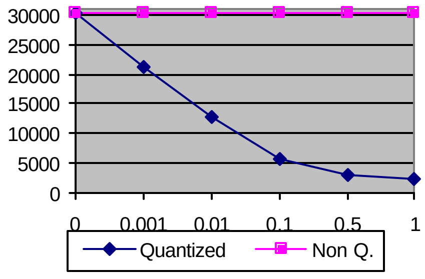


Figure 43. DEVS Quantization [5].

The cost/benefit analysis between reduced traffic and increased error was discussed in [28]. It was also shown that the technique is useful when it is applied to Cell-DEVS models [29]. It was found that the number of messages involved in the simulation of Cell-DEVS models presented a reduction in the number of messages involved, as it can be seen in the following figure (including the results for two different applications). The curves belong to the class of curves $f(x) = bx^{-a}$ with $x \in (0,1]$. These results approximate the theoretical optimum results presented in [5].

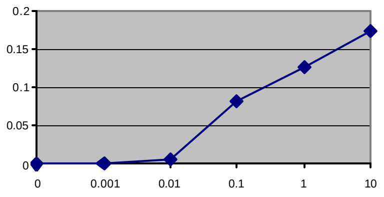




(a) (b)

Figure 44. Number of messages involved.

The introduction of quantizers introduces an error factor, which value is a function of the local computing function, the number of simulation steps and the quantum. The future input values for a cell are dependent of the present results for the cell. This dependence can lead to a nonlinear behavior of the error, depending on the interconnection of the cell. In any case it can be seen that the higher the quantum, the worse the error. The use of a higher quantum reduces the number of steps, but each of them has higher error. The experimental results validate this behavior.



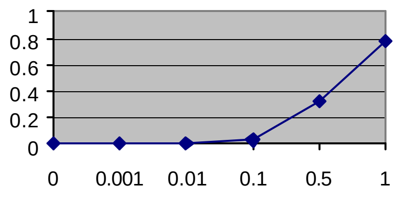


Figure 45. Accumulated error behavior.

The error grows as $f(x) = ax^b$, and it can be linear when there is no influence between cells. We can see that, in the (a) case, the error hardly increases while the messages go down by approximately 1/10. Considering these factors, the tool has included quantization facilities. The simulators are able to run quantized models in a more efficient fashion.

SIMULATING MODELS

Once a model has been generated and its description is included in the modeling 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 following figure shows a printout of the log file generated by the tool when the Life game is simulated.

It can be seen that the simulation process begins by sending an initialization (**I**) message to the external model (top), which is in charge to distribute them through the cell space (see (a) in the figure). Each of the cells replies with a **done (D)** message (see (b) in the figure), telling the time for the next internal event (in this case, 100 milliseconds, that is, the delay defined in the specification).

When the top-level coordinator receives all the **done** messages – (c) in the figure-, it transmits the time of the imminent child to the root coordinator. In this way, the global time of the simulation is updated. The root coordinator replies using a *-message (see (d) in the figure), that is sent to the imminent children. They execute, and their simulators generate **Y** and **done** messages (see (e) in the figure).

Finally, the coordinator analyzes which output of a model should be sent to others (see (f) in the figure), generating new **X** messages. The cycle is repeated up to the moment where all the external events are consumed, or when the simulation time is consumed (in this case, 500 milliseconds, as it can be seen in the item (g) of the figure).

```
Message I / 00:00:00:000 / Root(00) for top(01)                (a)
Message I / 00:00:00:000 / top(01) for life(02)
Message I / 00:00:00:000 / life(02) for life(0,0)(03)
Message I / 00:00:00:000 / life(02) for life(0,1)(04)
...
Message I / 00:00:00:000 / life(02) for life(19,19)(402)
Message D / 00:00:00:000 / life(0,0)(03) / ... for life(02)
Message D / 00:00:00:000 / life(0,1)(04) / 00:00:00:100 for life(02)     (b)
Message D / 00:00:00:000 / life(0,2)(05) / 00:00:00:100 for life(02)
...
Message D / 00:00:00:000 / life(19,19)(402) / ... for life(02)
Message D / 00:00:00:000 / life(02) / 00:00:00:100 for top(01)           (c)
Message D / 00:00:00:000 / top(01) / 00:00:00:100 for Root(00)
Message * / 00:00:00:100 / Root(00) for top(01)                (d)
Message * / 00:00:00:100 / top(01) for life(02)
Message * / 00:00:00:100 / life(02) for life(0,1)(04)
...
Message * / 00:00:00:100 / life(02) for life(19,17)(400)
Message Y / 00:00:00:100 / life(0,1)(04) / out / 0.000 for life(02)             (e)
Message D / 00:00:00:100 / life(0,1)(04) / ... for life(02)
Message Y / 00:00:00:100 / life(0,2)(05) / out / 0.000 for life(02)
...
Message D / 00:00:00:100 / life(19,16)(399) / ... for life(02)
Message Y / 00:00:00:100 / life(19,17)(400) / out / 1.000 for life(02)
Message D / 00:00:00:100 / life(19,17)(400) / ... for life(02)
Message X / 00:00:00:100 / life(02) / neighborchange / 0.000 for life(19,0)(383)        (f)
Message X / 00:00:00:100 / life(02) / neighborchange / 0.000 for life(19,1)(384)
...
Message X / 00:00:00:100 / life(02) / neighborchange / 1.000 for life(18,18)(381)
Message D / 00:00:00:100 / life(19,0)(383) / ... for life(02)
Message D / 00:00:00:100 / life(19,1)(384) / ... for life(02)
Message D / 00:00:00:100 / life(19,2)(385) / 00:00:00:100 for life(02)
...
Message D / 00:00:00:100 / life(18,18)(381) / ... for life(02)
Message D / 00:00:00:100 / life(02) / 00:00:00:100 for top(01)
Message D / 00:00:00:100 / top(01) / 00:00:00:100 for Root(00)
Message * / 00:00:00:200 / Root(00) for top(01)
Message * / 00:00:00:200 / top(01) for life(02)
Message * / 00:00:00:200 / life(02) for life(19,2)(385)
...
[The simulation cycle is repeated]
Message D / 00:00:00:500 / life(12,8)(251) / ... for life(02)
Message D / 00:00:00:500 / life(12,9)(252) / ... for life(02)
Message D / 00:00:00:500 / life(02) / ... for top(01)                 (g)
Message D / 00:00:00:500 / top(01) / ... for Root(00)
```

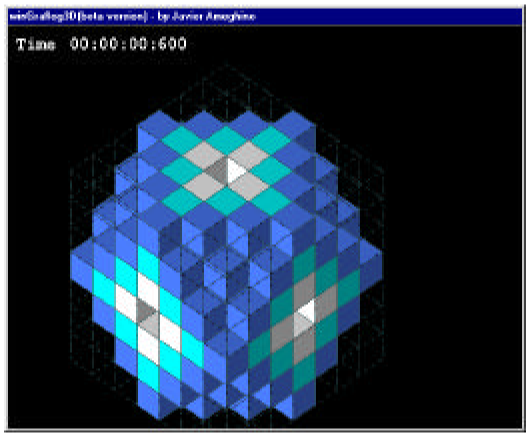
Figure 46. Message traffic during the execution of the hierarchical model.

Based on the log file, a simple text output interface was defined. It allows the user to see the execution of Cell-DEVS models in a fast way, allowing to debug executable models. For instance, the following figure shows the results obtained when executing the three-dimensional Life game defined earlier (showing the three-dimensional space as three independent planes).

```
Time: 00:00:00:000                                Time: 00:00:00:100
     0123456      0123456      0123456                 0123456      0123456      0123456
    +-------+    +-------+    +-------+               +-------+    +-------+    +-------+
   0|1      |   0|       |   0|1      |              0| 1    1|   0|11    1|   0| 1    1|
   1|1 1  11|   1|11   11|   1|  111  |              1|1 1   1|   1|1     1|   1|1 11  1|
   2| 1   1 |   2|   11 1|   2| 1 11  |              2|11  1 1|   2|1    1 |   2|11   11|
   3|       |   3|  1  11|   3|     11|              3|    111|   3|  1 1 1|   3|    1 1|
   4|  1  11|   4|  1 1  |   4| 1   11|              4|       |   4|     11|   4|       |
   5|  11  1|   5|   1 1 |   5| 11  1 |              5|1  111 |   5|1 111 1|   5|1  11 1|
   6|1  1  1|   6| 1   1 |   6| 1 11 1|              6|       |   6| 1     |   6| 1  1  |
    +-------+    +-------+    +-------+               +-------+    +-------+    +-------+
Time: 00:00:00:200                                Time: 00:00:00:900
     0123456      0123456      0123456                 0123456      0123456      0123456
    +-------+    +-------+    +-------+               +-------+    +-------+    +-------+
   0|1     1|   0|      1|   0|1     1|              0|       |   0|       |   0|       |
   1| 11  1 |   1| 1   1 |   1| 11  1 |              1|       |   1|       |   1|       |
   2|    11 |   2| 1    1|   2|       |              2|       |   2|       |   2|       |
   3|1   1  |   3|1   11 |   3|1   11 |              3|       |   3|       |   3|       |
   4|   1111|   4|   11  |   4|   1111|              4|1    1 |   4|1    11|   4|1    1 |
   5|1  1   |   5|1  1  1|   5|1  1  1|              5|       |   5|       |   5|       |
   6|11   11|   6|11   11|   6|11  111|              6|       |   6|       |   6|       |
    +-------+    +-------+    +-------+               +-------+    +-------+    +-------+
      ...          ...           ....
```

Figure 47. Execution results for the 3D Life Game.

These values can be used to visualize the results using 3D tools. Following we show the outputs of the 3D Life Game in one of our current 3D visualization engines:



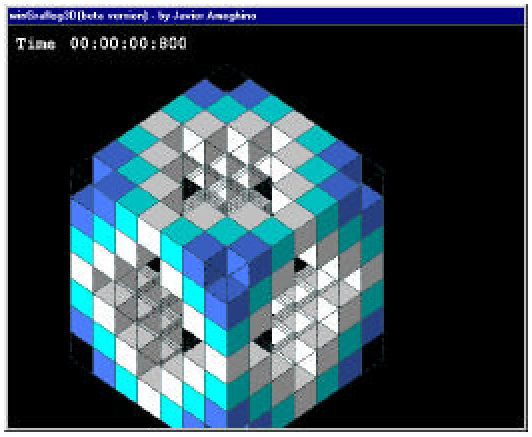


Figure 48. Graphical display for the 3D Life Game.

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).

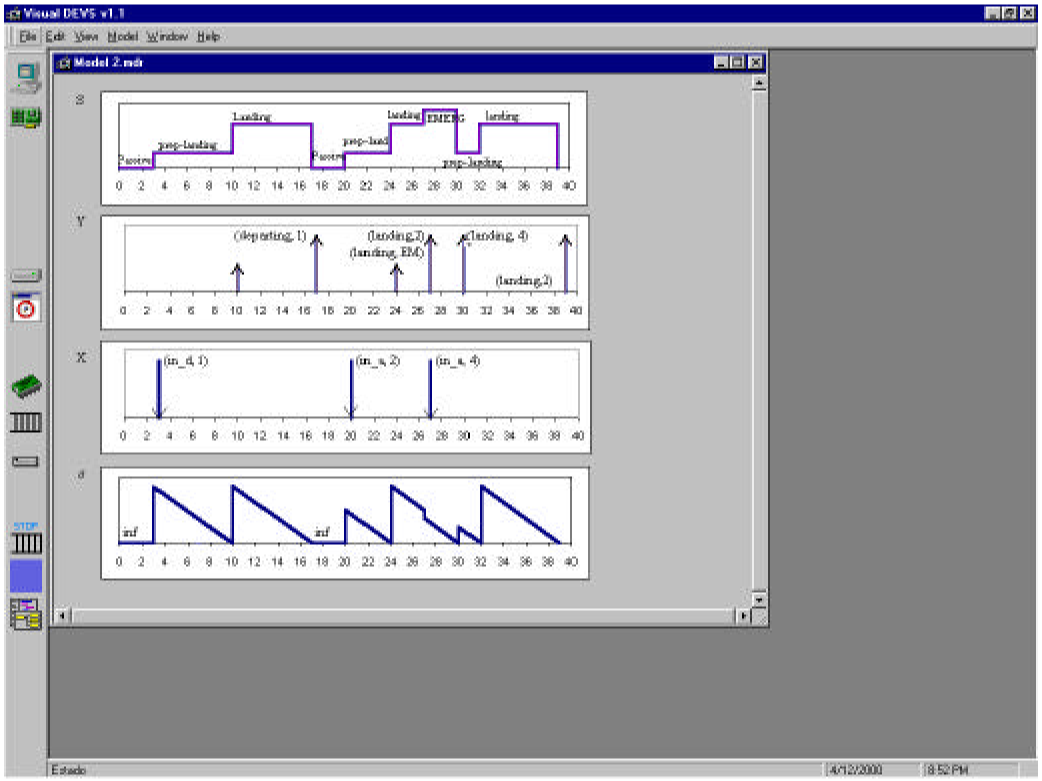


Figure 49. Execution results for Airport model.

In this case, 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_rnwy-use* 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 *rnwy-use* 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 runway, and lands (the values related with this flight are kept in the control tower queue).

Besides these standard graphical outputs, a SCADA tool built with industrial application purposes was modified to receive inputs from the CD++ tool [30]. The SCADA receives input from the simulator, and reacts to them as if they were real-time data. In this way, complex man-machine interaction can be provided. The user can define what values in a model wants to be analyzed, and how. Graphical mimics can be associated with different It also can associate alarm conditions to the central databases, combine the results, store them as historical data to be analyzed, etc.

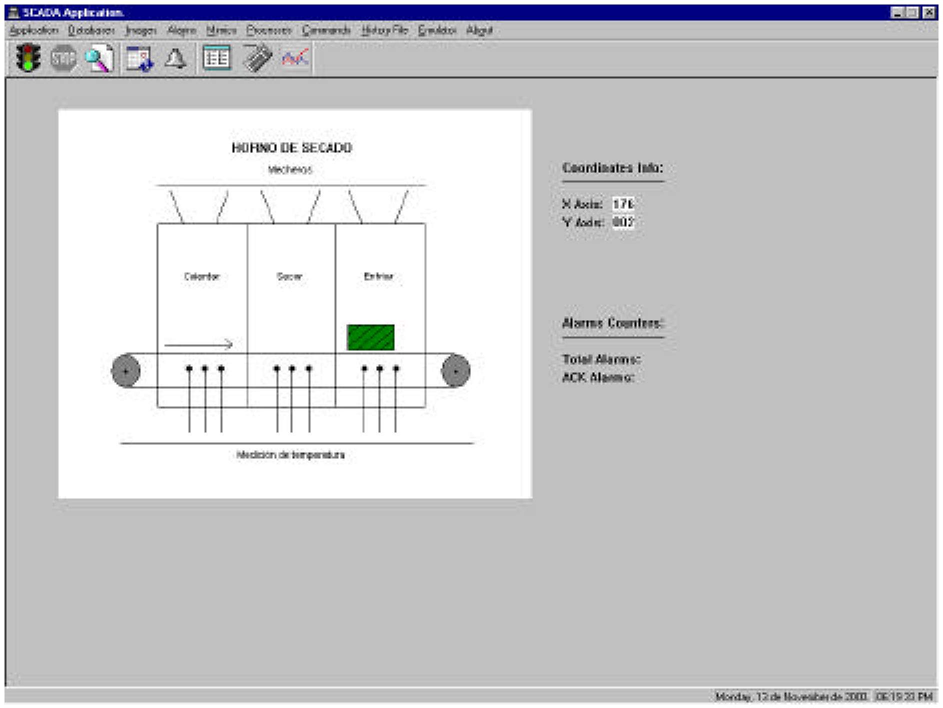


Figure 50. Output visualization using IGNATIUS.

Another graphical output included shows the execution results of Cell-DEVS models. For instance, the following figure includes the execution results of the original Rothermel model presented earlier.

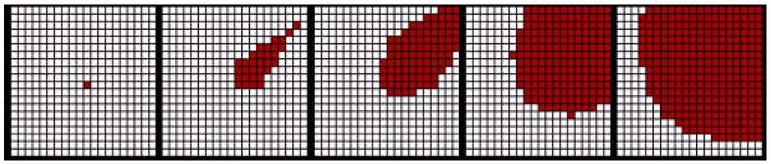


Figure 51. Fire propagation results. The fire is originated in cell (11,11) with wind blowing NE.

Each of these graphical tools is independent from the simulation engine. Therefore, different kind of visualization methods can be employed for the same simulation results. Some of these results can be seen using a centralized tool using Windows. Others were developed in Java and can be embedded in a web front-end that can communicate with the simulation server, providing facilities for web-based simulation. Recently we have extended these results to provide 3D visualization using VRML.

Finally, the following figure shows the execution results of different quantized versions of a model. It shows that the error introduced by the quantizers can lead to undesired behavior for the execution models.

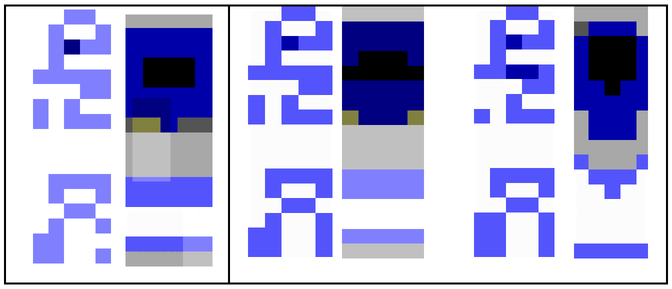


***Figure 52*. Error behavior for the heat seekers. (a) non-quantized and q=0.001; (b) q=0.1; (c) q=1.**

We show the behavior of devices seeking heat sources in a diffusion model. This three-dimensional model uses one plane for the heat diffusion, and a second one for the heat field. The seekers move in the plane looking for the heat maxima in the second plane. The figure presents, in the left, the behavior of the heat seekers, and in the right, the heat surface for a given simulated time (using different quantum in each case). We can see that the use of a high quantum produces much higher errors, producing a result completely different from the desired in both fields. The use of a smaller quantum also introduces errors, but most seekers can reach the local maximum. Therefore, several orders of improvement in execution times can be achieved using small quanta, preserving the behavior of the model. In this case, a speedup of almost 40% was achieved with almost no error.

CONCLUSION.

We have introduced several features CD++, a toolkit for DEVS modeling and simulation. The tool was built using the DEVS formal modeling technique, 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. The separation of visualization tools from the simulation engines make easier the development of new improvements in visualization, whereas making easier its use for web-based simulation.

We showed that the abstract simulation mechanism enables defining different simulation techniques without needing changing the models developed, as they follow the formal specifications of DEVS formalism. The models are easily reusable thanks to the hierarchical construction. Consequently, the costs of development are reduced, the quality of the models improves, and non-expert users are able to start developing new applications with a fast learning curve. 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 [14].

The tool was used to develop several kinds of application examples, which allowed us to show the flexibility of the toolkit. Some of them include:

. Alfa-1: a complete simulated processor emulating the SPARC CPU (used with educational purposes);

. ATLAS: a high-level specification language for traffic models (mapped in DEVS and Cell-DEVS);

. A watershed model;

. Rothermel fire spread analysis;

. Robot moving in a manufacturing plan;

. Diverse physical problems: heat diffusion, crystal growth, excitable media, particle collision, heat seekers, surface tension, etc.;

. An airport model (and a three dimensional model of flying planes coming to the airport);

. A model of a manufacturing plant;

. A library of network performance analysis (based on client/server applications for web servers);

. Cryptokey generation;

. Ant foraging models and other ecological systems.

At present some other applications are being developed, including analysis of PCS systems, behavior of the heart tissue, a flow injection model, and an extended library for modeling network systems.

The tools are public domain and can be obtained at "http://www.sce.carleton.ca/faculty/wainer/celldevs". The developed models are publicly available, and will be incorporated to a web-based environment that can be applied to the development of DEVS models.

REFERENCES

[1] HO, Y. "Special issue on discrete event dynamic systems". *Proceedings of the IEEE*, 77 (1), 1989.
[2] ZEIGLER, B. "Theory of modeling and simulation". *Wiley*, 1976.
[3] ZEIGLER, B.; KIM, T.; PRAEHOFER, H. "Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems". Academic Press. 2000.
[4] ZEIGLER, B. "DEVS Theory of Quantization". DARPA Contract N6133997K-0007: ECE Dept., UA, Tucson, AZ. 1998.
[5] ZEIGLER, B., CHO, H.; LEE, J.; SARJOUGHIAN, H. "The DEVS/HLA Distributed Simulation Environment And Its Support for Predictive Filtering". DARPA Contract N6133997K-0007: ECE Dept., UA, Tucson, AZ. 1998.
[6] GIAMBIASI, N.; ESCUDE, B. GHOSH, S. "GDEVS: A Generalized Discrete Event Specification for Accurate Modeling of Dynamic Systems," *Transactions of the Society for Computer Simulation*, Vol. 17, No. 3, 2000.
[7] WAINER, G.; GIAMBIASI, N. "Timed Cell-DEVS: modeling and simulation of cell spaces ". In "Discrete Event Modeling & Simulation: Enabling Future Technologies", Springer-Verlag. 2001.
[8] WOLFRAM, S. "Theory and applications of cellular automata". Vol. 1, *Advances Series on Complex Systems*. World Scientific, Singapore, 1986.
[9] WAINER, G.; BARYLKO, A.; BEYOGLONIAN, J. "Experiences with DEVS modeling and simulation". In *IASTED Journal on Modeling and Simulation*. June 2001.
[10] WAINER, G. "Discrete-event cellular models with explicit delays". *Ph.D. Thesis. Université d'Aix-Marseille III.* 1998.
[11] WAINER, G.; CHRISTEN, G.; DOBNIEWSKI, A. "Defining models with the CD++ toolkit". *Proceedings of the European Simulation Symposium.* Marseille, France. 2001.
[12] RODRIGUEZ, D.; WAINER, G. "New Extensions to the CD++ tool". *Proceedings of Summer Computer Simulation Conference*. Chicago, IL. USA. 1999.
[13] MUZY, A.; INNOCENTI, E.; SANTUCCI, J.F.; WAINER, G. "Comparing simulation methods for fire spreading across a fuel bed". *Proceedings of AIS 2002 – Simulation and planning in high autonomy systems*. Lisbon, Portugal. 2002.
[14] WAINER, G.; GIAMBIASI, N. b "Application of the Cell-DEVS formalism for cell spaces modeling and simulation". *Simulation*. 2001.
[15] DE SIMONI, F.; BUROTTI, M.,; WAINER, G. "Definition of an airport model using the CD++ tool ". Report, Comp. Science Dept. Universidad de Buenos Aires. http://www.dc.uba.ar/people/proyinv/celldevs. 1999.

[16] MORIHAMA, L.; PASUELLO, V.; WAINER, G. "Automatic Verification of DEVS Models". In Proceedings of 2002 Spring Simulation Interoperability Workshop, Orlando, FL. U.S.A. 2002

[17] ZEIGLER, B.; SONG, H.; KIM, T.; PRAEHOFER, H. "DEVS Framework for Modeling, Simulation, Analysis, and Design of Hybrid Systems". In *Proceedings of HSAC*, 1996.

[18] GARDNER, M. "The fantastic combinations of John Conway's New Solitaire Game 'Life'.". *Scientific American*. 23 (4). pp. 120-123. April 1970.

[19] AMEGHINO, J.; TROCCOLI, A.; WAINER, G. "Modeling and simulation of complex physical systems using Cell-DEVS". *Proceedings of the 33rd SCS Summer Computer Simulation Conference*. Seattle, WA. USA. 2001.

[20] AMEGHINO, J.; WAINER, G. "Application of the Cell-DEVS formalism using N-CD++". *Proceedings of the 32nd SCS Summer Computer Simulation Conference*. Vancouver, Canada. 2000.

[21] ROTHERMEL, R. “A mathematical model for predicting fire spread in wildland fuels”. Research Paper INT-115. Ogden,UT: U.S. Department of Agriculture, Forest Service, Intermountain Forest and Range Experiment Station; 1972. 40 p.

[22] WEIMAR, J. “*Simulation with Cell spaces*”. Berlin, Logos – Verl, 1997

[23] ZEIGLER, B. "Multifaceted Modeling and discrete event simulation". *Academic Press*, 1984.

[24] KIM, K.; KANG W.; SAGONG, B.; SEO, H. “Efficient Distributed Simulation of Hierarchical DEVS Models: Transforming Model Structure into a Non-Hierarchical One”. *Proceedings of the 33rd Annual Simulation Symposium*. 1998.

[25] Glinsky, E. and Wainer, G. 2002a. Definition of Real-Time simulation in the CD++ toolkit. Accepted for publication in *Proceedings of SCS Conference*, San Diego, CA.

[26] GLINSKY, E. AND WAINER, G 2002b. Performance analysis of DEVS environments. In *Proceedings of AI Simulation and Planning*. Lisbon, Portugal.

[27] TROCCOLI, A.; WAINER, G. "Performance analysis of cellular models with Parallel Cell-DEVS". *Proceedings of Summer Computer Simulation Conference*. Orlando, FL. 2001.

[28] ZEIGLER, B.; BALL, G.; CHO, H.; LEE, J. S; SARJOUGHIAN, H. "Bandwidth Utilization/Fidelity Tradeoffs in Predictive Filtering". SISO SIW '99. Orlando, Florida. March 1999.

[29] WAINER, G.; ZEIGLER, B. "Experimental results of Timed Cell-DEVS quantization". In *Proceedings of AIS'2000*. Tucson, Arizona. U.S.A. 2000.

[30] BENITEZ, S.; SEOANE, J.; WAINER, G.; BEVILACQUA, R. "Experiences with a tool to develop SCADA systems". *Proceedings of the 1997 IEEE Conference on Systems, Man and Cybernetics.* Orlando, USA