Automatic verification of DEVS models

Gabriel Wainer
Systems and Computer Engineering
Carleton University
1125 Colonel By Drive
Ottawa, ON. K1S 5B6. Canada.
gwainer@sce.carleton.ca

Liliana Morihama *Viviana Passuello*
Departamento de Computación
Universidad de Buenos Aires
Pabellón I. Ciudad Universitaria (1428)
Buenos Aires. Argentina.
{lmoriham, vpassuel}@dc.uba.ar

Keywords: DEVS, model verification, Cell-DEVS

ABSTRACT: *The Discrete Event System Specification (DEVS) formalism is an abstract basis for model specification that is independent of any particular simulation implementation. We have developed a tool following DEVS theory that allows a user to define complex models that can be executed using different abstract mechanisms. Recently, we have included a set of automatic verification facilities. In this way, the model interaction can be verified with reduced user intervention. We have employed the new techniques applying them to existing DEVS models, finding errors in the specifications. This approach helped improving the development times in simulation models.*

1. Introduction

In recent years, several efforts have been devoted to define modelling paradigms, allowing improving the analysis of complex dynamic systems through simulation of these models. A 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 [1]. In [2] the Cell-DEVS formalism was presented, as a means to describe cell spaces as a DEVS models including explicit timing delays.

We have built a toolkit with the goal of develop models and simulate them based on the DEVS and Cell-DEVS paradigms. The core of the toolkit is the CD++ environment [3, 4], which implements the DEVS and Cell-DEVS theory. The toolkit has been built as a set of independent software pieces, each of them independent of the operating environment chosen. The tool includes a set of automatic verification tools, based on the construction of Experimental Frameworks used as signal generators/acceptors. We have also included automatic verification for Cell-DEVS rules. In the following sections we will present these features. First, we will briefly recall the basic ideas related with DEVS and Cell-DEVS theory. After, we introduce the main features of the toolkit related to the model definition and verification. Finally, we show some examples of application of the techniques developed.

2. The DEVS formalism

DEVS is a systems theoretical approach that permits defining 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). 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:

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

where
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 provided with an interface consisting of input and output ports to communicate with other models. Input external events (those events received from other models) are collected in input ports. The external transition function specifies how to react to those inputs. The internal transition function is activated after a period defined by the time advance function. The goal is to produce internal state changes. Model execution results are spread through output ports. This is done by the output function, which executes before any internal transition.

A DEVS coupled model is composed by several atomic or coupled submodels. Coupled models are closed under coupling, therefore they can be integrated to a model hierarchy, allowing model reuse. Coupled models are formally defined as:

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

where
X is the set of input events;
Y is the set of output events;
D is an index for the components of the coupled model, and
$\forall$ i $\in$ D, $\mathbf{M_i}$ is a basic DEVS (that is, an atomic or coupled model),
$\mathbf{I_i}$ is the set of influencees of model i (that is, the models that can be influenced by outputs of model i), and $\forall$ j $\in$ I_i,
$\mathbf{Z_{ij}}$ is the i to j translation function.

Coupled models are defined as a set of basic components (atomic or coupled), which are interconnected through the model's interfaces. 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 [2] has extended the DEVS formalism, allowing 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 is to improve performance 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:

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

Here,
X defines external input events,
Y are the 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;
δ_{int} is the internal transition function,
δ_{ext} the external transition function,
τ is a local computing function,
λ the output function, and
ta is the state's time advance function.

Each cell uses N inputs to compute its next state. These inputs, which are received through the model's 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 time advance functions, as in other DEVS models.

Once the cell behavior is defined, a coupled Cell-DEVS can be created by putting together a number of cells interconnected by a neighborhood relationship. A Cell-DEVS model is defined by:

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

Here,
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 composed of an array of atomic cells, whose size and dimensions are defined by the n and {t1,..,tn} parameters. Each cell is connected to its neighborhood, whose shape must be defined. As the cell space must execute within finite boundaries, border cells should be provided with a different behavior than the rest of the space. Otherwise, the space is considered to be "wrapped", meaning that the cells in a border are connected with those in the opposite one. Finally, the Z function defines internal and external couplings, which uses the Xlist and the Ylist (devoted to define external coupling) and the neighborhood definition (for internal coupling.

3. CD++

CD++ [3, 4] is a modeling tool that was defined using the specifications presented in the previous section, and the basic simulation techniques introduced in [1, 2]. The toolkit includes facilities to build DEVS and Cell-DEVS models. DEVS Atomic models can be programmed and incorporated onto a class hierarchy programmed in C++. Coupled models can be defined using a built-in specification language. Cell-DEVS models are built following the formal specifications of the previous section, and we also provided a built-in language to describe them. The following sections will define the facilities available in the toolkit. We begin by introducing the definition of Cell-DEVS models and the verification tools associated. Then, we show how to define DEVS models and how to use the automatic verification tools associated with them.

3.1 Cell-DEVS models definition

The CD++ tool includes an interpreter for a specification language that allows describing the behavior of each cell of a cellular model, including its delay and neighborhood. In addition, it allows to define the size of the cell space and their connection with other DEVS models, the border and the initial state of each cell. These definitions are based on the formal specifications defined earlier, and can be completed by considering a few parameters: size, influencees, neighborhood and borders. The following figure shows the definition for the "Life" Game [5]. This model represents a cell space with entities that evolve according to the neighbors' states.

```
[life]
type : cell
width : 20            height : 2
delay : transport
border : wrapped
neighbors : life(-1,-1) life(-1,0) life(-1,1)
neighbors : life(0,-1)  life(0,0)  life(0,1)
neighbors : life(1,-1)  life(1,0)  life(1,1)
localtransition : new-life-rule

[new-life-rule]
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 1. Definition of the Life game.

As we can see, the model specification includes all the parameters defined in section 2 for Cell-DEVS models. We also see that the behavior of the local computing function is defined using a set of rules. Each rule has the form:

VALUE DELAY { CONDITION }

These indicate that when the *CONDITION* is satisfied, the state of the cell will change to the designated *VALUE*, whose output it will be *DELAY*ed for the specified time. If the precondition is *false*, the next rule in the list is evaluated until a rule is satisfied or there are no more rules.

In the Life game example, the rules define that 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 (`(0,0) = 0`)and the neighborhood has 3 active cells, the cell activates (represented by a value of 1 in the cell). In every other case, the cell remains inactive (*t* indicates that whenever the rule is evaluated, a *True* value is returned).

Complex cellular models can be defined with simple rules (see, for instance, [6, 7]), and several useful operations are included: boolean (*AND*, *OR*, *NOT*, *XOR*, *IMP* and *EQV*), comparison (=, !=, <, >, <= and >=), and arithmetic (+, -, * and /). In addition, different types of functions 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's 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 conditions. Some common constants are predefined: *pi*, *e*, gravitational constant, light speed, Planck's constant, etc. Finally, pseudorandom numbers generation is included, using different probability distributions.

We have included diverse verification facilities for Cell-DEVS models. The most simple ones include checks on the number of cells in each dimension, initialization of each of the cells, position of the border cells (and zones with specialized behavior) and positions of the input/output cells representing the Xlist and Ylist.

The most complex verification aids are related with the definition of the local computing function rules. For instance, if no rule satisfy a precondition, an error will be raised, aborting the simulation process. This error indicates 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 will be aborted.

3.2 Defining DEVS Atomic models

CD++ was built as a class hierarchy of models related with a simulation processing entity. DEVS Atomic models can be programmed and incorporated onto the Models basic class hierarchy using C++. A new atomic model is created as a new class that inherits from the *Atomic* base class. The state of a model is defined in the *AtomicState* class. When creating a new atomic model, a new class derived from *Atomic* has to be created.

```
class Atomic : public Model  {
public:
virtual ~Atomic();    // Destructor

protected:
//User defined functions.
virtual Model &initFunction() = 0;
virtual Model &externalFunction( const External-
Message & );
virtual Model &internalFunction( const Internal-
Message & ) = 0 ;
virtual Model &outputFunction( const CollectMes-
sage & ) = 0 ;
virtual string className() const

//Kernel services
Time nextChange();
Time lastChange();

Model &holdIn(const AtomicState::State &, const
Time &);
Model &sendOutput(const Time &time, const Port &
port , Value value);
Model &passivate();
virtual ModelState* getCurrentState() ;
};     // class Atomic
```

Figure 2. The Atomic Class

Atomic is an abstract class that declares a model's API and defines some service functions the user can use to write the model. The *Atomic* class provides a set of services and requires a set of functions to be redefined:

- holdIn(state, Time) : tells the simulator that the model will remain in the state *state* for a period of Time *time*. It corresponds to the ta(s) function of the DEVS formalism.

- passivate(): sets the next internal transition time to infinity. The model will only be activated again if an external event is received (this function is equivalent to **holdIn(passive, Infinite)**).

- sendOutput(Time, port, BasicMsgValue*): sends an output message through the port. The time should be set to the current time.

- nextChange(): Returns the remaining time for the next internal transition.

- lastChange(): Returns the time since the last state change.
- state(): Returns the current model's phase.

The new class should override the following functions:

- virtual Model &initFunction(): this method is invoked by the simulator at the beginning the simulation and after the model state has been initialized.

- virtual Model &externalFunction(const External-Message &): this method is invoked when one external event arrives to a port. It corresponds to the δ_{ext} function of the DEVS formalism.

- virtual Model &internalFunction(const Internal-Message &): this method corresponds to the δ_{int} function of the DEVS formalism.

- virtual Model &outputFunction(const CollectMessage &): it is in charge of sending all the output events of the model. It corresponds to the λ function of the DEVS formalism.

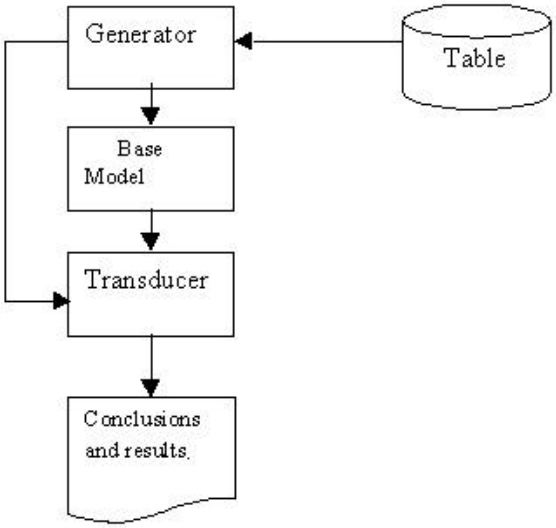


Figure 3. Format for the verification framework.

We have developed an Experimental Framework that automatically verifies DEVS base models. Once an atomic model has been built and incorporated to the modelling hierarchy, we can control if the model being verified returns the expected results at a given time. The Experimental Framework composed by a Generator and a Transducer doing these tasks for any existing atomic model. These models are coupled to the base model to be verified. The Generator recognizes the input ports of the base model, and connects its outputs to the model's inputs. The Transducer 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 input for the generator is a three-column table using the following format :

	Transition Type	Time	Value
1	E	00:00:08:00	28
2	E	00:00:10:00	40
3	I	00:00:20:00	30
4	I	00:00:22:00	42

Figure 4. Input format for the verification framework.

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 28 enters the model at 00:00:08:00*". In the same way, for an *I* type transition, data in the table should be interpreted as "*Model must output value Y at time T*". Line 3 of the example should be read "*Model must output value 30 at 00:00:20:00*".

The data corresponding to external transitions, is sent by the Generator to the base model in order to be processed, while the ones corresponding to internal transitions are sent to the Transducer. The Transducer stores this information, and later on, compares it with the real values issued by the model.

As a result, output error messages are issued. For every error found, the following message is issued:

```
Invalid result
        Expecting: XXX At: Expected_Time
        Getting: YYY At: Output_Time
```

The Results file allows checking the differences between the expected data and the outputs issued by the model. Also, differences in timing can be analyzed.

3.3 Defining DEVS Coupled models

Once an atomic model is defined, it can be combined with others 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.

The coupled model at the higher level is always named `[top]`. As showed in formal specifications presented in section 1, four properties must be configured: components, output ports, input ports and links between models. The following syntax is used:

`Components`: `model_name1[@atomicclass1] [model_name2[@atomicclass2] ...`
Lists the components integrating the coupled model. A coupled model might have atomic models or other coupled model as components. For atomic ones, an instance name and a class name must be specified. This allows a coupled model to use more than one instance of an atomic class. For coupled models, only the model name must be given. This model name must be defined as another group in the same file.

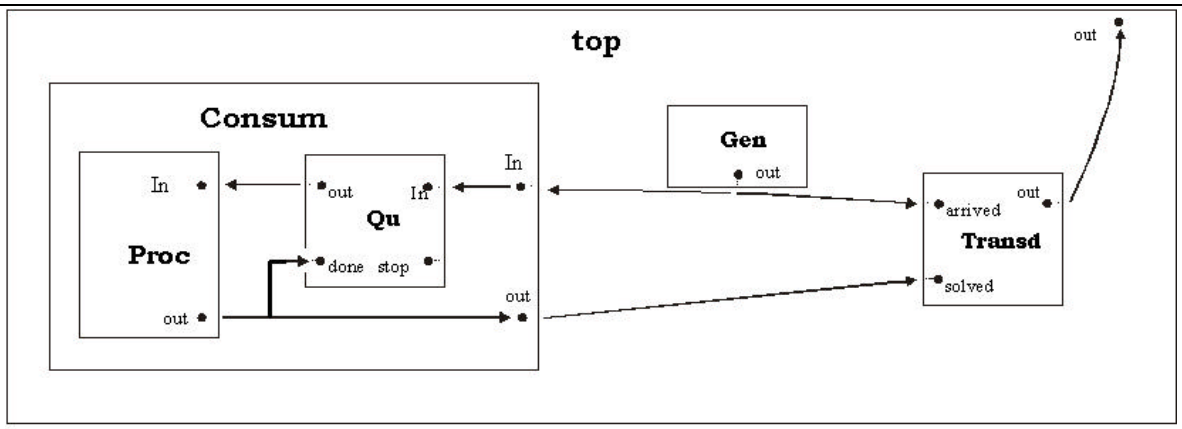
`Out`: `portname1 portname2 ...`
Enumerates the model's output ports. This clause is optional because a model may not have output ports.

`In`: `portname1 portname2 ...`
Enumerates the input ports. This clause is also optional because a coupled model is not required to have input ports.

`Link`: `source_port[@model] destination_port[@model]`.
It describes the internal and external coupling scheme. The name of the model is optional. If it is not included, the model used by default will be the coupled model currently being defined.

The following figure shows a sample coupled model and its specification in CD++. It consists of three models: a generator, in charge of creating data, a consumer, and a transducer, in charge of measuring the consumer speed. The consumer is also a coupled model, composed by a processor and a queue to keep waiting jobs.

In the top level of this example, the *Generator* influences the *Transducer* and the *Consumer*. The *Consumer* also influences the *Transducer*. The *Consumer* influences the *Queue* and the Processor also influences it. Then, the *Queue* influences the *Processor*. Finally, the *Transducer* influences the top model. These influences define the influencee's sets for each of the components, which is used to define the translation functions. The figure shows the influences for this example, which are carried out by transmitting information through the input/output ports in the models.




```
[top]
components : Transd@Transducer Gen@Generator Consum
Out : out
Link : out@generator arrived@transducer
Link : out@generator in@Consumer
Link : out@Consumer solved@transducer
Link : out@transducer out

[Consum]
components : Qu@Queue Proc@CPU
in : in
out : out
Link : in in@qu
Link : out@qu in@Proc
Link : out@Proc done@qu
Link : out@Proc out
```

Figure 5. Definition of a DEVS coupled model [3]

The verification tools for coupled models are in charge of authenticate ports and their links between DEVS Models. It will be checked that every input port is associated to an output port, and vice-versa.

A list of influencees is defined and associated to each output port. This ***Influencee_List*** holds all the input ports linked to the current output port. The influencee lists are 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.

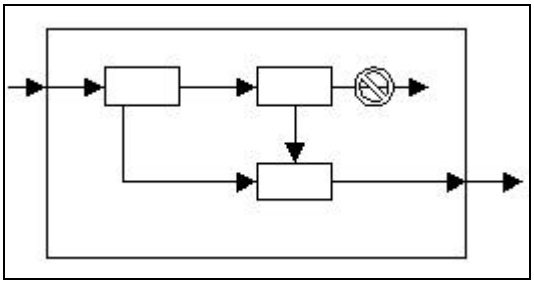


Figure 6. Output ports not linked.

A global influencees 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.

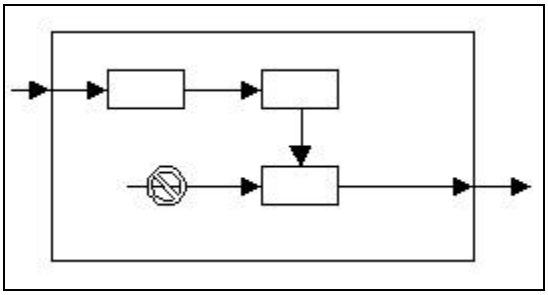


Figure 7. Input ports not linked.

In both cases, the simulation will be aborted, and an error message will be displayed, 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. As we can see, 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 cor-

rect. The tool includes type validation, ensuring compatibility of the data shared through input/output ports.

4. Some verification examples

As we explained earlier, the automatic verification facilities can improve the definition of models. We will first show some aspects related to Cell-DEVS models, and then we will explain how to verify DEVS models using the toolkit.

4.1 Cell-DEVS models verification

As we explained in section 3.1., we have included different verification techniques for the rules defining the cell behavior in Cell-DEVS models. 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 8. Redefinition of the Life game.

In these situations, an error will be 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 evaluate TRUE!
  Model used is: new-life-rule
  The state of the Neighbours is:
  +---------------------------------------+
  |      0.00000       1.00000       2.00000 |
  |      1.00000       2.00000       1.00000 |
  |      0.00000       1.00000       2.00000 |
  +---------------------------------------+
Aborting simulation...
```

Figure 9. 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 will be *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 10. 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 will result in an *Undefined* state. In that case, when we evaluate `(0,1) = ?`, the result is ?, and the result of the AND operation will be *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 11. 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 will be 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 12. 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 Neighbours is:
+--------------------------------------+
|      1.00000       0.00000       0.00000 |
|      1.00000       0.00000       1.00000 |
|      1.00000       0.00000       0.00000 |
+--------------------------------------+
Aborting simulation...
```

Figure 13. Error detected: ambiguous rules.

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

```
[new-life-rule4]
rule : 2 100 { (0,0) = 1 }
rule : 1 100 { (0,0) = 1 and random < .8 }
rule : 0 100 { (0,0) = 2 or (0,0) = 0 }
```

Figure 14. Ambiguous rules with random values.

In this case, when the value of the cell is 1, and the random number generated is smaller than 0.8, the first and second rule are valid. The cell's future value is different for both rules. Therefore, the first rule is considered as the valid one, and the following message is generated:

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

Starting simulation. Stop at time: 00:00:05:000
Warning! – Stochastic model with two or more
   rules evaluated to TRUE
...
```

Figure 15. Warning: ambiguous stochastic model.

In other cases, the random execution can make every precondition to be evaluated to *False*. In this case, the cell will be assigned an *Undefined* value, and will use the default delay time for the cell. For instance, the following figure shows a set of rules producing this kind of error:

```
[new-life-rule5]
rule : 1 100 { (0,0) = 1 and random <= .8 }
rule : 0 100 { (0,0) = 2 }
rule : 2 100 { (0,0) = 0 }
```

Figure 16. Ambiguous rules with random values.

In this case, if the cell has a value 1 and the random function generates a value higher than 0.8, all the rules are evaluated to *False*. In this case, 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 evaluates to True
   in an Stochastic model
...
```

Figure 17. Warning: ambiguous stochastic model.

4.2 DEVS models verification

Let us show now how to verify different examples of DEVS models. Figure 5 presented a coupled model representing a Producer/Consumer scheme. We have checked the *Proc* model separately using the verification Experimental Frame, using the following input data:

```
E 00:00:08:000 2
I 00:00:11:000 2
E 00:00:10:000 15
I 00:00:10:003 15
E 00:00:12:000 20
I 00:00:12:003 40
E 00:00:15:000 60
I 00:00:15:003 60
E 00:00:19:000 70
I 00:00:22:000 70
```

Figure 18. Input data for the Proc model.

We have injected different types of inputs, to show the possible errors that can be obtained. The behavior of the *Proc* model is to act as a standard server model: it receives inputs that are served, providing outputs. Every time an input is received (representing a job id), it is stored during 0.003 seconds, and an output (representing the job id) is generated. When we run the verification frame in this model, we obtain the results presented in Figure 19.

In the first case, we inject a value 2 to the model, and we say that the model should generate a value 2 in simulated time 11:000. Nevertheless, the processor outputs job 2 only 0.003 seconds after the input. This results in an output error. Then, we inject the value 15, and we say that we expect the output 15 at 10:003. As we can see, no errors were raised in this case. The following error shows that, after injecting a value 20 at 12:000, the model returns a 40 at 12:003, which is an unexpected value according to the input definition. The final error shows another timing problem according to the input specification.

```
Output file:

00:00:08:003 out 2
00:00:10:003 out 15
00:00:12:003 out 20
00:00:15:003 out 60
00:00:19:003 out 70

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

Invalid result:
   Expecting: 40.000000 At: 00:00:12:003
   Getting: 20.000000 At: 00:00:12:003

Invalid result:
   Expecting: 70.000000 At: 00:00:22:000
   Getting: 70.000000 At: 00:00:19:003
```

Figure 19. Output data for the Proc model.

As explained in section 3.3, we also included verification mechanisms enabling automatic checking of model coupling. We used 18 previously existing models, and they were verified with the tool. In most cases no errors were found. Nevertheless, the following models presented inconsistencies.

The producer/consumer example presented in Figure 5 includes a *stop* port in the *Qu* model. This port is used to send stop signals, in order to avoid buffer overruns. In this example, the port is not being used, therefore, it is not linked to any input port. When the model is verified, the following error message is issued:

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

Starting simulation. Stop at time: Infinity

Exception thrown!
Model Name: Qu
Output Port Name: stop, has not influences.
Description: Output Ports without Influences!

Aborting simulation...
```

Figure 20. Error detected: Unlinked ports.

Another model in which we detected inconsistencies represents an Operation Room in a hospital Emergency Room. The idea is to enable better scheduling of the existing resources. This model is composed by the following submodels:

- Scheduler: it handles the waiting list of operations. When a patient arrives, it will first do some filing and recording. After a certain time, it will output a request to a doctor. The requests are added into a waiting list that can be managed using different strategies.
- Doctor: it models the examination time of a patient, after which the patient is passed on to a nurse.
- Nurse: models routine check-ups on the patient before sending him into the operation room.

The Operation Room includes two components:

- Monitoring: it models the check-ups done to the upon arrival to the operation room.
- Operation: when the operation is finished, this model will send a "done" message to the scheduler so that it can output the next patient on the list to get the doctor's examination.

This example was executed with different configurations, in order to analyze scheduling strategies. The couple model representing the system is shown in the following figure:

```
[top]
components : scheduler@Scheduler doctor@DOCTOR
components : nurse@Nurse Operationroom
out : out              in : in

Link : in in@scheduler
Link : out@scheduler in@doctor
Link : out@doctor in@nurse
Link : out@nurse in@Operationroom
Link : out@Operationroom out
Link : done@Operationroom done@scheduler

[Operationroom]
components : equipment@Equipment
components : operation@Operation

in : in        out : out done
Link : in in@equipment
Link : out@equipment in@operation
Link : out@operation out
Link : done@operation done
```

Figure 21. Operation room model [8].

When the model was verified, we found the following error:

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

Starting simulation. Stop at time: Infinity

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

Aborting simulation...
```

Figure 22. Error detected: Unlinked ports.

Studying the model, we could see that the coupled model addresses the ***Operation*** class. Here, ***done*** is defined as an output port, while in source code it is defined as an input port.

Another model verified represents a switching station for mobile phones. The coupled model representing the station has the following structure:

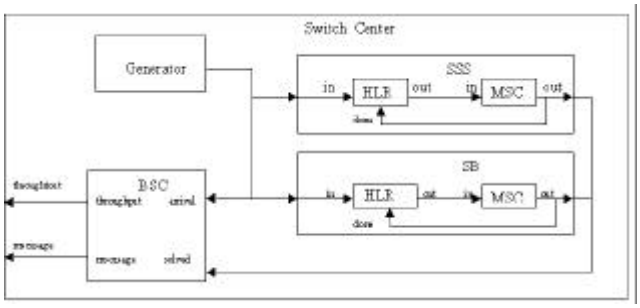


Figure 23. Switching station coupled model [8].

HLR model is in charge of managing the call flow. When it receives a request, it enqueues it, and advances the switching time associated to the call. When the call is sent, it will receive an ACK signal. The MSC model is in charge of connecting a call. It runs individual switching tasks in a non preemptive fashion. If the model receives a new call and it is already processing one, the new one is discarded. The BSC is in charge of computing the numbers of call finished in a given time unit. It computes the usage ratios for the MSC and transfers the outputs with a given frequency.

When this 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 24. Error detected: Unlinked ports.

In this case, the ***HLR*** class includes an input port named ***stop***. This port was not used in the MA file.

The last model we will describe represents a resin processing tank. The different types of raw material are put into a storage tank. When the raw material is ready, a control system is activated, and a mix tank and mould tank will heat up to certain temperature. After heating, the material should be mixed for certain time, and then cast to the mould tank. The mix and mould tanks are only able to process one batch of raw material at a time, so new material is stored in the storage tank.

The structure of this coupled model can be seen in the following figure:

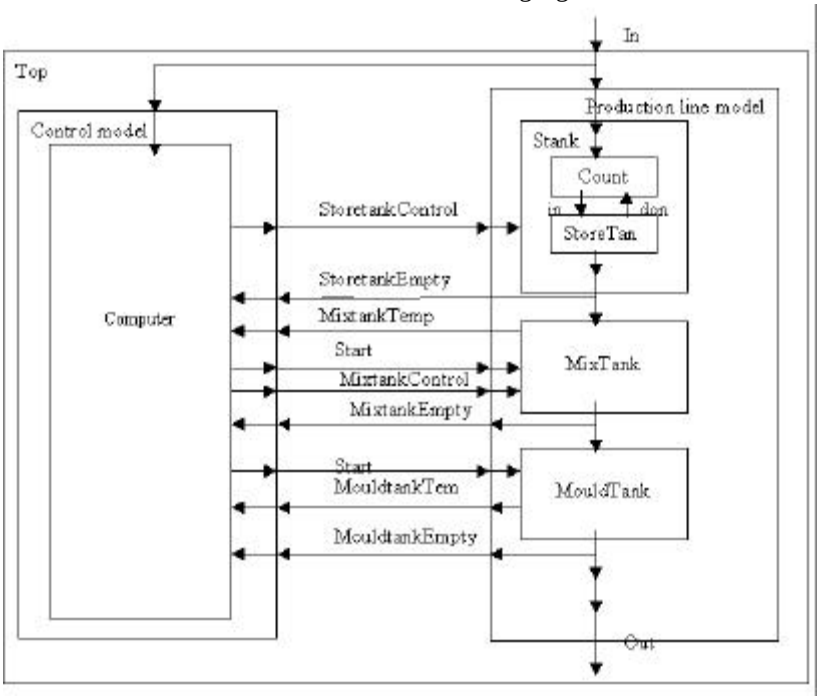


Figure 25. Mix Tank structure.

```
[top]
components : controlsystem   prodline
out : out     in : in
Link : in in@controlsystem
Link : in in@prodline
Link : storetankEmpty_out@prodline
       storetankEmpty_in@controlsystem
Link : mixtankTemp_out@prodline
       mixtankTemp_in@controlsystem
Link : mixtankEmpty_out@prodline
       mixtankEmpty_in@controlsystem
Link : mouldtankTemp_out@prodline
       mouldtankTemp_in@controlsystem
Link : mouldtankEmpty_out@prodline
       mouldtankEmpty_in@controlsystem
Link : out@controlsystem  start_in@prodline
Link : storetankControl_out@controlsystem
       storetankControl_in@prodline
Link : mixtankControl_out@controlsystem
       mixtankControl_in@prodline
Link : out@prodline out

[controlsystem]
components : compu@Computer
in : in   storetankEmpty_in mixtankTemp_in
in : mixtankEmpty_in mouldtankTemp_in
in : mouldtankEmpty_in
out : out  mixtankControl_out storetankControl_out
Link : in  in@compu
Link : storetankEmpty_in storetankEmpty_in@compu
Link : mixtankTemp_in mixtankTemp_in@compu
Link : mixtankEmpty_in  mixtankEmpty_in@compu
Link : mouldtankTemp_in mouldtankTemp_in@compu
Link : mouldtankEmpty_in mouldtankEmpty_in@compu
Link : out@compu out
Link : storetankControl_out@compu
       storetankControl_out
Link : mixtankControl_out@compu mixtankControl_out
```

```
[prodline]
components : Stank mtk@mixtank motk@mouldtank
Out : out  storetankEmpty_out mixtankTemp_out
out : mixtankEmpty_out mouldtankTemp_out
out : mouldtankEmpty_out
in : start_in mixtankControl_in
in : storetankControl_in   in
Link : in in@stank
Link : start_in start_in@mtk
Link : start_in start_in@motk
Link : storetankControl_in
       storetankControl_in@stank
Link : mixtankControl_in
       mixtankControl_in@mtk
Link : out@stk in@mtk
Link : out@mtk in@motk
Link : out@motk out
Link : out@stank storetankEmpty_out
Link : mixtankTemp_out@mtk mixtankTemp_out
Link : mixtankEmpty_out@mtk mixtankEmpty_out
Link   :   mouldtankTemp_out@motk   mouldtank-
Temp_out
Link : out@motk mouldtankEmpty_out

[Stank]
components : cou@count stk@storetank
in : in  storetankControl_in
out : out
Link : storetankControl_in
       storetankControl_in@stk
Link : in  in@cou
Link : out@cou in@stk
Link : out@stk out
Link : out@stk done@cou
```

Figure 26. Mixing tank coupled model.

In this case, when the model was verified, the following error raised:

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

Starting simulation. Stop at time: Infinity

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

Aborting simulation...
```

Figure 27. Error detected: Unlinked ports.

The ***Computer*** model defined an input port named ***out***. Nevertheless, this port was used as an output port in the coupled model defined in the previous figure.

5. Conclusion

We have presented some of the verification features of CD++, a toolkit for DEVS modelling and simulation. The tool was built using the DEVS formal modelling paradigm, improving the safety and development times of the simulations.

We used existing models, and found inconsistencies. Some of the errors found do not affect the model execution, but others could generate logical errors difficult to be found. Besides, we were able to be sure that no errors exist in other models.

Verification tools can improve the security and cost in the development of the simulations. The main gains are in the testing and maintenance phases, the more expensive for these systems. The use of a formal approach like DEVS made easy the development of the verification tools. The next step is to attack automatic verification based on the mathematical properties that can be derived from DEVS specifications.

The tools are public domain and can be obtained at "http://www.sce.carleton.ca/faculty/wainer/celldevs".

References

[1] ZEIGLER, B.; KIM, T.; PRAEHOFER, H. Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems. Academic Press. 2000.

[2] WAINER, G.; GIAMBIASI, N. "Timed Cell-DEVS: modelling and simulation of cell spaces ". In "Discrete Event Modeling & Simulation: Enabling Future Technologies", Springer-Verlag. 2001.

[3] WAINER, G.; BARYLKO, A.; BEYOGLONIÁN, J. "Experiences with DEVS modelling and simulation". In *IASTED Journal on Modelling and Simulation*. March 2001.

[4] CHRISTEN, G.; DOBNIEWSKI, A.; WAINER, G. "Defining DEVS models with the CD++ toolkit". In *Proceedings of European Simulation Symposium*. Marseilles, France. 2001.

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

[6] AMEGHINO, J.; WAINER, G. "Application of the Cell-DEVS paradigm using CD++". In *Proceedings of the 32nd SCS Summer Multiconference on Computer Simulation*. Vancouver, Canada. 2000.

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

[8] CHEN, S.; DU, H.; HU, R.; XIE, P.; WAINER, G. "Modelling and simulation of DEVS models". Internal report. SCE, Carleton University. 2001.

Gabriel A. Wainer received the M.Sc. (1993) and the Ph.D. degrees (1998, with highest honours) at the Universidad de Buenos Aires, Argentina, and DIAM/IUSPIM, Université d'Aix-Marseille III, France. He is Assistant Professor at the Systems and Computer Engineering, Carleton University (Ottawa, Canada). He was Assistant Professor at the Computer Sciences Dept. of the Universidad de Buenos Aires, Argentina, and a visiting research scholar at the Arizona Center of Integrated Modelling and Simulation (University of Arizona). He has published more than 50 articles in the field of operating systems, real-time systems and Discrete-Event simulation. He is author of a book on real-time systems and another on Discrete-Event simulation. He has been the PI of several research projects, and participated in different international research programs. Prof. Wainer is a member of the Board of Directors of The Society for Computer Simulation International (SCS). He is the coordinator of a group on DEVS standardization. He has been appointed Associate Editor of the Transactions of the SCS. He is also a Coassociate Director of the Ottawa Center of The McLeod Institute of Simulation Sciences.

LILIANA MORIHAMA received her B.Sc. degree at the Universidad de Buenos Aires in 1999. At present she is a M.Sc. student at the same University, and an independent software developer in Buenos Aires, Argentina.

VIVIANA PASSUELLO received her B.Sc. degree at the Universidad de Buenos Aires in 1998. At present she is a M.Sc. student at the same University, and an independent software developer in Buenos Aires, Argentina.