# Modeling State-Based DEVS Models in CD++

Gastón Christen Alejandro Dobniewski Gabriel Wainer

Computer Science Department
Universidad de Buenos Aires
Planta Baja. Pabellón I.
Ciudad Universitaria (1428)
Buenos Aires. Argentina.

Dept. of Systems and Computer Engineering
Carleton University
4456 Mackenzie Building
1125 Colonel By Drive
Ottawa, ON. K1S 5B6. Canada.
E-mail: gwainer@sce.carleton.ca

**Keywords:** DEVS, DEVS Graphs, CD++.

**Abstract:** *We introduce the features of CD++, a toolkit for modeling and simulation based on the DEVS formalism. We show recent extensions that permit the users to write the models using state machines, and we show how to use it through application examples. The use of this formal approach allowed developing safe and cost-effective simulations, reducing significantly the development times of simulation software.*

## 1. INTRODUCTION

In recent years, several efforts have been devoted to define new modeling paradigms, allowing improving the analysis of complex dynamic systems through simulation of these models. DEVS (Discrete Event systems Specifications) allows modular description of models that can be integrated using a hierarchical approach [1, 2].

We have built a toolkit with the goal of developing models based on the DEVS formalism and simulating them. The core of the toolkit is the CD++ environment [3], which implements the DEVS theory. Here, we focus in the development process of simulated models using a graph-based definition of DEVS models. Graphical-only notations have some limitations in building complex systems [4]. Therefore, we permit the users to define intermediate functions in C++. Likewise, if the complexity of the models is such that graphical notations are not adequate, the models can be still defined using a standard representation in C++, thus permitting the integration of simple state-based specifications with more complex models defined in a programming language.

## 2. THE DEVS FORMALISM

DEVS was originally defined in the '70s as a discrete-event modeling specification mechanism [1]. 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.

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:

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

Each model is seen as having input (***X***) and output (***Y***) 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 ($\boldsymbol{\delta_{int}}$) defines the behavior under such inputs. The internal transition function ($\boldsymbol{\delta_{ext}}$) is activated after the lifetime of the present state has been consumed, which is defined by the time advance (**ta**) 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 ($\boldsymbol{\lambda}$), which executes before the internal transition.

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

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

Each model is seen as having input (***X***) and output (***Y***) ports to communicate with other models. Coupled models are defined as a set (**D**) of basic components ($\mathbf{M_i}$ atomic or coupled), which are interconnected. The translation function ($\mathbf{Z_{ij}}$) is in charge of converting the outputs of a model into inputs for the others. To do so, an index of influencees ($\mathbf{I_i}$) is created for each model. 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 $I_i$.

The CD++ tool [3] allows defining models following these specifications. 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++. New atomic models must be incorporated into this class hierarchy as subclasses of the Atomic Model class.

Defining models in C++ allow the users to have great flexibility to define behavior. Nevertheless, a non-experienced user can have difficulties in defining models using this approach. Likewise, having a graphical specification enhances the interaction with customers during system specification [5]. Graph-based notations have the advantage of allowing the modeler to think about the problem in a more abstract way. Therefore, we have used an extended graphical notation to allow the user define atomic models behavior [6]. Each DEVS graph defines the state changes according to internal and external transition functions, and each is translated into an analytical definition. DEVS graphs can be formally defined as:

$$GGAD = < X_M , S, Y_M , \delta_{int}, , \delta_{ext} , \lambda, D >$$

$X_M = \{(p,v) | p \in IPorts, v \in X_p \}$ set of input ports;

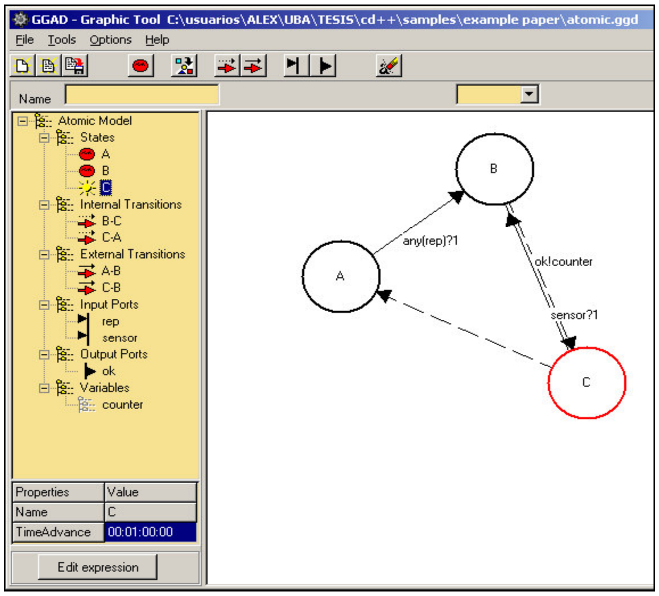
$Y_M = \{(p,v) | p \in OPorts, v \in Y_p \}$ set of output ports;

$S = B \times P(V)$ states of the model,

$B = \{ b \mid b \in Bubbles \}$ set of model states.

$V = \{ (v,n) \mid v \in Variables, n \in R_0 \}$ intermediate state variables of the model and their values.

$\delta_{int}$, $\delta_{ext}$, $\lambda$, and $D$ have the same meaning as in traditional DEVS models.Each model is defined by a unique identifier, and it can include a graphical specification or C++ code. When we use the state-based notation, states are represented by bubbles including an identifier and a state lifetime. When the lifetime is consumed, an internal transition function is executed.



| Properties | Value |
|---|---|
| Name | C |
| TimeAdvance | 00:01:00:00 |

Figure 1: An atomic model defined as a DEVS graph.

Figure 1 shows a simple atomic model using this notation. The model includes three states: A, B and C. Dotted lines represent internal transitions, while full lines define external transitions. In this case, if the model is in state A and it receives an external event through the *rep* input port (shown in the left panel), the *any* function is evaluated. If the result of this evaluation is 1, the model changes to the state B. While in B, the model waits its lifetime to be consumed. It then executes the output function, which will send the value of the intermediate state variable *counter* through the output port *ok*. After that, the internal transition function executes, and the model changes to the state C.

While graphical models are best for design, simulators require text representations (or extended notations in XML or other high level modeling languages). Thus, we defined a textual representation for each of the elements in the graphical notation. Every GGAD model is converted into this language, called GADscript. GGAD models are then simulated by CD++, as shown in Figure 2.

In order to translate DEVS graphs into text specifications, we use the following syntax [7]:

`[modelname]` defines the atomic or coupled model name, which will be used subsequently. Model states are declared as: `state: state1 state2 ...`

States are associated to a time advance value. This attribute are initialized with the name of the object and the list of valid attributes for that object, as follows:

```
state1 : time-expression
```

One of the states must be declared as the initial state of the model: `initial: statename`

Then, I/O ports are declared either as follows:

```
in :  inport1 inport2 ...
out : outport1 outport2 ...
```

Temporary variables are declared by:

```
var : var1 var2 var3 ...
```

and they can be optionally initialized as:

```
var1 : value1
var2 : value2
```

The internal transitions use the following syntax:

```
int:source destination [outport!value]* ( { (action;)* } )?
```

Here, the source and destination represent the initial and final states associated with the execution of the transition function. As the output function should also execute before the internal transition, an output value can be associated with the internal transition. Finally, if the user wants to execute a complex function to generate an output, one or more actions can be defined. External transitions are defined using the following expression:

```
ext : source destination EXPRESSION ( { (action;)* } )?
```

In this case, the expression should hold, and then the model will change from state *source* to state *destination*, while also executing one or more actions.

As we can see, transition functions have associated actions to modify the value of the intermediate variables in the model. Each action is an assignment to a variable of the result of evaluating an expression. Actions are defined using simple mathematical expressions. They can include constants, variables and functions. The value of a message in a port can be referenced by the name of the port, like a variable.Functions are actually evaluated in native C++ code. It is possible to incorporate new functions by programming them into CD++. The current version of the simulator includes a library of basic arithmetic and Boolean functions, presented in the following figure.

| Function | Description |
|---|---|
| Add(n1,n2) | sum of n1 and n2. |
| And(n1,n2) | true if both arguments are true. |
| Any(port) | true if the port has a valid value. |
| Between(n1,n2,n3) | true if n1 <= n2 <= n3 |
| Compare(n1,n2,n3,n4,n5) | n3, n4 or n5 if n1 is reater, equal or less than n2 respectively. |
| Divide(n1,n2) | n1 divided by n2 |
| Equal(n1,n2) | true if n1 = n2, false else. |
| Greater(n1,n2) | true if n1 > n2, false else. |
| Less(n1,n2) | true if n1 < n2, false else. |
| Minus(n1,n2) | n1 - n2 |
| Multiply(n1,n2) | n1 multiplied by n2 |
| Not(n) | negation of n. |
| NotEqual(n1,n2) | true if n1 is different of n2. |
| Or(n1,n2) | true if one of the parameters true. |
| Pow(n1,n2) | returns n1 power n2 |
| Rand(n1,n2) | random value between (n1,n2) |
| Value(n) | returns n. |

Figure 2: Action functions available.

## 3. MODELS OF DISCRETE EVENT SYSTEMS

We will exemplify the use of the toolkit by using a model of a traffic light radar. This device takes photos of cars violating red lights or speeding in a crossing. For doing this, it uses a radar sensor and a presence sensor. Figure 4 shows the structure of the topmost model in this application.

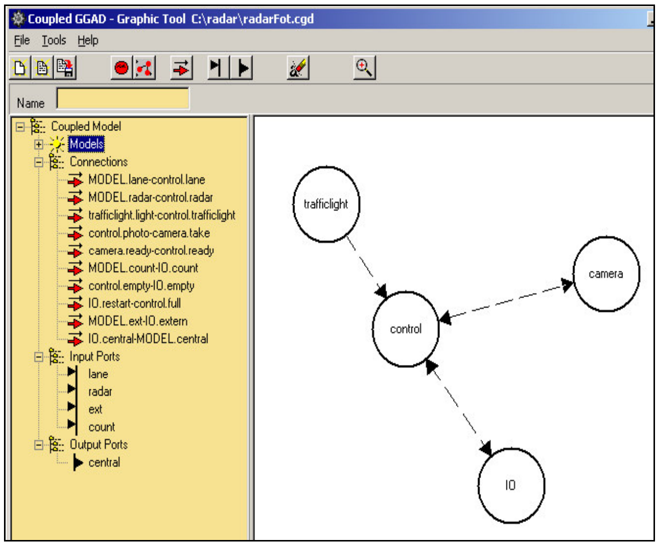


Figure 3: Structure of the photographic radar model.

The top level coupled model is composed is composed of the following submodels.

| Model | Description |
|---|---|
| Control | Manages the system. |
| TrafficLight | Cycles the red, yellow and green lights. |
| Camera | Takes the shots. |
| IO | Does communications tasks. |

The *Control* atomic model instructs the *Camera* atomic model to take a photo. This coupled model uses two external ports. The first one, called *lane*, detects the presence of a car over the pedestrian zone at the corner of the street. A second one, called *radar*, represents a sensor detecting a car running faster than the maximum speed. This sensor is active when the *lane* sensor warns about a car over the pedestrian zone and the traffic light is red (*traffic light* atomic model). If the memory of the digital *camera* is full, it warns the *Control* atomic model to request to download the pictures and to empty the memory *IO* atomic model.

As we can see in the leftmost panel, this model uses the following input-output ports:

| Port | Description |
|---|---|
| Lane (in) | Receives a signal alerting the presence a vehicle on the pedestrian crossing. |
| Radar (in) | Alerts a vehicle over the speed limit. |
| Ext (in) | Commands controller |
| Count (in) | Count memory available. |
| Central (out) | Alerts to the Central. |

Using this specification, the graphic tool generates a CD++ specification of the coupled models, as showed in the following figure. As we can see, the specification represents the model coupling (*link* sentences), and it defines the atomic models as *cdd* files. Each of these files contain the description of the atomic models using GGAD notation.

```
[Top]
components : trafficlight@GGad control@GGad
camera@GGad IO@GGad
out : central      in : lane radar ext count
Link : lane lane@control
Link : radar radar@control
Link : light@trafficlight trafficlight@control
Link : photo@control take@camera
Link : ready@camera ready@control
Link : count count@IO
Link : empty@control empty@IO
Link : restart@IO full@control
Link : ext extern@IO
Link : central@IO central
[trafficlight] source : trafficlight.cdd
[control]      source : control.cdd
[camera]       source : camera.cdd
[IO]           source : IO.cdd
```

Figure 4: Text specification of the radar coupled model

Let us show the details of the *Control* atomic model. Figure 6 represents its graphical design:

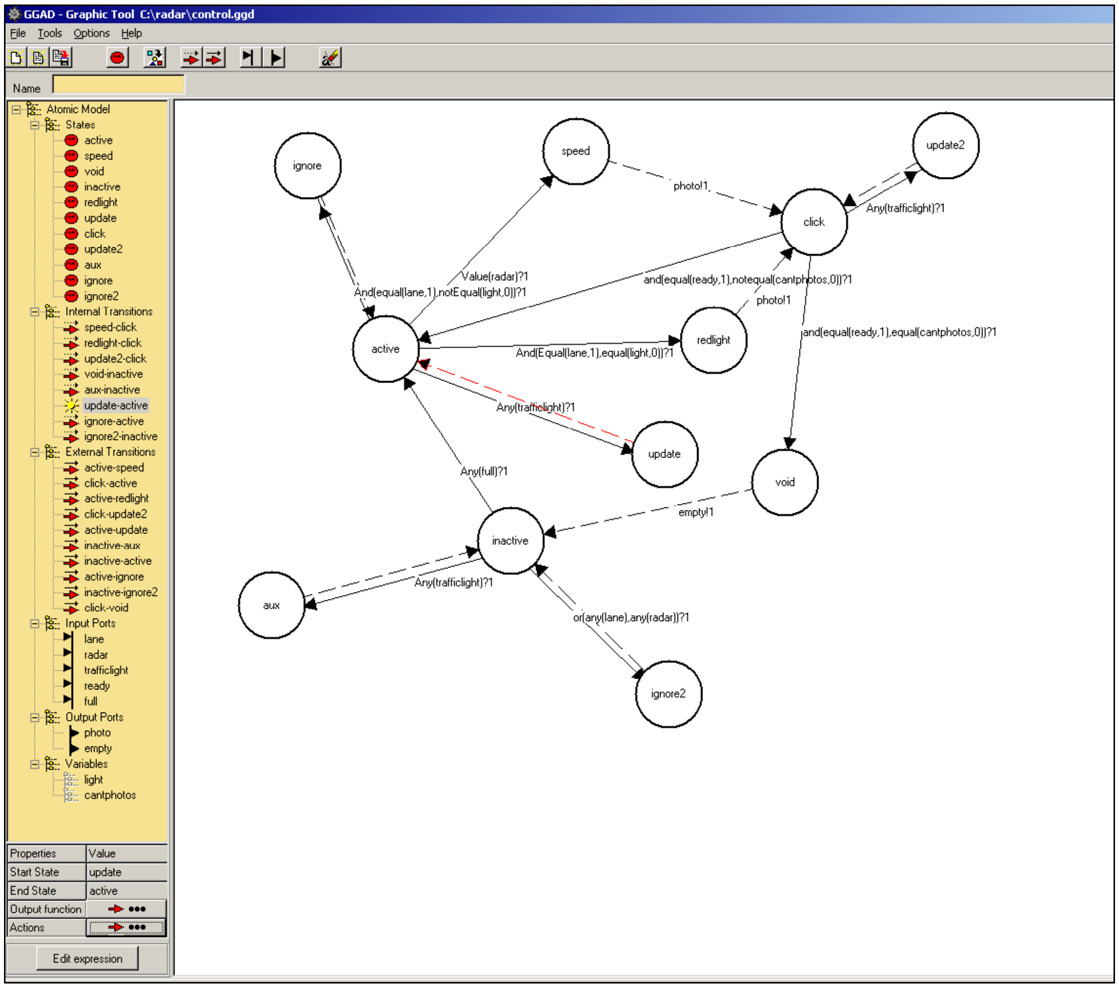


Figure 5: DEVS graph for the Control model.

This atomic model starts in the *Active* state. If the *Radar* ports receive messages with high speed or over the lane when the traffic light is in red, then it commands the *Camera* to take a shot and returns to *Active*. If the remaining shots counter gets to zero sends a warning on Empty port and waits the memory to be emptied on the port *Full*. Every time the traffic light cycles the model updates the value of the variable *light*. The graphic tool generates the following GADscript code that can be simulated using CD++.

```
[control]
in: lane radar trafficlight ready full
out: photo empty
var: light qph
state: active speed void inactive redlight up-
date click update2 aux ignore ignore2
initial: active
int: speed click photo!1 {qph = minus(qph,1);}
int: redlight click photo!1 {qph = mi-
nus(qph,1);}
...
ext: active speed Value(radar)?1
ext: click active and(equal(ready,1),
notequal(qph,0))?1
...
active: infinity
speed:00:00:00:00
void:00:00:00:00
inactive: infinite
...
light:0
qph:2
```

Figure 6: Text specification of the control atomic model

We also need to define the remaining submodels. We will briefly describe their behavior (for a detailed description, see [8]):

- **Camera** is in *StdBy* state until receives the shot order on the port *Take*. The time spent storing the picture is represented by the time advance of state *Prepare*. When the picture is stored, a message is emitted in the port *Ready*.
- **Traffic Light** is a simple timed cycle between the three states of the traffic lights. Each change of lights is announced on the port *light*. There are no input ports.

- **IO** takes the responsibility of communicating with the traffic central, in order to inform the lack of memory available for new pictures and downloading them. If a signal is received on port *Empty*, it sends the information through the port *central* to communicate the lack of memory available for new pictures.

After defining each submodel, we can execute the simulation in order to analyze the behavior of this system. We log the following information:

- Event Types.
- Simulated time.
- Details (varies upon the event being considered):
  - Model initialization (type : C): `init state, { (var, initial value),(var2, initial value),... }`
  - Input Message (type : ?): recorded every time a message is received on an input port. `{port receiving the message, value received}`.
  - Output Message (type : O): recorded each time the output function is executed. `{output port, value}`.
  - Internal Transition Function (type : I): {`init, final, {(var-before,val),(var-after, val) ,...} }`.
  - External transition function (type : E): `init state, final state, {(var1, value), (var2, value), ... }`

By running a simulation of this model we can analyze the system behavior in detail. For instance, we simulated the complete system using the following inputs:

| Time | Port | Value |
|---|---|---|
| 00:00:02:00 | lane | 1 |
| 00:00:04:00 | radar | 1 |
| 00:00:12:00 | lane | 1 |
| 00:00:15:00 | lane | 1 |
| 00:00:19:00 | lane | 1 |
| 00:00:24:00 | radar | 1 |
| 00:00:26:00 | ext | 1 |
| 00:00:26:00 | count | 10 |
| ... | | |

The following log files show the execution of the four atomic models that compose the coupled system. By analyzing these log files we can completely study the model behavior and their interaction.

**Control**

```
C 00:00:00:000 : active , (qph=2) (light=0)
? 00:00:02:000 : lane , 1
E 00:00:02:000 : active ,redl(qph=2) (light=0)
O 00:00:02:000 : photo , 1
I 00:00:02:000 : redl,click (qph=1),(light=0)
? 00:00:04:000 : radar , 1
? 00:00:04:000 : ready , 1
E 00:00:04:000 : click, active(qph=1),(light=0)
? 00:00:05:000 : trafficl, 2
E 00:00:05:000 : active,update(qph=1),(light=2)
I 00:00:05:000 : update,active(qph=1),(light=2)
? 00:00:06:000 : trafficl , 0
```

```
E 00:00:06:000 : active, update(qph=1)(light=0)
I 00:00:06:000 : update,active(qph=1),(light=0)
...
? 00:00:19:000 : lane , 1
E 00:00:19:000 : active,redl(qph=1),(light=0)
O 00:00:19:000 : photo , 1
I 00:00:19:000 : redl , click (qph=0),(light=0)
? 00:00:21:000 : ready , 1
E 00:00:21:000 : click , void (qph=0),(light=0)
O 00:00:21:000 : empty , 1
I 00:00:21:000 : void,inact(qph=0),(light=0)
? 00:00:22:000 : trafficl , 1
E 00:00:22:000 : inact,aux(qph=0),(light=1)
I 00:00:22:000 : aux,inact(qph=0),(light=1)
? 00:00:24:000 : radar , 1
E 00:00:24:000 : inact,ignore2(qph=0) (light=1)
I 00:00:24:000 : ignore2,inact(qph=0) (light=1)
? 00:00:26:000 : full , 10
E 00:00:26:000 : inact,active(qph=10) (light=1)
? 00:00:27:000 : trafficl , 2
E 00:00:27:000 : active,update(qph=10)(light=2)
I 00:00:27:000 : update,active(qph=10)(light=2)
? 00:00:28:000 : trafficl , 0
E 00:00:28:000 : active,update(qph=10)(light=0)
...
```

**Camera**

```
C 00:00:00:000 : stdby ,
? 00:00:02:000 : take , 1
E 00:00:02:000 : stdby , run
I 00:00:02:000 : run , prepare
O 00:00:04:000 : ready , 1
I 00:00:04:000 : prepare , stdby
? 00:00:19:000 : take , 1
E 00:00:19:000 : stdby , run
I 00:00:19:000 : run , prepare
O 00:00:21:000 : ready , 1
I 00:00:21:000 : prepare , stdby
...
```

**IO**

```
C 00:00:00:000 : receive ,  (qph=0)
? 00:00:21:000 : empty , 1
E 00:00:21:000 : receive , send (qph=0)
O 00:00:21:000 : central , 1
I 00:00:21:000 : send , receive (qph=0)
? 00:00:26:000 : count , 10
? 00:00:26:000 : extern , 1
E 00:00:26:000 : receive , resume (qph=10)
...
```

**TrafficLight**

```
C 00:00:00:000 : green ,
O 00:00:05:000 : light , 2
I 00:00:05:000 : green , yellow
O 00:00:06:000 : light , 0
I 00:00:06:000 : yellow , red
O 00:00:11:000 : light , 1
I 00:00:11:000 : red , green
O 00:00:16:000 : light , 2
I 00:00:16:000 : green , yellow
O 00:00:17:000 : light , 0
I 00:00:17:000 : yellow , red
O 00:00:22:000 : light , 1
I 00:00:22:000 : red , green
...
```

As we can see, the control model starts in the state "active". There are 2 photos left (qph = 2) and the traffic light is in red (light = 0, the light is encoded as red=0, green=1, yellow=2). The camera starts in state "stdby" which means it is ready to take a shot.

A message is received after 2 seconds on input port *lane*, with value 1. This event fires the external transition function. A suitable transition is found by the simulator from state *active* to state *redlight*. This state represents the fact that a car has crossed the lane when the light was red. An instantaneous (i.e. ta=0) internal transition is fired to send a message to the camera to take a photo. The message is generated by the output function through port *photo*. This output port is connected at the coupled model level with the atomic model of the camera. As a result, When it receives a message on port *take* (from *control* model) it simulates the process of taking the photo and also of advancing to the next memory position to store a new photo. After evaluating the output function an internal transition is fired from state *redlight* to *click*, decrementing the count of remaining photos. A while later a message from the radar alerts of a car exceeding the speed limit at the same time the camera ends the previous request. Because the camera was not ready the car must be ignored. After that the control returns to *active*, so it can process new infractions. The traffic light sends its first message to indicate a light change. The control updates its internal variable *trafficlight* to match the status received. At 00:00:19:000 we have a real infraction; a car over the lane while the light is red. The *control* model commands the *camera* model to get a shot of the car. Then the model goes to state *click* to wait to the camera to complete the take.

At 00:00:21:000 the camera sends its *ready* message, and the controller decrements the count of photos and discovers that there are no more available photos, changing to state *void*. It could be observed that the controller takes the last photo and changes to the *void* state, sending a message on the port *empty*. This request is received by the model *IO*, which on time 00:00:26 warns on port *full* that the pictures are downloaded and there is enough more to store 10 new photos. On that lapse the controller was on *inactive* state, and only process messages incoming from the traffic lights to keep the light active.

The *trafficlight* model just cycles over the three lights. Note that the yellow light has a 1 second time advance while the other two has a 5 second time advance. There are no variables or input ports.

The *IO* model is activated when the control model want to communicate the central that needs is out of photos. The output port *central* of *IO* is connected at the coupled model level to the outside of the system. Now, *IO* receives a message from *count* of 10 new photos, so it passes the news to the control model through output port *restart*, which is connected to input port *full* of control.

## 4. 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 development time of simulations. The tool was used to develop different application examples, which allowed us to show its flexibility.

Several types of models can be integrated in an efficient fashion, allowing multiple points of view to be analyzed using the same model. The tools are public domain and can be obtained at http://www.sce.carleton.ca/faculty/wainer/celldevs. At present we started studying the use of DEVS graphs to analyze temporal constraints of the DEVS models involved. In such a way, we will be able to provide analytical tools to study the execution times of the time constraints of the models, while having facilities to analyze the model execution in detail by analyzing the simulated results. We are also using the Eclipse platform [9] to integrate the existing tools and facilitate the definition of applications, their execution and analysis.

## REFERENCES

[1] ZEIGLER, B. "Theory of modeling and simulation". *Wiley*, 1976.

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

[3] WAINER, G. "CD++: a toolkit to define discrete-event models". G. Wainer. *Software, Practice and Experience*. Wiley. Vol. 32, No.3. pp. 1261-1306. November 2002.

[4] MARCA, D. A., McGOWAN, C. L. "SADT - Structured Analysis and Design Technique", *McGraw-Hill*, New York, New York, 1988.

[5] SOMMERVILLE, I. "Software Engineering". 6th Edition. Addison-Wesley. 2000.

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

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

[8] CHRISTEN, G.; DOBNIEWSKI, A. "Extending the CD++ toolkit to define DEVS graphs". M. Sc. Thesis. Computer Science Dept. Universidad de Buenos Aires. 2003.

[9] OBJECT TECHNOLOGY INTL. INC. "Eclipse Platform Technical Overview". http://www.eclipse.org/whitepapers/eclipse-overview.pdf. 2001.