Converting high level models into DEVS modeling and simulation applications

Gabriel A. Wainer

Department of Systems and Computer Engineering. Carleton University. 1125 Colonel By Dr. Ottawa, ON. K12 5B6. Canada.

Abstract We discuss a number of methods for converting high level modeling formalisms and languages into lower level discrete-event systems specifications using the DEVS formalism. We present the implementation of such methods in the CD++ open source toolkit, and discuss different case studies. We focus on a variety of methods, ranging from Petri Nets and Finite State Machines up to Modelica and advanced Traffic modeling languages, showing the generality of DEVS based solutions, and the definition of user libraries in different domains.

1 Introduction

As discussed in the Introduction chapter, I have attended SummerSim since 1999, presenting a variety of results of our research to the Modeling and Simulation community. Besides making lifetime friends and colleagues, I was able to witness (and collaborate with) research on different modelling and simulation methodologies. In the beginning, the conference focused in a number of methodologies and applications for continuous and discrete-event simulation, which were mature and well established. In parallel, there were a large number of research efforts focusing on modeling formalisms, and a number of those investigations focused on the transformation of models into DEVS [Zeigler et al., 2000]. In this chapter, we introduce our efforts in this field, which are summarized in Figure 1 (the reader can find information about these topics at htpp://cell-devs.sce.carleton.ca/ars).

Although the research in modeling methodologies in our laboratory includes numerous areas, ranging from Graphical User Interfaces (CD++Builder), Real-Time Simulation (RT-DEVS, Embedded CD++, E-CD Boost, I-DEVS, I-DEVS Schedulability, E-CDBoost), Parallel and Distributed Simulation (Parallel CD++, Conservative and Time-Warp/PCD++, Mashup SOA CD++, Remoting .NET DEVS, RESTful middleware, multi-core optimistic PCD++, DEVS as a service, SamSaas Mashups), our main focus has been on theory of modeling and simulation. This included the definition of Cell-DEVS and its abstract simulator and its extension to

Parallel Cell-DEVS and Quantized Cell-DEVS, as well as the first definition of Activity Tracking as a method to reduce cell activation in Cell-DEVS models, and the extension to N-Dimensional Cell-DEVS models. We defined the ATLAS traffic language and the ATLAS traffic simulation compiler. We also worked in the transformation of different methodologies into DEVS and Cell-DEVS. For instance, we showed how to transform Petri Nets [Jacques and Wainer, 2002], Finite State Machines [Zheng and Wainer, 2003]. We also worked in various extensions to the DEVS formalism and proved important properties with the Stochastic DEVS formalism [Castro et al., 2010], how to check models based on the Rational Time-Advanced DEVS [Saadawi and Wainer, 2010], and how to model real-time systems with constraints using Imprecise RT DEVS. We also introduced formal verification of Hybrid DEVS models, presented finite forkable DEVS and DEVS with uncertain inputs. We were also able to show the transformation of other methods, including Finite Element (FEM), Lattice Gas and Computational Fluid Dynamics. As research on modeling of continuous and hybrid systems advanced, the concept quantization of the state variables obtaining a discrete event approximation of the continuous system proved to be an important method to allow the integration of continuous and discrete models. This method that could be defined using DEVS [Kofman and Junco, 2001], and was later extended to the family of QSS methods [Pietro et al, 2018]. We used these methods to define models of continuous systems, including VHDL-AMS, and later Bond Graphs [D'Abreu and Wainer, 2006] and Modelica [Chechiu and Wainer, 2005].

Figure 1. Modeling Methodologies at the Advanced Real-Time Simulation Lab

In this chapter, we summarize some of these efforts, presenting a tutorial on different modeling methods and their translation into DEVS. Most of the research presented has been discussed in numerous articles and books [Wainer 2009], and this serves as a generic introduction for the readers interested in advancing in this research area. The tools are open source, and all the models presented here are publicly available.

We first discuss the definition of simple Finite State Machines and its definition as DEVS models. Then, we introduce Timed Petri Nets, and show a library of DEVS models for modeling and simulating them. After, we show a method we defined, in which we have Timed Automata models converted into DEVS (and vice-versa), allowing advanced model checking using tools like UPPAL, in particular using hybrid models that can be defined through quantization of the input/output signals. We then elaborate on the definition of Bond Graph models and their implementation using Quantized DEVS models. Bond Graphs were used to build a Modelica compiler, which is described next. We also built a library to build digital circuits using VHDL-AMS, which is described next. Finally, we present ATLAS, a specification language defined to depict city sections and to model and simulate traffic flow. Road segments are defined by their size, number of lanes, traffic direction, maximum speed, etc. Once the city section is outlined, the constructions are translated into Cell-DEVS models, and the traffic flow is automatically set up. The rule generation for describing the traffic behavior is based on macro templates, enabling changes in the model implementation in a flexible way.

2 Modeling Finite State Machines in DEVS

Finite State Machines have been used in systems engineering applications as they can represent complex artificial devices in an abstract fashion. They are characterized as abstract mathematical entities that can take a finite number of states, and they respond to external inputs to trigger transitions between states. A deterministic FSN can be formally defined, using a systems theoretical notation, as follows [Hopcroft and Ullman, 1979]:

$$FSM = < S, X, Y, \delta, \lambda >$$

Where	X:	finite input set
	Y:	finite output set
	S:	finite state set
	δ:	$X \, x \, S \rightarrow S$ the next state function
	λ:	S → Y is the output function (Moore machine) or $X \, x \, S \rightarrow Y$ (Mealy machine)

In [Zheng and Wainer, 2003] we showed how to define FSM as DEVS. The basic idea is to represent the behavior of a generic state as an Atomic model, and then to build the FSM by combining a number of those atomic models into a coupled model representing the FSM. All the states in a FSM are encoded as integers, and a unique global value, *stateCode,* is assigned to each state. A *phase* is used to indicate if a given state is active or passive (and only one state is can be active at a time, according to FSM semantics). The events are encoded as integers as well.

The formal DEVS specification for the FSM for the atomic model *State* is:

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

```
X = {eventIn, transitionIn}
Y = {stateOut, transitionOut }
S = { phase, events[],isEvent stateCode, stateValue, nextActiveState }
δext ((phase, events[],isEvent stateCode, stateValue, nextActiveState), e, x)
    case phase
        active:
            if x is from eventIn
                get nextActiveState from events[ ];
                  isEvent = true;
                  ta(active, 0); //trigger an immediate internal event for output
        passive:
                  if x is from transitionIn
        isEvent = false;
        ta(passive, 0);
}

δint (events[ ], stateCode, phase, stateValue, nextActiveState, isEvent)   {
        if (isEvent)  // always passivate after receiving an event
                passivate();
        else        // new state activated by the input from the transition
                ta(active, Infinity);
}

λ(events[ ], stateCode, phase, stateValue, nextActiveState) {
        if (isEvent)          // inform the next active state
           send nextActiveState to port transitionOutput
        else                              // indicate the current state
           send stateValue to port stateOutput
}
```

The state variable *events[ ]* is an array that records the legal events of the state and the associated *nextActiveState*; and *stateCode* includes the unique state code of the state (0 represents the initial state in a FSM). The *phase* can be active or passive; *stateValue* is the assigned output of the state; *nextActiveState* is the state code of the next active state; *isEvent* indicates if an event is received or if a transition signal is received. The *eventIn* receives encoded numbers representing external events. If a state receives a legal event listed in *events[ ]* when it is active, the *stateOut* port sends out the current *stateValue*, and the *transitionOut* port sends out the *nextActiveState* signal to all the *transitionIn* ports in the FSM reporting which state will be active in the next step. A state becomes active if the encoded number received from *transitionIn* port is same as its *stateCode*.

In order to create an FSM we connect various *States*:

- All *transitionOut* and *transitionIn* ports should be connected together inside the FSM.
- All *eventIn* ports of *States* should be connected to an input port of a FSM.
- Each *stateOut* port of *State* could either be connected to an individual output port of a FSM or connected together as one output port of a FSM
- All the states in a FSM are encoded as integer numbers (*stateCode*) during simulation. The state with the *stateCode* 0 is the initial state in the FSM.

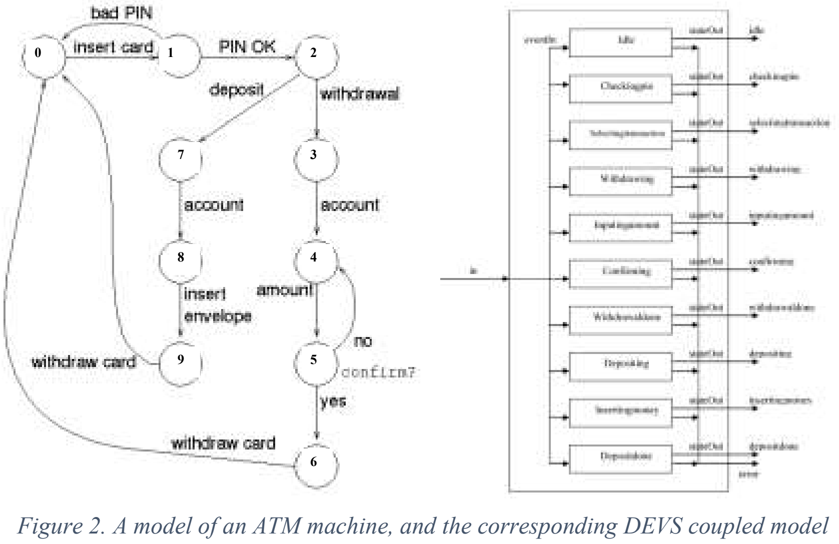


Figure 2. A model of an ATM machine, and the corresponding DEVS coupled model

After executing this model in CD++ with a set of external events, we obtain the results in Figure 3. The left column shows the inputs, and the right column, the outputs obtained when we test it.

As we can see, initially we insert a card, which produces a *checkingpin* output. When we receive *PIN-OK,* we trigger an output in *selectingtransaction* (in this case, a *withdrawal*, which generates a *withdrawing* output). We then choose the account type and the amount. As we can see, the user does not confirm the amount (*confirm_no*), as they made a mistake. This is selected again at 00:80, after which we confirm, withdraw the card, and start the process all over again.

```
00:00:10 in insertCard           00:00:010 checkingpin
00:00:30 in PIN-OK               00:00:030 selectingtransaction
00:00:40 in withdrawal           00:00:040 withdrawing
00:00:50 in choose_account_type  00:00:050 inputingamount
00:00:60 in choose_amount        00:00:060 confirming
00:00:70 in confirm_no           00:00:070 inputingamount
00:00:80 in choose_amount        00:00:080 confirming
00:00:90 in confirm_yes          00:00:090 withdrawaldone
00:00:100 in withdraw_card  ...  00:00:100 idle ...
```

Figure 3.Execution of the ATM machine model in CD++

The library allows the users to define FSM and to use DEVS as the intermediate modeling and simulation language, obtaining an equivalent model that can be executed and combined with other models easily. For instance, it could be combined with a Petri Net model like the one described in the next subsection.

2 Petri Nets

Petri Nets (PN) originally defined by C.A Petri, is a modeling formalism developed to study concurrent systems [Peterson 1977]. They became very popular as they can represent the models both using a formal mathematical notation which is adequate for formal proofs and software development, as well as a graphical representation, which is well suited for communication and analysis.

The static properties of PN are defined by bipartite graphs in which the nodes are called *Places* (represented graphically as bubbles) and *Transitions* (represented as bars), and the links in the graph connect places to transitions (and vice-versa). PN are executed by *firing* transitions that are *enabled*, one at a time, for as long as there is at least one enabled transition. This dynamic behavior is observed by using *tokens* that are added to places, and by the instantaneous firing of the transitions that are enabled to do so. A transition is enabled when they have at least one token on each input link to the transition. When a transition fires, a token is removed from each one of its input places and another token is deposited in each one of the output places. When more than one transition is enabled, the one that fires is selected in nondeterministic fashion.

Through the years, numerous extensions to PN have been introduced, allowing the modeler to define complex behavior [Liu and Zhang, 2018]. Our PN library in DEVS, allows the users to define PN and to run them using a DEVS tool like CD++. Our library supports some of these extensions.

- Inhibitor arcs: a special arc is placed between a place and a transition, and this new arc enables the transition only if the place is empty, as opposed to containing at least one token.

- Multiple Arcs: they indicate that the number of tokens being transferred when firing is more than one.

- Time: PN are *logical time* models, meaning that only the behavioral aspects are considered (deadlock, livelock, concurrent execution, etc.). Timed PN (TPN) introduced the concept of time, and therefore can be used for non-behavioral analysis (performance, throughput, timeliness, etc. [Bowman and Gomez, 2006]). Timing can be represented as delays associated to the model execution.

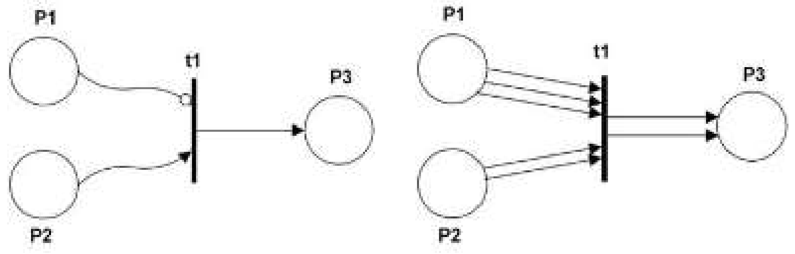


Figure 4. (a) Inhibitor Arc; (b) Multiple Arcs

We built a library of Petri Nets in CD++, whose main components are based on our FSM model in the previous section [Jacques and Wainer, 2002]. The library includes an atomic model to represent a place, and another to represent a transition. The Place atomic model can receive token inputs from transition models, and when a transition fires, we generate an input to tell the place to remove tokens (we support multiple arcs, and timed PN, as the DEVS message also includes a timestamp for the incoming tokens). We report the number of tokens the Place contains so transitions can determine if they are enabled (including TPN, in whose case we only consider enabled transitions if timestamps are equal to or lower than the current time). We consider two kinds of delays:

- Delayed Transition Firing: these are special transitions where the firing condition must be maintained for a certain time before the actual firing occurs.

- Timestamped Tokens: we can use timestamps for the tokens in a place. The firing condition needs enabled tokens whose timestamp is equal to or lower than the current time. When a transition fires, the output tokens should ALSO include a timestamp.

The TPN Place atomic model can be defined as:

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

$\boldsymbol{X} = \{IN \in \boldsymbol{N}^{+} x\ \boldsymbol{R}\}$
$\boldsymbol{Y} = \{OUT \in \boldsymbol{N}^{+}\}$

$S = \{\{tokens \in \textbf{bag(R)}\} \cup \{id \in \mathbf{N}^+\} \cup \{phase \in \{active, passive\} \cup \{id \in \mathbf{R}_0^+\}\}$ where *tokens* is the list of tokens with their timestamps contained in the place, and *id* is the identifier of the place as assigned by the simulator.

```
δext ( (number of tokens N, id, timestamp T) ∈ S, e, x ∈ X) {
        case id
                0:        // generic message
                  add 'N' tokens with timestamp 'T' to tokens;
                  k = max(0 , min_timestamp(tokens) – e );
                  ta (active, k);

                != 0:  // specific message
                    id matches id of this place?
                        no:          discard the message
                yes: if there are enough enabled tokens
                            decrement tokens by the number specified
                  ta(active, 0)
} // end of δext

δint ((number of tokens N, id, timestamp T) ∈ S) {
        if there are tokens with timestamp larger than the current time
                no: passivate
                yes: ta (active, min_timestamp(tokens) – e)
} } // end of δint

λ ((number of tokens N, id, timestamp T) ∈ S) {
    send ( (id, number of tokens), out);  }
```

The external transition function receives new tokens with timestamps, and adds them to the list of tokens, or decrement the number of tokens (only the enabled ones based on their timestamp) to fire a transition. After this, an internal transition is scheduled; an output combining *id* and *tokens* state variables is generated and transmitted on the **OUT** port. The number of advertised tokens is that of enabled tokens, whose timestamp is equal to or lower than the current simulation time. The internal transition function keeps scheduling events until all tokens have been enabled.

The TPN transition atomic model is defined as:

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

$\mathbf{X} = \{IN0, IN1, IN2, IN3, IN4 \in \mathbf{N}^+\}$

Y *= {OUT1 ∈* ***1****x****R****, OUT2 ∈* ***2****x****R****, OUT3 ∈* ***3****x****R****, OUT4 ∈* ***4****x****R****, FIRED ∈* $\mathbf{N}^{+}$*}*

S *= {{inputs ∈* ***N****} ∪ {enabled ∈ bool} ∪ {active ∈ bool }}.*

```
δext (s,e,x) {
    case port
        IN0: arc width = 0;      IN1: arc width = 1;
        IN2: arc width = 2;      IN3: arc width = 3;
        IN4: arc width = 4;
    extract the id of the place sending the message;
    if this is the first message we get from this id, inputs++;
    save (id, arc width);
   extract number of tokens in the place that sent the message and save it;
   if all input places have enough tokens to enable the transition
  if transition is enabled enabled = true
              if transition is active
                 ta(active, nextChange)
             else
                 active = true;
                 ta(active, DELAY_FIRE);
 else
       enabled = active = false
       passivate
} end of δext

δint (s) {
       if inputs = 0 // transition is a source,
          ta (active, random() ).
       else
          passivate
       active = false
}

λ (s) {
       firing_time = now + DELAY_TOKEN;
       send ((1, firing_time), OUT1);      send ((2, firing_time), OUT2);
       send ((3, firing_time), OUT3);      send ((4, firing_time), OUT4);
       send a message to every input place via the FIRED port;
} //  end of λ (s)
```

The model uses a number of input and output ports:
- **IN1**: it is used to receive the number of enabled tokens in the places connected to this input. Places that connect to this port have a single connecting arc; if the transition fires, only one token will be removed from the input places.
- **IN2-4** are used for multiple arcs (2 to 4 arcs)
- **IN0**: inhibitor arc. The input place must contain zero tokens (including those enabled by time for TPN) for the transition to fire and when it does, no token is removed from the place.
- **OUT1-4** are used to transmit 1-4 tokens to the corresponding ports.
- **FIRED** is used to remove tokens from the input places which must have their **IN** port connected to this output port in addition to being connected to one of the input ports.

The *inputs* state variable contains the number of input places for the transition, and *enabled* indicates if the transition is enabled or not. The model uses the DELAY_TOKEN parameter to determine delayed firing (i.e., the tokens will have a timestamp which is the current time plus DELAY_TOKEN). After being enabled DELAY_FIRE time, the transition actually fires.

One factor that is complex when simulating TPN in a DEVS simulator like CD++ is that when two or more transitions are enabled, one must be chosen and fired in a non-deterministic fashion. This implies that a controlling agent, aware of the state of all transitions in the model, would be required to determine which one should fire. In order to do this, the transition model schedules its own firing including a random amount of time after the transition is enabled. Given that all transitions do the same, this result in a near non-deterministic decision process.

When simulating the TPN results, we need to see the evolution of the marking, as seen in the following figure. The first two lines list the name of the places and transitions that make up the model. Then, we show the initial marking of the TPN counting the number of enabled tokens. We see transition *t2* firing at time 00:00:05, which makes places *p2* and *p3* update their token count. Then 2 seconds later, *p1* and *p4* do the same (as this is a TPN) producing the marking (1,4,4,1).

```
Petri Net places: p1 p2 p3 p4
Petri Net transitions: t1 t2
[00:00:000]      p1: 0 enabled tokens  p2: 5 enabled tokens
                 p3: 5 enabled tokens  p4: 0 enabled tokens
   (0,5,5,0)
       |
t2 [00:00:05:000]
       |
       V
[00:00:05:000] p2: 4 enabled tokens    p3: 4 enabled tokens
[00:00:07:000] p1: 1 enabled token     p4: 1 enabled token

   (1,4,4,1)...
```

Let us assume now that we want to study the behavior of an elevator. The elevator starts at the first floor, and it waits for people. Once people are inside, it closes the doors and it moves to the second floor, where opens its doors, waits for people leaving the elevator, closes the doors, and it returns to the first floor. A part of the TPN of this elevator model can be seen in Figure 5. All places use time-stamped tokens. For instance, the *PeopleWaiting* place, each token represents one person, and their timestamp represent their time of arrival to the elevator. In the transition T1, the DELAY_FIRE parameter represents the time that the doors remain open if the button is not pressed and no one is entering the elevator. The DELAY_TOKEN parameter represents how long it takes to close the door. In T3, DELAY_TOKEN represents how long it takes to open the door. In T6, DELAY_FIRE models the time it takes to enter the elevator. In T7, DELAY_FIRE is how long it takes to start moving after the doors are closed. Finally, in T8 DELAY_FIRE represents how long it takes to reach the second floor.

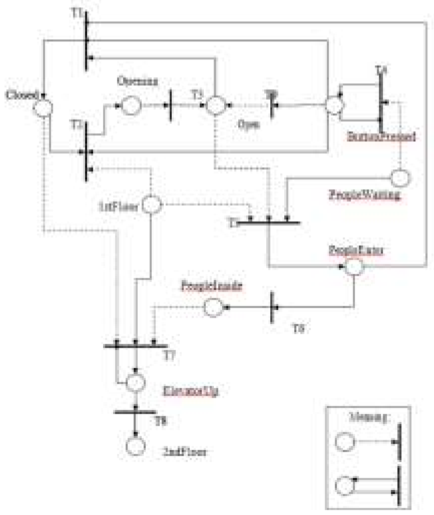


Figure 5. A TPN for modeling an elevator

The following test case shows a simulation run in which *PeopleWaiting* initially holds 3 tokens with timestamps 00:00:05, 00:00:10 and 00:00:15, representing the arrival of three persons at those times.

```
Petri Net places: Closed Opening Open ButtonPressed
                  PeopleWaiting PeopleEnter PeopleInside
                  PeopleArrived 1stFloor ElevatorUp
                  2ndFloor leavingElevator
Petri Net transitions: t1 t2 t3 t4 t5 t6 t7 t8 t9 t10
                       t11 t12 t13 t14
```

```
    [00:00:000]     Closed: 1 enabled  Opening, Open,
                    PeopleWaiting, PeopleEnter, ElevatorUp,
                    PeopleInside, PeopleArrived: 0 enabled
                    1stFloor: 1 enabled
                    2ndFloor, leavingElevator: 0 enabled
    [00:00:05:000]  PeopleWaiting: 1 enabled

    (1,0,0,0,1,0,0,0,1,0,0,0)
                 |
                 t4 [00:00:05:017]
                 |
                 V
    [00:00:05:017]  ButtonPressed, PeopleWaiting: 1 enabled

    (1,0,0,1,1,0,0,0,1,0,0,0)
             |
             t2 [00:00:05:031]
             |
             V
    [00:00:05:031]  Closed: 0 enabled ; Opening: 1 enabled
                    ButtonPressed: 0 enabled ; 1stFloor: 1 enabled
    (0,1,0,0,1,0,0,0,1,0,0,0)
             |
             t3 [00:00:05:047]
             |
             V
    [00:00:05:047] Opening: 0 enabled

    ...

    [00:00:49:044] ElevatorUp, Closed: 0 enabled; Opening: 1 enabled;
                     PeopleInside: 3 enabled; 2ndFloor: 1 enabled
    (0,1,0,0,0,0,3,0,0,0,1,0)
            |
            t3 [00:00:49:060]
            |
            V
    [00:00:49:060] Opening: 0 enabled ; Open: 1 enabled
    (0,0,1,0,0,0,3,0,0,0,1,0)
            |
            t10 [00:00:54:083]
            |
            V
    [00:00:54:083] Open: 0 enabled ; PeopleInside: 2 enabled ;
                         PeopleArrived: 1 enabled
...
    [00:01:00:153] Open, PeopleInside: 0 enabled ; 2ndFloor: 1 enabled
```

As we can see, initially the elevator is in Floor 1, and the door is closed. No transition is enabled. However, at 05:000, the first person arrives, enabling one token in the *PeopleWaiting* place. Transition *t4* is enabled, and it fires, representing that the elevator button has been pressed (and it takes 17ms to react). Transaction *t2* is enabled 14ms after that, which fires, triggering the opening of the door, and disabling the button pressed. At 05:047, *t3* is enabled, and now the door is open. A while after the three individuals have arrived, the elevator has moved to the second floor. We can see that at 49:044 the elevator is in the second floor, it has stopped moving up and there are three individuals waiting to leave the elevator. After 14 ms, the door is enabled to open, and it opens, allowing the individuals to leave the elevator. They leave one by one, until 01:00:153, where the elevator is empty in the second floor.

As we can see, TPN can be simulated using DEVS atomic models for PN transitions and places with an unmodified DEVS simulator, including a method in the internal transition function enabling timed Petri Nets. The DEVS specification allows transforming the methods with ease, and building user libraries for developing multimodels. In the next section, we discuss an extension for more advanced models based on Timed Automata.

3 Timed Automata and DEVS

New theoretical advances in model checking have allowed guaranteeing properties about models of real world systems using a formal approach. Model checking techniques can be automated, and Timed Automata (TA) theory [Alur and Dill 1994], in particular, has provided many practical results in this area. However, these formal methods are difficult to apply, and in many cases, they do not scale up well. Instead, using Modeling and Simulation (M&S) to gain confidence about the model correctness can be used to improve the study of experimental conditions during model definition, experimenting with virtual systems, explore options, including those cases where testing under actual operating conditions may be impractical. Nevertheless, no practical, automated approach exists to perform the transition that exists between the modeling and the development phases, and this often results in initial models being abandoned. Simultaneously, M&S frameworks are not as robust as their formal counterparts are. If the models used for M&S are formal, their correctness would also be verifiable, and a designer could see the system evolution and its inner workings even before starting a simulation [Saadawi and Wainer, 2010]. In order to deal with these issues, we showed a mechanism to represent TA with DEVS, and extended it to model hybrid systems using QSS methods, hence enabling formal verification of hybrid models within DEVS formalism.

We showed that, in order to be able to transform a DEVS model to a TA, we need that:

a) the TA variables bounded integers, in order to guarantee the finiteness of state space and hence the termination of the reachability algorithm (nevertheless, for

QSS, state variables are real numbers), and

b) the time of the next event be approximated to an integer number (in doing so we need to preserve the original behavior of QSS).

The first issue was handled by converting rational real numbers to integers by multiplying all values by the least common multiple of all the denominators. For any irrational values, we introduced a new method in [Saadawi and Wainer, 2010]. For the second issue, we use abstraction by over-approximation. With this technique, we approximate the real value of the event time t_i with a bounded time interval such that $t_c \in [T_L, T_H]$. This interval is bounded by floor(t_i) and ceiling(t_i) respectively. To obtain a TA that contains the behavior of a QSS model, we need a simulation relation with the QSS model (i.e., we need to show that the TA simulates QSS). To do so, each state in QSS would be simulated by a corresponding state in the TA, and each target state in QSS simulated by a corresponding target state in the TA. We will show these aspects using an example for an elevator controller originally introduced in [Saadawi and Wainer, 2010]. A summary of this case study is given below.

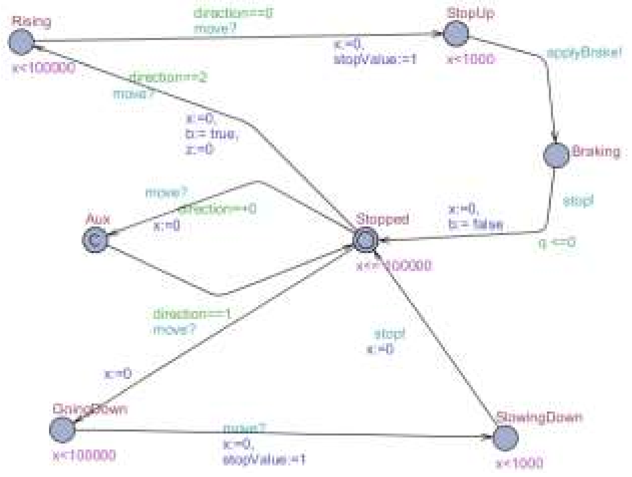


Figure 6: Elevator TA model.

In this model, whenever the elevator receives a command to stop, it synchronizes with a braking elevator motion model (*applyBrake!*). The elevator waits in state *Braking* for the quantized speed q value to reach zero. Once the elevator speed reaches zero, the transition from *Braking* to *Stopped* would be enabled and executed, then the elevator sends *stop!* to the elevator-controller. This hybrid model allows the designer to verify the control system with different parameters of the elevator physical system such as different braking values of de-accelerations, different elevator initial speeds, or other parameters in a more detailed QSS model.

This is an important addition to the elevator system verification as relevant physical factors to the controller performance can be identified and formally verified during design phase. The elevator de-acceleration motion and its speed are described by:

$$\frac{dv}{dt} = a \qquad\qquad v = at + v_i$$

where v is the elevator speed, a is the acceleration constant, and the speed is v at any point in time t, with v_i the initial elevator speed before applying the brakes.

To simulate and verify this hybrid model, we obtained a discrete representation of the elevator braking model, and we employed DEVS and QSS. We use a quantized variable q related to $v(t)$ system variable by quantization. To enable the formal verification, we transformed the QSS model to an equivalent TA model, resulting in the TA shown in figure 7.

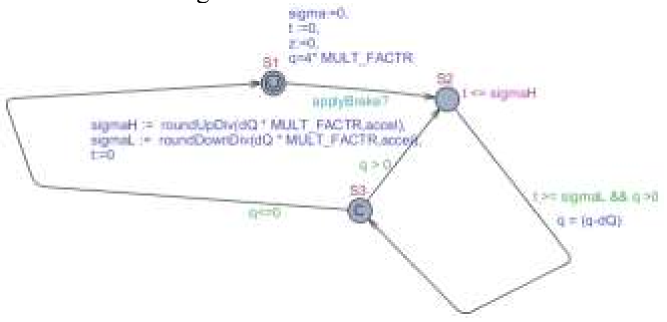


Figure 7. TA model of braking elevator motion.

Here, *S1* represents the initial elevator speed (4 m/s), the quantum value is dQ = 0.5 m/s, and *sigma* is the time interval between the outputs of two successive quantized values. When this model receives a synchronization input event (?) on the *applyBrake* channel, it changes to the state *S2* and starts a loop *S2-S3-S2...* in which we calculate the next quantized output *q* and the next values of *sigmaL* and *sigmaH*. When the quantized speed *q* reaches zero, the model moves back to *S1* and waits for another *applyBrake* event. As discussed earlier, sigma (the time advance) is over-approximated with an integer interval $\sigma \in$ [*sigmaL*, *sigmaH*].

Figure 8 shows the execution of the simulated hybrid DEVS model with braking de-acceleration equals -0.12 m/s^2. In this case, the time needed for the elevator to stop is approximately 33 seconds. This would contradict the user requirements, as the user expects the elevator to reach third floor within 27 seconds at most, and after this time the requirement for the elevator controller to be ready to accept another as shown on the transition *S5* $\rightarrow$ *S6.* However, the slow-braking elevator would not be able to fulfill the second request in time, hence we have a time lock and the model cannot progress beyond *S5*.

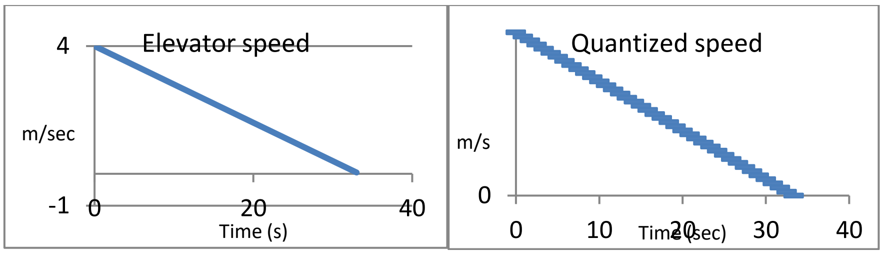


Figure 8. (a) Elevator speed; (b) Quantized speed, acceleration= - 0.12 m/s2.

It is important to count with advanced methods for modeling hybrid systems (where continuous and discrete phenomena interact), as they are found in many natural and artificial systems. Methods like those that we presented in this section can include verification of cyber-physical systems, which usually include discrete-event controllers interacting with a continuous plant. The combination of RTA-DEVS, hybrid Timed Automata and QSS allows verifying real-time hybrid systems modeled by DEVS. We showed a methodology to verify hybrid DEVS models. Some limitations, however, for this method of over approximation is that for systems described with nonlinear derivatives, it can lead to a wide flow pipe around the actual system trajectory. Other limitation is the inherit problem with model checking technique of state-space explosion that limits the ability to scale verification to larger models. In the next section, we show how to model these kind of hybrid systems using the Bond Graphs formalism and its transformation to DEVS.

4 Bond Graphs

The Bond Graphs formalism (BG) is a mathematical modelling method that focuses on the representation of continuous dynamic systems that can be described hierarchically [Karnopp et al., 1990; Vila and Rico, 2018]. BG represents the physical systems as a directed graph with hierarchical components, and its basic theory is based on the energy conservation law and the use of a lumped approach, separating dynamic system properties from each other, using submodels, and then linking those using ideal connectors. These connectors represent energy flow, and, it is assumed they have power continuity, and that no energy is generated or dissipated in the ideal connections. The physical systems they represent are modeled using directed graphs whose nodes define the physical processes and whose links (bonds) represent the ideal exchange of energy between them. Energy (or its time derivative, power), is the fundamental exchange between elements of the system. Power is the product of *flow* and *effort* (and they have no specific semantics; it can be used in translation mechanics, as *force* and *velocity*; in electrical systems, as *voltage* and *current*, etc.). The energy flow is represented elements exchanging effort and flow through the bonds. In order to represent the exchange of power between elements, we need to show the flows between components and their causality, as no component can determine the two power variables (effort and flow) at the same time. Given a pair of elements connected through a bond, their causality determines which causes flow and which causes effort.

There are a number of BG elements, including: *Capacitor* (*C*), *Inductor* (*I*), *Resistor (R), Effort source* (Se), *Flow source* (*Sf*), *Transformer (TF), Gyrator (GY), 1-junction, 0-juction.* To illustrate how they are defined, let us consider the *R* elements, which can represent resistors in the electrical domain, dampers the in mechanical context, etc. Their constitutive equation is defined by an algebraic equation relating *flow* and *effort*: *e = r. (f).* The electrical resistor is mostly linear, and the corresponding equation is *u=R.i,* where *R* is the resistance's constant.

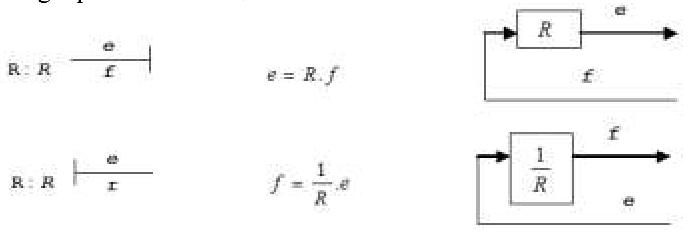


Figure 9. Top: R element with ***flow-in causality****, equations and block diagram representation; bottom: R element with* ***effort-in causality****, equations and block diagram.*

Based on BG concepts, we defined a discrete event library for Bond Graphs using DEVS [D'Abreu and Wainer, 2006], in which we define Quantized BG (QBG), that is, a BG where all the storages and sources are quantized elements. This method combines BG and DEVS models with QSS by adding quantizers equipped with hysteresis to the integrators output. The library consists of a number of atomic DEVS models developed on CD++, which implement QSS and QBG. The following code snippet shows a part of the quantizer model equipped with hysteresis implemented in CD++.

```
External transition
if ( msg.port() == in ) {
 if ( state() == passive ) {
   double inputValue = msg.value();    // gets message value
   currQValue = hquantize( inputValue); // applies QSS
   holdIn(active, Time::Zero); // schedule instantaneous transition }
}
```

```
Internal transition
  passivate();   //waits the reception of external messages

Output function
  if ( firstValue || ( currQValue != lastQValue ) ) {
      firstValue = false;
      lastQValue = currQValue;
      sendOutput( msg.time(), out, currQValue ) ;  }
// If first value received or boundary crossed, Sends quantized value

hquantize function
  if ( firstValue ) return QMethod->quantize( value );

  if ( ( value > lastQValue ) || ( value <= lastQValue - ∊ ) ) {
     return QMethod->quantize( value );  // hysteresis window size ∊
  }
  return lastQValue;
```

The model receives external inputs, and then computes the quantization function (with hysteresis) of the value received. Then, transmits the calculated value only if a boundary is crossed. The model parameters included the *Quantization method* (we support *uniform* and *intervals* quantization), their *parameters* (quantum size, lower and upper saturation values), the i*ntervals quantization method* (array of intervals and length of array of intervals, and the *hysteresis window size*.

A coupled DEVS representing a QBG model can be formally defined as:

$$CQBG = <X_{self}, Y_{self}, D, \{M_i\}, \{IC\}, select>$$

$X_{self} = \{\varnothing\}$ (no external inputs)

$Y_{self} = \{\varnothing\}$ (no external outputs)

D is the set of elements representing BG components, and for each i in D,

M_i is a DEVS atomic model representing a QBG component

IC is the internal coupling set defined as: $IC = \{ice_{ui,vj}\} \cup \{icf_{vj,ui}\}$ where $ice_{ui,vj}$ and $icf_{vj,ui}$ represent the coupling between effort and flow ports on u and v, being the effort calculated by element u.

$$ice_{ui,vj} = \begin{cases} ((u, out_{ei}), (v, in_{ej})) & \text{if } v \text{ is not a source(flow source)} \\ \varphi & \text{otherwise} \end{cases}$$

if u is a serial junction then i = 1 *(only one effort-out port)*

$$icf_{vj,ui} = \begin{cases} ((v, out_{fj}), (u, in_{fi})) & \text{if } u \text{ is not a source(effort source)} \\ \varphi & \text{otherwise} \end{cases}$$

if v is a parallel junction then j = 1 *(only one flow-out port)*

select gives priority to structural components (junctions, transformer, gyrator).

The library includes the following models: *QBGCapacitorFlowIn, QBGInductorEffortIn, QBGResistanceFlowIn, QBGResistanceEffortIn, QBGSourceEffort_Constant, QBGSourceEffort_Step, QBGSourceEffort_Sine, QBGSourceEffort_Pulse, QBGSourceFlow_Constant, QBGSourceFlow_Step, QBGSourceFlow_Sine, QBGSourceFlow_Pulse, QBGTransformer, QBGGyratorFlowIn, QBGGyratorEffortIn, QBGSerialJunction, QBGParallelJunction*.

In order to show how to use the library, we define a mechanical model of the response of mass against the application of effort, shown in Figure 10.

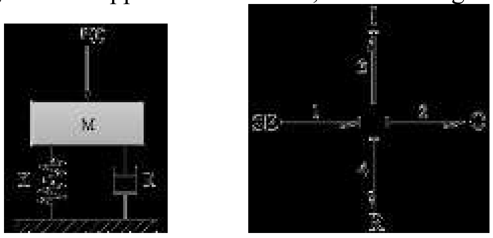


Figure 10. Mechanical circuit and its Bond-Graph representation

Following, we show two cases of execution of the model, in which we use different frequencies. Figure 12 the evolution of mass speed over time. It can be seen how speed is modified every time that effort is imposed over M and how it tries to return to its original value. We use I=40; C=2; SE: signal=pulse, period=200ms, pulse duration=2ms, and different resistance. We can see that the variation of resistance produces changes in the number and speed of oscillations, as expected.

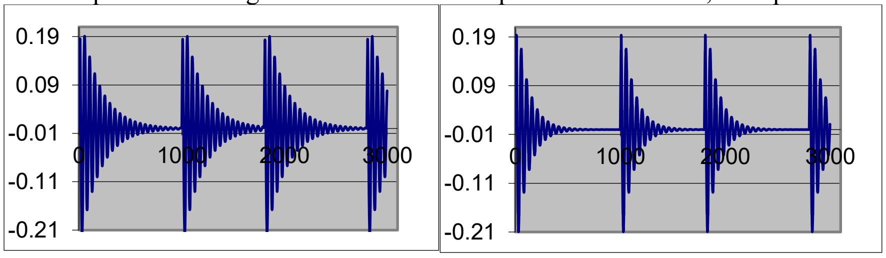


Figure 11. Mass speed; (a) R=1.5; (b) R=3.

Using our BG library, we then conducted an extended effort, building the first open source Modelica compiler, which was implemented on top of the QBG library, and will be described in detail in the following section.

5 Modelica

Modelica is an object-oriented language, defined for modeling physical systems and built to support library development and model exchange. Modelica models are described using differential, algebraic and discrete equations, and there are various libraries of standard components using ODEs, block diagrams, electrical and mechanical formalisms [Fritzson, 2004].

We defined a compiler that understands models from the *electrical library* in Modelica [D'Abreu and Wainer, 2006]. A source file in Modelica is compiled into an equivalent Bond Graph representation of the circuit. The generated BG constitutes the output, which is used for simulation. The BG generation algorithm is based on Karnopp's circuit construction method [Karnopp et al, 1990]. The approach is to build a BG that resembles the circuit structurally, and then to simplify it based on selected properties, as follows:

`- For each node with a distinct potential add a 0-junction:` we used transitive closure applied to every node on the graph (we check for paths between arbitrary nodes *x* and *y*, given only adjacency information). As there are different ways to specify the parallel coupling between elements in the circuit, we need to ensure that the *0-junction* elements are correctly inserted (independently from the definitions in the Modelica file).

`- Insert each 1-port circuit element by adjoining it to a 1-junction, inserting 1-junctions between the appropriate pair of 0-junctions (C, I, R, Se, Sf elements):` we add a *1-junction* to each *1-port* element, inserting it between the corresponding pair of *0-junctions*.

`- Assign power directions to all bonds:` a standard convention assumes positive direction of power when it flows out of sources (*Se* and *Sf*) and into *C*, *I* and *R* elements. For two-port elements, *TF* and *GY*, we consider that power flows into the elements. We use a power propagation algorithm that traverses the graph and assigns power using the standard conventions and the information in the Modelica file. At the end of this step we obtain a directed BG is obtained.

`- Erasing ground potential:` all the explicit ground potentials are deleted from the graph; if there is no explicit ground potential, we delete the 0-junction nearest to each source element. The 0-junctions selected are only those associated with the negative pin of every source's port.

`- Simplification:` a junction between two bonds with through power direction can be deleted; likewise, a bond connecting two junctions of the same type can be deleted and the junctions joined

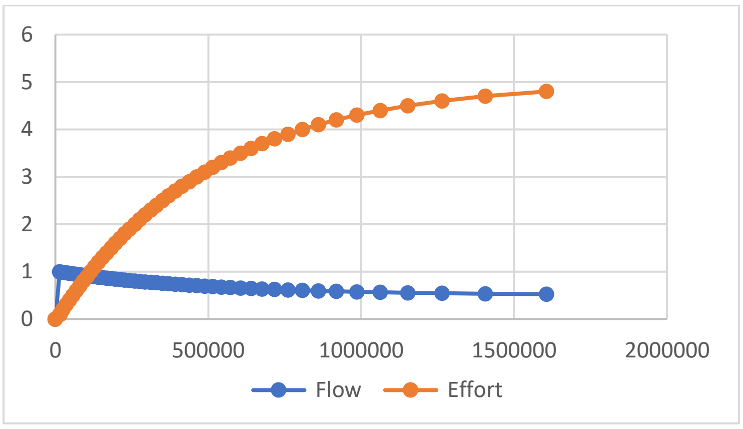
In the final BG, we check for algebraic loops and singularities (elements that have discontinuities e.g. diode), and we generate an optimized QBG corresponding to the electrical circuit, which is used to generate a coupled DEVS model in CD++.

```
model circuit
  Modelica.Electrical.Analog.Sources.PulseVoltage
                      V(V=10,width=50,period=2);
  Modelica.Electrical.Analog.Basic.Resistor R1(R=10);
  Modelica.Electrical.Analog.Basic.Capacitor C(C=50);
  Modelica.Electrical.Analog.Basic.Ground Gnd;
equation
  connect(V.p, R1.p);           connect(R1.n, C.p);
  connect(C.n, V.n);            connect(V.n, Gnd.p);
end circuit;
```

(a)

```
[top]
components : $SJ1@QBGSerialJunction C@QBGCapacitorFlowIn
               R1@QBGResistanceEffortIn V@QBGSourceEffort_Pulse

link : e1n@$SJ1 e1p@R1       link : f1p@R1  f1n@$SJ1
link : f2n@$SJ1 f1p@C        link : e1p@C   e2n@$SJ1
link : f3p@$SJ1 f1n@V        link : e1n@V   e3p@$SJ1

[C]
quantum : 0.1  hystWindow : 0.01     C :     50     initialLoad :  0

[R1]
R :     10

[V]
quantum : 0.1    hystWindow : 0.01    signal : Pulse    offset : 0
startTime : 0    amplitude : 10       period : 2        width : 50
```

(b)



(c)

Figure 12. (a) Modelica model (b) QBG translation (c) Model execution

Figure 12 shows a Modelica model of a circuit, first, and the translation to QBG after. As we can see, we generate a coupled model for each of the Modelica components. The corresponding QBG are built for each of the components, and, in this case, they are connected by the Serial Junction SJ1 (which is automatically generated by the compiler using the procedure explained above). The equations are converted into couplings in the DEVS coupled model. After the coupled model definition, we can see the initialization values for each of the components, which also follow the Modelica specification. The Capacitor and the Pulse Voltage generator are approximated using QSS (the quantum size and the hysteresis window parameters can be adjusted for simulation). Finally, we show outputs in both Flow and Effort ports for Capacitor C, which matches the expected behavior of the circuit.

6 VHDL

Digital designers have often relied on M&S for the design of circuits, as simulated studies allow reducing the number of design bugs and integration errors, while easing product maintenance and reducing the overall cost. Design and simulation of digital logic with HDLs (Hardware Descriptor Languages) is a well-proven methodology. Mixed signal HDL simulators allow combining discrete time digital and continuous time analog models. Here, we discuss a method for simulating mixed signal HDLs by converting the designs to DEVS [Mehta and Wainer, 2005]. The discrete-event nature of DEVS is well suited to model digital logic, and signal quantization with QSS allows modeling the continuous systems components.

VHDL-AMS is targeted toward register transfer level modeling of digital circuits with limited behavioral modeling and analog constructs. The main construct used by the language is the *Entity,* which describes the interface to a VHDL-AMS design or design unit. The entity declaration contains a list of *ports*, each of which is assigned a *type* and an optional *mode*. Ports of type *std_logic* or *std_logic_vector* (a standardized type for digital logic) are used for digital signals while electrical ports are used for analog signals. In the case of digital signals, ports will have mode *in*, *out*, *inout* or *buffer*. Analog ports do not require a mode. The syntax of an entity declaration is as follows:

```
entity  entity_name is
     port ( [signal |  terminal | quantity] identifier {,
            identifier}: [mode | ] signal_type | electrical
        {; [signal | terminal | quantity ] identifier {,
            identifier}: [mode | ] signal_type | electrical});
          ) ;
end [entity] [entity_name] ;
```

For instance, Figure 13 shows the declaration for an analog low pass filter entity.

```
entity LPF is
   port (
   terminal tout, tin,
        tgnd: electrical
   );
end entity LFP;

architecture top of LPF is
  signal clk : std_logic;
  signal vin : std_logic;
  quantity vout across tout to tgnd;

begin
  vout'dot = (1/(R*C))*(vin-vout);
```

```
clk: entity clk
       port map (clk=>clk);
       vin<=clk;
end architecture top;
```

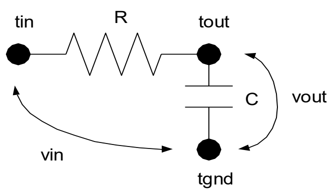


Figure 13. Low-pass Filter

A design *architecture* describes the functionality of a design or design unit; this may be a structural, dataflow or behavioral description. A single architecture is associated with exactly one entity.

```
architecture architecture_name of entity_name is
     signal_declaration
     | constant_declaration
     | component_declaration
begin
     {process_statement
     | concurrent_signal_assignment_statement
     | component_instantiation_statement
     | simultaneous_statement}
end [architecture] [architecture_name] ;
```

The body of an architecture consists of statements that may be categorized as concurrent, sequential or simultaneous. These statements operate on signals and quantities that are declared within the scope of the architecture, and on ports that are declared in the entity.

```
  signal    signal_name    :    std_logic_vector    (upper_bound    downto
lower_bound) | std_logic ;
  quantity identifier : REAL | Voltage | Current | Charge ;
  quantity identifier {, identifier} across identifier {, identifier}
through free_terminal to reference_terminal ;
```

Signals and quantities are defined in the declarative region. These belong to the scope of the architecture in which they are declared and may only be referenced within that architecture. Quantities may also be declared relative to terminals in an entity declaration. These may be either *across* or *through* quantities. Across quantities, represent the voltage at the free terminal relative to the reference terminal.

Through quantities represent the current from the free terminal into the reference terminal. *Process*, *Simultaneous*, *Concurrent Assignment* and *Conditional Concurrent Assignment* Statements execute concurrently within an architecture. The conditional concurrent assignment statement assigns the target signal the value of an expression if the condition is true and the value of a different expression otherwise. A *process* executes the statements between begin and end process when an event occurs on a signal in its sensitivity list. No signals modified by the process are updated until the process body is completed. The statements between *begin* and *end* clauses are referred to as *sequential statements*, and they are executed in sequence.

```
[process_name:]
process (sensitivity_list)
   { type_declaration      }
begin
  {signal_assignment_statement
  | if_statement
  | case_statement
  end process [process_name] ;
```

The *if* statement has identical semantics to that of an if-then-else statement in C/C++; the *case-when* statement runs the sequence of statements that are listed under the when clause whose expression matches that of the expression in the case statement. Simultaneous statements are generally used for describing Differential Algebraic Equations, and may consist of quantities or signal, for instance:

```
x1'dot'dot == -f*(x1 - x2) / m1;
x2'dot'dot == -f*(x2 - x1) / m2;
…
```

The *'dot* notation denotes the derivative with respect to time of the quantity listed before the *'dot*. For example *signal'dot* is the first derivative with respect to time of *signal*, while *signal'dot'dot* is the second derivative.

Figure 14 shows the software architecture of the tools used to build VHDL-AMS models. We start with a syntax check phase, in order to ensure that the VHDL-AMS model is syntactically correct. During the elaboration phase, each component description is assigned to a structure in the VHDL-AMS design hierarchy. The description of the architecture and entity for each component in the design is parsed, and a Netlist is produced, which includes interconnected integrators, algebraic operators, processes, signals and sub-component instances. We then generate DEVS models in CD++, using a library for each components of the design and a coupled model to define the architecture. The CD++ models are then compiled, after which the Netlist and model library are used by the model file generation, after which we obtain a DEVS coupled that can be used for simulation. In order to convert VHDL-AMS models to CD++ coupled models (done during the Model Code and Netlist Generation phases above) we first need to identify the components that constitute the design hierarchy. Basic components do not contain sub-component instances in their architectures, while aggregate components do. We generate a dependency tree must in which the leaves are basic components, while branches are aggregate components, and the root is the top-level model. VHDL-AMS sub-component instances are connected to the architecture in which they are instantiated as defined by the port map clause in their component instantiation statement.

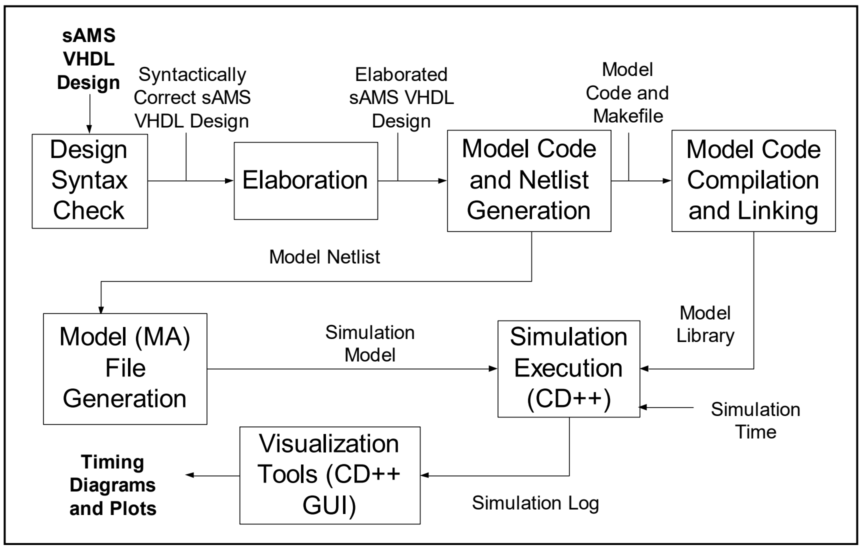

sAMS VHDL Design
Syntactically Correct sAMS VHDL Design
Elaborated sAMS VHDL Design
Model Code and Makefile
Design Syntax Check
Elaboration
Model Code and Netlist Generation
Model Code Compilation and Linking
Model Netlist
Model Library
Model (MA) File Generation
Simulation Model
Simulation Execution (CD++)
Simulation Time
Timing Diagrams and Plots
Visualization Tools (CD++ GUI)
Simulation Log

Figure 14. Architecture of the VHDL to DEVS compiler

The process body is implemented within the external transition function in CD++. Buffering is also done within the external transition function for each signal referenced in a rising_edge or falling_edge operation. Buffers are also created for each output port on the model in CD++. The output ports on the model represent all the signals that are driven from within the VHDL-AMS process. The values that are assigned to these buffers will be output on their respective ports when the model schedules an output event. Expressions and case statements are converted directly to equivalent ones in C++. The CD++ signal model is used to implement transport delay on messages sent between process model ports. The implementation of the process model in DEVS does not allow assignment statements to have transport delays, since the output events for all driven signals must occur simultaneously. Therefore, we implemented transport delays in the CD++ signal model, which receives and buffers data on its input port, enters the active state for the time specified by the assignment statement transport delay, and outputs the buffered data on its output port.

As discussed earlier, simultaneous statements in VHDL-AMS allow the definition of continuous time systems through differential algebraic equations (DAE); in our case, we approximated this by solving ordinary differential equation systems with initial conditions, and combining them with a DEVS approximation of Runge-Kutta and Euler integration methods. For instance, in the case of Euler, we compute $y_{n+1} = y_n + h.f(t_n, y_n)$. The method extrapolates the solution over the interval using an approximation to the derivative at the beginning of the interval (Figure 15).

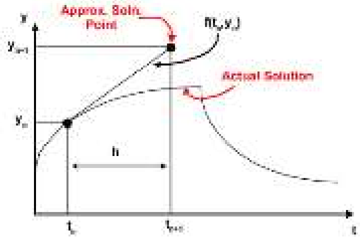


Figure 15. Euler integration method

We implemented both Euler and a fourth-order Runge-Kutta Method Integration method (which is more accurate and stable than Euler for a given step size, as it uses the derivative at the beginning of the interval, the derivative at two trial midpoints and the derivative at a trial end point). These two methods were defined using the QSS method (order two), providing accurate results for a small quantum size. To do so, we inverted the equations used by the numerical methods to determine at what time (relative to the present time) the integral of the first order differential equation will enter the quantum state above or below the current quantum state. We decompose the ODE into a set of first order differential equations, and convert them into a Quantized Integrator model.

The following figure shows a CD++ definition of the DEVS model derived from the low pass filter presented in Figure 13, and simulation results of the model.

```
[top]
components : int@rkIntegModel clock
out : clk y
Link : y@int y             Link : y@int dydt@int
Link : out@clock clk       Link : out@clock vin@int

[int]
y0 : 0      dydt0 : 0      C : 1.0E-6      R : 1000
[clock]
components : inv@Process_Inv sig1@Signal qm@QuantumMultiply
```

```
out : out
Link : out@sig1 in@inv     Link : out@inv in@sig1
Link : out@sig1 in@qm      Link : out@qm out

[sig1]
Transport_Delay : 00:00:1:000

[qm]
Transport_Delay : 00:00:000Attenuation : 100
```

Figure 16. DEVS implementation of the low pas filter in Figure 13.

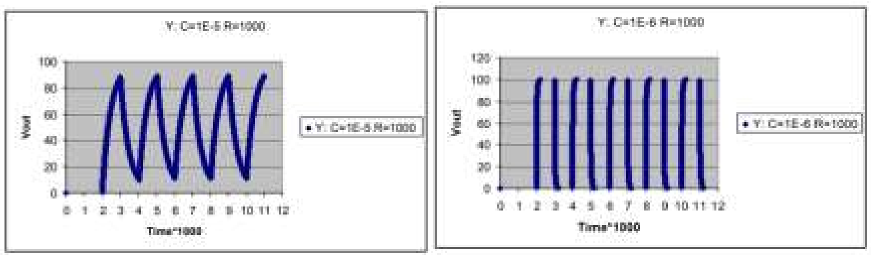


Figure 17. Simulation results of the low-pass filter.

7 The ATLAS Traffic Modelling Language

Urban traffic analysis is a problem whose complexity makes the analysis with traditional analytical methods difficult. The use of simulation is now the tool of choice urban traffic analysis. ATLAS (Advanced Traffic Language Specifications) is devoted to build models of city sections using microsimulation [Davidson and-Wainer, 2006]. The basic language constructions allow defining a static topology of the section to be studied. The dynamic behavior of the section can be modified by including traffic lights, traffic signs, etc. Once the urban section is outlined, models are converted into cell spaces and the traffic flow is automatically set up. Language constructions were mapped into DEVS and Cell-DEVS models. The models were formally specified, which made easier the verification of the language constructions.

ATLAS allows to represent the structure of a city section defined by a set of streets connected by crossings. The language constructions define a static view of the model, and they are considered as cellular models. The main constructions are

Segments, *Crossings*, *Parking segments*, *Traffic lights*, *Railways*, *Construction sites*, and *Traffic Signs*. Different *Experimental frameworks* can be used to conduct experimentation and analysis.

ATLAS formal language defines the structure and behavior of these constructs, and we used the formal specification with validation purposes. Based on ATLAS, we built TSC, the Traffic Simulation Compiler, which understands ATLAS models and translates them into DEVS models. In order to make the model definition easier, we defined two different tools for model definition and simulation visualization. MAPS allows the user to build maps and decorations (traffic signs, speed, etc.), and the results can be animated using CD++ (or Google Maps) to visualize the simulation results. Figure 18 shows a workflow and the software stack used for ATLAS.

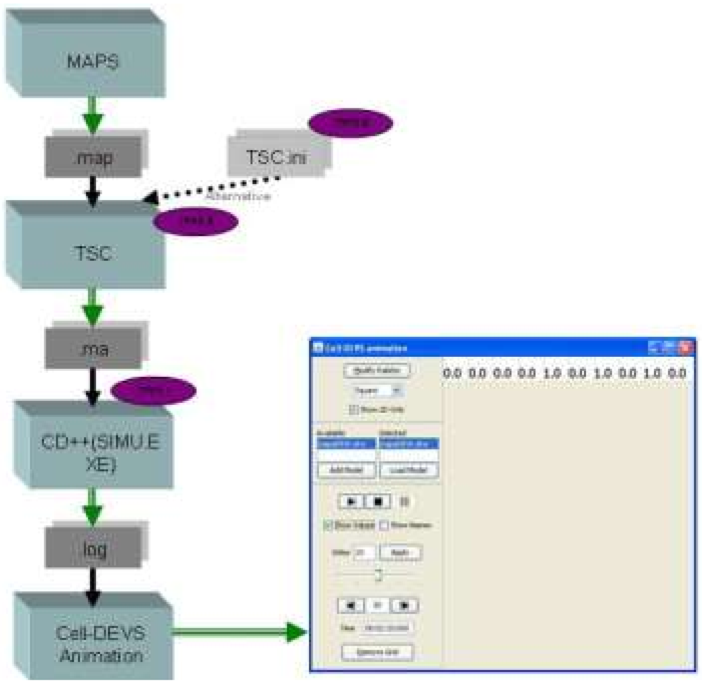


Figure 18. ATLAS Tool stack

Following, we will summarize the constructions of ATLAS, their formal specifications, and their implementation in the TSC simulation language (the reader interested in further details about the formal language and the compiler tools can refer to [Lo Tártaro et al. 2001]).

The main construction is the *segment,* which represent a section between two intersections. Each lane in a given segment has the same direction (i.e., they are one-way) and they have a maximum speed. They are formally specified as:

Segment={(p1,p2,n,a,dir,max)}

where *p1* and *p2* represent the boundaries of each segment, n is the number of lanes, and *dir* represents the vehicle direction, 0 represents the traffic direction is from *p2* to *p1* and 1 represents the traffic direction is from *p1* to *p2*. The *a* parameter defines the shape of the segment (0-straight or 1-curve, allowing to define the city shape precisely, and to include the exact number of cells), and max is the maximum speed allowed in this segment.

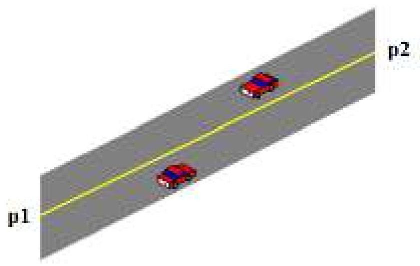


Figure 19. A road segment

The syntax of TSC [Lo Tártaro et al., 2001] allows defining the segments by delimiting them using the sentences *begin segments* and *end segments*. At least one segment must be defined, using the following syntax:

```
id = p1, p2, lanes, shape, direction, speed, delay, parkType
```

where,

`p1`: (x, y), integers; the start of the segment.

`p2`: (x, y), integers; the end of the segment.

`lanes`: integer, the number of lanes in the segment.

`shape: [curve | straight]`, the shape of the segment.

`direction: [go | back]`, the direction of the segment.

`speed`: integer, maximum speed in the segment.

`delay`: integer, defines a delay value used for parking lanes.

`parkType: [parkNone | parkLeft | parkRight | parkBoth]`, defines if parking is allowed in the segment, and where.

The following example shows the definition of the segments section; in this case there is only one segment with start/end points at cells (4,1) and (4,3), two lanes wide. It is a straight segment, and traffic moves from origin (cell (4,3)) to destination (cell (4,1)). The maximum speed is 60 km/h. You can park on the right lane, and the simulation will receive a parameter of 1100 s as the average delay

```
begin segments
    S1 = (4,3), (4,1), 2, straight, go, 60, 1100, parkRight
end segments
```

As we can see in Figure 18, these TSC specifications are translated into CD++, as a set of rules based on the Cell-DEVS formalism. We defined models for segments from 1 to 5 lanes (unidirectional), and we formally verified the correctness of the rules. We will now discuss this mapping for 2-lane segments.

A segment s t = (p1, p2, 2, a, dir, max) is defined as a 2D Cell-DEVS model, using transport delays, with the structure presented in Figure 20.

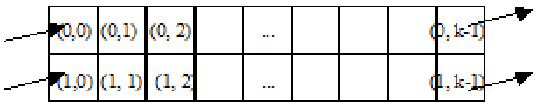


Figure 20. 2-lane segment

Each row has a different specification, based on the asymmetry of the cell space. The vehicles in the first row can move straight or to the right (and the ones in the second row, in the opposite direction). The first row is defined as:

$$C_{0j} = < I, X, S, Y, N, \delta_{int}, \delta_{ext}, \text{delay}, d, \tau, \lambda, D >$$

I = < η, P^X, P^Y>, with $\eta = 6$; and P^X = { (X_1, boolean), (X_2, boolean), (X_3, boolean), (X_4, boolean), (X_5, boolean), (X_6, boolean) }; P^Y = { (Y_1, boolean), (Y_2, boolean), (Y_3, boolean), (Y_4, boolean), (Y_5, boolean) , (Y_6, boolean) }.

X = Y = boolean;

S:

$$s = \begin{cases} 1 \text{ if there is a vehicle;} \\ 0 \text{ otherwise.} \end{cases}$$

delay = transport;

d = convert_to_delay(speed(max)); where *speed* is a random function that uses using a probabilistic distribution based on vehicle traffic. One expects a few vehicles with maximum and minimum speed, and a majority between them. Based on the maximum speed, we compute the mean $\bar{x} = \frac{2}{3} * max(km/h)$ and the standard deviation, $\sigma = \frac{1}{3} * max(km/h)$, which are passed to the function, which returns a natural number representing the random speed in km/h for the vehicle. Based on this, we compute the delay to cross a cell, which is 7.5 m (the size needed for a vehicle [Chopard et al. 1996]), dividing by the speed in km/h, and multiplying by 60^2 (to convert the delay in km/h into seconds).

N = { (0,0), (0,1), (-1,0), (-1,1), (0,-1), (-1,-1) };

τ is:

New State	Preconditions	Rule Name
1	((0,0) = 0 and (0,-1) = 1) or	From_Behind
	((0,0) = 0 and (0,-1) = 0 and (-1,-1) = 1 and (-1,0) = 1)	From_Right
0	((0,0) = 1 and (0,1) = 0) or	Move_Forward
	(0,0) = 1 and (-1,1) = 0 and (-1,0) = 0	Right_of_Way_Right
(0,0)	True	Default

The vehicles can only arrive from the cell behind, or from the right (due to the neighborhood definition). For diagonal movements, we need to consider the right of way. In this way, we avoid collisions. The rules in the second lane are symmetrical to these ones, but we also need to consider that the vehicles from the right have the right of way (so, we evaluate those rules first).

The coupled model corresponding to the segment is defined as:

S2L(k, max) = < Xlist, Ylist, I, *X*, *Y*, n, $\{t_1,...,t_n\}$, η, N, C, B, Z, select >

Ylist = { (0,0), (1,0), (0,k-1), (1,k-1) }

Xlist = { (0,0), (1,0), (0,k-1), (1,k-1) }

I = <P^x, P^y>, with P^x = {x-c-vehicle, boolean>, x-c-vehicle, boolean>, <x-c-free, boolean>, <x-c-free, boolean>} and P^y = {<y-c-free, boolean>, <y-c-free, boolean>, <y-c-vehicle, boolean>, <y-c-vehicle, boolean>}

X = Y = Boolean;

n = 2; t1 = 2; t2 = k; η = 6 and N is described for each lane.

C = { C_{ij} / i $\in$ [0, 1] $\wedge$ j $\in$ [0, k-1] }, with Cij a Cell-DEVS atomic model

B = { (0, k-1), (1, k-1), (0,0), (1,0) }

select = { (0,1), (1,1), (0,0), (1,0), (0,-1), (1,-1) }

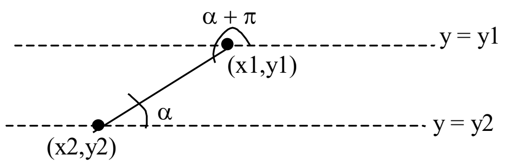


Figure 21. Computing the length based on inclination angles.

In this case, S2L(k, max) is a segment of 2 lanes, k cells long, and maximum sped *max* km/h. The number of cells *k* is computed automatically as the distance between start/end of the segment divided by 7.5 m, using the inclination angle as in Figure 21.

The model's interface includes the cells of the first and last column of the segment; these ones can interchange vehicles from/to the crossings. Therefore, the behavior of these cells is different to the rest of the segment, and the formal definitions of the borders change as follows, for cell (0,0):

η = 4; **N** = { (0,0), (0,1), (-1,0), (-1,1) }

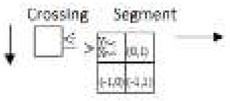


Figure 21. Neighborhood definition and ports for border cell (0,0)

τ is:

New State	Preconditions	Rule Name
1	(0,0) = 0 and portvalue(x-c- vehicle) = 1	From_Crossing
0	((0,0) = 1 and (0,1) = 0) or	Move_Forward
	(0,0) = 1 and (-1,1) = 0 and (-1,0) = 0	Right_of_Way_Right
(0,0)	True	Default

This cell only receives vehicles that are coming from a crossing, while the rules to advance are similar to the other cells. We define a similar set of rules for cell (1,0), the second input border. Similarly, cells (k,0) and (k,1) need a modification of neighborhood and input/output ports to allow vehicles to leave the segment in the direction of a crossing. In this case, τ is:

New State	Preconditions	Rule Name
1 send(0, y-c-vehicle)	(0,0) = 0 and (0,-1) = 1	From_Behind
	((0,0) = 0 and (0,-1) = 0 and (-1,-1) = 1 and (-1,0) = 1)	From_Right
0 send(1, y-c- vehicle)	(0,0) = 1 and portvalue(x-c-free) = 0	To_Crossing
(0,0)	True	Default

This cell only receives vehicles coming from a crossing, while the rules to advance are similar for lane one and the rest of the model do not change. We have symmetric rules for cell (1,0). Likewise, cells (0, k-1) and (1, k-1) must generate outputs to the crossings, and the behavior generated is similar.

Parking defines different behavior in the border cells in a segment, as these can be used for parking, as seen in Figure 2. They are formally defined as:

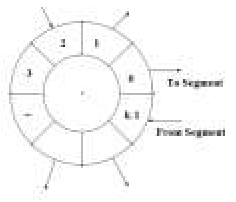
Parking={(s,n1)}

Every pair (s1, n1) identifies the segment and the lane where car parking is allowed. If n1= 0, the cars park on the left; if n1 = 1, the cars park on the right. As seen in the *segments* definition, the parking has been added as an attribute of the segment for each of them in the TSC compiler.

Crossings are defined in those points in the plane where several segments intersect. They are formally defined as:

Crossings={(c,max)}

which represents the points where the crossings are located, and the maximum speed to cross a segment (this allows defining a different speed for the crossing and the segments they are connected to, and defining specialized behavior for countries where speed in crossings should be lower than in the crossing streets. Likewise, it allows defining reduced speeds for roundabouts. Finally, if there are segments with different speeds entering a given crossing – for instance, a main avenue crossing a residential street – this lets us defining the desired speed for the crossing). Crossings are translated to a cellular model built as a ring of cells with moving vehicles. A vehicle in a cell of the crossing has higher priority to obtain the next position in the ring than the cars outside the crossing (see Figure 22). The number of cells in the ring is calculated as the number of input and output segments connected to the crossing, as shown in Figure 22.



$$k = \sum_{\substack{t \in T \wedge t=(c1,c2,n,a,dir) \\ \wedge (c1=p \vee c2=p)}} n$$

Figure 22. Crossing definition and calculation of number of cells in the crossing

The TSC definitions for crossings are delimited by the separators `begin crossings` and `end crossings`. Each sentence defines a crossing using the following syntax:

```
id = p, speed, tLight, crossHole, pout
```

Parameters `p` and `speed` represent the coordinate of the crossing `(p1,p2)` and max `speed` for the crossings. `Pout` defines the probability of a vehicle to leave the crossing, used to simulate random routing of different vehicles.

Crossings with *traffic lights* are defined as:

TLCrossing={(c | c∈Crossings)}

Here, $c \in TLCrossings$ defines a set of models representing the traffic lights in a corner and the corresponding controller. Each of these models is associated with a crossing input. The model sends a value representing the color of the traffic light to a cell in the intersection corresponding to the input segment affected by the traffic light. The following qualifier is added to a standard crossing definition in TSC when a crossing must include traffic lights:

```
tLight : [withTL | withoutTL]
```

The crossings C1 and C2 in the following definition show two crossings, one at cell (4,3), and another at cell (4,14). Both have a maximum speed of 40 km/h and a probability to leave the cell of 70%.

```
begin crossings
   C1 = (4,3), 40, withTL, withoutHole, 0.7
   C2 = (4,14), 40, withTL, withoutHole, 0.7
end crossings
```

Railways are defined as a set of level crossings overlapped with the road segments (see fig. 5). The railway network is defined by:

$$RailNet=\{(Station,Rail)$$

Where *Station* is a model and $Rail \in RailTrack$ and *RailTrack* is defined as:

$$RailTrack=\{(s,\quad \delta,\quad seq)\}$$

Railtrack associates a level crossing with other existing constructions in the city section. Each element identifies the segment that is crossed (*s*) and the distance to the railway from the beginning of the section (δ). Finally, a sequence number (*seq*) is assigned to each level crossing, defining its position in the RailTrack. *RailNet* represents a set of stations connected to railways. When a railway is defined in TSC, we use the *begin railnets* and *end railnets* clauses. Each *RailNet* is defined using the following syntax:

```
id =(s1, d1)  (s2, d2)  (s3, d3)…
```

where `si` defines an identifier of a segment crossed by the railway, and `di` defines the distance between the beginning of the segment `si` and the railway. The compiler automatically generates the sequence number.

Similarly, we have defined language constructions for potholes, traffic signs, construction zones, etc.

The following picture shows a main section in downtown Ottawa. The traffic in this area is usually crowded, and congestion occurs frequently. The area includes two parks, several one-way streets and avenues. In several of these streets, parking is allowed, while in others it is forbidden. There are traffic lights in several of the crossings.

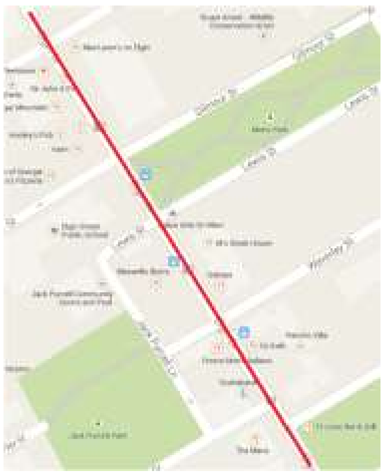


Figure 23. Traffic section for case study: Elgin Street in Ottawa

Figure 24 shows the ATLAS definition for this area, labeling the segments and crossings. Using MAPS, we built a model of this section, which is then translated to ATLAS TSC as follows:

```
begin segments
     S1 = (4,1), (4,3), 2, straight, back, 60, 1100, parkRight
     S2 = (5,1), (4,3), 2, straight, go, 60, 1100, parkRight
     S3 = (4,3), (4,14), 2, straight, back, 60, 1100, parkRight
     S4 = (4,3), (5,14), 2, curve, go, 60, 1100, parkRight
  ...

      S28 = (4,36), (10,36), 1, straight, go, 40, 1100, parkNone
      S29 = (10,36), (13,36), 1, straight, go, 40, 1100, parkNone
      S30 = (4,39), (1,40), 1, straight, go, 50, 1100, parkNone
      S31 = (4,39), (1,39), 1, straight, back, 50, 1100, parkNone
end segments

begin crossings
  C1 = (4,3), 40, withTL, withoutHole, 220, 110
  C2 = (4,14), 40, withTL, withoutHole, 221, 111
  ...
  C8 = (10,28), 20, withoutTL, withHole, 229, 119
  C9 = (10, 36), 20, withoutTL, withHole, 230, 120
end crossings

begin ctrElements
  in S21 : pedestrian crossing, 1, 110
  in S22 : pedestrian crossing, 1, 111
  in S22 : school, 2, 112
  in S23 : stop, 7, 113
  in S24 : stop, 6, 114
```

```
   ...
   in S30 : pedestrian crossing, 1, 117
end ctlElements
```

For instance, we can see, that segment S1 is a two-lane road, which includes all the parameters as discussed earlier. Following the method to compute the cells in the cell space, this model must be mapped into a two-dimensional Cell-DEVS with transport delays. Similarly, the TSC definition of segment S29 represents is a one way street starting at (10, 36) and ending at (13, 36). The traffic direction on this road is from the source to the destination. The maximum speed is 40 km/h and parking is not permitted. The crossing C1 is at position (4, 3) and the maximum speed in this intersection is 40 km/h. The crossing has traffic lights, and there are no potholes. We can also see that in road segment 22, there is a school located 2 cells away from the beginning point, and that in segment 28, there is a pedestrian crossing located 1 cell away from the beginning point.

The compiler generates a CD++ file, using the TSC template rules, and the model generated is as follows, following the rules for 2-lane models described above:

```
[TOP]
components : S1  S2  S3 ... S31   ; segments
       C1   C2   ... C7           ; crossings
       S17tl@TrafficLight S17Gen@Generator S15Cons@Consumer ...
       C1stl@SincroTrafficLight C2stl@SincroTrafficLight
       C21stl@SincroTrafficLight C4stl@SincroTrafficLight
       C41stl@SincroTrafficLight
...

link : y-t-car0@S17Gen x-ge-car00@S17
link : y-t-car1@S17Gen x-ge-car10@S17
...

[S17]
type : cell
width : 3    height : 2      delay : transport    border : nowrapped
neighbors : (1,-1) (1,0) (1,1) (0,-1)(0,0)(0,1) (-1,-1) (-1,0) (-1,1)
in : x-ge-car00 x-ge-car10 x-c-space02 x-c-space12
out: y-c-car02 y-c-car12
link : x-ge-car00 x-ge-car@S17(0,0)
link : x-ge-car10 x-ge-car@S17(1,0)
...
localtransition : S17-segment2-lane0-rule
[S17-segment2-lane0-rule]
#Macro(S17-From_Behind)
#Macro(S17-From_Right_Lane)
#Macro(S17-Move_Forward)
#Macro(S17-To_Right_with_Right_of_Way)
#Macro(S17-Default)

   ...
#BeginMacro(S27-From_Behind)
```

```
rule : 1          40 { (0,0) = 0 and (0,-1) = 1 }
```

As we can see, the segments definitions are translated into CD++ coupled models using the definitions discussed earlier. For instance, we can see the definition of segment S17, a small Cell-DEVS model, of 2 lanes (3 cells long), connected to a *generator* model (which feeds the section with vehicles through the ports *x-ge*. The local transition rule is defined using the formal specifications discussed above, and in particular, we show a definition of the macro used to define the rule "From_Behind", which represents movement to a cell from the vehicle behind it. When we simulate the model, we can change initial conditions, for instance, the traffic lights, number of vehicles generated, etc. Figure 24 shows the simulation results at different times in segment S17, in which we injected a similar amount of vehicles, and we can see the influence of traffic lights (lower line) that help regulating the traffic in the section.

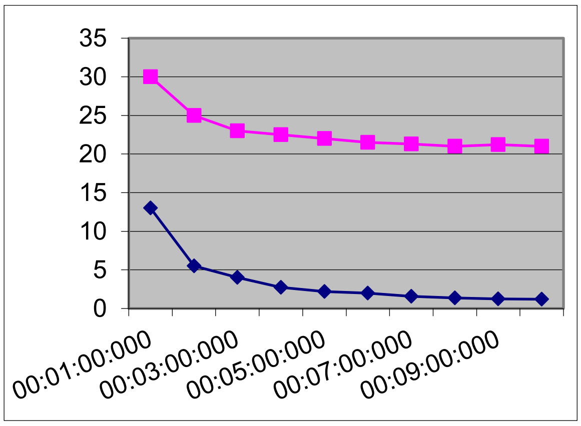


Figure 24. Simulation scenario with traffic lights and without traffic lights

8 Conclusion

The last 50 years of SummerSim have brought a wide range of advances to the field of Modeling and Simulation. Here, we have discussed our lab's research in the area, focusing on using DEVS as a common formalism for simulating a diverse variety of models. DEVS provides the means of building complex models evolving incrementally from simple subcomponents in incremental, hierarchical fashion. This view enables the reuse of simulations and components, where the integration of simulations and components is seamless.

The experiments were carried out using CD++, a DEVS tool that has been built following the formal definitions of DEVS and Cell-DEVS. The modeler can then to focus on the modeling formalism of interest, and to use all the advantages of DEVS (in terms of integration with other models, multimodeling, sharing of repositories, proving the validity of simulation code, etc.) for model composition and simulation.

The use of formal modeling techniques enhances model verification. Specifically, automated rule verification, based on meeting basic logical properties in cellular models and coupled model definitions, can be provided. Our tools also provide mechanisms for automating the verification of multicomponent model coupling. In the same sense, we showed how to provide multimodeling and hybrid modeling. The models are able to include both continuous and discrete event model components. For instance, the behavior governing the physics of a vehicle can be described with Bond Graphs (continuous modeling technique), while vehicle cruise control system might be better modeled using a discrete event formalism. We showed how to integrate these model views in a seamless fashion, using DEVS and the CD++, combined with the QSS method and model transformations. Spatial notions can provide extra facilities for understanding and visualizing the resulting simulation. For example, it allows incorporating geographical data obtained in GIS software.

Acknowledgements

This work was partially funded by NSERC. The research has been conducted by a large number of students in our lab in the last 20 years, including: M. Lo Tártaro, C. Torres, T. Zheng, M. D'Abreu, L. Checiu, C. Jacques, H. Saadawi, D. Vicino, R. Castro, S. Mehta, A. González and numerous others, who have spent long hours in this research.

References

R. Alur, D. Dill. "Theory of Timed Automata". Theoretical Computer Science, volume 126, pg. 183-235, 1994.

H. Bowman, R. Gomez. Concurrency Theory: Calculi and Automata for Modelling Untimed and Timed Concurrent Systems. Springer-Verlag London 2006.

R. Castro, E. Kofman, G. Wainer. (2010) "A Formal Framework for Stochastic DEVS Modeling and Simulation". SIMULATION: Transactions of the Society for Modeling and Simulation International. Vol. 86, No. 10. pp. 587-611.

Chechiu, L., and Wainer. G. (2005). Experimental results on the use of Modelica/CD++. In Proceedings of the 2005 SCS Summer Computer Simulation Conference. Philadelphia, PA.

Chopard, B. Queloz, P. A.; Luthi, P. "Cellular Automata Model of Car Traffic in two-dimensional street networks". J.Phys. A.vol.29, pp.2325-2336, 1996

D'Abreu, M., and Wainer, G. (2006) A Bond-Graph mapping mechanism for M/CD++. In Proceedings of the 2006 SCS Summer Computer Simulation Conference. Calgary, AB, Canada.

A. Davidson, G. Wainer. ATLAS: a specification language for traffic modelling and simulation". Simulation, Practice and Experience. Elsevier. Volume 14, No. 3, pp. 317-337. April 2006.

Fritzson, P. (2004). Principles of Object-Oriented Modeling and Simulation with Modelica 2.1. Wiley-IEEE Press.

Hopcroft, J. Ullman, J. (1979). Introduction to Automata Theory, Languages, and Computation. Reading/MA: Addison-Wesley.

Jacques, C., and Wainer, G. (2002). Using the CD++ DEVS toolkit to develop Petri Nets. Proceedings of the 2002 Summer Computer Simulation Conference, San Diego, CA. USA.

Karnopp, D., Margolis, D., Rosenberg, R. 1990: System Dynamics: a unified approach, Wiley.

Kofman, E., and Junco, S. (2001) Quantized State Systems. A DEVS Approach for Continuous System simulation. Transactions of the SCS, 18(3): 123-132.

Liu, F., & Zhang, H. (2018). A class of extended time Petri nets for modeling and simulation of discrete event systems. SIMULATION, 94(8), 753–762.

M. Lo Tártaro, C. Torres, G. Wainer. "TSC: a compiler for the ATLAS language". In Proceedings of 2001 Winter Simulation Conference. Arlington, VA. U.S.A. IEEE Press. 2001.

S. Mehta, G. Wainer. (2005) "DEVS for mixed-signal Modeling based on VHDL". Proceedings of 2005 DEVS Integrative M&S Symposium, Spring Simulation Conference. San Diego, CA.

Peterson, James L. "Petri Nets". ACM Computing Surveys, Vol 3, No. 5. September 1977. pp 221-252.

Pietro, F. D., Migoni, G., & Kofman, E. (2018). Improving Linearly Implicit Quantized State System Methods. SIMULATION. https://doi.org/10.1177/0037549718766689

H. Saadawi, G. Wainer. 2010. "Rational time-advance DEVS (RTA-DEVS). In Proceedings of DEVS Symposium 2010, Orlando, FL., April 11-15.

Villa-Villaseñor, N., Rico-Melgoza, J. J. (2018). Complementarity framework formulation from bond graphs to model a class of nonlinear systems and hybrid systems with fixed causality. SIMULATION, 94(9), 783–795.

Wainer, G. A. (2009). Discrete-Event Modeling and Simulation: A Practitioner's Approach. Boca Raton, FL, USA: CRC Press.

B. P. Zeigler, T. Kim, and H. Praehofer. 2000. Theory of Modeling and Simulation. San Diego, CA: Academic Press,

Zheng, T., and Wainer, G. (2003). Implementing finite state machines using the CD++ toolkit. Proceedings of the 2003 Summer Computer Simulation Conference. Montreal, QC. Canada.

Author's Resume

GABRIEL A. WAINER, FSCS, SMIEEE, received the M.Sc. (1993) at the University of Buenos Aires, Argentina, and the Ph.D. (1998, with highest honors) at UBA/Université d'Aix-Marseille III, France. In July 2000, he joined the Department of Systems and Computer Engineering at Carleton University (Ottawa, ON, Canada), where he is now Full Professor and Associate Chair for Graduate Studies. He has held visiting positions at the University of Arizona; LSIS (CNRS), Université Paul Cézanne, University of Nice, INRIA Sophia-Antipolis, Université de Bordeaux (France); UCM, UPC (Spain), University of Buenos Aires, National University of Rosario (Argentina) and others. He is one of the founders of the Symposium on Theory of Modeling and Simulation, SIMUTools and SimAUD. Prof. Wainer was Vice-President Conferences and Vice-President Publications, and is a member of the Board of Directors of the SCS. Prof. Wainer is the Special Issues Editor of SIMULATION, member of the Editorial Board of IEEE Computing in Science and Engineering, Wireless Networks (Elsevier), Journal of Defense Modeling and Simulation (SCS). He is the head of the Advanced Real-Time Simulation lab, located at Carleton University's Centre for advanced Simulation and Visualization (V-Sim). He has been the recipient of various awards, including the IBM Eclipse Innovation

Award, SCS Leadership Award, and various Best Paper awards. He has been awarded Carleton University's Research Achievement Award (2005, 2014), the First Bernard P. Zeigler DEVS Modeling and Simulation Award, the SCS Outstanding Professional Award (2011), Carleton University's Mentorship Award (2013), the SCS Distinguished Professional Award (2013), and the SCS Distinguished Service Award (2015). He is a Fellow of SCS.