ProDEVS : an Event-Driven Modeling and Simulation Tool for Hybrid Systems using State Diagrams

Le Hung Vu
University of Toulouse,
CNRS-LAAS
7 Avenue du Colonel Roche
Toulouse, France
lhvu@laas.fr

Damien Foures
University of Corsica, UMR
CNRS 6134 SPE
Campus Grimaldi
Corte, France
foures@univ-corse.fr

Vincent Albert
University of Toulouse,
CNRS-LAAS
7 Avenue du Colonel Roche
Toulouse, France
vincent.albert@laas.fr

ABSTRACT
This paper introduces a new event-driven modeling and simulation tool for the simulation of hybrid systems. The particularity of this software called ProDEVS lies in its graphical language to define model components behaviour. This graphical language customizes state diagrams for DEVS and quantized based numerical methods. In this paper, syntax and operational semantic of the language are explained, and a mapping from DEVS to this language is illustrated across two simple examples in discrete-event and continuous domain. Finally a complete hybrid system is modeled and simulated to show the usability and the efficiency of this model.

General Terms
Languages

Keywords
State diagram, DEVS, Hybrid systems, Event-driven simulation

1. INTRODUCTION
DEVS (Discrete EVent System Specification) [19] is a general formalism for specifying modular and hierarchical model of dynamic systems. A DEVS specification is executed by an event-driven simulator which ensures the scheduling of timed events and increases the simulation clock to the time of next event. Algorithms and methods to implement an event-driven simulator for DEVS models are also given in [19].

There are numerical methods like the Quantized State System (QSS) family [3] that have shown to efficiently approximate ordinary differential equations. QSS is based on state space discretization, also called *quantization*, rather than time discretization given by conventional numerical integration methods. Within QSS, a *quantization function* maps real-valued numbers onto a discrete set of real values also called *quantization levels* [3]. A continuous-time system is then approximated by computing the required time for a state variable to reach the next level. This technique has a straightforward representation in DEVS. A first-order non-stiff Quantized State System (QSS1) algorithm was introduced by Kofman in 2001 [9], followed by second and third-order accurate non-stiff solvers, called QSS2 [7] and QSS3 [8], respectively. QSS family techniques include now also stiff system solvers (LIQSS [11]) as well as solvers for marginally stable systems (CQSS). QSS methods have been theoretically analyzed to exhibit nice stability, convergence, and error bound properties, and in general come with several advantages over classical approaches.

DEVS is even considered [1] as the most general formalism since other discrete event languages such as Petri Nets and State Charts but also discrete time systems can be seen as particular cases of DEVS. Moreover DEVS provides a unified framework for representing hybrid systems which combines discrete and continuous dynamics and the usage of QSS based integration methods are noticeably efficient to simulate hybrid systems due to their ability to handle discontinuities [1]. Taking into account these remarks, DEVS is a good candidate for modern modeling and simulation environment.

There is a huge variety of tools which support DEVS: DEVSJAVA [18], aDEVS [12], CD++ [17], PowerDEVS [1], JAMES II [20], VLE [15], PyPDEVS [16], DEVS-Ruby [4]. A comparison of this software list is given in [5]. If most of them have a Graphical User Interface, none of these tools are able to support at the same time a graphical syntax for atomic component description and the mentioned numerical integration of ordinary differential equations. PowerDEVS is hybrid system oriented but atomic components are defined in C code. The idea of using visual diagrams for DEVS specification is not new. Similar approaches can be found in [10], in [14] and in CD++ [17]. However, these approaches do not address numerical integration of ordinary differential equations which requires advanced specification of actions using algebraic equations.

In this context, we design and implement ProDEVS, an integrated modeling and simulation environment oriented to hybrid systems based on discrete-event simulation theory and DEVS. We define a language which specializes state diagram for building, animating and simulating graphical DEVS specification. We implement an event-driven simulation engine according to the algorithms and the methods previously mentioned. This idea is motivated by the trend for using high-level specification language rather than code for dynamic system design.

The next section gives an overview of the tool and its features. Section 3 introduces the syntax and the semantic of a ProDEVS model. Section 4 shows the mapping between a DEVS atomic component and a ProDEVS State diagram for discrete-event system and continuous system. Section 5 describes an example to illustrate the use of this model in hybrid simulation. Section 6 gives conclusions and perspectives.

2. TOOL FEATURES

Our software is implemented in Java and it is based on the platform of services OSGI (Open Service Gateway initiative). Thus, each component of the application (editor, model, simulation engine, plot, code generator) is in the form of bundle that can be started or stopped dynamically. OSGI framework also facilitates the addition of new features.

Figure 1 shows a screenshot of the editor. It gives users an easy navigation within the hierarchical levels of the model and allows simulating the model at different level of the hierarchy. The diagram panel in the center allows editing models. There are two type of pallets if it is coupled component or atomic component. A pallet for the description of a block diagram with port and connector is given for the definition of a coupled component and a pallet for ProDEVS State diagram description is proposed when editing an atomic component. We use the JGraphX library for graphical design of the model.

The WEST region contains a project explorer, a model explorer and a library of components. Components can be dragged and dropped directly from the library to the diagram (import) or from the diagram panel to the library (export). The components of the library are stored in XML files.

The EAST region contains a properties tab for specifying information of models such as initial values of its variables and its parameters. The simulation tab is used to specify the simulation time and the execution algorithm. Two simulation algorithms are available: classic or parallel [19]. In classic simulation only one event at a time can occur. In parallel simulation, more than one event can be received by a component at the same time. In case of non-determinism, the receiver component randomly chooses an event to execute. A complete set of mecanisms (priorities, guards) allows to resolve conflicts in the execution and improves the semantic of our language.

The simulation can be performed in step-by-step mode, where each variable value can be observed. At the end of the simulation, a new frame gives the plots for selected ports or variables. The bundle for the visualization of trajectories is based on the JFreeChart library.

For a sake of performance, the simulation engine executes compiled code. A specific bundle for loading the model is called when a new simulation is started. For each atomic component, this bundle generates a .java file that is compiled and executed. Compiled code improve significantly the performances compared to interpreted code that needs parsing boolean and arithmetic expressions. However, the user can access the source file to introduce more complex treatments like loops that can not be expressed graphically until now.

A model verifier is used to check static properties on the model. These properties are about the correct construction of the model (typing, valid expression, non-initialised variable,...). Verification result is given as warning and error in the console tab.

We finally design a ProDEVS model transformation to extended TPN (Timed Petri Nets) with data handling called (TTS) Time Transition Systems [2] to generate a finite representation for the accessible states of a ProDEVS model. We consider the Finite and Deterministic subset of DEVS [6], classic DEVS and parallel DEVS. This feature is no further explain here and will appear in another paper.

3. PRODEVS MODEL

We create a customized component-based state diagram model that supports DEVS syntax and semantic. In substance, we reuse and specialize many concepts of UML 2.4 (Unified Modeling Language), including packages *BehaviorStateMachines, Communications* and *Kernel*. This section gives a description of this model.

3.1 Model structure

The abstract syntax for the structure of a ProDEVS model is given by the UML class diagram figure 2. A ProDEVS model has (composition link) exactly one component which is either atomic or coupled (inheritance links). A component that is coupled or atomic consists of input or output ports (portDirectionKind attribute). An atomic component is associated with a *DEVSStateMachine* describing its behavior. Coupled component is composed of one to several atomic or coupled components and zero to several connectors. A port must be typed (the class *Port* inherits from the class *TypedElement*). A port acting as a source is connected to zero or more ports acting as a target via connectors. A port that serves as a target is the destination of zero or more ports acting as source via connectors (associations between classes *Port* and *Connector*). A connector can only be associated with two and only two ports at a time. A connector connects either an output port of a component to an input port of another component (IC) or an output port of a component with an external output (EOC) or an external input to an input port of a component (EIC).

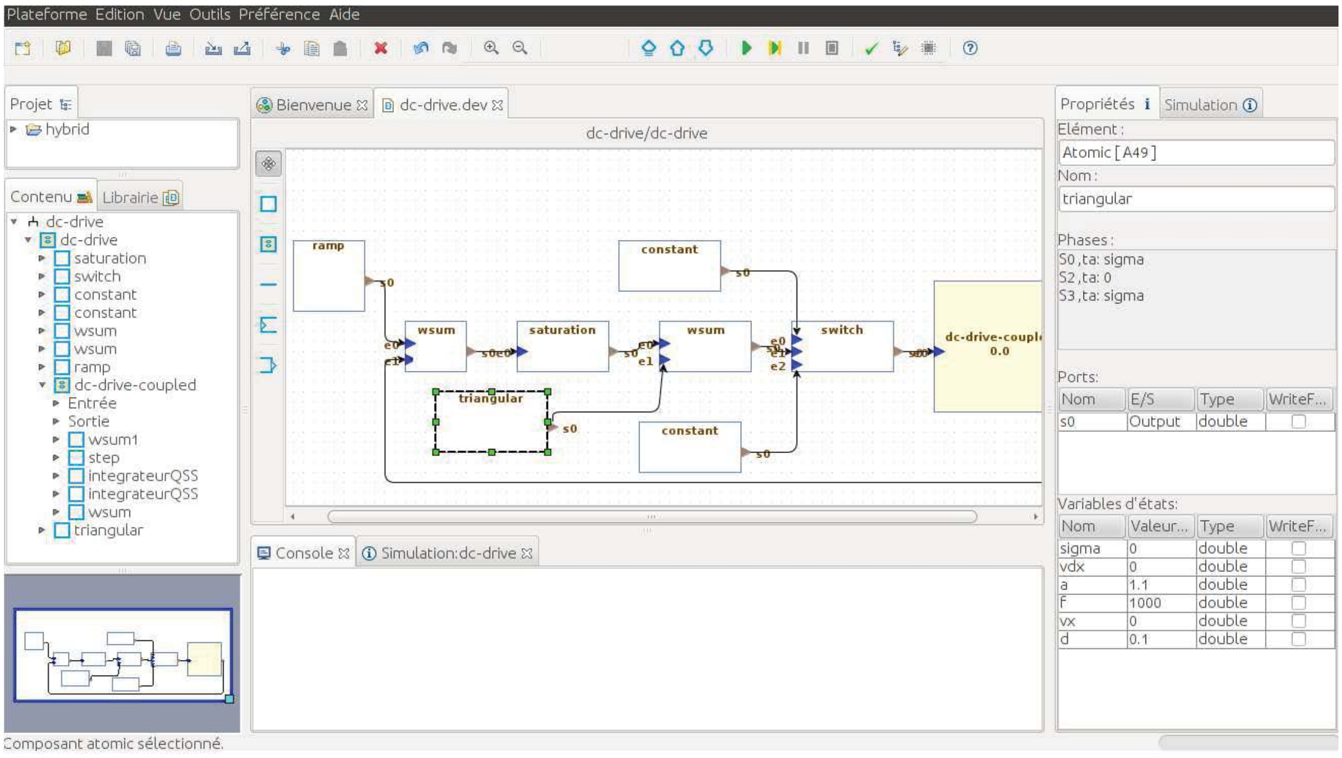
TypedElement defines the type of value *DataType* that is a comprehensive list of literal values: boolean, double, integer and float. *PortDirectionKind* is a comprehensive list of literal values: in, out. *ConnectorKind* is a comprehensive list of literal values: IC, EOC, EIC.

We define constraints to check the correctness of the model. Examples of constraint are given through this section in natural language and in Object Constraint Language (OCL):

If the connector is of type IC, source port must be of type out and the target port must be of type in. In addition, source port and target port must belong to different sub-components of the same current coupled component.

```
Coupled.AllInstances -> forAll(connector.connectionKind
<> IC) implies (source.portDirectionKind <> out) and
(destination.portDirectionKind <> in) and
(self.includes(source.component) and
(self.includes(destination.component) and
(source.component != destination.component)
```

3.2 ProDEVS State diagram



Elément :
Atomic [A49]

Nom :
triangular

Phases :
S0 ,ta: sigma
S2 ,ta: 0
S3 ,ta: sigma

Ports:

Nom	E/S	Type	WriteF...
s0	Output	double	☐

Variables d'états:

Nom	Valeur...	Type	WriteF...
sigma	0	double	☐
vdx	0	double	☐
a	1.1	double	☐
f	1000	double	☐
vx	0	double	☐
d	0.1	double	☐

Composant atomic sélectionné.

Figure 1: ProDEVS editor

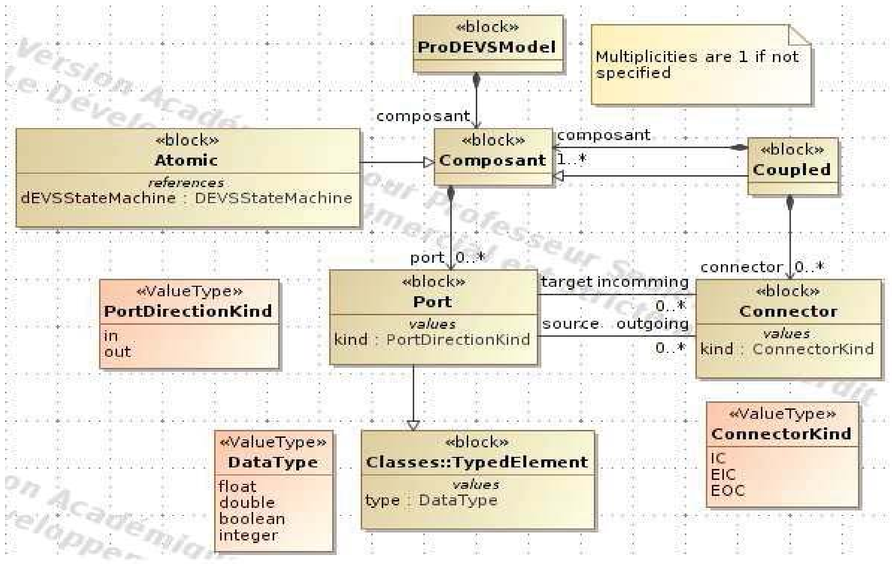


Figure 2: ProDEVS model structure

Syntax.

The abstract syntax for a ProDEVS State diagram is given by the UML class diagram figure 3.

A *ProDEVSStateMachine* is a set of states-transitions that defines the behavior of an atomic component. A phase is a specific parameter that represents a period in the life of the atomic component where it is expected some events to occur (input event or time event). It has a set *outgoing* of transitions departing from this phase, a set *incoming* of transitions entering this phase and a *timeAdvance* attribute to specify the time in which the system has to stay before triggering the output and the internal transition function (a time advance value can also be given by a variable of the component).

A phase with a time advance equal to infinity must not have an outgoing transition of kind internal and must have at least one outgoing transition of kind external.

```
self.timeAdvance.val <> infinity implies
forAll(t | self.outgoing->not(t.kind <> internal)) and
exists(t | self.outgoing->t.kind <> external)
```

A phase with a time advance not equal to infinity must have at least one outgoing transition of kind internal

```
not(self.timeAdvance.val <> infinity) implies
exists(t | self.outgoing->t.kind <> internal)
```

A transition is an oriented relationship between a source phase and a target phase. It is associated to :

- a trigger which specifies an input that may fire an external transition in case of external event or an output in case of time event (time elapsed in a phase is reached).
- a guard which provides a fine-grained control over the firing of the transition using boolean expression. The guard is evaluated when an event, external event or time event, is dispatched by the state diagram. If the guard is true at that time, the transition may be enabled, otherwise, it is disabled. A guard constrains a set of properties [1] of the component.
- an action which specifies an optional assignement to be performed onto property when the transition fires.
- a source which designates the originating phase of the transition.
- a target which designates the target phase that is reached when the transition is taken.

3.3 Operational semantic

The system is in phase φ at a given time and must be in that phase for a period $e = \varphi.timeAdvance$, if no external event occurs. When the time e has elapsed without any external event has occurred, the system triggers a time event and calculates and propagates the output. Then, for each outgoing internal transitions the guards are assessed. The system triggers one of those transitions which are enabled. If instead, an external event occurs on the input before the expiration of e, the system triggers the corresponding external transition, if it is enabled. In any case, the system reaches a new phase $\varphi' = \varphi.outgoing.target$ for a period defined by $\varphi'.timeAdvance$, the actions associated to the triggered transition are computed and the same algorithm is applied.

When $\varphi.timeAdvance = \infty$, it means that it is a passive phase and only an external event will leave the phase. When $\varphi.timeAdvance = 0$, it means that this is a transient phase so the output computation and the internal transition triggering are immediately performed.

In ProDEVS, like in DEVS, communications are weak synchronous, i.e., non blocking with (possible) message loss. If both sender and receiver are ready to communicate, the output event is converted into an input event which is instantly received. If the receiver is not ready, the message is lost.

As a result of coupling of concurrent components, there may be multiple components with simultaneous events. Thus, there may be multiple components which are candidates for the next internal state transition. Such components are called imminents in the DEVS terminology. In classic DEVS, a *Select* function is added at the coupled component that allows to select the component to execute among a set of imminent components. In parallel DEVS, every imminent components are executed. A component may receive a bag of inputs. A confluent transition function is added to refine a transition in case of simultaneous time event and external event.

The label for internal/output function and external transition function are defined by the following BNF expression:

```
<transition> ::= [<guard>] <trigger> '/' [<action>]
```

The details of the syntax for the trigger event are defined by different kind of events:

```
<trigger> ::= <time-event> | <external-event>
```

- output events (at time event) are denoted by ! followed by the name of the triggered output port, followed by an assignment specification:

  ```
  <time-event> ::= [!<name> = <output-assignement>]
  ```

- external events are denoted by ? followed by the name of the triggering input port:

  ```
  <external-event> ::= ?<name>
  ```

There are no assignement on port when the trigger is of kind externalEvent.

```
self.kind <> externalEvent implies self.output->isEmpty()
```

There is an assignement on output port when the trigger is of kind timeEvent.

```
self.kind <> timeEvent implies self.output->notEmpty() and
self.port = self.output.assignedPort
```

4. PRODEVS IN PRACTICE

This section describes how a ProDEVS State diagram can describe a DEVS atomic component for discrete-event system and continuous system.

[1]The term *property* or *structural feature* is used in UML. In this model the class *Property* simply represents a variable of the component.

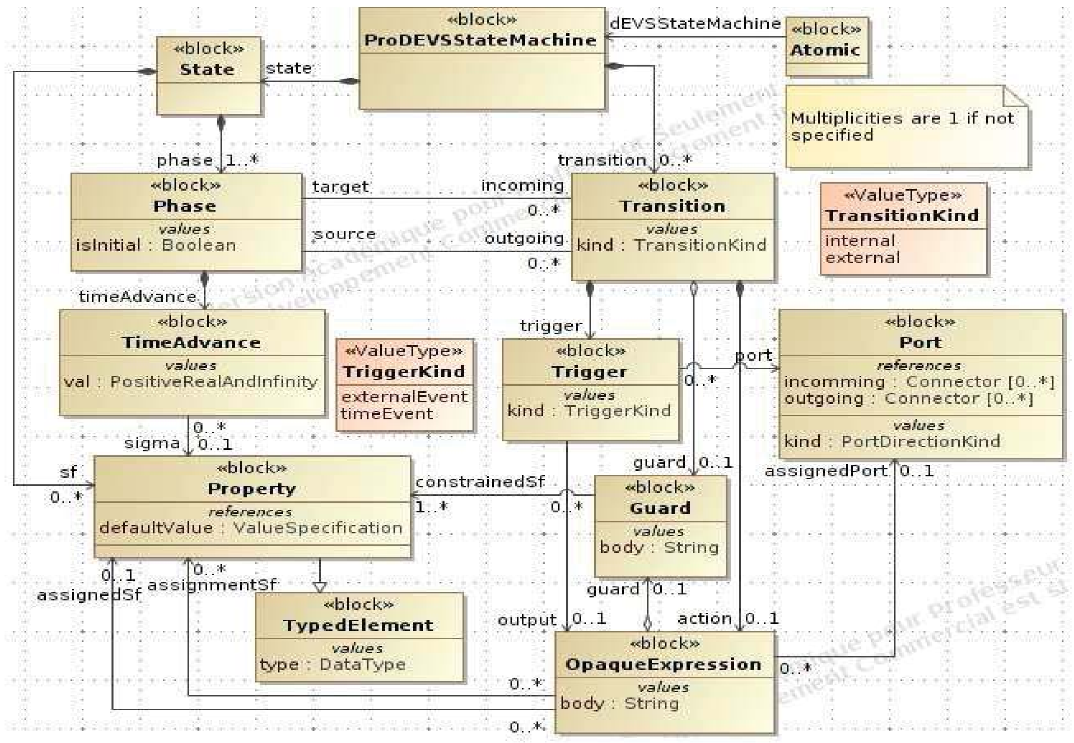


Figure 3: ProDEVS State diagram

Discrete-event system.
In DEVS, external transition function, internal transition function, output function and time advance function are defined on S [19]:

- $\delta_{ext} : Q \times X \rightarrow S$ with $Q = \{(s,e) | s \in S, 0 \leq e \leq ta(s)\}$
- $\delta_{int} : S \rightarrow S$
- $\lambda : S \rightarrow Y$
- $ta : S \rightarrow \mathbb{R}^{+}_{0,\infty}$

An element,i.e., a state, of S is a set of pair of a variable and value. For example, the set of states S_{BUF} for a buffer BUF is defined by:

$$S_{BUF} = \{(proc\_status, queue) | \\ proc\_status \in \{free, busy\}, queue \in \mathbb{N}_0)\} \quad (1)$$

Or the set of states S_I of a quantized integrator I is defined by:

$$S_I = \{ql, q, \dot{q}, \sigma | ql \in \mathbb{R}, q \in \mathbb{R}, \dot{q} \in \mathbb{R}, \sigma \in \mathbb{R}\} \quad (2)$$

Often, but this is not mandatory, a DEVS atomic component has a special state variable called phase that can take a value from a set list of literal values.

A initial state is given as a list of assignment. For example, if the state variables *queue* and *proc_status* have the initial values empty and free, then the initial state is given as $queue := 0, status := free$. The general form of the internal state transition function is given as $current\_state \Rightarrow next\_state$ where $current\_state$ is expressed as a boolean expression and $next\_state$ as a list of assignment. For example an internal transition buffer BUF could be:

$$(queue > 0 \;\&\&\; proc\_status == free) \Rightarrow \\ (queue := queue - -, proc\_status := busy) \quad (3)$$

The general form of a time advance function is given as $current\_state \Rightarrow \mathbb{R}^{+}_{0,\infty}$ where $current\_state$ is expressed as a boolean expression. For example a time advance for the buffer BUF could be:

$$(queue > 0 \;\&\&\; proc\_status == free) \Rightarrow 0 \quad (4)$$

In ProDEVS we assume that an atomic component features a set of state variables including exactly one of them named *phase* that can take many values as necessary from a comprehensive list of literal values. A value for this variable will be represented by a UML state. In the remainder of this paper, for simplicity, we will call a phase one value of state variable phase. Thus, in ProDEVS, time advance function and transition functions are associated to each value of phase, or simpler, to each phase, all which is actually restrictive according to DEVS specification.

However, within ProDEVS, the assignments are actions associated with the transitions and the boolean expressions are guards also associated with transitions. For example, a buffer that would be modeled without phase in DEVS will be modeled in ProDEVS by the state diagram in Figure 4 [2].

[2]A dotted arrow represents an internal transition and a full

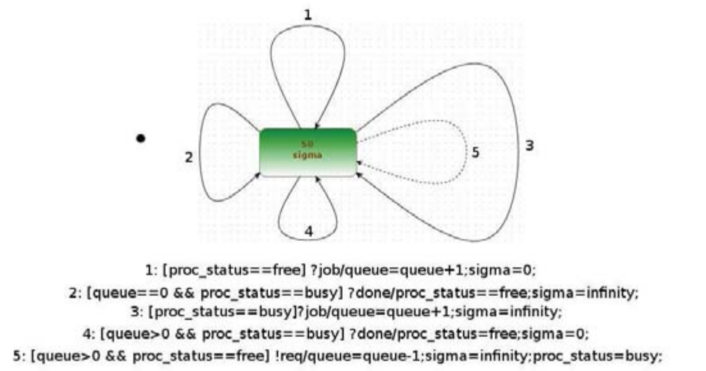

1
2
3
4
5
50
sigma
1: [proc_status==free] ?job/queue=queue+1;sigma=0;
2: [queue==0 && proc_status==busy] ?done/proc_status==free;sigma=infinity;
3: [proc_status==busy]?job/queue=queue+1;sigma=infinity;
4: [queue>0 && proc_status==busy] ?done/proc_status=free;sigma=0;
5: [queue>0 && proc_status==free] !req/queue=queue-1;sigma=infinity;proc_status=busy;

Figure 4: Atomic component BUF with one phase

This mechanism of actions and guards associated with transitions which are themselves a directed relationship between the phases can represent exactly a DEVS model. Indeed, the internal transition given by equation (1) is encoded by the transition 5 in Figure 4. Furthermore the time advance function given by equation (2) is encoded by the transition 4. The guard expresses the *current_state*, here, ($queue > 0$ && $proc\_status == busy$). The action contains the assignments on the state variables from the *current_state*, here, queue is unchanged and $proc\_status := free$. The assignment of the state variable *sigma* allows setting the time advance function for *next_state*.

In fact there are a multitude of ways to model a buffer always with the same behavior. For example it is possible to allocate the state variable *proc_status* in phases. Thus, BUF becomes as shown in Figure 5:

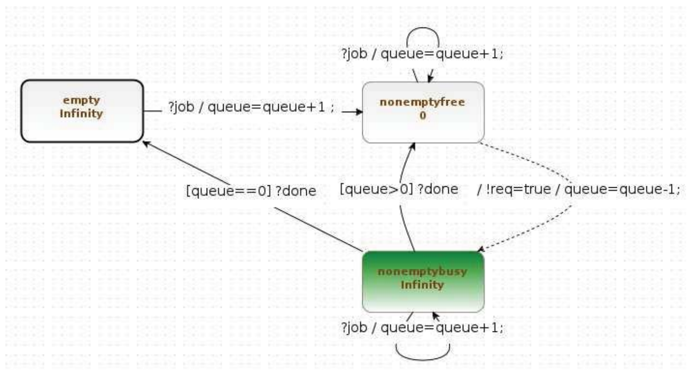


Figure 5: Atomic component BUF with three phases

The *finest* BUF model would allocate *queue* and *proc_status* into phases. This gives a model without guards and assignements as shown in Figure 6.

Continuous system.

Basically, the QSS method consists in discretizing the space of state variables using a fixed value called the *quantum* size D. According to this quantum, a variable q can only take values among $q \pm kD$ where k is an integer. The solution $q(t)$ of a system described by a differential equation is approximated on a grid in the phase space of the system. The resolution of the phase space grid is D. The time h required to move from one phase space grid point to another on $q(t)$ is approximated and a state change will be proceed only at this time.

arrow represents an external transition.

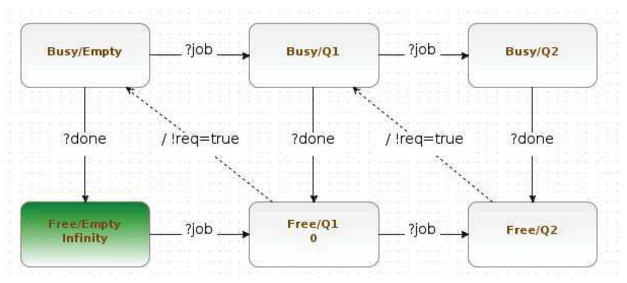


Figure 6: Atomic component BUF with six phases for a buffer with capacity of 2 (taken from [19] page 465).

As shown in [13], h may be computed from classical ODE solvers, e.g. for Euler or Runge-Kutta. Consider an ordinary differential equation in the form of

$$\dot{q} = f(q(t)) \tag{5}$$

We consider the simple Euler integration method

$$q(t+h) = q(t) + h\dot{q}(t) \tag{6}$$

Let the quantum D be defined by

$$D = |q(t+h) - q(t)| \tag{7}$$

Then the time required for a change of size D to occur on $q(t)$ is approximatively

$$h = \begin{cases} \frac{D}{|\dot{q}(t)|} & \text{if } \dot{q}(t) \neq 0 \\ \infty & \text{otherwise} \end{cases}$$

This approach can easily be extended to a set of coupled ordinary differential equations in the form of $\dot{q} = f(q(t))$, where q is a vector of differential variables. For each ordinary differential equation i of such a system, two variables are necessary. A variable q_i which is the position of state variable i on its phase space axis and a variable ql_i which contains the last grid point occupied by the variable q_i. These two variables are necessary because the function f_i is now computed at grid points in the discrete phase space of each element of the vector q. A variable q_j may have reached a grid point in its discrete phase space while the variable q_i has not reached its next grid point yet. Then the time required for the variable ql_i to be updated becomes

$$h = \begin{cases} \frac{D-|q_i-ql_i|}{|\dot{q}_i(t)|} & \text{if } \dot{q}(t) \neq 0 \\ \infty & \text{otherwise} \end{cases}$$

where $|q_i - ql_i|$ is the distance already traveled along the phase space axis of state variable i.

A DEVS description of the quantized integrator is:

$$\delta_{int}(ql, q, \dot{q}, \sigma) = (qn, qn, f(qn, x), \frac{d}{|f(qn, x)|}) \tag{8}$$

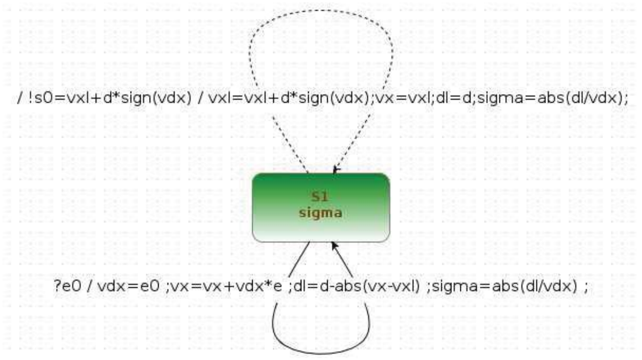
$$\delta_{con}(ql, q, \dot{q}, \sigma) = (qn, qn, f(qn, x), \frac{d}{|f(qn, x)|}) \quad (9)$$

$$\delta_{ext}((ql, q, \dot{q}, \sigma), e, x) = (ql, q + \dot{q} * e, f(q, x), \frac{d - |q + \dot{q} * e - ql|}{|f(q, x)|}) \quad (10)$$

$$\lambda(ql, q, \dot{q}, \sigma) = qn \quad (11)$$

$$ta(ql, q, \dot{q}, \sigma) = \sigma \quad (12)$$

with $qn = ql + d * sgn(\dot{q})$ the next value of the integral and function $sgn(v)$ returns -1 if $v < 0$, 0 if $v = 0$ or 1 if $v > 0$.



/ !s0=vxl+d*sign(vdx) / vxl=vxl+d*sign(vdx);vx=vxl;dl=d;sigma=abs(dl/vdx);
S1
sigma
?e0 / vdx=e0 ;vx=vx+vdx*e ;dl=d-abs(vx-vxl) ;sigma=abs(dl/vdx) ;

Figure 7: Quantized integrator

A ProDEVS State diagram for this quantized integrator is given Figure 7 [3]. It is a generic component that can be taken from the library to construct a model with only two parameters to define: the quantum d and the initial state value vdx.

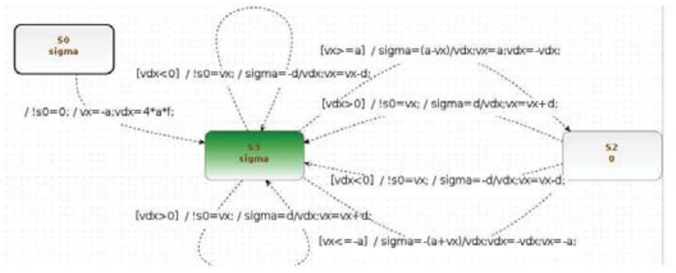
The atomic component figure 8 generates a triangular wave. The internal transition from phase s0 to phase s3 is used to initialise the component with the given parameters a for amplitude and f for frequency. vdx is the derivative and vx is the signal. From state s3 a new value of vx is sent on the output port s0 every $\sigma = d/abs(vdx)$ where d is the quantum. If vdx is positive (negative), vx is increased (decreased) by d. When vx reaches a or -a, the derivative is inverted. The transitions which map s3 to s2 must have higher priority than the transitions which map s3 to s3.

Hybrid system.

Hybrid systems are systems which features continuous and discrete dynamics. Typical application is switching circuit in electronics for controling a plant. The main problem when simulating this class of system is to handle a sudden and unexpected change of the behavior of the trajectory when integrating a continuous variable. The discontinuities in dynamic systems are linked to the notion of events, i.e. time event or state event. All discrete-time numerical integration algorithms used in simulation tools are based, either implicitly or explicitly on Taylor series expansions. The simulation trajectories are approximated by polynomial functions (never discontinuities) or rational (exhibits only poles but no discontinuity) in a step size h around the current time t_k.

So all these algorithms will have problems integrating a discontinuity. The algorithms will perceives a discontinuity because the trajectory has changed suddenly. This results in a new eigen value largely outside the complex plane. The algorithm will react by reducing the integration step to try to reduce this eigen value in the asymptotic region of the plane. But it will never succeed. It will reduce the integration step until the minimum tolerance is reached then give up (because the behavior is infinitely stiff). This never happens to QSS since it is the discontinuity itself which is responsible for the advancement of time. There is a simulation step only when the discontinuity occurs.

[3]Character e is a reserved character that may not be used as variable name of a component. It contains the elapsed time since the last transition



S0
sigma
[vx>=a] / sigma=(a-vx)/vdx;vx=a;vdx=-vdx;
[vdx<0] / !s0=vx; / sigma=-d/vdx;vx=vx-d;
[vdx>0] / !s0=vx; / sigma=d/vdx;vx=vx+d;
/ !s0=0; / vx=-a;vdx=4*a*f;
S3
sigma
S2
0
[vdx<0] / !s0=vx; / sigma=-d/vdx;vx=vx-d;
[vdx>0] / !s0=vx; / sigma=d/vdx;vx=vx+d;
[vx<=-a] / sigma=-(a+vx)/vdx;vdx=-vdx;vx=-a;

Figure 8: Triangular wave

Dymola and OpenModelica manage discontinuities using zero-crossing functions that are evaluated at each step. When one of these functions changes sign, an iterative process is performed in order to approximate the exact time of the event. The QSS methods provide dense outputs, so there is no need to iterate to detect a discontinuity, they are predicted. Everything that need to be ensured is that the quantum is able to discriminate the zero-crossing function. Quantization of the continuous variable is not handled automatically by the tool yet, it must be done externally by the user.

This feature, besides improving on the overall computational performance of these solvers, enables real-time simulation. Since in a real-time simulation the computational load per unit of real time must be controllable, Newton iterations are usually not admitted for use in real-time simulation.

The next section introduces the simulation of an hybrid system. The principle is to generate a logical signal (being 0 or 1), with high frequency but with a duty cycle digitally controlled. The average of the output signal is equal to the duty cycle.

5. EXAMPLE

This section describes an example that shows simulation results within ProDEVS. The application is the control of a DC motor with pulse width modulation. It is taken from the hybrid example folder in PowerDEVS software and detailed in [3]. In this application the power signal of the motor is replaced by a switch in which the duration of the *on* state is proportional to the desired voltage. The controller compares the speed $\omega(t)$ of the motor with the input reference, and calculates the desired input voltage of the motor, u_{ref}. The desired voltage is compared with a fast triangular waveform, obtaining the actual input voltage u_a that oscillates between

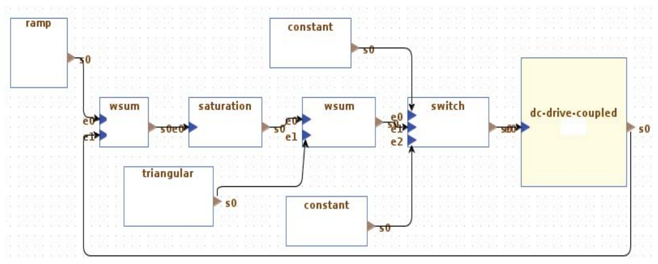


Figure 9: dc-drive model level 0

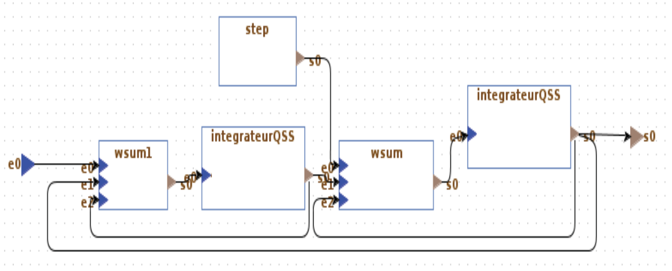


Figure 10: dc-drive model level 1 (dc-drive-coupled)

two values.

The whole control and plant systems were built using blocks from ProDEVS library which are atomic components described with ProDEVS State diagram. The motor is represented by a second order model (see Figure 10). A torque step is applied after 3 seconds of simulation. The control system is composed of the following components (see Figure 9):

- The input voltage of the motor switches between +500V and -500V depending on the PWM control law.
- For the PWM law, a triangular waveform of 1KHz frequency and an amplitude of 1.1V is considered. The moment at which the triangular wave crosses the value given by u_{ref}, determines the actual voltage applied to the motor.
- The control is using a proportional law, i.e. u_{ref} is proportional to the error $\omega_{ref}(t) - \omega(t)$.
- The angular velocity reference signal, $\omega_{ref}(t)$ is a ramp signal that increases from 0 to 60 rad/sec in 2 seconds.

The simulation results are shown in Figure 11. A simulation across 5 seconds of simulation time took about 27 seconds on a 1.80GHz running under Ubuntu. The same experiment with PowerDEVS took about 15 seconds. The real time of the simulation can be improved using QSS2 methods (205ms with PowerDEVS). We think we can improve the performance of the tool with a different storage management for variable values. At each step of the simulation values of each port and variable is stored as type String. The tool performs a type conversion in each step to make the calculations. This cast is expensive. Furthermore we believe we can optimize simulation engine processing such as searching ports connection through the hierarchy.

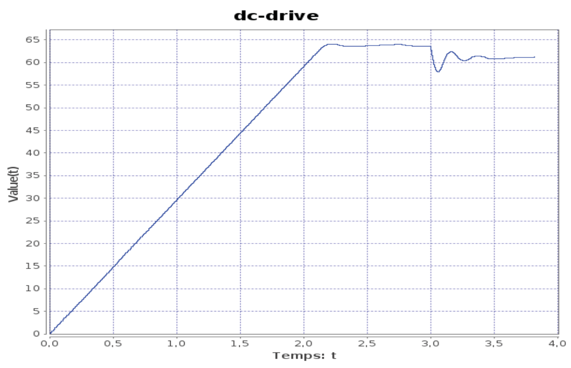


Figure 11: Output of the DC motor

6. CONCLUSIONS

This paper has introduce a new event-driven modeling and simulation tool for the simulation of hybrid systems. The particularity of this software lies in its graphical language to define the behavior of the atomic components. This graphical language customizes state diagrams for DEVS and quantized based numerical methods. ProDEVS proposes an intuitive and user-friendly interface for designing DEVS model without experience in programming. It provides a block diagram interface for coupled components where users can go up and done the hierarchy with a just a double-click. A library promotes the reuse of models where parameters can be tuned without entering the atomic component. This library can be extended by users thanks to the export feature by a simple drag and drop from the diagram panel to the library panel. ProDEVS provides a model verifier to quickly detect inconsistencies and incompleteness.

The example of the DC-motor demonstates the usability of the graphical language. This example was developed using only components of the library never requiring to look inside an atomic component. However, the development of new atomic component can quickly become impractical using the graphical language, just like stateflow in Matlab/Simulink. Also, the example demonstrates the expressiveness of the graphical language since discrete-event, discrete-time and continuous systems coexist in a same environment.

We are now developing a formal verification feature of a ProDEVS model. Our approach is based on a transformation to extended TPN (Timed Petri Nets) with data handling called (TTS) Time Transition Systems to generate a finite representation for the accessible states of a ProDEVS model. This feature will be used for detection of possible deadlock or model illegitimacy.

Finally we are considering a VHDL coder for hardware implementation of parallel architectures on FPGA to boost the performance and provide rapid prototyping features.

7. REFERENCES

[1] F. Bergero and E. Kofman. Powerdevs: a tool for hybrid system modeling and real-time simulation. *SIMULATION*, 2010.

[2] B. Berthomieu, F. Peres, and F. Vernadat. Model checking bounded prioritized time petri nets. In *ATVA*, Lecture Notes in Computer Science. Springer,

2007.

[3] F. E. Cellier and E. Kofman. *Continuous System Simulation*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[4] R. Franceschini, P.-A. Bisgambiglia, P. Bisgambiglia, and D. Hill. Devs-ruby: A domain specific language for devs modeling and simulation (wip). In *Proceedings of the Symposium on Theory of Modeling & Simulation - DEVS Integrative*, San Diego, CA, USA, 2014.

[5] R. Franceschini, P.-A. Bisgambiglia, L. Touraille, P. Bisgambiglia, and D. Hill. A survey of modelling and simulation software frameworks using Discrete Event System Specification. In *2014 Imperial College Computing Student Workshop*, Dagstuhl, Germany, 2014.

[6] M. H. Hwang. Qualitative verification of finite and real-time devs networks. In *Proceedings of the 2012 Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium*, San Diego, CA, USA, 2012.

[7] E. Kofman. A Second Order Approximation for DEVS Simulation of Continuous Systems. *Simulation: Transactions of the Society for Modeling and Simulation International*, 78(2):76–89, 2002.

[8] E. Kofman. A Third Order Discrete Event Simulation Method for Continuous System Simulation. *Latin American Applied Research*, 36(2):101–108, 2006.

[9] E. Kofman and S. Junco. Quantized State Systems. A DEVS Approach for Continuous System Simulation. *Transactions of SCS*, 18(3):123–132, 2001.

[10] C. Laroque, J. Himmelspach, R. Pasupathy, O. Rose, A. M. Uhrmacher, M. A. Hamri, and G. Zacharewicz. Automatic generation of object-oriented code from devs graphical specifications.

[11] G. Migoni, M. Bortolotto, E. Kofman, and F. Cellier. Linearly Implicit Quantization-Based Integration Methods for Stiff Ordinary Differential Equations. *Simulation Modelling Practice and Theory*, 35:118–136, 2013.

[12] J. Nutaro. Adevs (a discrete event system simulator). *Arizona Center for Integrative Modeling and Simulation (ACIMS), University of Arizona, Tucson*, 1999.

[13] J. Nutaro. *Discrete event simulation of continuous systems*. In Fishwick, P. (Ed.), Handbook of Dynamic System Modeling, 2007.

[14] H. Praehofer and D. Pree. Visual modeling of devs-based multiformalism systems based on higraphs. In *Proceedings of the 25th Conference on Winter Simulation*, WSC '93, pages 595–603, New York, NY, USA, 1993. ACM.

[15] G. Quesnel, R. Duboz, and E. Ramat. The Virtual Laboratory Environment – An operational framework for multi-modelling, simulation and analysis of complex dynamical systems. *Simulation Modelling Practice and Theory*, 2009.

[16] Y. Van Tendeloo and H. Vangheluwe. The modular architecture of the python(p)devs simulation kernel: Work in progress paper. In *Proceedings of the Symposium on Theory of Modeling & Simulation - DEVS Integrative*, San Diego, CA, USA, 2014.

[17] G. A. Wainer. Cd++: a toolkit to define discrete-event models. *Software, Practice and Experience. Wiley*, 2002.

[18] B. Zeigler and H. Sarjoughian. Introduction to devs modeling and simulation with java: A simplified approach to hla-compliant distributed simulations. *Arizona Center for Integrative Modeling and Simulation*, 2000.

[19] B. P. Zeigler, T. G. Kim, and H. Praehofer. *Theory of Modeling and Simulation*. Academic Press, Inc., Orlando, FL, USA, 2nd edition, 2000.

[20] S. Zinn, J. Himmelspach, A. M. Uhrmacher, and J. Gampe. Building mic-core, a specialized m&s software to simulate multi-state demographic micro models, based on james ii, a general m&s framework. *Journal of Artificial Societies and Social Simulation*, 16(3):5, 2013.