DEVS as a Common Denominator for Multi-formalism Hybrid Systems Modelling

Hans L.M. Vangheluwe
School of Computer Science, McGill University
McConnell Engineering Bldg. room 328
3480 University Street
Montreal, Quebec, Canada H3A 2A7
hv@cs.mcgill.ca

Abstract

When modelling complex systems, complexity is usually not only due to a large *number* of coupled components (sub-models), but also to the (perceived) *diversity* of these components and to their intricate interactions (*i.e.*, high degree of feedback). One would like to use a variety of formalisms (DAE, Bond Graph, Forrester System Dynamics, Petri Nets, Finite State Automata, DEVS, State Charts, Queueing networks, ...) to "optimally" describe the behaviour of different system components, aspects, and views. The choice of appropriate formalisms depends on criteria such as the application domain, the modeler's background, the goals, and the available computational resources.

In this article, a Formalism Tranformation Graph (FTG) is presented. In the FTG, vertices correspond to formalisms, and edges denote existing formalism transformations. A transformation is a *mapping* of models in the source formalism onto models in the destination formalism (with behaviour invariance). This traversal allows for meaningfully coupling models in different semantics. Once mapped onto a common formalism, closure under coupling of that formalism makes the meaning of the coupled model explicit.

In the context of hybrid systems models, the formalism transformation converges to a common denominator which unifies continuous and discrete constructs. Often, this is some form of event-scheduling/state-event locating/DAE formalism and corresponding solver. A different approach is presented which maps all formalisms, and in particular continuous ones, onto Zeigler's DEVS. Hereby, the state variables of the continuous models are discretized rather than time and the DEVS transition function progresses from one discretized state value to either the one just above or the one just below, giving as output, the time till the next transition.

1 Multi-formalism Modelling

Complex systems are characterized, not only by a large number of components, but also by the diversity of these components. For the analysis and design of such complex systems, it is not sufficient to study the diverse components in isolation, using the specific formalisms these components were modelled in. Rather, it is necessary to answer questions about properties (most notably behaviour) of the overall *multi-formalism* system.

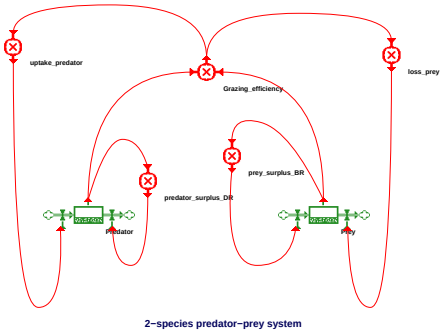

Figure 1: Predator-prey, System Dynamics Formalism

1.1 The Forrester System Dynamics Formalism

To introduce formalism transformation, we briefly present the semantics of the Forrester System Dynamics formalism [1] often used to model biological, sociological, and economical systems. It describes the variation of material-like quantities or levels. The variation is determined by birth rates (*BR*) and death rates (*DR*). *BR* and *DR* are graphically represented as valves to the left and right respectively of boxes denoting the levels. Levels may influence each other by influencing each other's *BR* and *DR*. Figure 1 shows a typical predator-prey interaction. The (product) interaction between predator and prey populations influences the predator's birth rate and the prey's death rate. The System Dynamics semantics is given by mapping each of the level/*BR*/*DR* combinations onto an Ordinary Differential Equation

$$\frac{d\ level}{dt} = BR - DR.$$

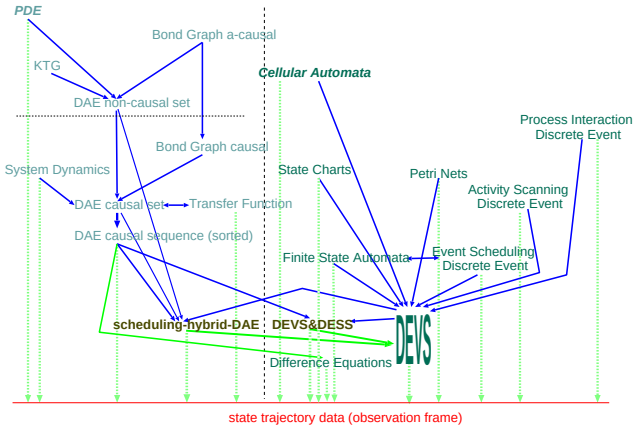

Figure 2: Formalism Transformation Graph

The operations such as product and sum are mapped onto the appropriate algebraic equations and couplings are mapped onto algebraic equalities.

1.2 Formalism Transformation

Based on the mathematical relationship between the System Dynamics and the ODE formalisms, translation of any model in the first formalism to a behaviourally equivalent model described in the second formalism is possible. In Figure 2, a part of "formalism space" is depicted in the form of a Formalism Transformation Graph (FTG). The different formalisms are shown as nodes in the graph. The vertical striped line in the middle denotes the distinction between continuous models (on the left) and discrete models (on the right). The well known difference equations formalism is often implicitly used in numerical simulators: ODEs are discretised by means of a suitable numerical scheme and the resulting difference equations are iteratively solved. Suitable refers to the nature of the equations as well as to the accuracy requirements. The arrows denote a homomorphic relationship "can be mapped onto", implemented as a symbolic transformation between formalisms. The vertical, dotted lines denote the existence of a *solver* or *simulation kernel* which is capable of simulating a model, thus generating a trajectory. A trajectory is really a model of the system in the data formalism (time/value tuples). In a denotational sense, traversing the graph makes *semantics* of models in formalisms explicit: the meaning of a model/formalism is given by mapping it onto some known formalism. This procedure can be applied iteratively to reach any desired (reachable) level. In an operational sense, a mapping describes how model interpretation can be achieved. If the "trajectory" formalism is the target of the mapping, model interpretation is model simulation. Though a multi-step mapping may seem cumbersome, it can be perfectly and correctly performed by tools. The advantage of this approach is that the introduction of a new formalism only requires the description of the mapping onto the nearest formalism as well as the implementation of a translator to the latter formalism. It is often meaningful to introduce a new formalism for a specific application, encoding particular properties and constraints of the application. Often, translation involves some loss of information, though behaviour must obviously be conserved. This loss may be a blessing in disguise as it entails a reduction in complexity, leading to an increase in (simulation) performance. Usually, the aim of multi-step mapping is to eventually reach the trajectory level.

Another major use for formalism transformation is the *answering of particular questions* about the system. Some questions can only be answered in the context of a particular formalism. In case of a System Dynamics model for example, the visual inspection of the model can provide insight into influences. If the model is mapped onto a set of Algebraic and Ordinary Differential Equations, a dependency analysis may reveal algebraic dependency cycles not apparent at the System Dynamics level. At this same level, one may check whether parts of the model are linear. If so, these parts may be solved symbolically by means of computer algebra. Also, transformation to the Laplace domain (*i.e.,* to a Transfer Function form) opens avenues to a plethora of techniques for stability analysis. Finally, the transformation, through numerical simulation to the data level allows for quantitative analysis of problems posed in initial value form. Note how the larger the number of intermediate formalisms, the higher the possibility for optimization along the way.

Above all, the traversal described above is the basis for the meaningful coupling of models described in different formalisms. This is discussed next.

1.3 Coupled Model Transformation

When we describe a structured model in the *network* or *coupled* formalism, we can only make meaningful assertions about its structure, its outside connections (interface) and its components, not about its overall meaning or behaviour. Formally, a coupled model has the form

$$CM \equiv \langle id, interface, S, C \rangle$$

The model is identified by a unique identifier *id* (a name or reference). The $interface$ is a set of connectors or ports to the environment. Associated with the ports are allowed values as well as causality. Meaningful causalities are $\{in, out, inout\}$. The set S contains the sub-models (or at least their unique identifiers). The coupling information is contained in a graph structure C. For non-causal, continuous models, the graph is undirected. For causal models, the graph is directed. Obviously, a coupled model is only valid if types and causalities of connected ports are compatible. In certain cases, the graph may be annotated with extra information. In case of traditional discrete-event models, a tie-breaking function is usually required to select between simultaneous events [4].

If all sub-models are described in the *same* formalism F, it may be possible to replace the coupled model (at least conceptually) by one atomic model of type F. In this case, F is called *closed under coupling* (or under composition). The property often holds by construction. In case

of Differential Algebraic Equations (DAEs), connections $\{connect(port_i, port_j)\}$ are replaced by algebraic $port_i = port_j$ coupling equations. Together with the sub-models' equations, these form a DAE. In formalisms such as Bond Graphs, information about the physical nature of variables allows one to generate either the above type of equations in case of coupling of across variables (this corresponds to Kirchoff's voltage law in electricity) or an equation summing all connected values to zero for through variables (this corresponds to Kirchoff's current law in electricity). In discrete-event models, implementing closure involves the correct time-ordered *scheduling* of sub-model events. The most imminent event will always be processed first. The tie-breaking function is used to resolve conflicts due to simultaneous events (an artifact of the high level of abstraction).

If a coupled model consists of sub-models expressed in *different formalisms*, several approaches are possible:

- A *meta-formalism* can be used which subsumes the different formalisms of the sub-models. The different sub-models are thus described in a single formalism. The Hybrid DAE and DEVS&DESS [6] formalisms integrate continous and discrete modelling constructs. Meaningful meta-formalisms which truly add expressiveness as well as reduce complexity are rare. Bond Graphs are a good example of the integration of different domains (mechanical, electrical, hydraulical).
- Another approach is to *transform* the different sub-models to one *common formalism*. Which formalism to transform to depends on the questions asked. The closest common formalism for DAE and System Dynamics formalisms for example is the DAE formalism. By transforming a System Dynamics model to a set of DAEs, and using the closure property of the DAE formalism, it becomes possible to answer questions about the overall model.
- In the *co-simulation* approach, each of the sub-models is simulated with a formalism-specific simulator. Interaction due to coupling is resolved at the trajectory level. Compared to transformation to a common formalim before simulation, this approach, though appealing from a software engineering point of view (it is object-oriented) discards a lot of useful information. Questions can *only* be answered at the trajectory level. Furthermore, there are obvious speed and numerical accuracy problems for continuous formalisms in particular if one attempts to support non-causal models. The approach is meaningful mostly for discrete-event formalisms. In this realm, it is the basis of the DoD High Level Architecture (HLA) for simulator interoperability.

The transformation to a common formalism mentioned above proceeds as follows:

1. Start from a coupled multi-formalism model. Check consistency of this model (*e.g.*, whether causalites and types of connected ports match).
2. Cluster all formalisms described in the same formalism.
3. For each cluster, implement closure under coupling.
4. Look for the best common formalism in the Formalism Transformation Graph all the remaining different formalisms can be transformed to. In the worst case, this will be the trajectory level in which case the approach falls back to co-simulation. Which common formalism is best depends on a quality metric which can take into account transformation speed, potential for optimization, *etc.*
5. Transform all the sub-models to the common formalism.
6. Implement closure under coupling of the common formalism.

A side-effect of mapping onto a common formalism is the great potential for optimization of the flattened model, as well as the reduced number of (optimized) simulation kernels needed.

To describe which formalism transformation are possible, the Formalism Transformation Graph (FTG) mentioned above is used. A plethora of formalisms is depicted in Figure 2. Each of these has its own merits. Petri Nets are particularly suited for symbolic analysis (proof of dynamic properties) of concurrent systems. State Charts, an extension of Finite State Automata are a graphical formalism with a very appealing, intuitive semantics. In the UML, the State Chart formalism is used to specify the concurrent behaviour of software. Cellular Automata extend Finite State Automata with a (discretized) notion of space. As such, they are similar to Partial Differential Equations which add a spatial dimension to Ordinary Differential Equations. Apart from the formalism transformations described earlier, the central position of DEVS is striking. On the one hand, the expressiveness of DEVS makes many discrete-event formalisms DEVS-representable. Recently, it has been shown that continuous models can be quantized and described in the DEVS [5] formalism. This mapping will be described further on. It allows one to meaningfully handle discrete/continuous multi-formalism models. Also, the potential for parallel implementation increases drastically [3].

2 The DEVS Formalism

For the class of formalisms denoted as *discrete-event*, system models are described at an abstraction level where the time base is continuous ($\mathbb{R}$), but during a bounded time-interval, only a *finite number* of relevant *events* occur. State variables are considered to change instantaneously, only at event-times. Between events, the state of the system does *not* change. This is unlike *continuous* models where the state of the system may change continuously over time. The Discrete EVent system Specification (DEVS) was introduced by Zeigler [4] as a rigourous basis for discrete-event modelling and simulation. The semantics of well known

discrete-event formalisms such as Event Scheduling, Activity Scanning, and Process Interaction can be expressed in terms of DEVS, making it a common denominator for the representation of discrete-event models. DEVS allows for the description of system behaviour at two levels. At the lower level, an *atomic DEVS* describes the autonomous behaviour of a discrete-event system as a sequence of deterministic transitions between sequential states as well as how it reacts to external input events and how it generates output events. At the higher level, a *coupled DEVS* describes a system as a *network* of coupled components. The components can be atomic DEVS models or coupled DEVS in their own right. The connections denote how components influence each other. In particular, output events of one component can become, via a network connection, input events of another component. It is shown in [4] how the DEVS formalism is *closed under coupling*: for each coupled DEVS, a *resultant* atomic DEVS can be constructed. As such, any DEVS model, be it atomic or coupled, can be replaced by an equivalent atomic DEVS. The construction procedure of a resultant atomic DEVS is also the basis for the implementation of an *abstract simulator* or *solver* capable of simulating any DEVS model. As a coupled DEVS may have coupled DEVS components, *hierarchical* modelling is supported. In the following, the DEVS formalism is presented in more detail.

2.1 The atomic DEVS Formalism

The atomic DEVS formalism is a structure describing the different aspects of the discrete-event behaviour of a system:

$$atomicDEVS \equiv \langle S, ta, \delta_{int}, X, \delta_{ext}, Y, \lambda \rangle.$$

The *time base* T is continuous ($= \mathbb{R}$) and is not mentioned explicitly.

The *state set* S is the set of admissible *sequential states*: the DEVS dynamics consists of an ordered sequence of states from S. Typically, S will be a *structured* set (a product set) $S = \times_{i=1}^{n} S_i$. This formalizes multiple (n) *concurrent* parts of a system. It is noted how a structured state set is often synthesized from the state sets of concurrent components in a coupled DEVS model.

The time the system *remains* in a sequential state before making a transition to the next sequential state is modelled by the *time advance function*

$$ta : S \rightarrow \mathbb{R}^{+}_{0,+\infty}.$$

As time in the real world always advances, the image of ta must be non-negative numbers. $ta = 0$ allows for the representation of *instantaneous* transitions: no time elapses before transition to a new state. Obviously, this is an abstraction of reality which may lead to simulation *artifacts* such as infinite instantaneous loops which do not correspond to real physical behaviour. If the system is to stay in an end-state s^* *forever*, this is modelled by means of $ta(s^*) = +\infty$.

The internal transition function $\delta_{int} : S \rightarrow S$ models the transition from one state to the next sequential state. δ_{int} describes the behaviour of a Finite State Automaton; ta adds the progression of time.

It is possible to *observe* the system output. The output set Y denotes the set of admissible *outputs*. Typically, Y will be a *structured* set (a product set) $Y = \times_{i=1}^{l} Y_i$. This formalizes multiple (l) output ports. Each port is identified by its unique index i. In a user-oriented modelling language, the indices would be unique port *names*.

The output function $\lambda : S \rightarrow Y \cup \{\phi\}$ maps the internal state onto the output set. Output events are *only* generated by a DEVS model at the time of an *internal* transition. At that time, the state *before* the transition is used as input to λ. At all other times, the non-event ϕ is output.

To describe the *total state* of the system at each point in time, the sequential state $s \in S$ is not sufficient. The *elapsed* time e since the system made a transition to the current state s needs also to be taken into account to construct the total state set $Q = \{(s,e) | s \in S, 0 \leq e \leq ta(s)\}$. The elapsed time e takes on values ranging from 0 (transition just made) to $ta(s)$ (about to make transition to the next sequential state).

Up to now, only an *autonomous* system was described: the system receives no external inputs. Now, the *input set* X denoting all admissible input values is defined. Typically, X will be a *structured* set (a product set) $X = \times_{i=1}^{m} X_i$. This formalizes multiple (m) input ports. Each port is identified by its unique index i. As with the output set, port indices may denote *names*.

The set Ω contains all admissible input segments $\omega : T \rightarrow X \cup \{\phi\}$. In discrete-event system models, an input segment generates an input *event* different from the *non-event* ϕ only at a finite number of instants in a bounded time-interval. These *external events*, inputs x from X, cause the system to interrupt its autonomous behaviour and react in a way prescribed by the external transition function $\delta_{ext} : Q \times X \rightarrow S$. The reaction of the system to an external event depends on the sequential state the system is in, the particular input *and* the elapsed time. Thus, δ_{ext} allows for the description of a large class of behaviours typically found in discrete-event models (including synchronization, preemption, suspension and re-activation).

When an input event x to an atomic model is not listed in the δ_{ext} specification, the event is *ignored*.

In Figure 3, an example state trajectory is given for an atomic DEVS model. In the figure, the system made an internal transition to state $s2$. In the absence of external input events, the system stays in state $s2$ for a duration $ta(s2)$. During this period, the elapsed time e increases from 0 to $ta(s2)$, with the total state $= (s2,e)$. When the elapsed time reaches $ta(s2)$, first an output is generated: $y2 = \lambda(s2)$, then the system transitions instantaneously to the new state $s4 = \delta_{int}(s2)$. In autonomous mode, the system would stay in state $s4$ for $ta(s4)$ and then transition (after generating output) to $s1 = \delta_{int}(s4)$. Before e reaches $ta(s4)$ however, an external input event x arrives. At that time, the system forgets about the scheduled internal transition and transitions to $s3 = \delta_{ext}((s4,e),x)$. Note how an external transition

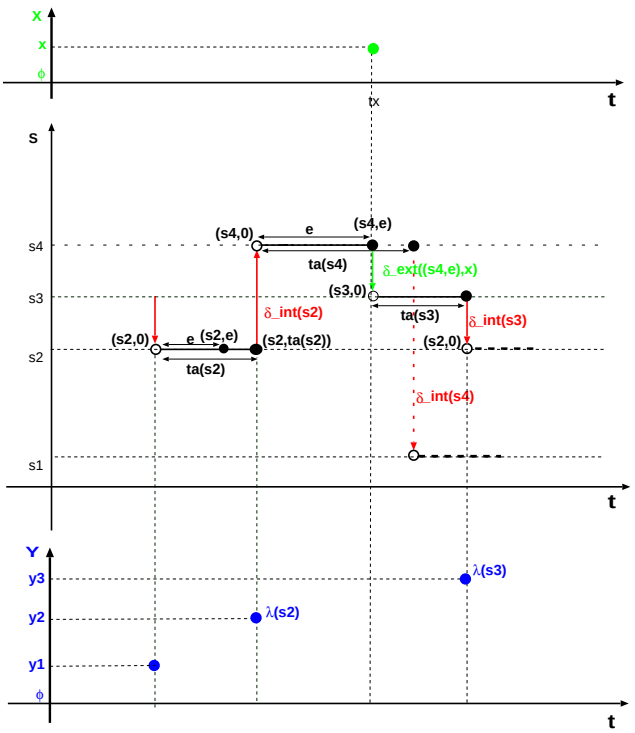

Figure 3: State Trajectory of a DEVS-specified Model

does not give rise to an output. Once in state $s3$, the system continues in autonomous mode.

2.2 The coupled DEVS Formalism

The coupled DEVS formalism describes a discrete-event system in terms of a network of coupled components.

$$coupledDEVS \equiv \langle X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, select \rangle$$

The component $self$ denotes the coupled model itself. X_{self} is the (possibly structured) set of allowed external inputs to the coupled model. Y_{self} is the (possibly structured) set of allowed outputs of the coupled model. D is a set of unique component references (names). The coupled model itself is referred to by means of $self$, a unique reference not in D.

The set of *components* is $\{M_i | i \in D\}$. Each of the components must be an atomic DEVS

$$M_i = \langle S_i, ta_i, \delta_{int,i}, X_i, \delta_{ext,i}, Y_i, \lambda_i \rangle, \forall i \in D.$$

The set of *influencees* of a component, the components influenced by $i \in D \cup \{self\}$, is I_i. The set of all influencees describes the coupling network structure $\{I_i | i \in D \cup \{self\}\}$. For modularity reasons, a component (including $self$) may not influence components outside its scope –the coupled model–, rather only other components of the coupled model, or the coupled model $self$: $\forall i \in D \cup \{self\} : I_i \subseteq D \cup \{self\}$. This is further restricted by the requirement that none of the components (including $self$) may influence themselves directly as this could cause an instantaneous dependency cycle (in case of a 0 time advance inside such a component) akin to an algebraic loop in continuous models: $\forall i \in D \cup \{self\} : i \notin I_i$. Note how one can always encode a self-loop ($i \in I_i$) in the internal transition function.

To translate an output event of one component (such as a departure of a customer) to a corresponding input event (such as the arrival of a customer) in influencees of that component, *output-to-input translation functions* $Z_{i,j}$ are defined:

$$\{Z_{i,j} | i \in D \cup \{self\}, j \in I_i\},$$

$$\begin{array}{rcll} Z_{self,j} & : & X_{self} \to X_j & , \forall j \in D, \\ Z_{i,self} & : & Y_i \to Y_{self} & , \forall i \in D, \\ Z_{i,j} & : & Y_i \to X_j & , \forall i, j \in D. \end{array}$$

Together, I_i and $Z_{i,j}$ completely specify the coupling (structure and behaviour).

As a result of coupling of concurrent components, multiple state transitions may occur at the same simulation time. This is an artifact of the discrete-event abstraction and may lead to behaviour not related to real-life phenomena. In sequential simulation systems, such transition *collisions* are resolved by means of some form of *selection* of which of the components' transitions should be handled first. This corresponds to the introduction of priorities in some simulation languages. The coupled DEVS formalism explicitly represents a *select* function for *tie-breaking* between simultaneous events: $select : 2^D \to D$. *select* chooses a unique component from any non-empty subset E of D: $select(E) \in E$. The subset E contains all components having a state transition simultaneously.

A DEVS solver or simulation kernel is based on the closure under coupling construction and can be used as a specification of a –possibly parallel– implementation of a DEVS solver or "abstract simulator" [4, 3]. The core of the closure procedure is the selection of the most *imminent* (*i.e.*, soonest to occur) event from all the components' scheduled events. In case of simultaneous events, the *select* function is used for tie-breaking.

2.3 The parallel DEVS Formalism

As DEVS is a formalization and generalization of sequential discrete-event simulator semantics, it offers little scope for parallel implementation. In particular, simultaneously occurring internal transitions are serialized by means of a tie-breaking *select* function. Also, in case of *collisions* between simultaneously occurring internal transitions and external input, DEVS ignores the internal transition and applies the external transition function. Chow [2] introduced the parallel DEVS (P-DEVS) formalism which alleviates some of the DEVS drawbacks. In an atomic P-DEVS

$$atomic\ P-DEVS \equiv \langle S, ta, \delta_{int}, X, \delta_{ext}, \delta_{conf}, Y, \lambda \rangle,$$

the model can explicitly define collision behaviour by using a so-called *confluent* transition function δ_{conf}.

3 Mapping the ODE Formalism onto DEVS

In the context of hybrid systems models, the formalism transformations in Figure 2 converge to a common denomi-

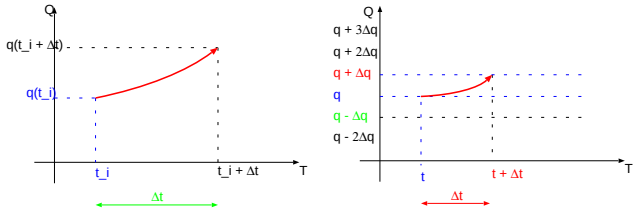

Figure 4: Time discretisation *vs.* State-discretisation

nator which unifies continuous and discrete constructs. Typically, this is some form of event-scheduling/state event locating/DAE formalism with its corresponding solver. A different approach, quantizing space rather than discretizing time, is presented here which maps continuous models, in particular algebraic and differential equations of the form

$$\begin{cases} \frac{dq}{dt} & = & f(q,x,t) & q \in Q \\ y(t) & = & g(q,t) & y \in Y \end{cases}$$

with $x(t) \in X$ a known input function, and initial conditions given by $q(0) = q_0$, onto Zeigler's DEVS formalism presented above. If the mapping is done appropriately, a discrete-event simulation of the DEVS model will yield a close approximation of the continuous model's continuous behaviour.

The normal approach of numerical mathematics is to approximate an ODE solution based on a Taylor expansion. Hereby, time is discretised, and subsequent state-variable approximations are calculated. Zeigler [5] proposes to discretize the state variables and to calculate the corresponding approximate time-increases. The DEVS transition function (constructed from the ODE) will repeatedly go from one discretised state value to either the one just above or the one just below. The transition function will also calculate the time till the next discrete transition (possibly $+\infty$ if the derivative is zero). Both approaches are shown side by side in Figure 4. In mathematical terms, the model above is mapped onto a DEVS

$$atomicDEVS \equiv \langle \hat{S}, ta, \delta_{int}, \hat{X}, \delta_{ext}, \hat{Y}, \lambda \rangle.$$

$\hat{x} \in \hat{X}$, $\hat{q} \in \hat{Q}$ and $\hat{y} \in \hat{Y}$ are the quantized variables. The simple quantization used here is based on a grid of quanta (Δ_x, Δ_q, Δ_y). Note how each of the quanta are hypercubes. The quantized state set $\hat{S} = \{(\hat{q},\hat{x},t) | \hat{q} \in \hat{Q}, \hat{x} \in \hat{X}, t \in T\}$. A memory of input and absolute time is kept in the DEVS model. The internal transition function

$$\delta_{int}((\hat{q},\hat{x},t)) = (\hat{q} + sgn(f(\hat{q},\hat{x},t))\Delta_q, \hat{x}, t + ta(\hat{q},\hat{x},t)).$$

The time advance function

$$ta((\hat{q},\hat{x},t)) = \left| \frac{\Delta_q}{f(\hat{q},\hat{x},t)} \right|$$

specifies after how much time the trajectory will leave the quantum hypercube. The external transition function describes how autonomous (integration) behaviour can be interrupted by an external input event (the input function exceeding a quantum boundary)

$$\delta_{ext}((\hat{q},\hat{x},t),e,\hat{x}') = (\hat{q},\hat{x}',t+e).$$

Note how ignoring the change in q (not $\hat{q}$) during e is a rough approximation and a better approach is to internally keep track of the non-quantized value q. In case an internal *and* external transition occur simultaneously, the confluent transition function of parallel DEVS describes how internal transition and external input are both taken into account

$$\delta_{confl}((\hat{q},\hat{x},t),\hat{x}') = (\hat{q} + sgn(f(\hat{q},\hat{x},t))\Delta_q, \hat{x}', t).$$

Quantized output is obtained after application of the output function

$$\lambda((\hat{q},\hat{x},t)) = \lfloor g(\hat{q},t)/\Delta_y \rfloor.$$

Coupling of the thus obtained atomic DEVS models into a coupled DEVS provides a means for –possibly parallel– simulation of hierarchically coupled continuous models. To achieve maximum performance, equations should first be symbolically manipulated and tightly coupled sets of equations must be clustered inside an atomic DEVS.

References

[1] François E. Cellier. *Continuous System Modeling*. Springer-Verlag, New York, 1991.

[2] A.C.-H. Chow. Parallel DEVS: A parallel, hierarchical, modular modeling formalism and its distributed simulator. *Transactions of the Society for Computer Simulation International*, 13(2):55–68, June 1996.

[3] Ki Hyung Kim, Yeong Rak Seong, Tag Gon Kim, and Kyu Ho Park. Distributed simulation of hierarchical DEVS models: Hierarchical scheduling locally and time warp globally. *Transactions of the Society for Computer Simulation International*, 13(3):135–154, September 1996.

[4] Bernard P. Zeigler. *Multifacetted Modelling and Discrete Event Simulation*. Academic Press, London, 1984.

[5] Bernard P. Zeigler and J.S. Lee. Theory of quantized systems: Formal basis for DEVS/HLA distributed simulation environment. In Alex F. Sisti, editor, *Enabling Technology for Simulation Science II, SPIE AeroSense 98*, volume 3369, pages 49–58, August 1998. Orlando, FL.

[6] Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of Modelling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press, second edition, 2000.