Implementation of Hybrid Systems Described by DEV&DESS in the QSS Based Simulator PowerDEVS

Franz Preyser[1], Irene Hafner[2], Matthias Rößler[2]
[1]Institute for Analysis and Scientific Computing, Vienna University of Technology, Wiedner Haupstraße 8-10, 1040 Vienna, Austria
[2]Simulation Services, dwh GmbH, Neustiftgasse 57-59, 1070 Vienna, Austria
franz.preyser@tuwien.ac.at

In this article, a method for implementing hybrid models, formulated as DEV&DESS, in the QSS based simulator PowerDEVS is presented. PowerDEVS is actually a pure DEVS simulator. However, it is specialised for simulating continuous Models (DESS) using QSS (Quantized State System) to translate them into discrete event models (DEVS). Therefore, it is perfectly suited for the simulation of hybrid models (DEV&DESS). When designing and simulating coupled DEVS models though, very soon a lot of difficulties occur caused by concurrent events and feedback loops. Hence, first some concepts are introduced of how to implement an atomic DEVS in a way that avoids those difficulties. Afterwards, an approach of how to implement an atomic DEV&DESS is introduced which makes use of those concepts.

1 Introduction

The factor 'costs for energy consumption' is gaining more and more importance in terms of production process optimization. Since most of the time energy intense processes in a production line are of continuous nature a demand is rising for being able to formulate those continuous aspects in addition to the usual discrete logistical aspects in one simulateable model [9]. See [11] for preceding work on this issue.

Herbert Praehofer introduced DEV&DESS (Discrete Event & Differential Equation System Specification) [10] for the formal description of hybrid systems. DEV&DESS is based on the two formalisms DEVS and DESS invented by Bernard Zeigler [13] for the description of discrete event systems and continuous systems, respectively.

However, as digital computers work in a purely discrete way, DESS models and therefore also DEV&DESS models have to be discretised somehow before they can be simulated on a digital computer. The usual method is to apply an ODE solver onto the differential equations describing the continuous model. A rather new alternative is called QSS (Quantized State System) [8] which is already mentioned in [13] as it transforms the continuous model (DESS) into an discrete event one (DEVS). Ernesto Kofman introduced a set of advancements to QSS [3], [4], [5], [6], [7] which have been implemented in PowerDEVS [1]. PowerDEVS is a DEVS simulator that supports graphical block orientated model description comparable to Simulink or Dymola.

Although PowerDEVS is specially designed for implementing continuous models in an discrete event manner, so far there has been no way to directly implement a DEV&DESS. A formal method of how to embed DEV&DESS in DEVS is presented in [12]. Based on this work, a generic PowerDEVS DEV&DESS block is developed.

However, creating and simulating coupled DEVS models turns out to be quite difficult. This mainly is owed to the various numbers of possible scenarios of concurrent events that may occur during simulation and that have to be considered when defining the DEVS of each single block involved. To relieve the modeller of these difficulties, a new PowerDEVS *Atomic DEVS* block is introduced, based on the DEVS extension *Parallel DEVS* (P-DEVS, see [2]). The

thereby developed methods for concurrency treatment are then also utilized for the mentioned generic DEV&DESS block, called *Atomic DEV&DESS*.

1.1 DEVS

DEVS denotes a formalism for describing systems that allow changes only at discrete points in time called *events*. An atomic DEVS is specified by the following 7-tuple.

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

where

$X \ldots$ set of possible inputs(e.g. $\mathbb{R}^n$)

$Y \ldots$ set of possible outputs(e.g. $\mathbb{R}^+ \times \mathbb{N} \times \mathbb{R}^m$)

$S \ldots$ set of possible states(=state space)

$Q = \{(s,e) | s \in S, e \in [0, ta(s)]\}$

$\delta_{ext} : Q \times X \rightarrow S \ldots$ *external state transition function*

$\delta_{int} : S \rightarrow S \ldots$ *internal state transition function*

$\lambda : S \rightarrow Y \ldots$ *output function*

$ta : S \rightarrow \mathbb{R}_0^+ \cup \infty \ldots$ *time advance function*

Figure 1 illustrates a DEVS graphically. It basically consists of an inner state s that can be altered by the external and internal state transition functions which are evaluated each time an external or an internal event occurs. External events are triggered by arriving input messages x, whereas internal events are triggered when the current state s has not change for the duration of $ta(s)$. On each internal event, right before δ_{int} is called, the output function λ is evaluated which may result in an output message y.

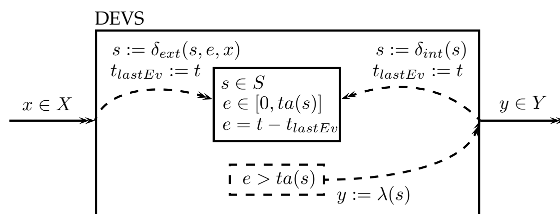

Figure 1: Graphical illustration of an atomic DEVS.

As an atomic DEVS has input and output ports, it can be coupled with other atomic DEVS resulting in a *coupled DEVS* whose behaviour again can be described by an atomic DEVS (see closure under coupling property in [13]). Figure 2 illustrates some possible coupling schemes of atomic and coupled DEVS.

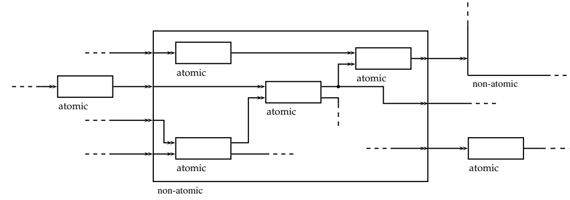

Figure 2: Coupling scheme of DEVS blocks.

Two types of DEVS can be distinguished:

Definition 1.1 *Mealy Type DEVS*
A DEVS is called Mealy Type DEVS *or* of Type Mealy*, if there exists an internal state s and an external input x in such a way that* $ta(\delta_{ext}(s,x)) = 0$*. Thus, a model described by a mealy type DEVS may produce an output as immediate response to an input.*

Definition 1.2 *Moore Type DEVS*
A DEVS is called Moore Type DEVS *or* of Type Moore*, if it is not of type mealy.*

1.2 DEV&DESS

DEV&DESS formalism is a composition of DEVS and DESS and therefore capable of describing hybrid systems. An atomic DEV&DESS can be described by the following 11-tuple:

$$< X^{discr}, X^{cont}, Y^{discr}, Y^{cont}, S, \delta_{ext}, C_{int}, \delta_{int}, \lambda^{discr}, f, \lambda^{cont} >$$

where

$X^{discr}, Y^{discr} \ldots$ set of discrete inputs and outputs

$X^{cont}, Y^{cont} \ldots$ set of continuous inputs and outputs

$S = S^{discr} \times S^{cont} \ldots$ set of states(=state space)

$Q = \{(s^{discr}, s^{cont}, e) \,|\, s^{discr} \in S^{discr}, s^{cont} \in S^{cont}, e \in \mathbb{R}_0^+\}$

$\delta_{ext} : Q \times X^{cont} \times X^{discr} \rightarrow S \ldots$external transition fct.

$\delta_{int} : Q \times X^{cont} \rightarrow S \ldots$ internal transition function

$\lambda^{discr} : Q \times X^{cont} \rightarrow Y^{discr} \ldots$discrete output function

$\lambda^{cont} : Q \times X^{cont} \rightarrow Y^{cont} \ldots$continuous output function

$f : Q \times X^{cont} \rightarrow S^{cont} \ldots$rate of change function

$C_{int} : Q \times X^{cont} \rightarrow \{true, false\} \ldots$ event condition fct.

As it can be seen, apart from ta, each component of an atomic DEVS is recurring in an atomic DEV&DESS. Additionally there are the right side of the ODE f and the continuous output function λ^{cont} describing the DESS part. The state event condition function C_{int} though, is new. It is responsible for triggering internal events in the DEV&DESS and therefore replaces ta. However, C_{int} does not only trigger time events, but also state events, i.e. events caused by the internal state q reaching some specific value. Nevertheless, both time and state events result in an execution of δ_{int}.

Figure 3 shows a graphical illustration of an atomic DEV&DESS. As well as DEVS also DEV&DESS is closed under coupling. However, continuous output ports cannot be coupled arbitrary to discrete input ports, but only if their output value is guaranteed to be piecewise constant. See [13] for more details about that and about DEVS and DEV&DESS in general.

2 DEVS Simulation in PowerDEVS

In PowerDEVS a DEVS of an atomic block consists of the specification of the six C++ functions: `init(t,parameters)`, `ta(t)`, `dint(t)`, `dext(x,t)`, `lambda(t)` and `exit()`. These functions are member functions of a C++ class describing the block. Thus, state variables, block parameters and auxiliary variables are defined as member variables of that class making them accessible in all the member functions.

These member functions are called by the PowerDEVS simulation engine whenever their execution is necessary to calculate the models dynamic behaviour. The function `init(t,parameters)` is called right before the actual simulation is started and can be used to initialise the system's state and to read block parameter values that have been entered by the modeller using PowerDEVS graphical user interface and the block's *Parameters Dialogue*. The function `exit()` is called after the simulation is finished and thus, can be used for example to free allocated memory.

The other methods represent the corresponding functions of the DEVS formalism. In case of an internal event they are executed in the following order:

1. call of `lambda(t)`
2. `e = t - tl`
3. call of `dint(t)`
4. `tl = t`
5. call of `ta(t)`: `tn = t + ta(t)`

where `tl` denotes the time of the last event, and `tn` the time of the next event in the corresponding DEVS.

In case of an external event they are executed in the following order:

1. `e = t - tl`
2. call of `dext(x, t)`
3. `tl = t`
4. call of `ta(t)`: `tn = t + ta(t)`

In coupled DEVS the so-called *Select* function, a kind of priority ranking of the involved blocks, selects which DEVS is allowed to execute its internal transition function first, when several of them are scheduled simultaneously. In PowerDEVS the *Select* function is implemented as a list in which all blocks occurring in a coupling are sorted descending according their priority in case of concurrent internal events. However, if a block produces an output message, the external transition function at the receiving block is always executed immediately, independent of its priority.

3 Problems with DEVS and Solution Approaches

3.1 Problem Identification

The definition of an atomic DEVS behaving exactly as intended in every possible situation of concurrent input messages at different input ports turns out to be quite challenging. The *Select* function regulating the resolution of such concurrencies is part of the coupling but nevertheless influences the behaviour of an atomic DEVS in such situations and therefore, it is hard to consider when formulating the atomic DEVS. For example, if an atomic DEVS with the current internal state s receives the input messages x_1 at its input

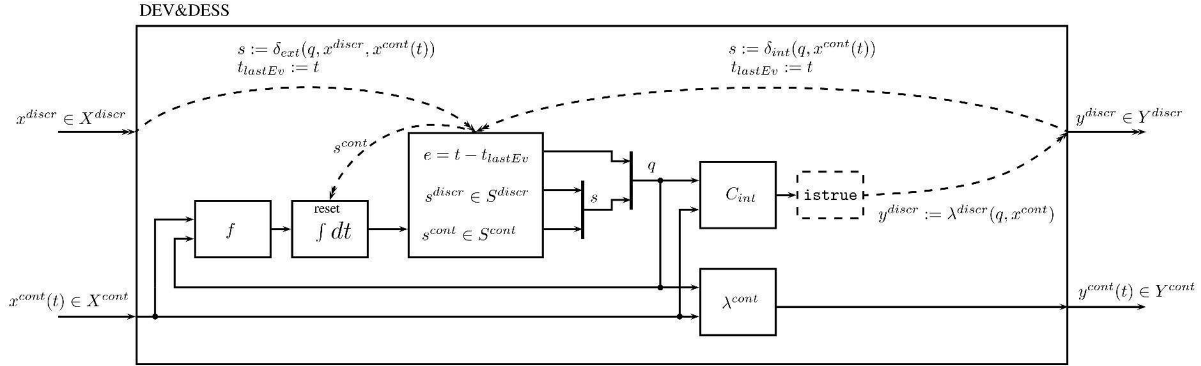

Figure 3: Graphical illustration of an atomic DEV&DESS.

port one and x_2 at its input port two at the same instant of simulation time, it depends on the *Select* function whether its new state calculates as $\delta_{ext}(\delta_{ext}(s,x_2),x_1)$ or as $\delta_{ext}(\delta_{ext}(s,x_1),x_2)$. Even more, if the block is of type Mealy, it may depend on the *Select* function if an output message is produced in reaction to the input messages or not.

Therefore, to define a DEVS in a rigorous way, it is necessary to consider each possible set of concurrent input messages and moreover, every possible treatment order.

3.2 Parallel DEVS Approach

The DEVS extension *Parallel DEVS* counters the problems mentioned in section 3.1 by collecting all input messages in so called *bags* before treating them all at once in one single call of δ_{ext}. If at the same time also an internal event is triggered, a so called *confluent transition function* $\delta_{conf} : Q \times X \rightarrow S$ is applied instead of δ_{ext}.

The idea is to first calculate λ of each immanent block, i.e. of each block experiencing an internal transition at the current simulation time, before calculating δ_{int} or δ_{conf} of any of them. However, due to blocks of type Mealy and due to feedbacks in the coupling it may be necessary to recalculate λ if the set of arrived input messages of the corresponding block has changed after its former calculation. Therefore, the determination of a stable set of input messages for each block is an iterative process. It even is possible that finally the stable set of input messages is empty although having not been empty in former iteration which makes it necessary to undo the state changes that may have been accomplished by a call of δ_{ext}.

If a coupled model contains an algebraic loop, it may not be possible to determine a stable set of input messages for each block in a finite number of iterations. Such models are called *illegitimate*.

3.3 DEVS Approach in PowerDEVS

As in Parallel DEVS the execution of λ is decoupled from the succeeding execution of the corresponding δ function, P-DEVS works with a simulation engine different to the one used for DEVS. PowerDEVS does not support Parallel DEVS simulation. Nevertheless, it is possible to implement P-DEVS functionality in PowerDEVS.

For this purpose, three mechanisms have to be installed. The first one addresses the gathering of concurrent input messages in a set $\mathbf{x}$ (bold letters denote sets). In order to do that the internal state of the DEVS is extended by an input buffer representing $\mathbf{x}$ and the external transition function only is allowed to change the input buffer (by storing arriving messages with their arrival time in it). The actual state change caused by the external event is shifted into the internal transition which is executed every time the input buffer has been modified. Thus, it is made sure that all input messages coming from blocks with higher priority are gathered in $\mathbf{x}$ before they are treated. However, due to feedback couplings it cannot be avoided that some input messages origin from blocks with lower priority. This problem is addressed by the second mechanism.

The second mechanism consists of a backing up of

the state s of the DEVS every time the first event at the current simulation time is triggered. In PowerDEVS this can be identified by `tl<t`. The δ functions (δ_{int}, δ_{ext}, δ_{conf}) are then using the backup s_{old} instead of s to calculate a new state. Therefore, if $\mathbf{x}$ is changed after one of the δ functions has already been evaluated it simply is re-evaluated: $s = \delta(s_{old}, \mathbf{x})$. As every calculation of δ is preceded by a calculation of λ though, it may occur that formerly output messages have been produced at output ports where finally no output messages are to be sent. However, these output messages have already altered the set $\mathbf{x}$ of its receiving blocks. These alterations have to be withdrawn somehow. This is what the third mechanism is dedicated to.

Like $\mathbf{x}$ also the set of output messages $\mathbf{y}$ is iteratively changing and hopefully finally stabilising. Thus, also for the output messages an output buffer is installed as part of the state of the DEVS. Each calculated output message is stored in it at the corresponding output port with its time of last change and with an 'already sent - flag'. So if λ is recalculated and there exists an entry in the output buffer that has been already sent at the current simulation time but is not marked to be resent (as it is not included in the result of λ anymore) a *retrieve message* is sent instead informing the receiving block to ignore the formerly received message and to recalculate its own λ and δ function.

4 Atomic PDEVS Block

The created *Atomic PDEVS* PowerDEVS library block is intended to overcome the problems mentioned in section 3.1 by implementing the three mechanisms discussed in section 3.3. In the following, its working principle is expressed in form of a description of the content of the C++ functions corresponding to δ_{ext}, ta, λ, and δ_{int}. Of course, a complete description of each detail that need to be considered when programming the *Atomic DEVS* block would go beyond the scope of this article.

1. `dext(`x`,t):`
   - If `tl<t` set $s_{old} = s$, $\sigma_n = \sigma - e$, $\sigma_{n,old} = \sigma_n$ and `flag='n'` but if additionally $\sigma_n = 0$ set `flag='i'`.
   - If x is a retrieve message, remove the corresponding entry from $\mathbf{x}$, set $\sigma = 0$ and if $\mathbf{x} = \emptyset$ set `flag='n'`.

   Else, if x differs from the last reception at the same port in value or in arrival time, modify $\mathbf{x}$ accordingly and set $\sigma = 0$ and update `flag`: `'n'`$\mapsto$`'e'`, `'i'`$\mapsto$`'c'`.
2. `ta(t)`:
   Return σ.
3. `lambda(t)`:
   - If `tl<t` set $s_{old} = s$ and `flag='i'`
   - If there is no pending output message, calculate $\mathbf{y} = \lambda(s_{old}, \mathbf{x})$, and add retrieve messages to $\mathbf{y}$.
   - If there are pending output messages left in $\mathbf{y}$, output one of them.
4. `dint(t)`:
   If there are no pending output messages and no newly arrived input messages:
   - if `flag='i'` calc. $[s, \sigma_n] = \delta_{int}(s_{old})$.
   - if `flag='e'` calc. $[s, \sigma_n] = \delta_{ext}(s_{old}, \mathbf{x})$.
   - if `flag='c'` calc. $[s, \sigma_n] = \delta_{conf}(s_{old}, \mathbf{x})$.
   - if `flag='n'` set $[s, \sigma_n] = [s_{old}, \sigma_{n,old}]$.
   - set $\sigma = \sigma_n$.

The variable `flag` is used to identify the type of event and therefore, which particular δ function is to be used for calculating the new state. The message retrieving mechanism though, only works if all blocks involved in a coupling implement it.

5 Atomic DEV&DESS Block

Based on the Atomic PDEVS block, an *Atomic DEV&DESS* PowerDEVS library block is developed.

5.1 Structure

The structure of the DEV&DESS embedded in PowerDEVS is depicted in Figure 4. The idea is to divide a DEV&DESS into a continuous (DESS) part that can be implemented graphically as block diagram and into a discrete part that can be implemented as atomic PDEVS block. For this purpose the function C_{int} is also divided into two parts: one for detecting state events (C_{int}^{se}) and one for scheduling time events

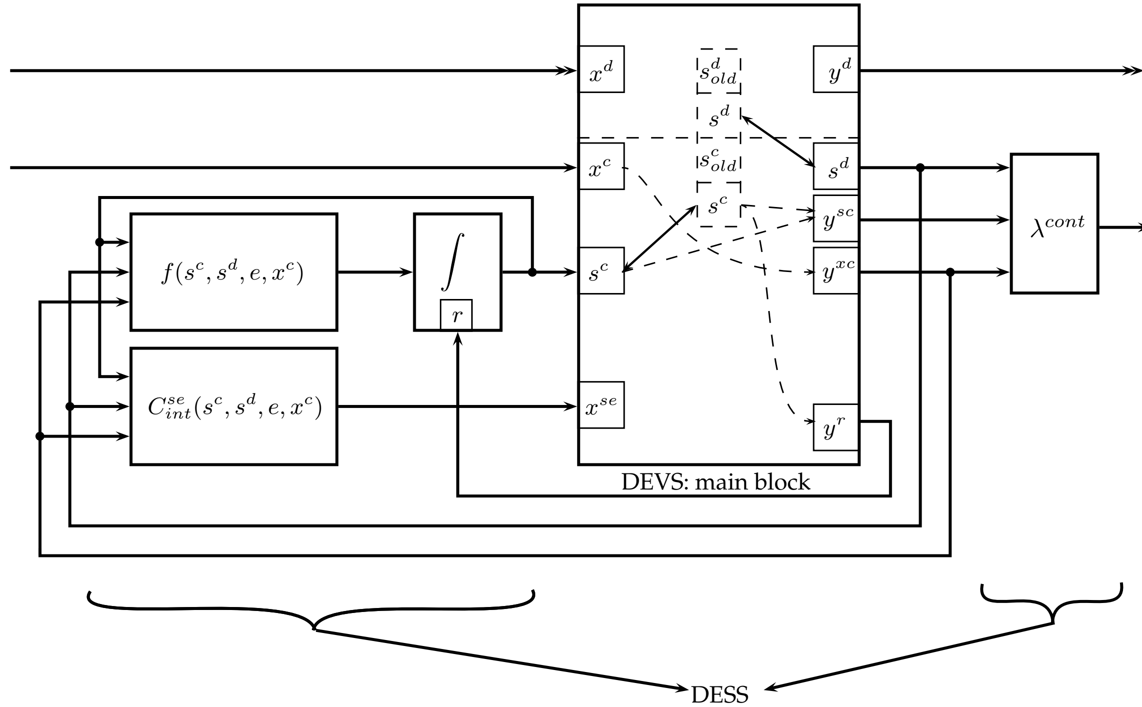

Figure 4: Graphical illustration of the construction of a DEV&DESS in PowerDEVS.

(C^{te}_{int}). The first one is a component of the continuous part and the second one is included in the atomic PDEVS definition.

The continuous part consists of λ^{cont}, f, an integrator $\int$, and of C_{int}. The discrete part, responsible for the implementation of δ_{ext}, δ_{int}, λ^{discr}, and C^{te}_{int} is realized as one single DEVS referred to as *main block*. The priority list of the coupling in Figure 4 is as follows: f, $\int$, C^{se}_{int}, `main block`, λ^{cont}.

5.2 QSS Signals

All continuous signals in DEV&DESS are described as QSS signals in PowerDEVS. That is, they are described as piecewise polynomial functions with jump discontinuities at the merging points of two polynomials which are smaller than a requested constant called *quantum q*. The polynomials represent the Taylor polynomials of the continuous signal with the expansion point at the discontinuities. Every time the difference between the current Taylor polynomial and the continuous signal or between Taylor polynomial and a Taylor approximation of higher order becomes equal to q, the expansion point is advanced in time and the coefficients of the new polynomial are sent as DEVS message. In this way, a continuous signal can be described in a discrete event manner. As in DESS piecewise continuous signals are allowed, jump discontinuities bigger than q may appear as well. Anyway, they are not caused by the signal discretisation but have already existed before.

5.3 State Events

Using QSS, state events, triggered by the continuous state reaching a specific value, simply can be calculated as the points in time when the current polynomial signal representation reaches the specified value. Thus, state events are transformed into time events.

All the signal lines in Figure 4 may transmit vectorial signals. However, in PowerDEVS vectorial signals are not sent at once but index by index resulting in temporally non-valid signals during the time after the first changed index is sent and before the last changed index is sent. This in turn may lead to state events wrongly indicated by C^{se}_{int}.

Further there is the case in which a state transition in the main block leads to a state that immediately triggers a state event. This case is declared to be prohibited. That is, the DEV&DESS is not allowed to define state transitions leading immediately to a state event.

5.4 Main Block

As the main block calculates δ_{ext} and δ_{int} it has to administer the entire state of the system consisting of a discrete part s^d and of a continuous part s^c. However, in between two state transitions in the main block s^c may change due to Taylor polynomial expansion point advancement triggered by the integrator. Therefore, the third input port of the main block is coupled to the integrator's output port. The input buffer of that port is simultaneously used as storage for the continuous state s^c itself. The same holds for s^d and the output buffer of the second output port. The fourth input port of the main block receives the result of the calculation of C_{int}^{se} and thus, input messages at this port trigger internal transitions.

f, C_{int}^{se}, and λ^{cont} depend on the continuous input signal x^c. However, they are not coupled directly to it but indirectly through the main block. This is to not forward each single, maybe even only temporary index change in x^c immediately to the continuous part. Instead the whole new x^c is forwarded just before the main block calculates one of the δ functions as it has to evaluate C_{int}^{se} before to being able to decide which one.

In the following, the working principle of the main block is expressed in form of a description of the content of the C++ functions corresponding to δ_{ext}, ta, λ, and δ_{int}.

1. `dext(x,t)`:
   If `tl<t` set $s_{old} = s$, $\sigma_n = \sigma - e$, $\sigma_{n,old} = \sigma_n$, `phase='g'` and `flag='n'`, if additionally $\sigma_n = 0$ set `flag='i'`.
   If input at port:
   - x^d: add/remove x to/from $\mathbf{x^d}$.
     update `flag`:
     if $\mathbf{x} = \emptyset$ `'e'`↦`'n'`, `'c'`↦`'i'`
     otherwise, `'n'`↦`'e'`, `'i'`↦`'c'`
   - x^c: add/remove x to/from $\mathbf{x^c}$.
   - s^c: add change in s^c to $\mathbf{y^{sc}}$.
     if `phase='g'`, apply change in s^c to s^c_{old}
   - x^{se}: add x to $\mathbf{x^{se}}$

   If any input buffer changed, set $\sigma = 0$.
2. `ta(t)`:
   Return σ.
3. `lambda(t)`:
   If `tl<t` set $s_{old} = s$, $\sigma_n = \sigma - e$, $\sigma_{n,old} = \sigma_n$, `phase='l'` and `flag='i'`
   if `phase=`
   - 'g' Set `phase='c'`.
   - 'c' If $\mathbf{x^c}$ changed, or $s \neq s_{old}$, forward $\mathbf{x^c}$ to $\mathbf{y^{xc}}$, set $s^d = s^d_{old}$ and $\mathbf{y^r} = s^c_{old}$. Further set `phase='r'`.
     Else set `phase='C'`.
   - 'r' If there is a pending output left in $\mathbf{y}$, set $\mathbf{x^{se}} = \emptyset$ and output next message.
     Else, set `phase='C'`.
   - 'C' If $x^{se} \neq \emptyset$, update `flag`: `'n'`↦`'I'`, `'e'`↦`'C'`
     `phase='l'`
   - 'l' if `flag='i'|'I'`:
     calc. $[s^d, s^c, \sigma_n] = \delta(s^d_{old}, s^c_{old}, e, \mathbf{x^c})$, and $\mathbf{y^d} = \lambda^{discr}(s^d_{old}, s^c_{old}, e, \mathbf{x^c})$.
     if `flag='e'`:
     calc. $[s^d, s^c, \sigma_n] = \delta(s^d_{old}, s^c_{old}, e, \mathbf{x^c}, \mathbf{x^d})$, and $\mathbf{y^d} = \lambda^{discr}(s^d_{old}, s^c_{old}, e, \mathbf{x^c})$.
     if `flag='c'|'C'`:
     calc. $[s^d, s^c, \sigma_n] = \delta(s^d_{old}, s^c_{old}, e, \mathbf{x^c}, \mathbf{x^d})$, and $\mathbf{y^d} = \lambda^{discr}(s^d_{old}, s^c_{old}, e, \mathbf{x^c})$.
     Add retrieve messages to $\mathbf{y^d}$.
     Add each change in s^c to $\mathbf{y^r}$.
     Set `phase='o'`.
   - 'o' if unsent element of $\mathbf{y^r}$ left, send it.
     Else, if s^c changed, update $\mathbf{y^{sc}}$.
     Else, if unsent element of $\mathbf{y^d}$ left, send it.
     Else, if unsent change of s^d left, send it.
     Else, if unsent element of $\mathbf{y^{sc}}$ left, send it.
4. `dint(t)`:
   If now pending output is left and there are no new input messages at input ports x^d and x^c, set $\sigma = \sigma_n$.

As the new state is part of the output of the main block the order in which λ and δ are calculated is reversed here. This is why δ is now calculated in the C++ function `lambda` instead of in `dint`.

Since each time before δ and λ are calculated, C_{int}^{se} has to be evaluated, there are several different phases in

`lambda` to be distinguished. This is what the variable `phase` is for.

Although the description above is quite elaborate, it still does not cover every detail of the complete source code of the *Atomic DEV&DESS* block. However, the basic principles are included.

6 Conclusion

With the ever growing computational power of modern computers also the possibilities to simulate more and more extensive and complex models increase. However, when including more and more details into a model, very soon a point is reached where pure discrete or pure continuous models do not suffice anymore. Production process models which include energy consumption behaviour provide an example for this trend. As a consequence the formulation and simulation of hybrid models is demanded. Concerning the formulation, DEV&DESS seems to be quite powerful. So far though, it lacks a simulator able to rigorously implement and simulate a DEV&DESS. An approach to implement DEV&DESS in PowerDEVS has been demonstrated in this paper. As the correct definition of coupled DEVS models is quite challenging, additionally a way of how to implement models in PowerDEVS similar to Parallel DEVS has been introduced. However, profound theoretical investigations and a formal proof of the correctness of the presented approaches are works that still needs to be done.

References

[1] F. Bergero and E. Kofman. PowerDEVS. A Tool for Hybrid System Modeling and Real Time Simulation. *Simulation: Transactions of the Society for Modeling and Simulation International*, 87(1–2):113–132, 2011.

[2] Alex Chung Hen Chow and Bernard P. Zeigler. Parallel devs: a parallel, hierarchical, modular, modeling formalism. In *Winter Simulation Conference'94*, pages 716–722, 1994.

[3] 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.

[4] E. Kofman. Quantized-State Control. A Method for Discrete Event Control of Continuous Systems. *Latin American Applied Research*, 33(4):399–406, 2003.

[5] E. Kofman. Discrete Event Simulation of Hybrid Systems. *SIAM Journal on Scientific Computing*, 25(5):1771–1797, 2004.

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

[7] E. Kofman. Relative Error Control in Quantization Based Integration. *Latin American Applied Research*, 39(3):231–238, 2009.

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

[9] N. Popper, I. Hafner, M. Rössler, F. Preyser, B. Heinzl, P. Smolek, and I. Leobner. A General Concept for Description of Production Plants with a Concept of Cubes. *Simulation Notes Europe*, 24(2):105–114, 2014.

[10] H. Praehofer. Systems Theoretic Formalisms for Combined Discrete-Continuous System Simulation. *International Journal of General Systems*, 19(3):219–240, 1991.

[11] A. Schmidt and T. Pawletta. Hybride modellierung fertigungstechnischer prozessketten mit energieaspekten in einer ereignisdiskreten simulationsumgebung. In *ASIM 2014 – 22. Symposium Simulationstechnik, Berlin, 03.-05.09.2014*, pages 109–116. ARGESIM/ASIM, 2014.

[12] B.P. Zeigler. Embedding dev&dess in devs. In *DEVS Integrative M&S Symposium (DEVS'06)*.

[13] B.P. Zeigler, H. Praehofer, and T.G. Kim. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press, 2000.