A Simulator for NSA-DEVS in Matlab

David Jammer[1*], Peter Junglas[2], Thorsten Pawletta[1], Sven Pawletta[1]

[1]Research Group Computational Engineering and Automation, University of Applied Sciences Wismar, Philipp-Müller-Straße 14, 23966 Wismar, Germany;
[*]*david.jammer@cea-wismar.de*

[2]PHWT-Institut, PHWT Vechta/Diepholz, Thüringer Straße 3, 49356 Diepholz, Germany;

Abstract. The PDEVS formalism is widely used for the description and analysis of discrete event systems. But PDEVS has some drawbacks in modeling Mealy behavior. A revised version (RPDEVS) has been invented to resolve them, but it has problems of its own, mainly because its complicated simulator structure. The recently proposed NSA-DEVS scheme tries to unite the advantages of both formalisms by using infinitesimal time intervals.

To further substantiate this claim we describe an abstract simulator for NSA-DEVS, implement it in Matlab and simulate a simple queue-server system. This shows that NSA-DEVS combines the Mealy-like model description of RPDEVS with the simple simulator structure of PDEVS, making it a promising approach to implement an improved modeling and simulation system.

Introduction

The DEVS formalism [1] and its most popular variant PDEVS [2] are a well established approach for the modeling and analysis of discrete event systems. Although a few modeling and simulation tools exist that are using PDEVS [3], the usual formalism does not directly support the implementation of component-based simulation programs.

A few formal problems can be fixed by simple variations of the basic formalism [4], a well-known example being the introduction of input and output ports. A more serious flaw has been found by Preyser et al. [5]: Due to the Moore-like structure of PDEVS the combination of Mealy-type components can sometimes be difficult to implement. Using the standard workaround of transitory states (i. e. states with transition times of 0) the behaviour of a complete system can always be modeled with PDEVS. But the description of the underlying components as individual ("atomic") blocks can lead to an ordering of concurrent events in the complete system, which does not agree with the intended behaviour.

Therefore Preyser et al. have introduced a Revised PDEVS (RPDEVS) formalism [6] that uses a Mealy-like scheme directly – without the introduction of transitory states – and allows for direct modeling of Mealy-like components, which behave correctly in the context of a larger system. To make this possible they had to define an abstract simulator for RPDEVS [7] that uses a complicated scheme of internal iterations.

However, for systems with a complex causal structure of concurrent events this iteration leads to problems, as has been shown in [8] using the example of a queue-server system. To solve these problems and to bring the modeling process closer to the underlying ideas of the modeler, the NSA-DEVS ("Non-Standard Analysis DEVS") formalism has been introduced in [8], which is a variant of RPDEVS and uses concepts of non-standard analysis [9]. Another approach has been suggested to cope with the ordering of concurrent events by augmenting the real time line [10], but it doesn't adress the Mealy-related problems that RPDEVS and NSA-DEVS try to solve.

The objective of the work presented here is to further investigate the soundness and usefulness of the NSA-DEVS formalism by defining a proper abstract simulator. To this end we first review the model and simulator specifications in PDEVS, then shortly introduce the hyperreal numbers and define the NSA-DEVS modeling formalism. Next we describe the abstract NSA-DEVS simulator and highlight some crucial points of its implementation in Matlab. Finally we implement the queue-server system from [8] and demonstrate that it works as intended.

1 Short review of the PDEVS formalism

The Discrete Event System Specification (DEVS) formalisms are divided into model specification and abstract simulator which are explained in more detail in the following for Parallel DEVS (PDEVS). The model specification differentiates between atomic and coupled models, which together form a hierarchical structure. In the following, we introduce a simplified model specification for PDEVS that uses ports instead of input

bags [11, p.108]. The model specification of an atomic model is an 8-tupel $< X, S, Y, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta >$ with

X	set of input ports and values,
Y	set of output ports and values,
S	set of sequential states,
$\delta_{int} : S \to S$	internal transition function,
$\delta_{ext} : Q \times X^+ \to S$	external transition function,
$\delta_{con} : S \times X^+ \to S$	confluent transition function,
$\lambda : S \to Y^+$	output function,
$ta : S \to \mathbb{R}^{\geq 0} \cup \{\infty\}$	time advance function.

Here $Q = \{(s,e) | s \in S, 0 \leq e < ta(s)\}$ and e is the elapsed time since the last transition. The input and output sets are defined as

$$\begin{aligned} X &= \{(p,v) | p \in P_{in}, v \in X_p\} \\ Y &= \{(p,v) | p \in P_{out}, v \in Y_p\} \end{aligned}$$

where P_{in} and P_{out} are the sets of input and output names and X_p and Y_p are the sets of possible values at input or output port p. Since inputs can arrive simultaneously at different ports, one needs the set

$$X^+ := \big\{ \{(p_1,v_1),\ldots,(p_n,v_n)\} \,|\, n \in \mathbb{N}_0,\, p_i \in P_{in},\, p_i \neq p_j \text{ for } i \neq j,\, v_i \in X_{p_i} \big\}$$

and similarly Y^+ for simultaneous outputs at several ports. Unlike in [11] simultaneous inputs at the same port are not allowed here. This makes the formulation of external transition functions easier, but prohibits the direct connection of several output ports to one input port. This is not a real limitation though, since one can insert an appropriate atomic component (*multiplexer*) for this purpose.

The formal specification of coupled models has changed several times in the development of DEVS. For practical purposes, the following specification is used in this article: $N =< X, Y, D, \{M_d\}, EIC, EOC, IC >$

X	set of input ports and values,
Y	set of output ports and values,
D	set of component names,
$\{M_d\}$	set of dynamic systems with $d \in D$,
EIC	set of external input couplings,
EOC	set of external output couplings,
IC	set of internal couplings.

Furthermore, the PDEVS formalism defines an abstract simulator, which describes the execution of a specified model [11, p.197]. It consists of the modules *root coordinator*, *coordinator* and *simulator*. They are combined in a hierarchical structure, which is shown in Fig. 1 for a simple example. The abstract simulator al-

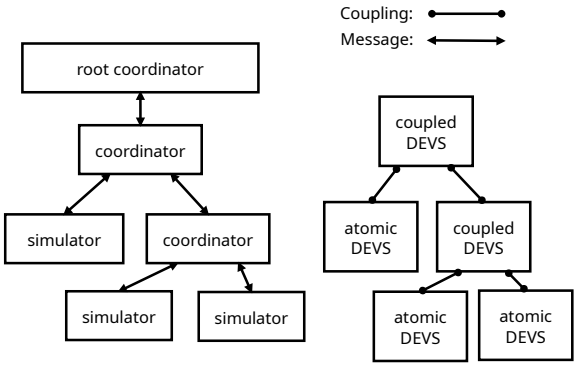

Figure 1: Hierarchical and distributed concept of the abstract simulator.

ways consists of exactly one root coordinator as the topmost instance. This is always followed by a coordinator that is attached to the uppermost coupled model. In addition, a coordinator is assigned to each coupled model of the underlying layers of the hierarchical structure, whereas a simulator is assigned to each atomic model. The coordinators and simulators form a tree structure that parallels the model structure (cf. Fig. 1), where the leaves on the left side are the simulators and on the right (model) side the atomic components.

The simulation is organized with a message concept. Messages are exchanged between root coordinator, coordinators and simulators, all downwards messages contain the current simulation time t. The following message types are used:

- i-message: downwards for initialization,
- *-message: downwards to initiate internal events,
- y-message: upwards to distribute outputs,
- x-message: downwards to trigger events,
- d-message: upwards to return information.

This terminology follows [7], in [11] the d-messages ("done") are only implicitly mentioned in the pseudocode.

The simulation starts with an i-message that is sent by the root coordinator to the topmost coordinator and distributed downwards. Each simulator initialises its atomic model and returns the time of its next internal

event to its parent coordinator. All coordinators collect the times of their children and report the smallest value upwards, until the root coordinator is reached, which stores the received value as the current simulation time.

Next the root-coordinator sends a *-message, which is forwarded according to the hierachical structure to all simulators that are *imminent*, i. e. the time of their next event is equal to the current simulation time. Each of these simulators executes the λ-function of its atomic model and sends the output to its coordinator via a y-message. Using the set IC of its coupled model the coordinator distributes the outputs to the appropriate child simulators and coordinators via x-messages and sends additional empty x-messages to the imminent children. Furthermore the coordinator collects the external outputs according to the set EOC of its coupled model and sends them upwards via a y-message.

On receiving an x-message a simulator executes one of the three transition functions of its atomic model, depending on the event type. An empty x-message means an *internal* event, which causes the execution of the δ_{int} function. A non-empty x-message represents an *external* or *confluent* event. If the atomic model is not imminent, δ_{ext} is executed, otherwise δ_{conf}. After the execution of a transition function, the time advance function ta is called to compute the time of the next internal event, which is sent upwards. A coordinator, that receives an x-message, forwards it to its active children, i. e. those that get a new input or are imminent.

As a result, the root coordinator receives the next event time, updates the current simulation time and sends a new *-message. This procedure is repeated until the root coordinator detects a termination condition. A complete description of the abstract simulator using pseudocode is given in [11, p.350-353].

2 The NSA-DEVS modeling formalism

The basic idea of the NSA-DEVS formalism is to start with the RPDEVS description, to add infinitesimal delays at the inputs of all components and to replace transitory states by states with infinitesimal transition times. This has two immediate advantages: Firstly, the complex iteration, that is necessary in the RPDEVS simulator to handle the transport of events through networks of Mealy-type components, is obsolete. Secondly, one can easily define the ordering of concurrent events by using appropriate delay times.

The introduction of infinitesimals to represent small real delays avoids an abundance of unknown additional parameters. Instead one can mainly use a default value ε, using different values only for special needs. Furthermore, the simulator handles the infinitesimal events mainly internally, so that from the user perspective, correct Mealy behaviour can be achieved.

For a precise mathematical description of finite or infinitesimal time delays we use the totally ordered field of hyperreal numbers ${}^*\mathbb{R}$. This is an extension of the real numbers including an infinitesimal $\varepsilon > 0$, which is smaller than any positive real number. Every finite hyperreal a is infinitely close to exactly one real number, called the standard part of a and denoted by $\mathrm{st}(a)$. The construction of ${}^*\mathbb{R}$ relies on advanced results from set theory and logic, but its use is rather straightforward. Exact definitions, theorems and proves can be found in [9]. For the implementation of a simulator, numbers of the form $a+b\varepsilon$ with $a,b \in \mathbb{R}$ are sufficient, they can be stored as a pair of floating point numbers. The standard part then simply is $\mathrm{st}(a+b\varepsilon) = a$. To represent *passive states*, i. e. states with an infinite transition time, the hyperreal number $\omega := 1/\varepsilon$, represented by the floating point value "infinity", can be used. ω is *unlimited*, i. e. it is larger than any real number. In the following we are mainly interested in the subset of positive finite hyperreals ${}^*\mathbb{R}^{>0}_{\mathrm{fin}}$.

One can now formally define an *atomic NSA-DEVS* as a 7-tuple $< X, S, Y, \tau, ta, \delta, \lambda >$ in the following way:

X	set of input ports and values,
S	set of states,
Y	set of output ports and values,
$\tau \in {}^*\mathbb{R}^{>0}_{\mathrm{fin}}$	input delay time,
$ta : S \to {}^*\mathbb{R}^{>0}_{\mathrm{fin}} \cup \{\omega\}$	time advance function,
$\delta : Q \times X^+ \to S$	transition function,
$\lambda : Q \times X^+ \to Y^+$	output function.

where the sets X, Y are defined as in section 1, but Q is changed slightly to $Q = \{(s,e) | s \in S, 0 \le e \le ta(s)\}$.

The main difference to the PDEVS formalism described above is the restriction to only one transition function and the extension of the output function, which is now called at all three kinds of events. This is identical to the RPDEVS definition in [6] and allows for a direct formulation of Mealy-type components. The formal difference to RPDEVS is small: All time values and intervals are now meant as subsets of the hyperreals ${}^*\mathbb{R}$ and ta is always > 0. But the semantics are slightly different: When an external event, i.e. a set of

inputs $x \in X^+$, occurs at time t, the output function λ is called at $t + \tau$, followed by an immediate call of δ. An internal event, i.e. a state change after a waiting time $ta(s)$, leads to a direct (undelayed) call of λ and δ. A concurrent incidence of a (delayed) external event and an internal event can be detected by both functions directly and doesn't need a special mechanism.

A *coupled NSA-DEVS* is defined just like in RPDEVS and PDEVS, outputs are transported as usual and a coupled component has no additional input delays. For the usual confirmation of closure under coupling, i. e. the formulation of a coupled system as an atomic component, one simply uses the smallest delay of all internal components that are connected to external inputs, and adds additional delays where necessary.

3 The abstract NSA-DEVS simulator

The general concept of the abstract simulator is the same as for PDEVS, it uses the hierarchical structure, the message system and the three modules that have been introduced in section 1. The algorithms of the coordinator and the root coordinator for NSA-DEVS are identical to the PDEVS versions described in [11, p.205] and [11, p.352-353], the only difference are the type of the current simulation time and the event times, which are now hyperreals instead of real numbers.

The basic difference lies in the algorithm of the simulator module: Though it looks similar to the PDEVS simulator in [11, p.351], it implements the NSA-DEVS scheme, which directly supports Mealy-like behaviour using infinitesimal input delays. Its simplified pseudocode description is presented in Listing 1.

Listing 1: NSA-DEVS Simulator.

```
1  properties:
2    parent
3    tl
4    tn
5    model (NSA-DEVS incl. τ and
                 total state (s,e))
6    y
7    x*
8
9  when receive i-message(i,t) at time t
10   tl = t - e
11   tn = tl + ta(s)
12
13 when receive *-message(*,t) at time t
14   e = t - tl
15   y = λ(s,e,x*)
16
17   send y-message(y,t) to parent
         coordinator
18
19 when receive x-message(x,t) at time t
20   if x == ∅
21     e = t - tl
22     s = δ(s,e,x*)
23     x* = ∅
24     if ta(s) == ω
25       tn = ω
26     else if st(ta(s)) == 0
27       tn = t + ta(s)
28     else
29       tn = st(t + ta(s))
30     tl = t;
31   else
32     if not (x* == ∅)
33       add events from x to x*
34     else
35       x* = x
36     tn = t + τ
```

Lines 1–7 list the variables used by the simulator. The first five are the same as for PDEVS: the parent coordinator, the times of the last and the next event, the attached model – with the atomic NSA-DEVS structure and its complete state – and the output values. Since the input delay is realized inside the simulator, input values must be stored temporarily, using the variable x^*. In lines 9–11 the i-message is handled, which just computes the times of the last and the next events. The *-message is processed in lines 13–18, where the elapsed time is calculated, the λ function is executed and the y-message is sent to the parent coordinator. In contrast to the PDEVS algorithm λ is now a function of the total state (s,e) and the input value x^* that has been stored before.

The new part - compared to PDEVS - is the way the x-message is processed, which is shown in lines 19–36. It discriminates between an internal event (lines 21–30, x == $\varnothing$) and an external event (lines 32-36). In the latter case it stores all incoming values in x^* and schedules a new internal event at the delayed time t + τ. All internal events are handled by calling the transition function δ and computing the time of the next event using ta in lines 24–29. This calculation deserves special attention: It guarantees that "real" time steps (i. e. non-infinitesimal ones) lead to real valued time values in order to implement a correct timing and to prohibit an accumulation of infinitesimal delays.

To better understand the operation of the abstract simulator, especially how it creates a proper Mealy be-

haviour, we introduce a simple example model N: It consists of a generator G, which outputs a value t/10 at times t = 1, 2, 3 . . . , a multiplication block M, which multiplies its input by a factor 3, and a terminator component T, which acts as a sink for the incoming values (cf. Fig. 2).

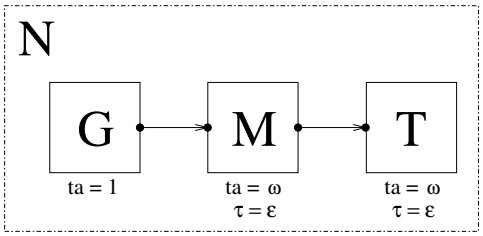

Figure 2: Simple example model.

The messages that are sent between the root coordinator *RC*, the coordinator C_N of coupled model N and the three simulators S_G, S_M and S_T with the associated models G, M and T are shown as a sequence diagram in Fig. 3. Downwards messages are denoted as (*msg type*, *current time*), for the x-messages the input value is added. Upwards messages are shown as (*msg type*, *result*).

In the initialisation step at t = 0, an i-message is sent and distributed to the simulators, returning the time t = 1 of the first internal event to *RC*. This is followed at t = 1 by a *-message sent to the simulator of the only imminent component G, which generates an output event and sends a y-message with its output 0.1 back to C_N. The coordinator now sends an empty x-message to S_G, which returns the time t = 2 of its next internal event. Moreover, C_N sends a non-empty x-message to S_M, which stores the value internally and schedules a new internal event according to the input delay time.

The next *-message at t = 1 + ε arrives at the simulator of the imminent component M, which calls its λ function and sends the output value 0.3 as a y-message to its coordinator. C_N now sends an empty x-message to S_M, which calls its δ and *ta* functions and returns t = ω to C_N. M is now in a passive state. The rest of the diagram shows how the output value propagates to S_T, which just terminates the incoming events. Since the coordinator stores all future event times of its children, it finally returns t = 2 (originally coming from S_G) as the time of the next event, which will repeat the whole cycle.

This example shows precisely, how the two parts of

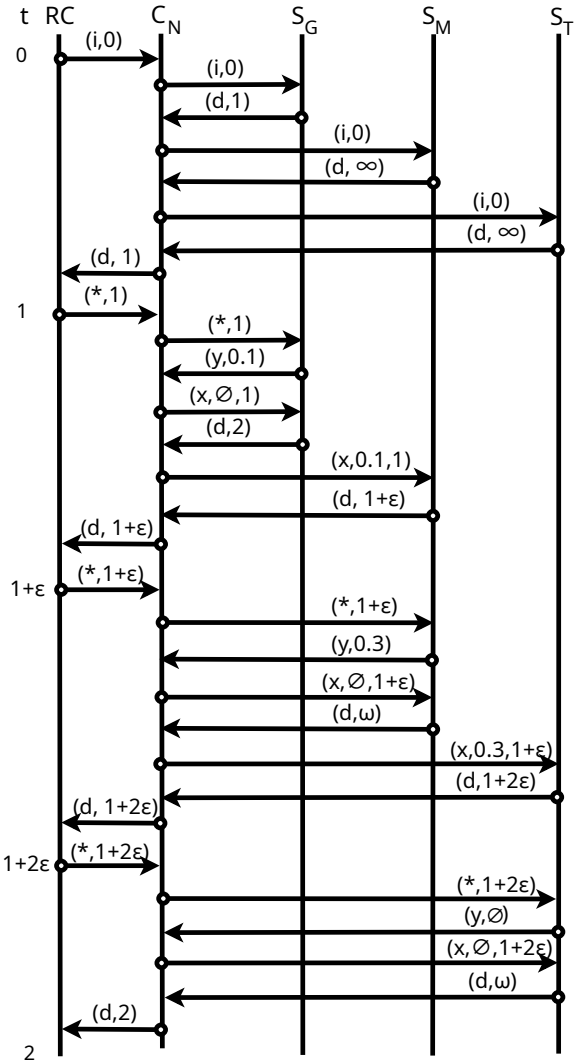

Figure 3: Example of message flow.

the x-message algorithm in the simulator module work together to implement the time delay and the Mealy behaviour of a simple Mealy block such as a multiplication function.

4 Implementation of the NSA-DEVS simulator

The "infinitesimal cloud" of hyperreal numbers around each real number has a complex structure with lots of different layers of smallness, e. g. using ε^2 or $\sqrt{\varepsilon}$. The purpose of using ${}^*\mathbb{R}$ in the formulation of NSA-DEVS is the possibility to introduce short time intervals with-

out defining their size explicitly, but still being able to order them. Therefore times of the form $a + b\varepsilon$ are sufficient here, they are stored as two-element vectors. The implementation of time comparisons and sorting has to be adapted accordingly.

Since the goal of NSA-DEVS is to provide a good basis for the concrete modeling of discrete event systems, an implementation should free the modeler from the tedious task of defining lots of additional infinitesimal parameters. Therefore the concrete simulator contains a variable $\tau_{def} = r\varepsilon$ (usually r = 1) that is used as a default value of τ for all atomic components. Furthermore the user can still define transitory states with a transition time ta(s) = 0, which is replaced automatically by setting ta(s) = $r\varepsilon$.

For debugging purposes it would be useful to make the infinitesimal delays explicitly visible. To this end the simulator contains a real (i. e. floating point) parameter μ, which is 0 normally, but can be set to a value larger than zero for debugging. In this case the infinitesimal ε is replaced by μ and all times are real values computed as

$$t' = \begin{cases} (t(1), t(2)) & \text{if } \mu = 0, \\ (t(1) + \mu\, t(2), 0) & \text{if } \mu > 0. \end{cases}$$

In complex models the value of μ has to be chosen carefully: It should be large enough to make the infinitesimal internal processes visible, but small enough to not induce any changes into the behaviour of the model. As a result of this extension, the implementation of the x-message gets more complicated, as can be seen in Listing 2.

Listing 2: Implementation of the x-message algorithm.

```
1  when receive x-message(x,t) at time t
2    if x == ∅
3      e = [t(1) - tl(1), t(2) - tl(2)]
4      s = δ(s,e,x*)
5      x* = ∅
6
7      tb = ta(s)
8      if tb == [0,0]
9        tb = [0, r]
10     if tb(1) == 0
11       if μ == 0
12         tn = [t(1), t(2) + tb(2)]
13       else
14         tn = [t(1) + μ*tb(2), 0]
15     else
16       tn = [t(1) + tb(1), 0]
17     tl = t;
18   else
19     if not (x* == ∅)
20       add events from x to x*
21     else
22       x* = x
23     if μ == 0
24       tn = [t(1) + tau(1), t(2) + tau(2)]
25     else
26       tn = [t(1) + tau(1) + μ*tau(2), 0]
```

The special case of passive states in Listing 1 (l. 24f) is done automatically in line 16 due to the handling of the value “infinity” in floating-point arithmetic.

5 Case study: A simple queue-server system

To test the operation of the complete NSA-DEVS formalism – model specification and simulator –, a prototype has been created in Matlab, which is named NSA-DEVSforMATLAB. It contains all features introduced in section 4 and is implemented in an object-oriented way. To show its functionality, the singleserver example from [8] has been chosen as an example for this article; it is shown as a block diagram in Figure 4. The

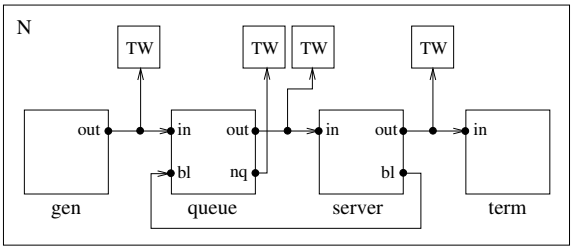

Figure 4: Example model singleserver.

following atomic models are used for the example:

- Generator: produces entities with an interval of one second,
- Queue: infinite queue,
- Server: service time 1.5 s,
- Terminator: terminates the entities,
- ToWorkspace (TW): logging data.

The special feature of this model is that the queue should only send entities to the server when it is not busy. The server announces this information via port blocked (bl), which is sent to port bl of the queue.

For functionality, the input delay of the queue must be greater than the input delay of the server. The input delays of the generator and terminator do not matter. A special role is played by the four ToWorkspace models, which are connected to the outport ports *out* and *nq* according to Fig. 4 and store the output values. They too – like every atomic model – have an input delay.

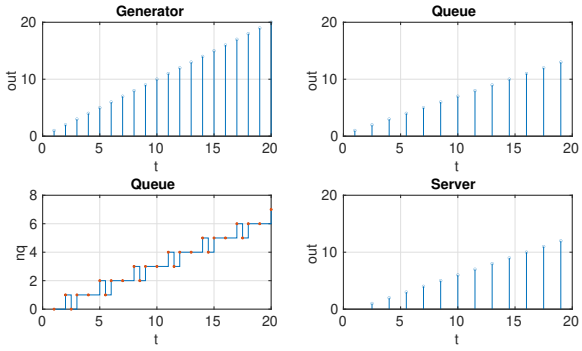

Figure 5: Simulation results with high input delay at ToWorkspace.

For the first simulation run, a large infinitesimal input delay was chosen for the ToWorkspace models. The result can be seen in Figure 5, which displays generator output (top left), queue output (top right), queue load (bottom left) and server output (bottom right).

The high input delay has the effect that output changes, which happen during a series of infinitesimal time intervals, are discarded inside a ToWorkspace block and only the final value before a finite time step is shown. This can be seen, for example, at time 10: The server has finished processing and is idle. Therefore the queue sends an entity to the server. At the same time, the generator also outputs an entity and sends it to the queue. In total, the load of the queue does not change.

However, if one uses a low input delay for the ToWorkspace models, one sees that the queue load at time 10 has the values 3 and 4 simultaneously. This means that the new entity enters the queue first and then an entity is sent to the server. This behavior is shown in Figure 6. One could use the debug mode, i. e. set the parameter μ to a finite value, to dissolve the "spike" at t=10 into a small step, thereby clearly showing the internal ordering of the events.

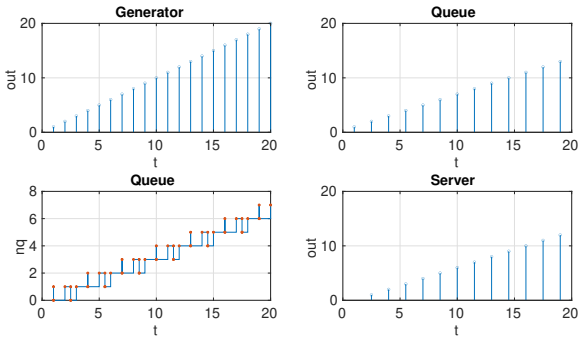

Figure 6: Simulation results with low input delay at ToWorkspace.

6 Conclusion

With the specification of an abstract simulator, which defines the behaviour of a model consisting of atomic and coupled components, the description of the NSA-DEVS formalism is now formally complete. We have shown that NSA-DEVS is able to directly describe Mealy-like models in the same way as RPDEVS, but with a much simpler simulator algorithm similar to the original PDEVS version. In this way NSA-DEVS combines the best of the two preceding formalisms.

Its principle usability has been demonstrated by the implementation of the simulator and a non-trivial example model in Matlab. The notoriously difficult modeling of concurrent events has been substituted by a clear definition of an ordering based on infinitesimal delays. The inclusion of a debugging mode further helps to understand the corresponding difficulties. An interesting side effect is the possibility to easily model systems with finite time delays.

At first sight, the NSA-DEVS approach seems to be very similar to the concept of superdense time [10], where a real time value is augmented by a natural number to order concurrent events. But the much richer structure of ${}^*\mathbb{R}$ – even of the small part that is used in the implementation – has profound consequences: On the practical side, one can use real infinitesimal delays to squeeze an event between existing ones, without the need to reorder the complete sequence. The conceptual difference, however, is the dynamic structure of NSA-DEVS: The order of concurrent events is defined by the infinitesimal delays in the complete model, which add up in a "realistic" way. While the fixed ordering of superdense time is similar to the *Select* function in Classical DEVS [11, p.104], NSA-DEVS – like PDEVS –

allows concurrent events on the infinitesimal scale and parallelism.

To further examine the practical usefulness of the NSA-DEVS formalism, one should next study a set of standard examples with complex event cascades and real-world case studies. This could help answering the crucial question, whether an abundance of new parameters is necessary in real models or if the use of a default delay is sufficient in many cases. Another interesting question is, whether one delay for an atomic model suffices, or if one needs port specific delay times.

Finally one should address the practical usefulness of the simulator and its implementation: How does it perform in comparison to existing PDEVS or RPDEVS simulators? Suitable benchmarks would address simulation times as well as the number of internal messages used inside a simulator. Although the definition of the simulator is a large step forward, much remains to be done before NSA-DEVS can be considered a solid approach for practical discrete event modeling and simulation.

References

[1] Zeigler BP. *Theory of Modeling and Simulation*. New York: Wiley-Interscience, 1st ed. 1976.

[2] Chow ACH. Parallel DEVS: A Parallel, Hierarchical, Modular Modeling Formalism and its Distributed Simulators. *Transactions of The Society for Computer Simulation International*. 1996;13(2):55–67.

[3] Franceschini R, Bisgambiglia PA, Touraille L, Bisgambiglia P, Hill D. A survey of modelling and simulation software frameworks using Discrete Event System Specification. In: *Proc. of 2014 Imperial College Computing Student Workshop*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2014; pp. 40–49.

[4] Goldstein R, Breslav S, Khan A. Informal DEVS conventions motivated by practical considerations. In: *Proc. of Symposium on Theory of Modeling &Simulation – DEVS Integrative M&S Symposium*. 2013; pp. 10:1–10:6.

[5] Preyser FJ, Heinzl B, Raich P, Kastner W. Towards Extending the Parallel-DEVS Formalism to Improve Component Modularity. In: *Proc. of ASIM-Workshop STS/GMMS*. Lippstadt. 2016; pp. 83–89.

[6] Preyser FJ, Heinzl B, Kastner W. RPDEVS: Revising the Parallel Discrete Event System Specification. In: *9th Vienna Int. Conf. Mathematical Modelling*. Wien. 2018; pp. 242–247.

[7] Preyser FJ, Heinzl B, Kastner W. RPDEVS Abstract Simulator. *SNE Simulation News Europe*. 2019; 29(2):79–84. doi: 10.11128/sne.29.tn.10473.

[8] Junglas P. NSA-DEVS: Combining Mealy Behaviour and Causality. *SNE Simulation News Europe*. 2021; 31(2):73–80. doi: 10.11128/sne.31.tn.10564.

[9] Goldblatt R. *Lectures on the Hyperreals*. New York: Springer. 1998.

[10] Sarjoughian HS, Sundaramoorthi S. Superdense time trajectories for DEVS simulation models. In: *SpringSim (TMS-DEVS)*. 2015; pp. 249–256.

[11] Zeigler BP, Muzy A, Kofman E. *Theory of Modeling and Simulation*. San Diego: Academic Press, 3rd ed. 2019.