Implementing Standard Examples with NSA-DEVS

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

[1]Research Group Computational Engineering and Automation, University of Applied Sciences Wismar, Philipp-Müller-Straße 4, 23966 Wismar, Germany; *peter@peter-junglas.de*

[2]PHWT-Institut, PHWT Vechta/Diepholz, Am Campus 2, 49356 Diepholz, Germany;

SNE 32(4), 2022, 195-202, DOI: 10.11128/sne.32.tn.10623
Received: 2022-09-27; Revised: 2022-11-07
Accepted: 2022-11-15
SNE - Simulation Notes Europe, ARGESIM Publisher Vienna
ISSN Print 2305-9974, Online 2306-0271, www.sne-journal.org

Abstract. To utilize the PDEVS formalism for the practical modeling and simulation of discrete-event systems, the recently proposed variant NSA-DEVS combines the Mealy behaviour of RPDEVS with a simple simulator algorithm by employing infinitesimal time delays. To further test the practical usefulness of this new approach, four simple systems showing non-trivial event-cascades are modeled and simulated within a concrete NSA-DEVS environment: A comparator-switch model, a digital circuit with flip-flops, a basic queue-server system and a more complex queuing system. Their simple implementations show that the potentially large number of delay parameters in NSA-DEVS in practice reduces to a single default value, which only occasionally has to be tuned to adapt to complex causal behaviour. In addition, by providing precise formal definitions of the models and by looking closely at the behaviour of the abstract simulator, the validity of the NSA-DEVS formalism is further substantiated.

Introduction

The PDEVS formalism [1] is a well-established method to concisely describe the hierarchical composition and dynamic behaviour of discrete-event based models. To make it directly applicable for concrete modeling and simulation environments, several variations have been proposed [2] ranging from the introduction of input and output ports to the revised version RPDEVS that incorporates a direct Mealy structure [3]. But as has been argued in [4], even then problems remain with the modeling and simulation of causal chains of concurrent events.

Therefore, the NSA-DEVS formalism (Non-Standard Analysis DEVS) is introduced in [4] that solves such problems with the drastic provision of prohibiting causal concurrent event chains altogether. To this end it introduces delay times at all inputs and forbids transitory states, i. e. states with a lifetime of zero. This allows to retain the mealy-type behaviour of RPDEVS, but makes the definition of an abstract simulator [5] much simpler than the corresponding simulator of RPDEVS [6].

Formally NSA-DEVS leads to the introduction of a large number of parameters for the necessary delays, whose usually very small values generally are of no interest at all. Therefore NSA-DEVS uses infinitesimal delays τ – often mostly given by a standard value τ_{def} –, and its abstract simulator clearly differentiates between the infinitesimal and finite time behaviour. This is possible in a mathematically precise way by resorting to the set ${}^*\mathbb{R}$ of hyperreal numbers, which are a well defined totally ordered field and form the basis of non-standard analysis [7]. ${}^*\mathbb{R}$ is an extension of the real numbers including the formal infinitesimal $\varepsilon > 0$, which is smaller than any positive real number, and the infinite number $\omega := 1/\varepsilon$. For the definition of NSA-DEVS one mainly needs the subset of positive finite hyperreals ${}^*\mathbb{R}^{>0}_{\text{fin}}$, occasionally enlarged by the single value ω, used as the lifetime of passive states.

The aim of this study is to further examine the practical usefulness of NSA-DEVS by implementing a set of examples with interesting event-cascades: The comparator-switch model from [8], a digital circuit containing flip-flops, a basic queue-server system and a complex queuing system. We will always start with a model that only contains default values for all input delays and transitory states, and then make the necessary fine-tuning to get the desired behaviour.This will show, whether the possible multitude of delay parameters can be tamed.

Another focus will lie on the exact definition of a model and its operation using the defined abstract simulator.

This will add confidence in the validity of the simulator algorithm, and show that modeling with NSA-DEVS – as with DEVS formalisms in general – can lead to a thorough understanding of a model and its behaviour.

1 The NSA-DEVS Formalism

The NSA-DEVS formalism is a variation of the basic PDEVS specification [1], which is divided into a model description and the definition of an abstract simulator. Two types of models are defined in PDEVS: an atomic model that describes the behaviour of a single component, and a coupled model, which shows how atomic models can be combined to build a hierarchical structure. The abstract simulator specifies the execution of a PDEVS model. It consists of three kinds of modules – a root coordinator, a coordinator for each coupled model and a simulator for each atomic model –, which exchange different types of messages to coordinate the behaviour of the atomic and coupled models.

In NSA-DEVS the definition of an atomic model is similar to the RPDEVS description, formally it is given by a 7-tuple $< X, S, Y, \tau, ta, \delta, \lambda >$ with

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

For the definition of X and Y one uses sets P_{in} and P_{out} of input and output names and corresponding sets X_p and Y_p of possible values at input or output port p. Then the input and output sets are given as

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

To describe the simultaneous arrival of input values at different ports, one additionally 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.

To specify the transition function δ and output function λ, which describe the basic behaviour of the model, one defines the set $Q = \{(s,e) | s \in S, 0 \leq e < ta(s)\}$ that combines a state and the elapsed time e since the last transition. As in RPDEVS, both event types (incoming event or internal state change) lead to a call of λ followed by a change to a new state according to δ.

The formal difference to RPDEVS is small, but important: All time values and intervals are meant here as subsets of the hyperreals ${}^*\mathbb{R}$, and the time advance function ta may be infinitesimal, but it is always > 0. A new element is τ, the delay time between the arrival of a set of inputs and the call of λ and δ. Generally it is an infinitesimal, often given by a default value $\tau_{def} = \varepsilon$, and is adapted if the need occurs.

A coupled NSA-DEVS model is defined as in PDEVS and RPDEVS, it consists of input and output ports and a set of atomic or coupled models, which are connected among themselves and to the external ports. The abstract NSA-DEVS simulator uses the same structure of modules and messages as in PDEVS, but it implements the input delays and the Mealy-like behaviour. Its details, the differences to the PDEVS simulator and an implementation in Matlab can be found in [5].

2 A Comparator-Switch Model

2.1 Theoretical analysis

In [8] a simple example is presented that consists of a switch controlled by a comparator (cf. Figure 1), such that an incoming entity is routed to the upper output, if it has a negative value, and to the lower output, if its value is positive or zero. As is shown in [8] this model can not be implemented straightforwardly in PDEVS using independent reusable components, but works perfectly well in RPDEVS. Therefore it is an excellent first example to test the capabilities of NSA-DEVS.

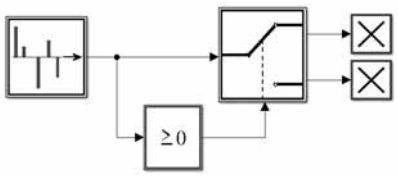

Figure 1: Example 1: Switch controlled by a comparator.

Using the NSA-DEVS specification of an atomic model, the comparator is defined by

$$P_{in} = \{in\},\ X_{in} = \mathbb{R} \Rightarrow X = \{(in,x) | x \in \mathbb{R}\}$$
$$P_{out} = \{out\},\ Y_{out} = \{0,1\} \Rightarrow Y = \{(out,0),(out,1)\}$$
$$S = \varnothing,\ ta(s) = \omega$$
$$\tau = \varepsilon \equiv \tau_{def}$$
$$\delta(s,e,x^+) = s$$
$$\lambda(s,e,\{(in,x)\}) = \begin{cases} \{(out,0)\} & | x < 0 \\ \{(out,1)\} & | x \geq 0 \end{cases}$$

The switch component needs an internal state to store the current routing behaviour of the switch, its formal definition is:

$$P_{in} = \{in, sw\},\ X_{in} = \mathbb{R},\ X_{sw} = \{0,1\}$$
$$\Rightarrow X = \{(in,x) | x \in \mathbb{R}\} \cup \{(sw,0),(sw,1)\}$$
$$P_{out} = \{out_1, out_2\},\ Y_{out1} = Y_{out2} = \mathbb{R} \cup \{\varnothing\}$$
$$\Rightarrow Y = P_{out} \times (\mathbb{R} \cup \{\varnothing\})$$
$$S = \{1,2\},\ ta(s) = \omega$$
$$\tau = r\varepsilon \quad (r = 1, \text{can be changed as parameter})$$
$$\delta(s,e,x^+) = \begin{cases} i+1 & | (sw,i) \in x^+ \\ s & | \text{otherwise} \end{cases}$$
$$\lambda(s,e,x^+) = \begin{cases} \{(out_s,x),(out_{3-s},\varnothing)\} & \\ \qquad\qquad | x^+ = \{(in,x)\} & \\ \{(out_{i+1},x),(out_{2-i},\varnothing)\} & \\ \qquad\qquad | x^+ = \{(in,x),(sw,i)\} & \\ \varnothing \qquad\qquad | x^+ = \{(sw,i)\} & \end{cases}$$

An important point to note is that the λ-function sends an empty value to the output port that is currently not used.

The example model contains two additional atomic components: a generator that outputs predefined values at given times, and a terminator that stores the last incoming value. Their formal definitions are straightforward and will not be needed in the folllowing. The complete model can then be easily defined as a coupled model.

To analyse the behaviour of this model in detail, we will retrace the simulator procedures. Generally, when an input value arrives at time $t = 1$ at a Mealy-like component with a time delay $\tau \in {}^*\mathbb{R}$, its simulator module S and the coordinator C of its enclosing coupled component perform the following steps [5]:

t	
1	C sends an x-message with the input value to S, which stores it in an internal variable x^* and sends back the time $1+\tau$ of its next internal event.
$1+\tau$	C sends a *-message to S, which now computes its output value $\lambda(s,e,x^*)$ and sends it to C via a y-message. C distributes it via x-messages to all connected components. After that C sends an empty x-message to S, which changes its state using $\delta(s,e,x^*)$ and resets x^* afterwards.

In the context of the example model the behaviour is more complicated. We will concentrate only on the switch with simulator S and assume that it is in state $s = 1$ (out_1 is active). At $t = 1$ the generator outputs a value $x = 1$. Now the following happens (cf. Figure 2):

t	
1	S gets $(in,1)$ via an x-message, stores it in x^* and returns $1+\varepsilon$.
$1+\varepsilon$	S gets a *-message, computes output $\{(out_1,1),(out_2,\varnothing)\}$ and sends it to C. Then the delayed output of the comparator arrives, therefore S doesn't get an empty x-message, but the input value $(sw,1)$. x^* now contains both values, the next event will be at $1+2\varepsilon$.
$1+2\varepsilon$	S gets a *-message, computes output $\{(out_1,\varnothing),(out_2,1)\}$ and sends it to C. Finally S gets an empty x-message, changes its state to $s = 2$ and clears x^*.

This chain of events boils down to the following behaviour: If a new input arrives during the waiting time of a component, it will either complement the output or overwrite it, and another delay time is added – which again could be extended. Consequently, intermediate outputs appear, only the final one representing the expected results. This behaviour reminds of the λ-iterations appearing in the RPDEVS simulator [6], but the underlying mechanism is quite different, as we will see in later examples.

Concentrating on the states the picture is simpler: The state only changes at the end of such an event chain and shows the anticipated behaviour. This becomes particularly clear, if one examines the terminator blocks: Though the value 1 appears at both outputs of the switch, it is only stored in the state of the (cor-

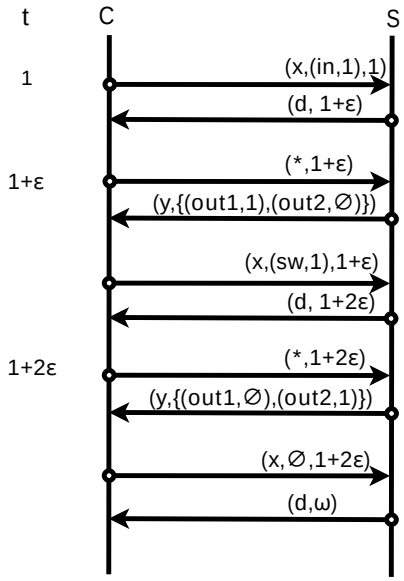

Figure 2: Internal messages from or to the Switch simulator.

rect) lower terminator. The incoming value in the upper terminator is erased by the empty second output and doesn't reach the state variable. To make this happen, it is important that the λ-function of an atomic component explicitely outputs empty values at unused ports.

2.2 Practical implications

What this complex behaviour means for a practitioner, who is not interested in the internal workings of the simulator, depends on the output values that a concrete simulator environment provides. In the PDEVS simulator hyPDEVS [9], a simulation run produces output values representing the states of the atomic components. In this case a similar NSA-DEVS simulator would show exactly the expected behaviour of the comparator-switch model.

The NSA-DEVS simulator that is described in [5] has no intrinsic output possibilities. Instead it uses an atomic component *ToWorkspace* that can be connected to an output port and copies the incoming values to a global output variable, which can be plotted or analysed after the simulation run. Like all NSA-DEVS atomics it has an internal delay τ. If one sets $\tau = r_{Out}\varepsilon$ for all ToWorkspace blocks and chooses a value r_{Out} that is larger than the delays of all other atomics, the intermediate outputs do not show up in the global output variables and the behaviour is again as expected. Choosing a very small value for r_{Out}, one can make these outputs visible, which could be useful for debugging purposes.

Another possible approach would be to set the delay of the switch component to 2ε. Since the input from the comparator now arrives, before the *-message of the switch is called, no intermediate outputs are generated and one can use arbitrary (especially: default) delays for ToWorkspace blocks. Thinking along these lines, one could take the appearance of intermediate outputs as a hint to properly adapt some delays.

3 Flip-flops and Shift Register

As has been shown in [10], the modeling of simple digital circuits containing flip-flops can be a challenge for discrete-event based systems. A solution for the case of RPDEVS has been given in [11]. Using the example of a simple shiftregister (cf. Figure 3), we will demonstrate in the following that NSA-DEVS can cope with such examples easily.

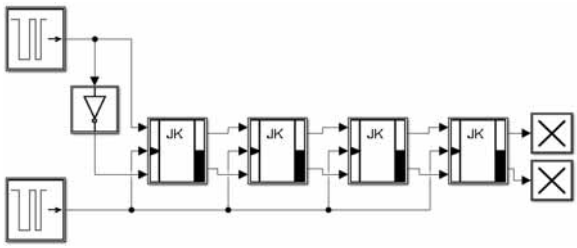

Figure 3: Example 2: Shiftregister with four JK flip-flops.

The basic component is the JK flip-flop, which has three binary inputs (J, CLK, K), two outputs $(Q, \overline{Q})$ and four internal states, three of them to store incoming values, and one for the proper state of the flip-flop. When the *CLK* input switches from 1 to 0 (a "clock tick"), the state changes according to the following function:

$$f(j,k,q) = ((j \vee q) \wedge \bar{k}) \vee (j \wedge k \wedge \bar{q}),$$

where j,k are the input values and q is the previous value of the internal state. Very important is the correct behaviour, when the arrival of inputs coincides with a clock tick: In this case the old (stored) values are used to compute the next state, after that the new values are stored internally. This is necessary to implement the correct behaviour of a shift register, where incoming values are shifted at a clock tick for one step along the line of flip-flops.

These considerations lead to the following formal definition of the JK flip-flop atomic model:

$$\mathbb{B} := \{0,1\}$$
$$P_{in} = \{J, CLK, K\},\ X = P_{in} \times \mathbb{B}$$
$$P_{out} = \{Q, \overline{Q}\},\ Y = P_{out} \times \mathbb{B}$$
$$S = \mathbb{B} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B},\ \text{where } s \equiv (j, clk, k, q) \in S$$
$$ta(s) = \omega$$
$$\tau = r\varepsilon \quad (r \text{ parameter})$$
$$\delta(s, e, x^+) = (j', clk', k', q') \text{ with}$$
$$q' = \begin{cases} f(j,k,q) & | clk = 1 \wedge (CLK, 0) \in x^+ \\ q & | \text{otherwise} \end{cases}$$
$$j' = \begin{cases} b & | (J, b) \in x^+ \\ j & | \text{otherwise} \end{cases}$$
$$clk' = \begin{cases} b & | (CLK, b) \in x^+ \\ clk & | \text{otherwise} \end{cases}$$
$$k' = \begin{cases} b & | (K, b) \in x^+ \\ k & | \text{otherwise} \end{cases}$$
$$\lambda(s, e, x^+) = \begin{cases} \{(Q, f(j,k,q)), (\overline{Q}, \overline{f(j,k,q)})\} & \\ & | clk = 1 \wedge (CLK, 0) \in x^+ \\ \varnothing & | \text{otherwise} \end{cases}$$

The complete example model is a coupled model that contains two binary generators producing test inputs, a not gate, four JK flip-flops and two terminators. The formal description of all these models is straighforward. Using the Matlab-based NSA-DEVS simulator from [5] the complete model is easily implemented and run.

Adopting the global default value for all input delays, the simulation results are as expected, no twisting of any parameters is necessary. This example shows that the key to a valid implementation of flip-flops is a precise definition of their behaviour – and that the NSA-DEVS formalism offers the tools do this easily.

4 A Simple Queue-Server System

The third example consists of a generator that creates entities in fixed time intervals $t_G = 1$ and sends them to a queue, which is connected to a simple server with fixed service time t_S = 1.5. Entities leaving the server are terminated (cf. Figure 4).

The queue outputs entities unless it is blocked; its blocking status is given by an additional input coming from the server. This model has already been used as a case study in [4] and [5], it shows a complex behaviour due to its cascade of causally related concurrent events. The queue and server components have an additional output for the number of stored entities (y_{nq} or y_{ns}, resp.), which will be used in the last example model.

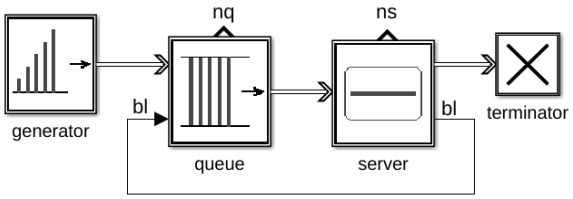

Figure 4: Example 3: Singleserver model combining a queue and a server component

The formal mathematical definition of the components is straightforward, but a bit cumbersome, especially for δ and λ. Therefore, their behaviour will be described by an enhanced state diagram like in Figure 5 for the server. These "macroscopic" states are not identical to the NSA-DEVS states, i. e. elements of the set S, which usually is much larger, but contain the essential information to conveniently describe the behaviour of the component.

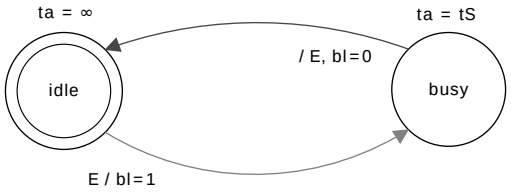

Figure 5: State diagram of the server component.

Nevertheless, it is a useful exercise to explicitely write down most of the formal structure of a component, to make its definition as precise as possible. For the server component this could be done in the following way (identifying an entity with a real number for simplicity):

$$P_{in} = \{in\},\ X_{in} = \mathbb{R} \Rightarrow X = \{in\} \times \mathbb{R}$$
$$P_{out} = \{out, ns, bl\}, Y_{out} = \mathbb{R}, Y_{ns} = \{0,1\} = Y_{bl}$$
$$\Rightarrow Y = \{out\} \times \mathbb{R} \cup \{\ (ns,0),(ns,1),(bl,0),(bl,1)\})$$
$$S = \{(\varnothing, \omega)\} \cup (\mathbb{R} \times [0, t_S])$$
$$ta((v,\sigma)) = \sigma$$
$$\tau = r\varepsilon \quad (r \text{ parameter})$$

While most of this definition is straightforward, the set of states needs some explanations: The state consists of the value v of a stored entity (or $\varnothing$) and the current lifetime σ of the state. This is a frequently used trick and reduces the time advance function to simply returning σ. It is necessary here to cope with a special situation: When an entity reaches the input, while the server is busy, the entity is discarded, and the waiting time of the currently stored entity has to be reduced. This can easily be done by changing the value of σ. Note that the "macroscopic" states *idle* and *busy* are only implicitely given by

$$idle \equiv (v = \varnothing)$$
$$busy \equiv (v \in \mathbb{R})$$

A crucial point in the definition of δ and λ is to take all possible values of $Q \times X^+$ into account. A helpful approach here is to divide these values into internal, external and confluent events – just as in PDEVS –, using the values of the function arguments s, e and x^+.

The definition of the queue can be done along these lines using the state diagram in Figure 6. Four macroscopic states are defined according to the blocking status and the size of the queue (empty or not). The critical state here is *queuing free*, which is the only state, where the queue outputs entities. It is a transitory state, which in the context of NSA-DEVS becomes a state with an infinitesimal delay $r_d\,\varepsilon$. The value of r_d is defined as a parameter with the usual default value of 1; as we will see, it plays a crucial role in the correct implementation of the singleserver example.

If one runs the complete model on the Matlab-based NSA-DEVS simulator, using default parameter values for all input delays and r_d, the result is incorrect: At time $t = 4$ the queue sends the fourth entity to the server instead of the third, which gets lost (cf. Figure 7). The basic reason for this behaviour is evident: After sending entity three, the queue stays in the state *queuing free* and sends the next entity after a delay of ε, because the

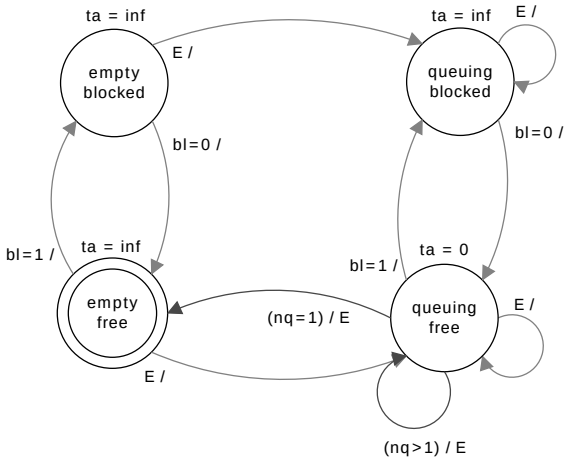

Figure 6: State diagram of the queue component.

blocking signal from the server has not arrived yet. In order to guarantee the desired ordering of events, one simply has to properly enlarge the value of r_d to get the expected simulation results.

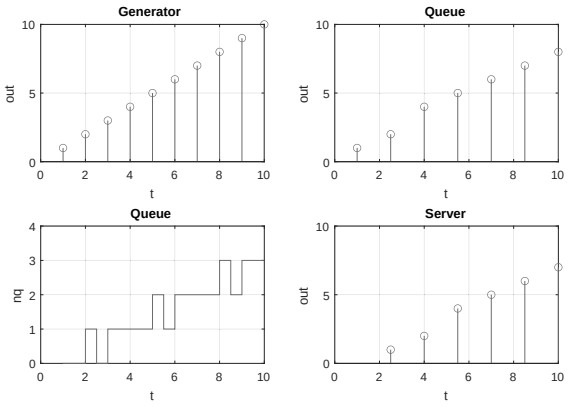

Figure 7: Simulation results of singleserver with default parameters.

Simple considerations like these are usually sufficient to cope with most problems coming from a wrong ordering of concurrent events. But if one looks more closely, there are still some fine points that are not immediately obvious, such as: Why are the results shown in Figure 7 so much different from the corresponding results in [4, Fig. 6]? And why is a value of $r_d = 1.1$ already large enough to produce correct results? One can always use the debugging features of the simulator to retrace its behaviour, which makes answering such questions a straightforward, if tedious, exercise [12].

5 A Complex Queuing System

The final example is a simplified version of the basic queuing system from the Argesim benchmark C22 [13]. It contains a generator and a set of three queue-server lines. Incoming entities are routed to the shortest line (including the server allocation) and leave the system after being served (cf. Figure 8). The benchmark defines two model variations according to the order of concurrent events: In variant A an entity leaves a server, before a new entity enters the system, in variant B the order is reversed.

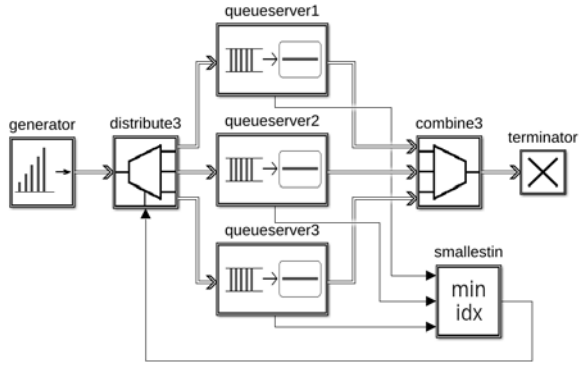

Figure 8: Example 4: Queuing system fifo3 containing three queue-server lines.

The queue-server lines are defined as coupled models consisting of the queue and server components from above and a simple atomic block that adds their loads. An additional ToWorkspace block is added for logging purposes (cf. Figure 9). According to the lesson learned from the last example, all input delays are identical, while the delay time of the transient queue state is twice as large.

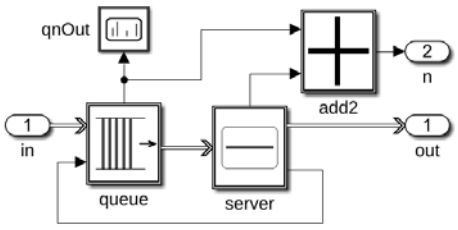

Figure 9: Internal structure of coupled model queueserver.

Three more atomic components are needed to complete the example:

- a distributor that routes incoming entities to the output that is defined by a control input,
- a combiner that accepts entities from its inputs and sends them to its single output,
- a computational block (smallestin) that gets the three loads, computes the minimal value and outputs the number of the smallest input where the minimum occurs.

They all can be implemented easily, the only interesting one is the combiner: When entities appear concurrently, they will be stored internally and output one after the other. The necessary transitory state as always induces an (infinitesimal) delay between the outgoing entities. If one uses a ToWorkspace block to display them, one has to set its delay to a small value, since otherwise one would only see the last outgoing entity.

Using only standard parameters for all delays, the complete model works without further ado and produces the expected results (cf. Figure 10). Having a close look at the order of the outgoing entities, one finds that the model realizes the variant B: Due to the delays of the adder and the smallestin block, the information that a server is empty arrives at the distributor after the new entity from the generator has passed. To implement variant A, one simply increases the delay of the distributor either to an arbitrary large value or – after chasing the delays through the diagram – to 3ε.

6 Conclusion

As the careful analysis of the examples has shown, the expected large number of delay parameters needed in NSA-DEVS usually boils down to one default value and a bit of fine-tuning in special cases. The most notable exception was the queue, where the lifetime of the transitory state has to be enlarged. But since a queue component with a properly adapted default value will usually be part of the model library, a user in practice won't come in touch with this exception.

Two simple provisions have been identified that often will help to hide the internal details: Firstly, if a component with several output ports sends values only to some of them, it should send an empty value to the remaining ports, in order to get rid of intermediate output values. In a Matlab implementation this could be the empty array `[]`. Secondly, one should choose a large input delay for the ToWorkspace blocks, which again could be predefined already in the model library.

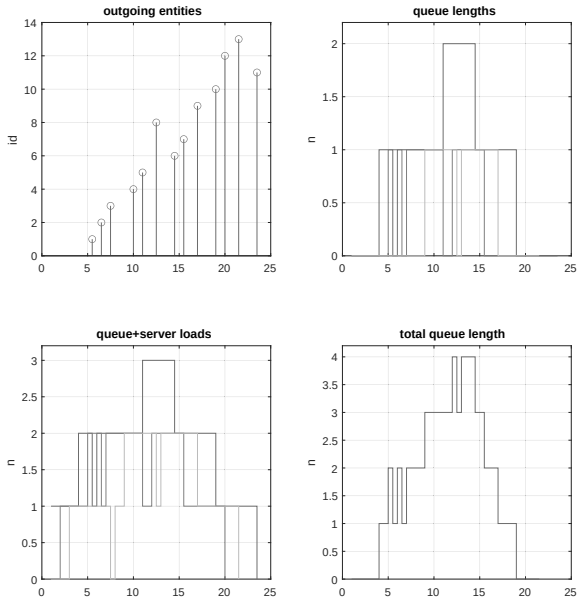

Figure 10: Simulation results of fifo3 with default parameters.

Of course there are situations, where one needs to think about the infinitesimal behaviour, e. g. to define a special ordering of concurrent events. Another example occurs in the comparator-switch model, when a chain of components before the comparator input leads to a large total delay, which has to be compensated by the delay of the switch. But such difficulties are basically inherent to discrete-event modeling, and the question is not, how to avoid them, but how to cope with them in a clear-cut way. One could argue that NSA-DEVS provides the right balance between hiding details in simple cases and giving access to the internals to solve these problems.

An open question at the end of [5] was, whether one delay time for an atomic model suffices, or if one needs port specific delay times. So far, the simple definition used in NSA-DEVS seems to work generally. If necessary, a simple workaround would be, to add a special delay component – basically a gain with factor 1 – before a port that needs a special delay. This question could be further examined in the context of the final remaining task from the todo list at the end of [4], which was the implementation of a complex case study in NSA-DEVS, and will be adressed in a future investigation.

References

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

[2] 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.

[3] 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.

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

[5] Jammer D, Junglas P, Pawletta T, Pawletta S. A Simulator for NSA-DEVS in Matlab. In: *Proc. of ASIM 2022 – 26. Symposium Simulationstechnik*. Wien. 2022; pp. 93–100. doi: 10.11128/arep.20.a2005.

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

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

[8] 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.

[9] Pawletta T, Deatcu C, Pawletta S, Hagendorf O, Colquhoun G. DEVS-based modeling and simulation in scientific and technical computing environments. In: *Proc. of DEVS Integrative M&S Symposium (DEVS'06) - Part of the 2006 Spring Simulation Multiconference (SpringSim'06)*. Huntsville/AL, USA: D. Hamilton. 2006; pp. 151–158.

[10] Junglas P. Pitfalls using discrete event blocks in Simulink and Modelica. In: *Proc. of ASIM-Workshop STS/GMMS*. Lippstadt. 2016; pp. 90–97.

[11] Fiedler C, Preyser FJ, Kastner W. Simulation of RPDEVS Models of Logic Gates. *SNE Simulation Notes Europe*. 2019;29(2):85–91. doi: 10.11128/sne.29.tn.10474.

[12] CEA Wismar. *NSA-DEVS on GitHub*. `https://github.com/cea-wismar/NSA-DEVSforMATLAB`.

[13] Junglas P, Pawletta T. Non-standard Queuing Policies: Definition of ARGESIM Benchmark C22. *SNE Simulation Notes Europe*. 2019;29(3):111–115. doi: 10.11128/sne.29.bn22.10481.