NSA-DEVS: Combining Mealy behaviour and Causality

Peter Junglas*

Dep. of Engineering "Dr. Jürgen Ulderup", PHWT Vechta/Diepholz, Schlesierstr. 13a, 49356 Diepholz, Germany; *peter@peter-junglas.de

Abstract. The RPDEVS ("Revised PDEVS") formalism has been introduced to allow for a simple description of Mealy-type components that behave consistently. This made it necessary to change the way the simulator handles event chains. Using a simple example model we show that the proposed algorithm has serious problems with the resulting sequence of concurrent events. Therefore we introduce NSA-DEVS, a variant formalism that is inspired by ideas from non-standard analysis (NSA). It uses infinitesimal time delays to make a natural ordering of concurrent events easy, without the need to fix lots of additional parameters. As proof of concept we describe the example model in NSA-DEVS and implement it in a suitably twisted RPDEVS simulator.

Introduction

More than 40 years after its invention the DEVS formalism [1] and its most popular variant PDEVS [2] are now standard tools for the mathematical modeling of discrete-event systems. If in doubt a quick search for "DEVS formalism" in Google scholar reveals over 6000 papers and that the seminal book of Zeigler et al. [3] has been cited about 7000 times.

Looking at widely-used simulation environments, the picture changes completely: Neither Arena [4] nor SimEvents [5] use DEVS internally or even mention it in their documentation. And though Mathworks has based its redesign of SimEvents on a proper modeling formalism, the developers didn't choose DEVS for this purpose [6].

On the other hand there are quite a few free simulation programs available that use DEVS or one of its variants for the definition of atomic components and the implementation of coupled systems [7]. But all of them twist the original DEVS formalism to make it a suitable foundation for a concrete simulation environment [8]. Some of the problems are just minor nuisances, like the addition of input and output ports, others are of a more fundamental nature.

Probably the most serious flaw has been named by Preyser et al., who show in [9], that PDEVS has difficulties modeling certain Mealy-type components: The necessary introduction of transitional states leads to delays that change the expected order of concurrent events and the behaviour of subsequent components. This is a serious drawback, if one wants to define a library of reusable blocks. Therefore the PDEVS formalism has been altered in [10] to allow for Mealy-like behaviour thereby introducing the revised version RPDEVS.

To make this work, one has to change the way chains of concurrent events are handled, which is a complex and possibly dangerous endeavour. Even after the careful analysis in [10] and the formal definition of an RPDEVS simulator [11] the question remains, whether the proposed scheme is capable of handling the subtle problems that appear in practical modeling tasks.

To further investigate the status of RPDEVS we will introduce a simple example that is plagued by a complex causal structure of concurrent events, and implement it in PDEVS and RPDEVS, using freely available simulators.

Since the results show that RPDEVS has problems with the example model, we will propose a different way of how to cope with concurrent event chains, which uses concepts of non-standard analysis [12]. After a short introduction to the basic mathematical ideas, we will define the new DEVS variant NSA-DEVS, which combines the ideas of RPDEVS with a more robust method to handle concurrent events.

Finally we will implement the standard example in NSA-DEVS and find that it can handle this model in a clear-cut, easily understandable way. This supports the assumption that NSA-DEVS might be a good basis for concrete modeling and simulation environments, since it combines the security in the handling of concurrent events from PDEVS with the modeling power of RPDEVS.

1 Singleserver - a fundamental example

The singleserver example used in the following 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. Fig. 1). Additionally the queue and the server output the current number of entities stored.

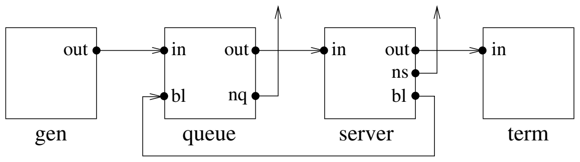

Figure 1: Example model singleserver.

Though this is probably by far the most studied system in discrete modeling, it is not trivial at all, especially if you try to model it with PDEVS. In the fundamental book of Zeigler et al. [3] a queue-server combination is modelled as one atomic component. But trying to create separate atomic models for a queue and a server is much more challenging due to the complex interaction of the two components.

The server can be implemented easily using the state diagram shown in Figure 2: When an entity E arrives, the server outputs the new blocking status and changes to the "busy" state. After the service time, it outputs the entity and the changed blocking status and returns to the "idle" state. In this and the following figures the annotation $(A)B/C$ on an arrow means: If condition A is true and input is B, then the output is C and the state changes. Any of the three parts may be missing.

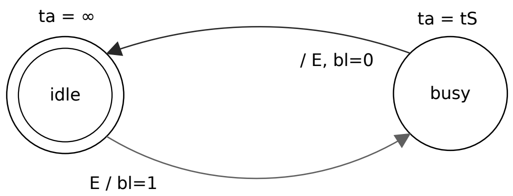

Figure 2: State diagram of the server component.

The behaviour of the queue is much more complicated, it is modelled here using the state diagram in Figure 3. The four states are distinguished by the size of the queue ("empty", "queuing") and the blocking status at the output of the queue ("free", "blocked"). The only internal transitions occur in the state "queuing free", they output an entity and have zero transition time. All other transitions are external, triggered by an incoming entity or a new blocking status.

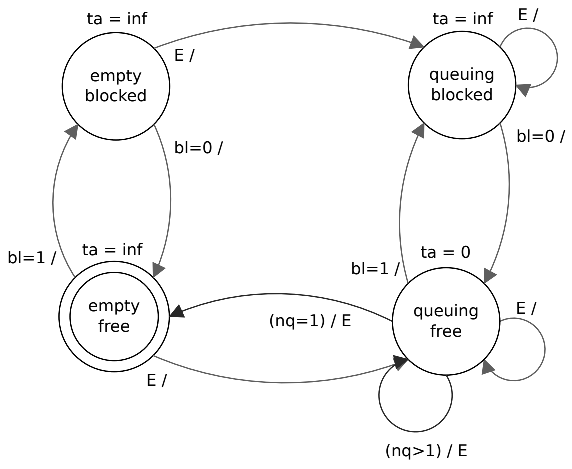

Figure 3: State diagram of the queue component.

This implementation of a queue uses a push strategy, where entities proceed as far as possible, until they are blocked. This is the basic idea behind many discrete event simulators, from GPSS to SimEvents. Alternatively, one could use a pull strategy, where entities only proceed, when they are called by a component. This would lead to a slightly simpler implementation of our example. It is an interesting question, whether pull or push strategies are better suited for complex simulation environments, but not the point of this investigation.

The simulation of the complete singleserver model leads to complicated cascades of concurrent events. For a typical example assume that the queue is in state "queuing blocked" with $nq > 1$ and the server gets ready, going from "busy" to "idle". It sends its new blocking status $bl = 0$ to the queue, which now transitions to "queuing free". Using an internal transition the queue outputs an entity, which arrives at the server, leading to a transition to "busy" and the sending of $bl = 1$ back to the queue. Now the queue has to change to "queuing blocked", *before* another entity is output via an internal transition.

2 Implementing singleserver with PDEVS

The fundamental component in the PDEVS formalism is an *atomic PDEVS* [3]. It is formally described by an 8-tuple $< X^b, S, Y^b, ta, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda >$ with the meanings

X^b	set of possible input bags
S	set of states
Y^b	set of possible output bags
$ta : S \to [0, \infty]$	time advance function
$\delta_{int} : S \to S$	internal transition function
$\delta_{ext} : Q \times X^b \to S$	external transition function
$\delta_{con} : S \times X^b \to S$	confluent transition function
$\lambda : S \to Y^b$	output function

where an element of Q combines a state and the time since the last internal transition, i.e.

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

It is important to note, especially for the present discussion, that the output function λ is only called directly before an imminant internal transition.

Atomic components can be combined to form a *coupled PDEVS*, which is formally defined by the set of components and their internal and outwards connections.

To implement the singleserver example in PDEVS one has to augment the state diagrams with transitional states that allow to produce output values, when an input appears, i. e. at an external event. For the server component (Fig. 2) one additional state is sufficient, and the definition of the transition, output and time advance functions is straighforward.

The definition of the queue component (Fig. 3) is much more complicated, its extended state diagram contains five additional states and a lot of corresponding additional transitions (Fig. 4). The purpose of the four states "n out A/B/C/D" and their corresponding transitions is evident: Whenever an entity arrives, the length of the queue changes, and a corresponding output value has to be sent.

But the new states lead to further complications, in particular for the definition of the external transition function δ_{ext}: When a new entity and a new blocking status arrive at the same time, the state has to proceed two "steps" at once to reach a necessary transitional state. E. g. when the queue is in state "queueing

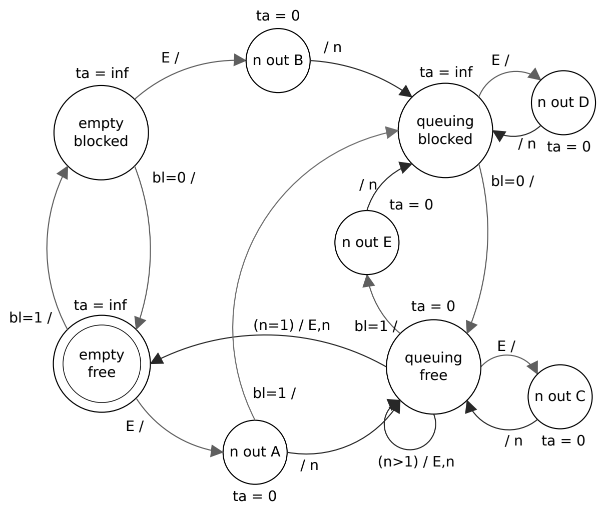

Figure 4: State diagram of the queue component with additional transient states.

blocked" and a new entity arrives together with $bl = 0$, the new entity is stored and the new state is "n out C".

Special care is needed for the definition of the confluent transition function δ_{con}. Usually it first calls the internal, then the external transition function, so that the entity at the head of the queue leaves, before the new entity is stored. But if a new value $bl = 1$ arrives, only the external transition function is used, so that no entity leaves the queue. This leads to a change of the queue size without a call of the internal transition function, therefore one needs another transitional state "n out E" to produce the corresponding output.

If one has taken proper care of all complications the complete model can be implemented in a PDEVS simulation environment like MatlabDEVS [13], where a simuation run will produce the expected results shown in Fig. 5. The "spikes" in the plots of the queue length and the server allocation are remnants of the concurrent event chains, where state variables have different values at the same time instant.

3 Trying to implement singleserver with RPDEVS

To make the direct definition of Mealy components possible, Preyser et al. define RPDEVS in [10] as follows: An *atomic RPDEVS* is a simplified version of the atomic PDEVS, which contains only a generic transition δ. Moreover its output funtion λ is called at

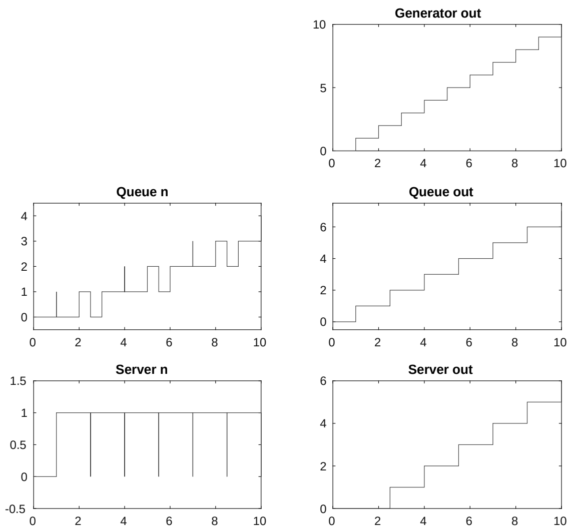

Figure 5: Simulation results of the PDEVS model.

any kind of event and depends on the state and the input. Formally an atomic RPDEVS is given by a 6-tuple $< X^b, S, Y^b, ta, \delta, \lambda >$ with the meanings

X^b	set of possible input bags
S	set of states
Y^b	set of possible output bags
$ta : S \rightarrow [0, \infty]$	time advance function
$\delta : Q \times X^b \rightarrow S$	state transition function
$\lambda : Q \times X^b \rightarrow Y^b$	output function

and the Q defined above.

A *coupled RPDEVS* is formally defined as in PDEVS, describing the subcomponents and the internal and outward connections. But its behaviour is different, in order to cope with possible Mealy components: The call of a λ-function produces outputs that are routed to other components, where they in turn may lead to a call of their λ-functions, creating cascades of λ steps, which might change already processed input values. In such a case earlier input values are withdrawn and replaced by new ones (or cancelled completely). For models without algebraic loops these λ iterations will finally lead to a situation, where all input bags are constant. Only then a single δ call is issued.

The singleserver example can be formulated in RPDEVS much easier than in PDEVS, since one can stick to the simple state diagrams shown in Fig. 2 and Fig. 3. This allows to specify the atomic components in a straightforward way by identifying the event type inside the generic δ function according to the input bag and the elapsed time. Using the PowerRPDEVS simulator [14], the components can be easily implemented in C++. Finally one can construct the complete singleserver example in a graphical environment.

Though all components work in simple test models, the simulation of the singleserver model aborts at t = 1. The error message states that the maximum number of λ steps has been reached and that the model is illegitimate due to a non-resolvable algebraic loop.

The reason for this behaviour becomes clear, when one analyses the internal chain of events in the simulator at t = 1 (cf. Table 1): The queue starts in state "empty free" and changes to state "queuing free" in line 3, while the server remains in state "idle". In line 4 the queue outputs its entity that is routed to the server, which now sends the blocking status bl = 1 to the queue. The basic problem now happens in line 6: The λ function of the queue is called again, now with bl = 1 in the input bag. The entity that has been sent before, is now blocked and has to be retrieved. This in turn leads to the withdrawal of the bl=1 message from the server, therefore the queue tries again to output its entity in line 7. The situation is now identical to line 4 and repeats, until it is stopped, when the maximal count of λ steps is reached.

No.	Block	Type	Out	Q in	Q bl	S in
1	Gen	λ	E1	E1		
2	Que	λ				
3	Que	δ				
4	Que	λ	E1			E1
5	Srv	λ	bl=1		1	
6	Que	λ				
7	Que	λ	E1			E1

Table 1: Events at t = 1.

The basic idea behind the state diagram in Fig. 3 was, that a component changes its state immediately after sending its output message. Therefore new input messages, that arrive due to event cascades, find the component in a changed state. But the repeated execution of λ steps without any state changing δ steps in RPDEVS leads to a completely different behaviour.

From the point of view of RPDEVS the singleserver is faulty, containing an algebraic loop. On the other hand the behaviour described in Fig. 3 is quite simple and can be easily implemented in PDEVS. Zeigler knew very well, why he didn't include Mealy-type behaviour.

But if one insists on it for the sake of better modularity, one has to think over the simulator behaviour, or better: the abstract model behind it.

4 Extending the time line by infinitesimals

Modeling experience teaches us that a mathematical problem in the description or simulation of a model often has its roots in an oversimplification of the system one wants to describe. This is of course true here: In real world systems small delay times are inevitable, whenever a message is sent or a state changes. If one includes them in the model description, the problems with cascades of concurrent events disappear immediately. But the price one has to pay for this solution, is high: Such a model contains a huge amount of delay times, whose values are not known, often not even their order of magnitude. In addition, the behaviour of the model gets much more complicated on a fine time scale, though one often is not interested in these details.

What we are looking for, are time steps that are larger than zero, but so small that they can be ignored for most purposes. Furthermore their actual size should not matter, even though we need different sizes of such steps. This is actually exactly what one commonly denotes as *infinitesimals*. Hewitt [15] and Robinson [16] have shown that one can implement such ideas in a mathematically rigorous manner. Therefore we will shortly introduce the basic concepts and use them afterwards for a new definition of discrete event systems. A precise and pedagogical introduction to the mathematical ideas and applications can be found in [12].

The set ${}^*\mathbb{R}$ of *hyperreals* is a totally ordered field that includes the ordinary real numbers. In addition it contains an infinitesimal element $\varepsilon > 0$ that is smaller than any positive real number. Using the field axioms one gets additional infinitesimals like $2\varepsilon, -\varepsilon, \varepsilon^2$. Each real number r is surrounded by an infinite cloud $r + \delta$ with infinitesimal δ, its *halo*. On the other end ${}^*\mathbb{R}$ contains $\omega := 1/\varepsilon$, which is *unlimited*, i. e. larger than any real number. Again one has lots of unlimited numbers like $2\omega, -\omega, \omega^2$, which are all surrounded by clouds $\omega + r + \delta$ with real r and infinitesimal δ, called their *galaxy*. Each limited element $h \in {}^*\mathbb{R}$, i. e. an element of the galaxy of 0, can be uniquely written as $h = r + \delta$ with real r and infinitesimal δ. r is called the *standard part* $st(h)$ of h.

The actual construction of ${}^*\mathbb{R}$ relies on heavy machinery from set theory and logic, like ultrafilters and the axiom of choice. From our current point of view the main reason for an explicit construction is to convince oneself that such a set exists in a precise mathematical way. Hyperreals have been used to reformulate the usual analysis with definitions that closely mimic the original ideas of Leibniz, an endeavor commonly designated as *nonstandard analysis*. This often leads to simple and intuitive proofs – once one accepts the basic properties of ${}^*\mathbb{R}$.

It is impossible to implement real numbers in a computer, much less hyperreals. For our purposes it is sufficient to use pairs (t, r) of floating point numbers, which correspond to the hyperreal $t + r\varepsilon$, where t could be the floating point value ∞ to include infinite time delays in passive states. This implementation looks similar to the concept of *superdense time* [17] that uses a pair of a real time value and a natural number for ordering of concurrent events. But the structure of the hyperreals is much richer, and the reasoning behind their use is more intuitive and better adapted to the problems that are adressed here.

5 Definition of NSA-DEVS

We will now use non-standard analysis ("NSA") to get rid of concurrent events by defining NSA-DEVS, a variant of RPDEVS. The basic idea is to forbid transient states, i. e. transition times are always > 0, though they may be infinitesimal. Furthermore we assume that the transport of data between components always takes a certain amount of time. Therefore we include an input delay $\tau > 0$ between the arrival of input and the call of the output function.

An *atomic NSA-DEVS* is given by a 7-tuple $< X^b, S, Y^b, \tau, ta, \delta, \lambda >$, where $\tau \in {}^*\mathbb{R}_{>0}$ is the *input delay time*. All other elements have the same meaning as in RPDEVS, but the definitions of the functions are changed to

$$\begin{aligned}
&Q := \{(s,e) | s \in S, e \in (0, ta(s)]\} \\
&ta : S \to (0, \omega] \\
&\delta : Q \times X^b \to S \\
&\lambda : Q \times X^b \to Y^b
\end{aligned}$$

The intervals $(0, ta(s)]$ and $(0, \omega]$ are meant as subsets of the hyperreals ${}^*\mathbb{R}$.

When an external event, i.e. an input $x \in X^b$, 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. an imminent 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 as in RPDEVS and PDEVS, outputs are transported as usual. Due to the (infinitesimal) delays a strict Mealy-type behaviour is impossible, therefore special provisions like the iterated λ calls in RPDEVS are unnecessary, each λ call is followed immediately by the corresponding δ call.

Formally we have introduced a lot of additional (infinitesimal) parameters, but a simulator might be able to free the user from this burden by using a simple default value of $\tau = \varepsilon$ for all components. Furthermore the transition times of previously transient states could be set to ε in many cases. It remains to be seen, whether such a simple scheme actually works in standard situations. In the case of the singleserver example manual finetuning is necessary, as will be shown below. On the other hand, if one insists on a special ordering of (ideally) concurrent events, one can use the infinitesimal delays to achieve any order in a quite intuitive way.

Since one is generally not interested in the infinitesimal behaviour, an NSA-DEVS simulator should show state changes and output values only at the end of an infinitesimal cascade, i.e. directly before a finite (non-infinitesimal) step. All short-lived states and overwritten outputs are then internal to the simulator. This behaviour is somewhat similar to that of an ODE solver that uses microsteps internally for stepsize adaptation, but outputs only completed steps. Optionally it should be possible to replace the value ε by a user supplied small real number for debugging purposes or to analyse the behaviour at the infinitesimal scale.

6 Implementing singleserver with NSA-DEVS

To adapt the RPDEVS description of singleserver to NSA-DEVS, one needs two modifications: All components get an additional parameter τ with a default value of ε, and the queue component gets a further parameter t_D, which defines the transition time of the state "queuing free", again with a default value of ε.

Until a proper NSA-DEVS simulator is available, one can use the PowerRPDEVS simulator to mimic the behaviour in debug mode, where the infinitesimal value ε is set to a small number ($\varepsilon = 10^{-4}$ in the following examples). To this end one creates a simple delay component, basically a simple server, and adds it before every input of each component. All delay times are set to the value of τ of the corresponding component. In particular, the delay times of several inputs of one component have to be identical to properly implement the NSA-DEVS behaviour defined above. These delays guarantee that no λ iterations occur, therefore the simulator works as required by the NSA-DEVS definition.

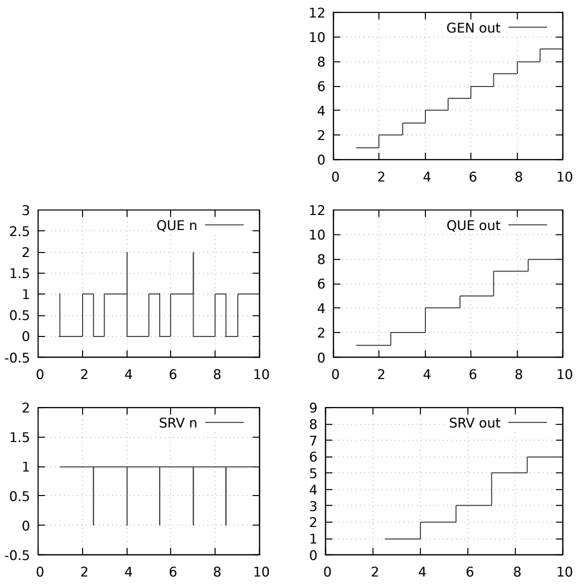

Figure 6: Simulation results of the NSA-DEVS model with default delays.

With these changes the singleserver model runs in "NSA-DEVS debug mode", the results are shown in Fig. 6. But they are not as expected: At $t = 4$ the entity 4 joins entity 3 in the queue and both leave the queue immediately. After the service time entity 3 leaves the server at $t = 5.5$, but entity 4 is lost. At first thought this problem seems to be related to the nullserver problem described in [18]: The delay at the input of the server acts as additional storage and accepts entity 4 though the server is busy.

While this is true, the real cause of the problem lies elsewhere. Even in a correct implementation of NSA-DEVS, where the delay is implemented directly in the simulator, entity 4 would get lost! This is due to the delay of the "busy" message from the server: Before it arrives, the queue has already output the next entity according to the internal transition shown in Fig. 3. This is an example, where it is not sufficient to use the default value ε for all infinitesimal delays.

In order to obtain the desired causal ordering of “concurrent” events, a fine tuning of the delays is necessary. One has to guarantee that the message from the server arrives, before the queue sends a new entity. Therefore one sets the transition time t_D of the “queuing free” state to a value larger than the sum of the two transport delays at the inputs of the server and the queue. Choosing $t_d = 2.1\,\varepsilon$ solves the problem and the results are as expected. They coincide with the results of the PDEVS implementation (cf. Fig. 5), including the “spikes”. Here they have a small, but finite width, like it should be in a proper NSA-DEVS simulator in debug mode. In standard mode output values that change in an infinitesimal time are suppressed and only the last values are shown.

7 Conclusions

Like RPDEVS, the variant NSA-DEVS proposed here allows for simple and consistent handling of Mealy-type behaviour. Furthermore it seems to solve the problems of RPDEVS with chains of concurrent events. Therefore it possibly could be a working basis for a concrete simulation environment.

The drastic measure of prohibiting real “concurrency” for causally ordered events generally causes serious side effects by introducing lots of additional time parameters. This is mitigated here by the introduction of infinitesimal delays used mainly internally and whose actual values do not matter. That such a scheme is mathematically sound has been shown by referring to the results of non-standard analysis.

A certain amount of fine-tuning can still be necessary to ensure a requested causal ordering. But this is not a speciality of NSA-DEVS: The proper behaviour of the system has to be modelled anyhow, in PDEVS this is done by a careful design of the confluent transition function. In case of causally unrelated events, one could twist some of the infinitesimal delay parameters to ensure a certain temporal order, if requested.

At first sight NSA-DEVS seems to destroy the potential of parallel execution. But this is not necessarily the case: For unrelated events one simply chooses identical delays – usually just ε –, so that they still occur at the same time $t \in {}^*\mathbb{R}$ and can be executed in parallel. Only causally depending events have different times, so that their order is fixed – as it should be. Moreover, the elimination of many transitional states in RPDEVS and NSA-DEVS could provide more opportunities for parallel execution than PDEVS.

This is only a first step in the analysis of a new DEVS-based scheme that could possibly be used for simulation environments and the definition of universally applicable component libraries. The next step would be the definition and implementation of a proper simulator. This could be followed by a thorough investigation of standard examples with complex event cascades, such as a switch that routes entities according to an input value [9] or models of digital circuits containing flip-flops [19, 20]. Finally one could try to implement a complex case study like the Argesim C22 benchmark [21] as a further step to investigate the practical usefulness of the proposed scheme.

Acknowledgement

The author gratefully acknowledges clarifying discussions with Christina Deatcu and Thorsten Pawletta.

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] Zeigler BP, Praehofer H, Kim TG. *Theory of Modeling and Simulation*. San Diego: Academic Press, 2nd ed. 2000.

[4] W David Kelton NBZ Randall P Sadowski. *Simulation with Arena*. New York: McGraw-Hill, 6th ed. 2015.

[5] Clune MI, Mosterman PJ, Cassandras CG. Discrete Event and Hybrid System Simulation with SimEvents. In: *8th International Workshop on Discrete Event Systems*. Ann Arbor. 2006; pp. 386–387.

[6] Li W, Mani R, Mosterman PJ. Extensible discrete-event simulation framework in SimEvents. In: *Proc. 2016 Winter Simulation Conference*. Arlington: IEEE. 2016; pp. 943–954.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] Preyser F. *PowerRPDEVS on sourceforge*. URL `https://sourceforge.net/projects/powerrpdevs/`

[15] Hewitt E. Rings of real-valued continuous functions I. *Transactions of the American Mathematical Society*. 1948;64(1):45–99.

[16] Robinson A. *Non-standard Analysis*. Amsterdam: North-Holland. 1966.

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

[18] Austermann L, Junglas P, Schmidt J, Tiekmann C. Conceptional problems of transaction-based modeling and its implementation in SimEvents 4.4. *SNE Simulation News Europe*. 2017;27(3):137–142. doi: 10.11128/sne.27.tn.10383.

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

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

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