Using component-based discrete-event modeling with NSA-DEVS – an invitation

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

[1] PHWT-Institut, PHWT Vechta/Diepholz, Am Campus 2, 49356 Diepholz, Germany; *peter@peter-junglas.de*

[2] Research Group Computational Engineering and Automation, University of Applied Sciences Wismar, Philipp-Müller-Straße 14, 23966 Wismar, Germany

Here will be SNE-specific data positioned.

To be filled out by editor.

Abstract. The quest for a simulation scheme that combines the preciseness of the PDEVS formalism with the ease of use of standard simulation environments has lead to the definition of NSA-DEVS, which has meanwhile been shown to provide a useful basis for real-world applications. A set of modeling and simulation tools for NSA-DEVS is freely available that uses Matlab as programming language and the graphical editor of Simulink for the construction of complex models from simple atomic components.
To demonstrate that these tools are ready for general discrete-event based applications, the implementation of a textbook example is presented in some detail. It is shown that components that are necessary for a transaction-oriented style can be modeled easily, leading to a comprehensible model with a solid mathematical basis.

Introduction

For modeling and simulation using the discrete event approach, practitioners can choose between a lot of commercial simulation environments, which provide users with a wide range of components and helpful tools [1]. However, the behaviour of complex models can sometimes be different than expected, especially because the documentation often does not provide all necessary details. In such situations users generally build a toolset of workarounds to make things work. But this often leads to conceptual problems and does not deepen the understanding of the precise behaviour of a model [2].

On the other hand, one could start instead with a precise description of the model and its components, using the well-established PDEVS formalism [3]. There are even a few free tools that provide a user-extensible set of DEVS-based components and a graphical user interface for combining components to build complex models [4]. But Preyser et al. have shown in [5] that the way, how PDEVS uses transitory states (i. e. states with lifetime 0), makes it hard to define some simple reusable components, especially when they show Mealy behaviour. Therefore they proposed a revised version of the PDEVS formalism [6] that allows for the direct description of Mealy components.

However, this formalism still has problems with chains of concurrent events [7]. Therefore it has been extended to NSA-DEVS (*Non-Standard Analysis DEVS*), which solves these difficulties by formally introducing infinitesimal delays. This idea has been analyzed thoroughly in [8, 9] and used to implement a large real-world example [10]. A corresponding simulation environment has been built, which contains graphical tools and a growing library of components. It is based on Matlab and the graphical editor of Simulink and is freely available from [11].

Since this article is primarily an invitation to use these new tools for modeling and simulation in discrete-event based studies, it concentrates on practical aspects, not on the underlying mathematical formalism. After a short definition of NSA-DEVS and a recapitulation of previous results, the structure of the tools and the basic workflow for concrete studies will be presented in some detail. A basic example from Law's textbook [12] will illustrate how to implement and apply components for standard entity-based applications.

1 Definition of NSA-DEVS

Like the PDEVS specification, the NSA-DEVS formalism describes two types of models: *atomic models*, which are the basic components, and *coupled models*, which combine atomic and coupled models in a hierar-

chical structure.

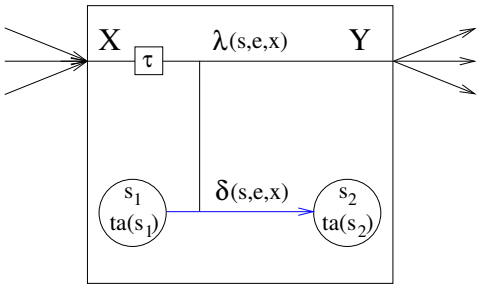

Figure 1: Dynamics of an atomic component.

An atomic model (cf. Fig. 1) is given by a set X of input ports, each with a name, a similar set Y of output ports, a set S of internal states and an input delay time τ. Each state s has a lifetime, given by the time advance function $ta(s)$. The behaviour is given by the output function λ, which defines output values y, and the transition function δ, which computes the next internal state. Both functions depend on three values: the current state s, the elapsed time e since the last transition and the input values x. When an external event, i. e. a set x of input values, occurs at time t, λ is called at time $t+\tau$, 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.

The essential modification of NSA-DEVS is the introduction of the input delay together with extended time values: Introducing an infinitesimal value $\varepsilon > 0$, times are defined as values of the form $a+b\varepsilon$. The delay time is usually defined as $\tau = \varepsilon$ and only changed at rare occasions to guarantee a given order of events. Furthermore, the lifetime of states can never be 0, but an "immediate" (transitory) state change needs at least an infinitesimal time τ_D.

A coupled model is basically a set of several lists that describe the submodels used (atomic or coupled), the input and output ports of the coupled model and all connections between the submodels and from or to the ports of the coupled model. Inputs of a coupled model are immediate, i. e. they have no additional input delays.

2 Current Status of NSA-DEVS

The fundamental insight of [5] was that while the basic PDEVS formalism is sufficient to model any discrete-event based system, this is generally not possible with every reusable component. After the definition of NSA-DEVS one had therefore to show that it was up to this task.

The first step was the definition of a corresponding abstract simulator [8]. According to the DEVS philosophy, the simulator is actually part of the definition of the formalism. It adds a semantic layer to the static definition of models by defining their exact behaviour. In a next step [9], a set of standard examples was defined formally and implemented using simple reusable atomic components. A special point of interest was, how one can define the infinitesimal parameters introduced by NSA-DEVS in a simple and systematic way. This was studied further with a large real-world example in [10] consisting of 391 atomic and 88 coupled models in 5 hierarchical levels. It contained 391 input delay times τ and 12 additional delays τ_D for transitory states. Its construction has been simplified by the introduction of a graphical model builder, which uses Simulink for the definition of coupled models.

In the course of these investigations, a basic set of atomic models has been defined and implemented:

- sources (constant, several generators),
- math operations (add, gain, multiply, divide, compare),
- logic operations and flipflops based on IEEE 1164,
- routing components (combine, distribute),
- a QSS-based integrator,
- a sink (toworkspace) for logging simulation results,
- common logistics components (queue, server, batch, unbatch, terminator),
- statistical computations (getmax, utilization).

For all atomic models, corresponding NSA-DEVS blocks are provided for the Simulink editor and assembled in libraries. They make it possible to construct coupled models using Simulink's graphical capabilities for positioning and connecting blocks and ports. In addition, block parameters can be set, among them the values for the input delay τ and, where necessary, the transition delay τ_D.

All delay values are predefined and usually set to the default value $\tau_{def} = \varepsilon$. An exception are components that emit trains of output values with infinitesimal time

distances, such as queue and combine atomics. They need larger values for τ_D, which are predefined in the library to $\tau_D = 2\varepsilon$. This often works, but has to be enlarged in special cases. If a special ordering is requested for loops that contain several sequences of components, one has to increase a few input delays to slow down some paths [10].

3 Implementation in Matlab

For the implementation of the example models, a set of tools and a component library have been constructed that are based on Matlab and the Simulink editor. Using a simple example, we will describe the basic workflow necessary to implement own models and running simulations.

Listing 1: Atomic model am_add2.

```
1  classdef am_add2 < handle
2    properties
3      s
4      in1
5      in2
6      name
7      tau
8      debug
9    end
10   methods
11     function obj = am_add2(name, tau,
           debug)
12       obj.s = "running";
13       obj.in1 = 0;
14       obj.in2 = 0;
15       obj.name = name;
16       obj.debug = debug;
17       obj.tau = tau;
18     end
19     function delta(obj,e,x)
20       if isfield(x, "in1")
21         obj.in1 = x.in1;
22       end
23       if isfield(x, "in2")
24         obj.in2 = x.in2;
25       end
26     end
27     function y = lambda(obj,e,x)
28       s1 = obj.in1;
29       s2 = obj.in2;
30       if isfield(x, "in1")
31         s1 = x.in1;
32       end
33       if isfield(x, "in2")
34         s2 = x.in2;
35       end
36       y.out = s1 + s2;
37     end
38     function t = ta(obj)
39       t = [inf, 0];
40     end
41   end
42 end
```

The atomic models are the basic building blocks. They are implemented in Matlab as classes that contain a constructor and the methods `delta`, `lambda` and `ta`. A simple example is the class `am_add2` for the addition of two input values (cf. Listing 1) and shows how to implement a Mealy component. On first sight, it looks similar to its counterpart in a continuous environment, but the discrete-event nature leads to a very typical change: Since input values are defined only, when an input event arrives, these values have to be stored internally using properties `in1` and `in2`. The property `s` is used throughout the whole library to denote "macroscopic" states, which is useful in more complex atomics. Here it is simply set to a constant value. The remaining properties store the values of external parameters, in this case the name of the component, the input delay and a debug flag.

The constructor provides initial values for all properties, its parameter list defines the set of external parameters of the component. The `delta` method just stores incoming values. The `lambda` method computes the output value, which is given here by the sum of the incoming or stored values. Finally the `ta` method returns the lifetime of the state, which is always given as a two-dimensional vector `[a, b]`, denoting the time $t = a + b\varepsilon$. In this case it is always infinite.

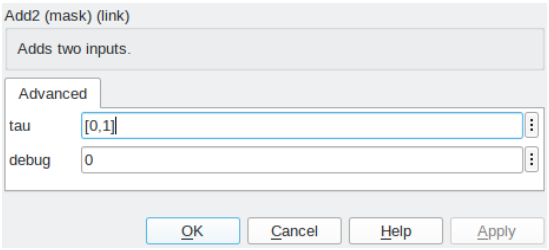

Figure 2: Mask of the `am_add2` atomic.

As a graphical representation of `am_add2` a Simulink subsystem with the name `am_add2` is stored in an NSA-DEVS library using the Simulink editor. Internally it just consists of unconnected input and output ports, which have the names of the ports that are used inside the atomic model. Additionally, it has a mask that defines the order and values of all parameters – except `name`, which is set to the name of the actual component

– and short description and help texts (cf. Fig. 2).

A coupled model, such as the simple example model `demo1` shown in Fig. 3, is defined by a Matlab function that creates all its atomic and – using recursion – its coupled components and all connections. To simplify this tedious and error prone programming task, the user instead builds the model from the Simulink representation by copying the components from the library and connecting them in the standard way. The `toworkspace` models are used here to collect the simulation results. In the example, their parameter `varname` is set to `"input"` and `"result"` respectively.

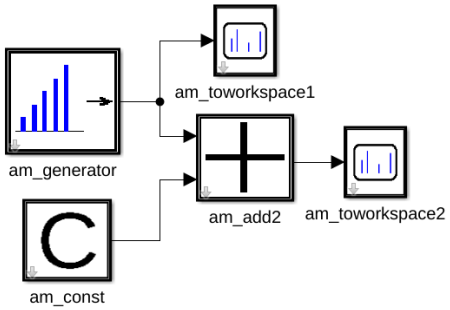

Figure 3: Coupled model `demo1`.

The very simple script shown in Listing 2 can now be used to create and run the model and plot the simulation results.

Listing 2: Run script for model `demo1`.

```
function testDemo()
  tEnd = 6;

  model_generator("demo1");
  out = model_simulator("demo1", tEnd);
  plot_results(out, tEnd);
end
```

From the Simulink representation, the `model_generator` creates the Matlab scripts for all coupled models. Next the `model_simulator` runs the model and collects all results in the struct variable `out`. In the example model it contains the two fields `out.input` and `out.results`, which each have subfields `t` and `y` for the time and result values. They can now be plotted easily with Matlab's standard functions.

A typical result is shown in Fig. 4. The `am_generator` creates increasing numbers, starting at $t = 1$, which are shifted by the constant value 3. The output event at $t = 0$ probably comes unexpected. It is due to the `am_const` atomic that sends its value only once at the beginning. It is then stored in `am_add2` and added to the initial value 0 stored for the other input port. One should bear in mind that the coupled models look like Simulink models, but the inner workings of discrete-event models are still very different.

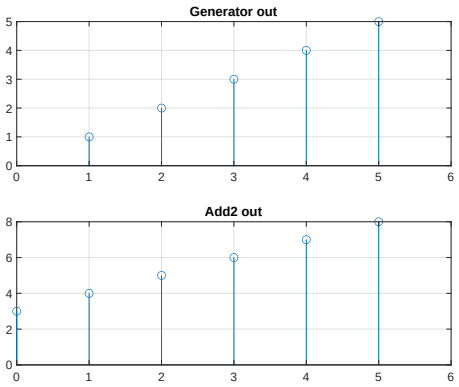

Figure 4: Plot of simulation results for the model `demo1`.

Discrete-event models can easily contain very hard to find errors. To support the debugging process, the toolset supports three different levels, from simple time stamps over debug outputs from individually chosen atomic components to a complete output of internal simulator messages. The last level creates a graphical representation of all messages with a huge amount of information and is usually only useful for simple test models.

4 Example Model

In order to show that the methods and tools presented can be used directly for transaction-based modeling, we will implement a standard textbook example [12]. The model describes a time-shared computer with N terminals, which submit jobs of varying computing time demands. These jobs are processed on a single CPU in time-slices of length q using a round-robin scheduler with a switching time t_{swap}. When a job completes, a new job is created after a waiting time. The waiting and processing times are exponentially distributed random variables with mean values t_W and t_S. After the completion of N_J jobs, the average response time and queue length and the CPU utilization are computed.

A common method to implement such a model uses entities describing the jobs, which contain attributes such as the service time, i. e. the remaining process-

ing time, or the start time. Entity attributes are implemented using struct variables. Four atomics have been created to handle such entity attributes (cf. Fig. 5): `am_adddata` adds a set of fields denoting new attributes to each incoming entity. If the input is not already an entity (i. e. of type struct), an entity is created with an additional attribute that stores the input value. `am_writedata` changes the value of an entity attribute using values from other attributes. The changing function is defined as string parameter describing an arbitrary Matlab command. `am_readdata` outputs the value of an attribute from the input entity and `am_deletedata` deletes a set of attributes.

Figure 5: Atomic models for entity handling.

A few existing atomics have been modified to optionally read an attribute from an incoming entity instead of using a parameter or an input, among them the `server` and `distribute` components. With these atomics, one can create a coupled model for a server with exponentially distributed service times t_S (cf. Fig. 6): The `addTS` component adds an attribute to store the value of t_S, `setTS` sets the value of this attribute using the Matlab command string

```
"out = -" + tS + "*log(rand());"
```

where the variable `tS` is the mean service time, given by a mask parameter. The `am_server` component uses the attribute of incoming entities to set the current service time. Finally, `deleteTS` deletes the attribute for the sake of better encapsulation. Using a different formula for the computation of t_S, which uses several attributes, one can implement complex strategies for calculating the service time.

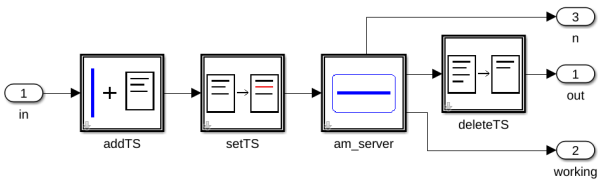

Figure 6: Coupled model of a server with entity-dependent service times.

A standard component in a transaction-based environment is the N-server, which can serve up to N incoming entities, each with its own service time. Its exact behaviour can be quite complicated and often is not transparent to the user of a commercial program. The NSA-DEVS description eliminates all ambiguities that arise e. g. with several incoming and outgoing entities at the same time. The diagram in Fig. 7 describes the basic behaviour of the `am_nserver` atomic, where monitoring compliance with the maximum server capacity has been omitted for better readability. Among its properties are a list E of entities in the server, a corresponding list σ of remaining service times and a list *qOut* of outgoing entities. An interesting difference to the simple server is the possibility that several entities can be ready at the same time. To handle this, the N-server moves all finished entities to *qOut*, changes to the state *emitting* and outputs them with a time delay of t_D. According to the rules stated above, t_D is predefined as 2ε, but may need to be enlarged in special applications. All details defining the exact behaviour can be found in the open source code of `am_nserver.m` [11].

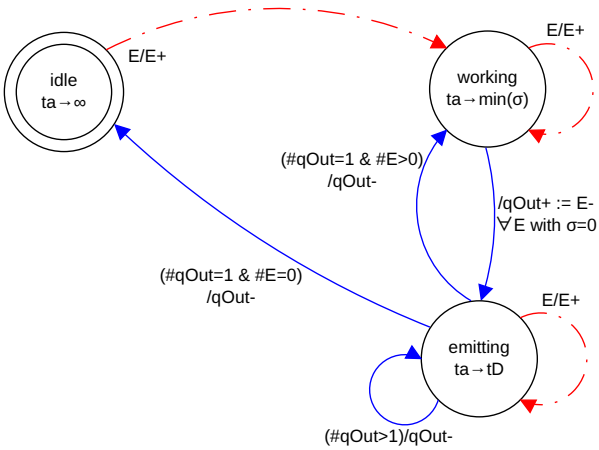

Figure 7: State diagram of the N-server component.

→	state transition due to internal event
--→	state transition due to external event
ta	lifetime of the state
E	external event (entity at input)
E+	insert entity in list E
E-	remove entity from list E
#E	number of entities in list E

With these atomics the three coupled models `Terminals`, `CPU` and the complete model

timeShared can be assembled easily. The Terminals model (cf. Fig. 8) starts with a generator initialJobs that creates N entities at time 0 (more precisely at times $n \cdot t_D$) with consecutive IDs. They get the attributes startTime, outPort and remainingServiceTime, which is set to the individual service times. An N-server implements the individual waiting times. The startTime is set to the current simulation time, using the utility function *get_time*(), and the entities proceed to the CPU. When a job is fully processed by the CPU, a corresponding ID is sent as external event to the input of coupled model Terminals, where it is promoted to a new job by the atomic model constAdder.

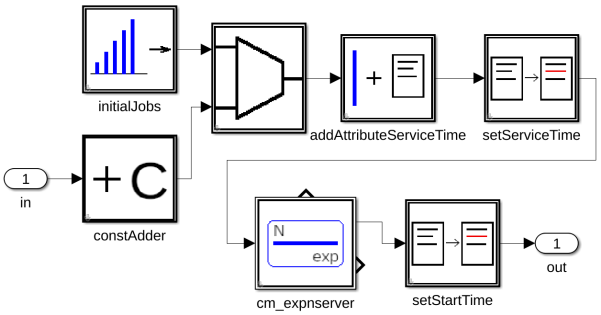

Figure 8: Coupled model Terminals.

The CPU model (cf. Fig. 9) starts with a queue for the waiting jobs, followed by a server representing the CPU proper. Its service time is set to $q + t_{swap}$ or less, if the job is almost ready. After processing, the entity attributes are updated: The time slice is subtracted from the remainingServiceTime and outPort is set to 1, if the job is ready, or 2 otherwise. The internal feedback from the server to the queue signals to the queue whether the server is available.

The complete model timeShared (cf. Fig. 10) shows the loop around the CPU that jobs are sent, until they have got their complete service time. The distribute atomic uses the outPort attribute to route finished jobs through the upper port. Finally the response time is computed by subtracting the value of attribute startTime from the current time and the entity is terminated. The terminator counts the outgoing jobs and sends this value back to the coupled model Terminals, where new jobs are created. A stop atomic halts the simulation, after N_J jobs have been processed.

Adding a few output blocks, one can gather enough

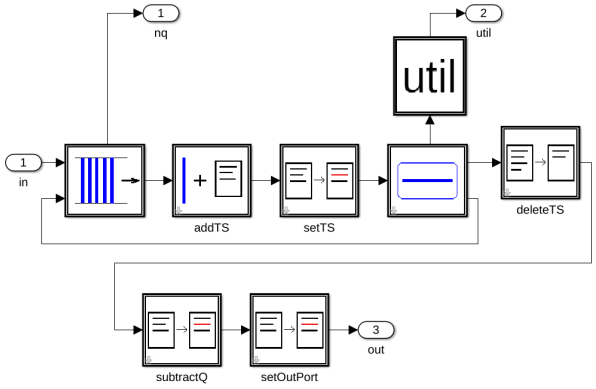

Figure 9: Coupled model CPU.

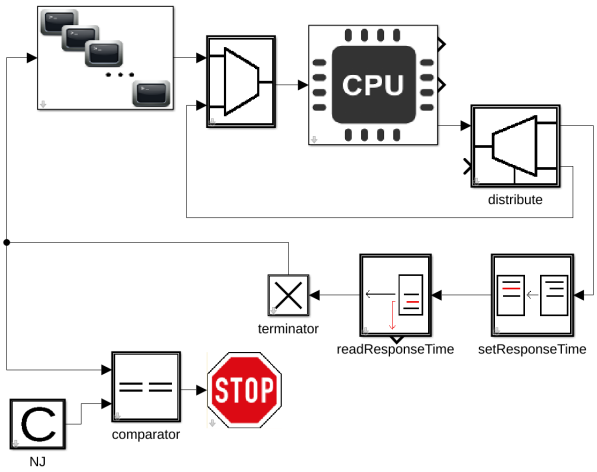

Figure 10: Complete coupled model timeShared.

information to get a complete picture of the model behaviour. The information displayed in Fig. 11 is sufficient to read off the waiting and service times of the jobs and follow each job individually through the model. This allows to thoroughly check the system behaviour. Especially intuitive is the plot of the remaining time after the CPU, which nicely displays the loops of the jobs around the CPU and the interaction of several jobs.

The component library contains the atomic am_getmean that calculates the running mean value of its input values, and the atomic am_utilization for the computation of the CPU utilization. Adding them to the timeShared model, one easily gets all requested statistical data. A typical example with $N = 20$ and $N_J = 1000$ is shown in Fig. 12. The results are similar to those of the SimEvents version of this model that had been used in [2], and consistent with

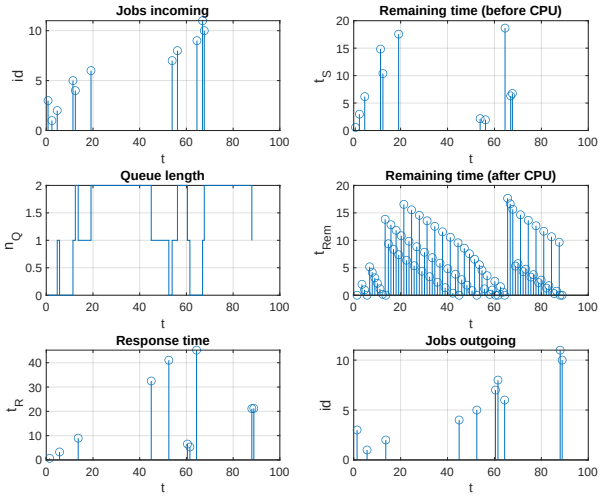

Figure 11: Detailed results of `timeShared`.

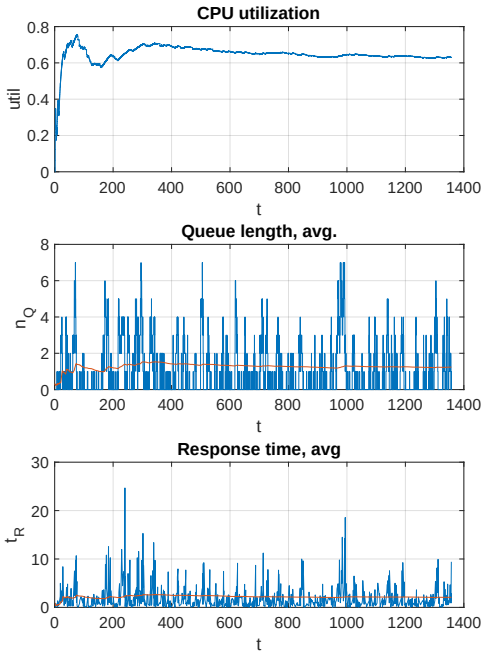

Figure 12: Statistical results of `timeShared`.

the results shown in [12].

5 Conclusions

The implementation of the `timeShared` model has once again shown that NSA-DEVS is a conveniant basis for component-based modeling of discrete-event systems with a sound mathematical foundation. Especially, not one of the delay parameters had to be changed from its default value. This should be the typical case for systems with stochastical elements, where the probability of concurrent events is quite small.

Furthermore, the toolset available freely from [11] has proven its versatility: After finding and implementing a suitable set of atomics for handling entities with variable attributes and adding some standard atomics to the library, the construction of a transaction-oriented application proceeded by standard graphical methods. We invite all modelers interested in discrete-event modeling to try out these tools, ask for enhancements or even provide useful new atomics to the library.

During the design of the free NSA-DEVS simulator and the library, the focus has mainly been on correctness and simplicity. This shows, when measuring its performance: The simulation of `timeShared` with $N = 40$ and $N_J = 1000$ has a runtime of around 45 seconds on a recent PC platform, while the corresponding SimEvents version needs less than 2 s. Of course, the comparison is not quite fair, since SimEvents compiles the Matlab code before a run. Nevertheless, there is definitely large potential for improvement by trading elegance of construction for runtime performance and by generally reducing the number of messages sent.

With the presented tools and methods, one can finally tackle the questions that have been raised in [2]:

> What are the shortcomings of current implementations? Which concepts or components are missing? How could a reasonable set of components be defined?

The atomic models for entity handling and the N-server introduced above are practical examples, how to precisely define fundamental building blocks due to their underlying NSA-DEVS based formulation. Another step along these lines would be the introduction of versatile queue models that are capable of supporting all the applications denoted in the ARGESIM benchmark C22 [13]. To cite [2] once again:

> For the advancement of transaction-based modeling it is vital that it is based on a thorough theoretical analysis to reveal the fundamental abstractions and basic components that are necessary.

This is true more generally for all discrete-event based modeling. NSA-DEVS and corresponding tools seem to be a promising path to promote such a program.

References

[1] Dias LMS, Vieira AAC, Pereira GAB, Oliveira JA. Discrete simulation software ranking—A top list of the worldwide most popular and used tools. In: *2016 Winter Simulation Conference (WSC)*. IEEE. 2016; pp. 1060–1071.

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

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

[4] 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.

[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] 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.

[8] Jammer D, Junglas P, Pawletta T, Pawletta S. A Simulator for NSA-DEVS in Matlab. *SNE Simulation Notes Europe*. 2023;33(4):141–148. doi: 10.11128/sne.33.sw.10661.

[9] Jammer D, Junglas P, Pawletta T, Pawletta S. Implementing Standard Examples with NSA-DEVS. *SNE Simulation Notes Europe*. 2022;32(4):195–202. doi: 10.11128/sne.32.tn.10623.

[10] Jammer D, Junglas P, Pawletta T, Pawletta S. Modeling and Simulation of a Real-world Application using NSA-DEVS. *SNE Simulation Notes Europe*. 2023; 33(4):149–156. doi: 10.11128/sne.33.tn.10652.

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

[12] Law AM. *Simulation Modeling and Analysis*. New York: McGraw-Hill, 5th ed. 2014.

[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.