Visual NSA-DEVS Modeling Using an Adapted DEVS Diagram

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

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

[*]*thorsten.pawletta@hs-wismar.de*

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

Abstract. NSA-DEVS (Non-Standard Analysis Discrete Event System Specification) is an advancement of the DEVS formalism for modeling and simulating discrete-event and hybrid systems. DEVS supports modular-hierarchical modeling and clearly separates model from simulator. The primary objective of NSA-DEVS is to simplify the modeling of components with Mealy behavior while maintaining a simple simulator structure and the exact set-theoretic definitions of DEVS. With the implementation of a modeling and simulation environment, a comprehensive library of generic components, and real-world applications, the use of NSA-DEVS has been evaluated. However, the implementation of complex applications revealed that set-theoretic modeling should be complemented by visual techniques to facilitate system design and documentation. This paper explores how a visual representation equivalent to NSA-DEVS can be developed based on the known DEVS diagram and Harel's Statecharts.

Introduction

The set-theoretic DEVS formalism [1, 2] and its popular version PDEVS [3] are widely used for studying discrete event systems. However, Preyser et al. [4] have shown that it is difficult to define some basic reusable components, especially when they have Mealy behaviour. They introduced a revised version called RPDEVS [5]. However, RPDEVS struggles with handling chains of concurrent events [6].

Junglas argues in [7] that mathematical problems often are due to oversimplification in modeling. The macroscopic abstraction of phenomena by events results in simultaneous events, while in the underlying microscopic dynamics the corresponding processes are separated by small time delays. NSA-DEVS (*Non-Standard Analysis DEVS*) [8, 9] uses hyperreal numbers to represent infinitesimal time delays, thereby solving the problem of simultaneous event cascades. In contrast to PDEVS, Mealy behavior can be modeled without transitory states, which simplifies the specification of generally reusable model components.

A modeling and simulation (M&S) environment based on NSA-DEVS has been developed using MATLAB and Simulink's graphical editor [10, 11]. A few things became clear when implementing a real application [12], which consists of several layers and hierarchies with a large number of model components. First, designing and testing atomic model components specified in purely textual form using set theory requires a lot of time when the specification involves a large number of events and states. Second, using the visual support of the Simulink editor has proven to be extremely beneficial for modeling networked systems and hierarchical structures. This insight is not new, and several approaches to represent DEVS models graphically have already been developed. Based on preliminary work, Song and Kim designed the *Revised DEVS Diagram* [13] for Classic DEVS. In the authors' opinion, the Revised DEVS Diagram provides the best compliant representation of atomic DEVS models, even taking into account newer UML-based approaches, such as in [14].

The paper explores how the Revised DEVS Diagram can be adapted to depict NSA-DEVS models. It starts by giving a brief overview of the NSA-DEVS formalism and the Revised DEVS Diagram, and then examines the key differences between modeling with Classic DEVS and NSA-DEVS. Next, it discusses the basics of the modified diagram proposed here. Finally, a case study is used to demonstrate how NSA-DEVS models can be specified using the customized diagram.

1 Background

This section gives a short review of the NSA-DEVS formalism and the Revised DEVS Diagram for Classic DEVS.

1.1 The NSA-DEVS Formalism

Analogous to the DEVS formalism, NSA-DEVS defines a set-theoretic model specification and an operational semantics, called abstract simulator, to execute the specification. As with all DEVS formalisms, a distinction is made in the model specification between *atomic* and *coupled* models. An abstract simulator is defined for each model type, which is referred to as the *simulator* for atomic models and the *coordinator* for coupled models. During the execution phase, an abstract simulator is assigned to each model. A *root coordinator* manages the hierarchy of abstract simulators.

An *atomic model* describes the dynamic behavior of an arbitrarily complex system and is defined 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,
$ta : S \to {}^*\mathbb{R}^{>0}_{\text{fin}} \cup \{\omega\}$	time advance function,
$\delta : Q \times X^+ \to S$	state transition function,
$\lambda : Q \times X^+ \to Y^+$	output function.

The input and output sets X, Y contain *pairs of ports and values*, where ports are given by names.

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

The sets X^+ , Y^+ consist of sets of pairs from X, Y to describe the simultaneous appearance of input or output values at different ports. Simultaneous input events at the same port are not allowed. The definition of the transition function δ and the output function λ contain the set $Q = \{(s,e) | s \in S, 0 < e \leq ta(s)\}$ that combines a state and the elapsed time e since the last transition.

According to the model specification, the *simulator* must process external, internal and confluent events. All three event types lead to a call of λ followed by a change to a new state according to δ. The time advance function ta may be infinitesimal or infinite (using $\omega := 1/\varepsilon$, with ε as infinitesimal value), but it is always > 0, thereby excluding proper transitory states. The delay time τ between the arrival of a set of inputs and the call of λ and δ is generally an infinitesimal, often given by a default value $\tau_{def} = \varepsilon$. Generally, all hyperreals used in models and the simulator have the form $a + b\varepsilon$ for real a,b. A more detailed description of the operational semantics of the simulator can be found in [10] and the full algorithm in [8].

A *coupled* NSA-DEVS model is defined as in *PDEVS with ports* [2]. 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. Outputs are transported as usual and a coupled model has no additional input delays. Since the operational semantics of the *coordinator* and the *root coordinator* are not essential for understanding a DEVS diagram for atomic models, they are not discussed in detail and reference is made to [8].

1.2 The Revised Diagram for Classic DEVS

To represent the dynamics of atomic models visually, Song and Kim formulated the Revised DEVS Diagram [13]. It is based on a series of preliminary works, whose origin probably goes back to Prähofer [15]. Based on the idea of Harel's *Statecharts* [16], Prähofer introduces a DEVS diagram which already contains descriptive elements adapted for atomic models.

The Revised DEVS Diagram in [13] is specifically designed for *Classic DEVS with Ports* [2]. According to the DEVS formalism, two types of diagrams are distinguished, one for atomic and another for coupled models. The latter is similar to the UML *Composition Structure Diagram*, which is extended by a field for specifying the *select function* specific to Classic DEVS.

The *diagram for atomic models* is based on a structuring of events and states. Events are grouped by categories into classes. Thus, events have a type and a value and are called a *message*. The set of *Ports* defines the interface of an atomic model. A port can only process messages of the same type, and only one message at a time, and is defined by *portName:messageType*. While an input port can only be connected to one source, an output port can be connected to several targets. The following syntax applies to input and output messages: *inportName?message* and *outportName!message*.

States are structured in *phases*, as in Harel's Statecharts [16]. A *phase* is a representative value of a set of states which produce the same output event and/or have the same time advance (remaining lifetime) at the states. A phase is represented by a rounded box with the value of the phase variable and its remaining lifetime $@T$ until the next internal event. In addition to the phase variable, there is the set of *ordinary state* variables S_v, so that the current system state $s \in S$ results from the values of the pairs $(phase, s_v)$ with $s_v \in S_v$.

Phase transitions are triggered by external or internal events. Transitions caused by external events are represented by a *solid* line, transitions caused by internal events by a *dashed* line. In Classic DEVS, there are no simultaneous internal and external (*confluent*) events in atomic models. These are resolved at the level of the coupled models using a *select function*. Furthermore, there is no Mealy behavior, i.e. outputs can only be generated by internal events. Both types of events lead to state changes, i.e. to a phase transition and/or to changes in the ordinary state variables, and to a recalculation of the lifetime $@T$ of the current phase. This DEVS-based dynamics can be described with a triple (*event, guard, action*) at the phase transitions using the following notation:

$$inportName?message@[guard]/\{actions\}$$
$$outportName!message@[guard]/\{actions\}$$

The first notation defines state changes triggered by external events and the second one outputs and state changes caused by internal events. The $@[guard]$ defines a logical expression that depends on the *message* or the ordinary state variables. A phase transition, the execution of actions, and the sending of output events will only become active if the guard is true.

Figure 1 shows the specification of the dynamics of a single server as DEVS diagram. The server receives entities E of type Ⓔ to be served via the port *in* :Ⓔ and sends processed ones via the port *out* :Ⓔ. Furthermore, it sends the current status via the port *working* : $\{0,1\}$, where 0 stands for the IDLE phase and 1 for BUSY. Additionally, the initial phase is denoted with a bold box. The types and initial values of the phase variable and the ordinary state variables are defined in the lower box. The transition annotation $in?E/\{job = E, \sigma = 0\}$ specifies the transition from IDLE to BUSY due to a message E on the port *in* without defining a guard. The two actions describe the allocation of the server with the entity $(job = E)$ and the scheduling of an immediately internal event $(\sigma = 0)$ in order to send the server's new status as a message on the port *working* (no Mealy behavior!). The behavior resulting from the internal event

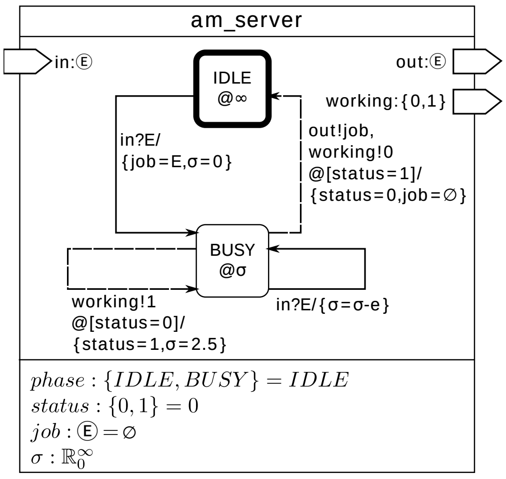


Figure 1: DEVS diagramm of a single server with discrete-event output of the server status.

specifies the dashed phase transition with the annotation $working!1@[status = 0]/\{status = 1, \sigma = 2.5\}$. However, the guard defines the condition for activating exactly this transition and the action $\sigma = 2.5$ sets the lifetime of the phase (fixed service time). Entities arriving in BUSY $(in?E/\{\sigma = \sigma - e\})$ are discarded, but the remaining time advance is recalculated using the internal variable e for the elapsed time since the last state change.

The transition from BUSY to IDLE with the annotation $out!job, working!0@[status = 1]/\{status = 0, job = \emptyset\}$ describes the output of the processed entity (job) on the port *out* when the service time has expired, and the output of the new server status at port *working*. Moreover, the ordinary state variables *status* and *job* are updated. The example shows how the guards control which of the two transitions takes place when an internal event occurs.

In addition to the techniques described, the Revised DEVS Diagram supports other mechanisms such as hierarchical phase compositions and parallel phases [13].

2 A Diagram for NSA-DEVS

This section begins with a short discussion of the key differences in modeling with Classic DEVS and NSA-DEVS. Subsequently, we introduce some adapted and new modeling elements for a DEVS diagram to represent atomic NSA-DEVS models.

2.1 Key Differences in Modeling with Classic DEVS and NSA-DEVS

The roots of NSA-DEVS are based on PDEVS [3] and RPDEVS [4]. Similar to these formalisms, concurrent events are not resolved at the level of coupled models. In contrast to Classic DEVS, there exists no *Select function* at the level of coupled models.

In NSA-DEVS, the handling of simultaneous internal and external events, termed *confluent events*, must be specified at the atomic model level. However, as outlined in Section 1.1, the approach differs significantly from that of PDEVS.

Analogous to Classic DEVS, NSA-DEVS prohibits multiple simultaneous external events on an input port. However, it does support simultaneous input messages on different ports.

With direct support for Mealy behavior in NSA-DEVS, the segmentation of state changes into event type-specific transition functions has been merged into a single state transition function δ. Moreover, all three event types (external, internal and confluent) prompt a call to the output function λ, followed by a transition to a new state using δ, and the re-scheduling of the next internal event via the ta function.

As detailed in Section 1.1, NSA-DEVS excludes proper transitory states with a lifetime of zero. Thus, internal events are always scheduled by the function ta with a value greater than zero, even if infinitesimal.

2.2 Adapted and New Modeling Elements

The differences in modeling and processing an atomic model between Classic DEVS and NSA-DEVS require the adaption and incorporation of some new modeling elements into the DEVS diagram. Figure 2 provides a summary of the diagram elements for NSA-DEVS.

The basic structure of the DEVS diagram shown in Figure 1 with the division into three parts: (i) name field, (ii) phase diagram, and (iii) variable definition, has been retained. The representation of ports and phases is also unchanged. The case study in Section 3 shows that it is sometimes useful to split the phase diagram into sub-diagrams. In order to always be able to clearly identify the initial phase, the initial transition has been adopted from Harel's statecharts. In NSA-DEVS, phase transitions can occur due to external, internal and confluent events. Although there are formally three event types in NSA-DEVS, only two line types are used for phase transitions in the adapted DEVS diagram. Phase transitions due to internal and confluent events are displayed with a dot-dash line. Due to the extended annotations at the phase transitions, explicitly internal events can still be clearly distinguished from confluent events in the diagram. In addition, the element *condition junction with priorities* already introduced by Freymann [17] for DEVS diagrams was adopted. This allows phase transitions to be summarized and case distinctions to be defined, which improves the clarity of the diagram.

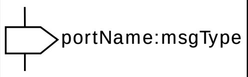
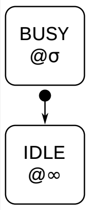
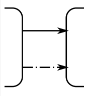
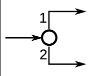
Item	Description
	**Input** or **output port** with name and permitted message type
	**State phase** with value of the phase variable and it's remaining lifetime until the next internal event; *Initial state phase*
	**Phase transition** due to an external (⟶) or internal/confluent (–·→) event
	**Condition junction** and **Priorities** for modeling case distinctions in phase transitions
inportName?msg	**Phase transition annotations**: *External event* as message with *type* and value at input port *Name*
inports?	*Read all input ports*
@[guard]	Definition of a *transition guard* with guard result true or false
/{outportName!msg,..., stateVar1=value,...}	*Transition actions*, defining output messages with: message *msg* at output port *Name*, and updates of ordinary state vaiables
%	Comment

Figure 2: Summary of modeling elements of the NSA-DEVS Diagram for atomic models.

Due to the different semantics of NSA-DEVS and Classic DEVS, the annotations at phase transitions have been partially changed. The annotation of an *external event* on a single input port is unchanged with *inportName*?*msg*. However, NSA-DEVS also supports simultaneous events on different input ports. For this purpose, the notation *inports*? is introduced. It specifies reading messages from all input ports simultaneously where ports without a message are set to *empty message* ($\emptyset$). The rules for defining *guards* have not been changed. The transition actions ($/\{actions\}$) specification has been extended by defining *output messages* and *updates of ordinary state variables* as actions. The change arises from the semantics of NSA-DEVS: Unlike Classic DEVS, Mealy behavior is modeled without transitory states, meaning that external events can immediately trigger output events. The general relocation of output messages to the action part ensures uniform indication of cause and effect for all event types.

Comments marked with % can be inserted in all three subfields of the DEVS diagram.

3 Case Study: Simple Queuing System

This section demonstrates the specification of a simple queuing system (SQS) using generic atomic models, usually organized in a model base. Figure 3 shows the structure of the SQS as a coupled model, developed within the NSA-DEVS M&S environment [11]. The associated model base provides an atomic model for recording statistical variables provided by ports such as *nq* (number of entities in queue) and *ns* (number of occupied servers).

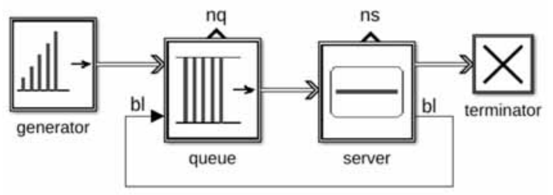


Figure 3: Coupled model `SimpleQueuingSystem`.

Figure 4 shows the DEVS diagram specifying the *generator*. The top box defines the name of the atomic model class, and the configurable variables, such as the interarrival time *tG* of the entities. The bottom box defines all variables. The state variable *E* is of type *Entity*, which defines a field variable *id*. This is initialized with the start ID, set by the configurable variable *n*0. The phase diagram in the middle shows the two phases with the initial phase PROD. After a time advance of *tG* time units, an internal event @*tG* is triggered. This results in two actions: (i) sending an output message with the current entity *E* at port *out* and (ii) incrementing the state variable *E*.*id*. The guard $@[E.id - n0 < nG]$ decides whether a phase transition occurs back to PROD or to FINISH. In FINISH, the generator becomes inactive due to the defined time advance $@\infty$. The variable τ is a default variable and should be defined for each atomic model according to the NSA-DEVS specification. But it is irrelevant for the generator and could be omitted here as it does not define input ports.

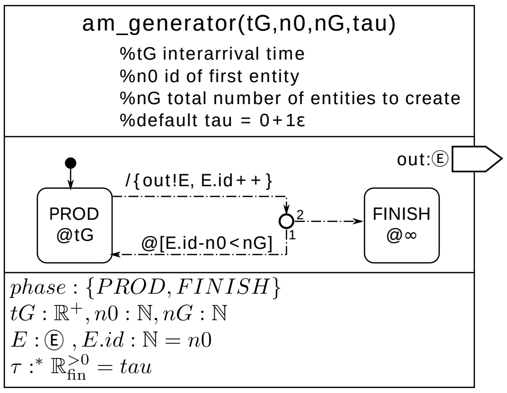

am_generator(tG,n0,nG,tau)
%tG interarrival time
%n0 id of first entity
%nG total number of entities to create
%default tau = 0+1ε
out:Ⓔ
/{out!E, E.id++}
PROD @tG
2
1
FINISH @∞
@[E.id-n0<nG]
$phase : \{PROD, FINISH\}$
$tG : \mathbb{R}^+, n0 : \mathbb{N}, nG : \mathbb{N}$
$E :$ Ⓔ $, E.id : \mathbb{N} = n0$
$\tau :^* \mathbb{R}^{\geq 0}_{\text{fin}} = tau$

Figure 4: Atomic model class `am_generator`.

Figure 5 shows the DEVS diagram specifying the atomic model class for the *queue*. The queue works according to the push principle, i.e. it releases entities until it receives a blocking event. The advantage over a pull queue policy is a reduction in the number of events when the queue stores entities for an N-server, i.e. a server with a capacity greater than one.

The diagram defines an input and output port for the entities *E*, an input port $bl : \{0, 1\}$ for blocking and unblocking events and a port for outputting the current number of entities in the queue ($nq : \mathbb{N}$). The *phase* variable defines four phases, and the three ordinary state variables are defined: (i) a list *q* for storing entities *E*, (ii) the input delay time τ and (iii) the internal delay time τ_D. The comment in the top box notes that the configurable variables are set to their default values, making $\tau_D > \tau$. This means that simultaneous input events are processed with an infinitesimal time interval before internal events. The type *List* for storing entities defines typical list operations, which are noted in the diagram as follows:

- #q . . . number of elements in queue
- q+(E) . . . insert element E into queue
- q–(1) . . . remove first element from queue
- q(1) . . . read first element from queue without delete

The phase diagram is divided into sub-diagrams, which are separated from each other by a solid line. The *first* diagram shows the four phases with the initial phase *EmptyFree* and all possible transitions from *EmptyFree* (marked with a bold box) to other phases. The *inports*? transition annotation indicates that this transition occurs, if a message is received at one of the input ports (in, bl) or at both ports simultaneously. If the first guard $@[in = E]$ is not satisfied, only a transition to *EmptyBlocked* occurs. Otherwise, an output message with the number of entities in the queue is sent to port *nq* and the new entity E is inserted into list q. The second guard $@[bl = 1]$ checks whether a blocking event is present at the same time and regulates the transition to *QueuingBlocked* or to *QueuingFree*. A transition to phase *QueuingFree* always involves the scheduling of an internal event with an infinitesimal time advance τ_D.

The *second* diagram describes all possible transitions originating from *EmptyBlocked*. This phase is also only left by an external message. If only an unblocked message $bl = 0$ is received, a switch to *EmptyFree* occurs. In the case of an incoming entity, the guard $@[in = E]$ is satisfied and the same actions are performed as described in the first diagram. If an unblocked message $bl = 0$ is also present, then the guard $@[bl = 0]$ is satisfied and a transition to *QueuingFree* occurs, otherwise to *QueingBlocked*.

The *third* diagram specifies the transitions that originate from *QueuingBlocked*. In case of an incoming entity $in = E$, the same actions are performed as in the two previous diagrams. The second guard $@[bl = 0]$ checks for an unblocking event and decides whether to switch to *QueuingFree* or to return to *QueuingBlocked*.

The most complex case are the transitions originating from *QueingFree*. For a better overview, the possible transitions are shown in two sub-diagrams. As mentioned above, *QueuingFree* schedules an internal event with infinitesimal time advance τ_D to send an entity stored in the list q on port *out*. However, the system can receive a blocking message $bl = 1$ at the same time.

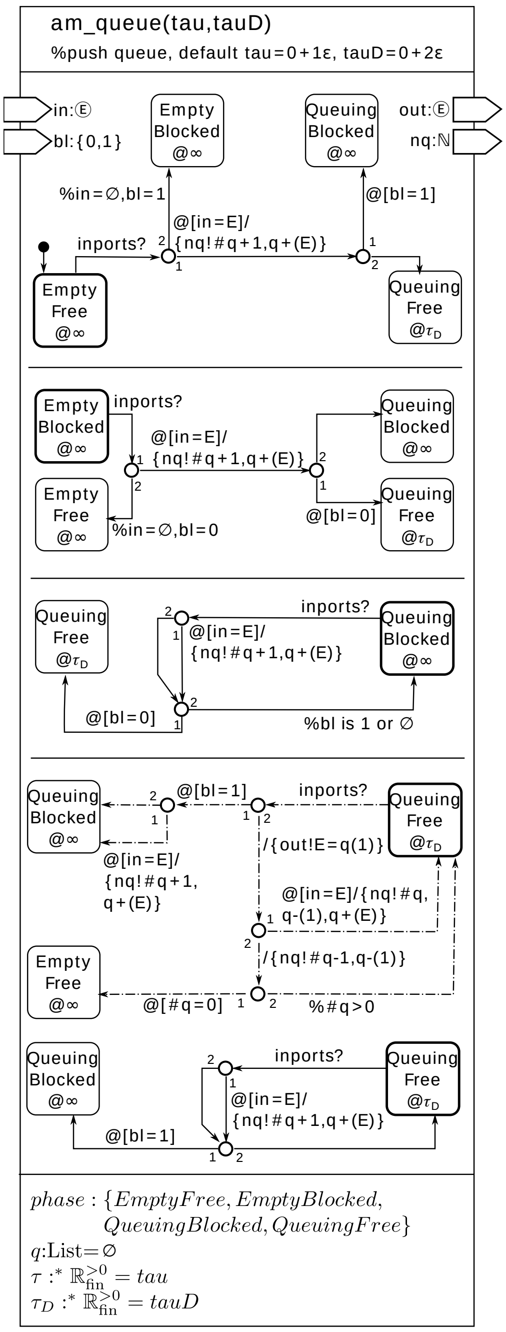


Figure 5: Atomic model class `am_queue`.

The simultaneous occurrence of both events is resolved by the hyperreal state variables τ and τ_D. The setting $\tau_D > \tau$ ensures that no entity is sent when a simultaneous blocking event is received. The specification of this confluent event situation is shown in the first sub-diagram of phase *QueingFree*.

The *inports*? transition annotation indicates that all input ports are checked for messages before the internal event is triggered. If a blocking message $bl = 1$ is present ($@[bl = 1]$ is satisfied), a transition to *QueingBlocked* occurs. The guard $@[in = E]$ checks whether an entity E is present at the input port *in* at the same time. If so, the current queue length is sent as an output message to port *nq* and the new E is inserted into the list q. If there is no message at port *bl* or $bl = 0$ the transition action $/\{out!E = q(1)\}$ is executed. The first entity is read from the list q and sent to port *out*. It is then checked whether an entity is present at the input port *in* at the same time. If so, an output event with the queue length is sent to port *nq*, the list q is updated (delete sent E and insert arrived E) and the system returns to *QueingFree* to output another entity. If no new entity has arrived, the current queue length is sent to port *nq* and the list q is updated (delete sent E). The last guard $@[\#q = 0]$ checks whether the list q is empty and regulates the transition to *EmptyFree* or the return to *QueuingFree*.

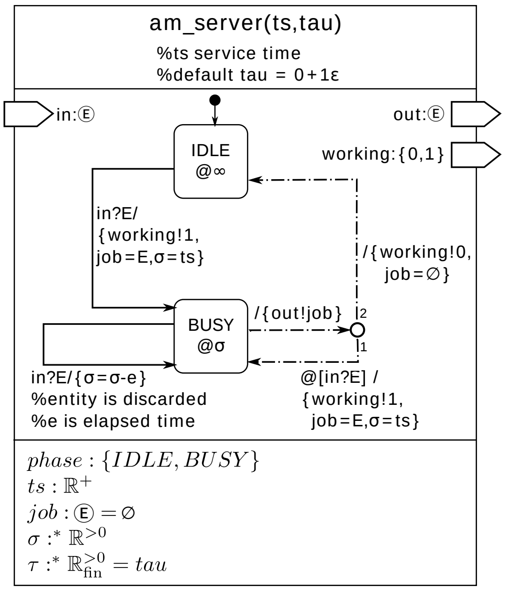


Figure 6: Atomic model class `am_server`.

Even though the next internal event is scheduled in *QueingFree* with infinitesimal time advance τ_D, external input messages may be present during this time span. The specification of this case is shown in the second sub-diagram for *QueingFree*. All input ports are checked by *inports*? for external messages. If the first guard $@[in = E]$ is satisfied, the new queue length is sent as output message and the new entity is inserted into the queue list q. The second guard $@[bl = 1]$ checks if a blocking message has been received and decides whether a transition to *QueingBlocked* or a return to *QueingFree* takes place.

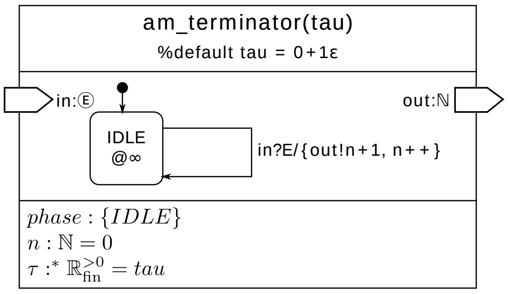


Figure 7: Atomic model class `am_terminator`.

Although the atomic model class `am_queue` supports N-server, only the specification of a *single server* is discussed below, shown as a DEVS diagram in Figure 6. The specification does not take into account the output port *ns* shown in Figure 3, which outputs the number of occupied servers, as this information is not of interest for a single server. The server is configurable with a service time *ts*. The infinitesimal input delay time τ can be set to the default value. The initial phase is IDLE. When an entity E arrives at port *in*, then: (i) a server occupancy event is sent to the port *working*, (ii) the entity and the service time are stored in ordinary state variables, and (iii) the server switches to BUSY, where an internal event $@\sigma$ is scheduled according to the service time. The server checks for external events in the BUSY phase, but does not register incoming entities (you could count them). The timeless return to BUSY, however, requires a rescheduling of the internal event using the action $\sigma = \sigma - e$. The internal variable e stores the elapsed time since the last event. If the internal event is triggered after the service time has expired, an external event may occur at the same time (confluent event). In any case, an output message is sent with the served entity *job* via port *out*. The guard $@[in?E]$ checks if an external event occurs at the same time. In this case a server busy output message ($working = 1$) is sent, the new entity E is scheduled as the current job in service, and the server returns to BUSY. Otherwise, a transition to IDLE occurs, a server is free output message ($working = 0$) is sent, and the state variable *job* is updated.

Figure 7 shows the specification of the *terminator*. In addition to the model structure in Figure 3, the model class `am_terminator` defines an output port $out : \mathbb{N}$. It counts the number of incoming entities and sends changes of the counter as an output message to port *out*.

4 Conclusion

At first glance, the specification of atomic systems with a DEVS diagram may seem confusing, especially the complexity of the notations at phase transitions. However, it must be emphasized that the adjusted DEVS diagram can represent the complete dynamic specification of atomic models and can be converted one-to-one into program code for the NSA-DEVS simulation environment, where atomic NSA-DEVS models are implemented as a MATLAB class and organized in a model base.

The developers' experience is that the specification with DEVS diagrams is very helpful after a short training period, especially in the system design and system documentation phase. This becomes especially clear, when implementing complex atomics like an N-server [10]. Its graphical description has also been very helpful during the implementation and debugging phases, and documents the complex code in a clear-cut way, that highlights the basic underlying ideas.

References

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

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] Junglas P. Mathematical Problems due to Oversimplication. In: *4th Northern-Light Symposium on Mathematical Education in Engineering*. Hamburg-Bergedorf. 2024; pp. 27–41.

[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] Junglas P, Jammer D, Pawletta T, Pawletta S. Using component-based discrete-event modeling with NSA-DEVS – an invitation. In: *Proc. of ASIM 2024 – 27. Symposium Simulationstechnik*. Munich, Germany. 2024; .

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

[12] 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.

[13] Song HS, Kim TG. DEVS Diagram Revised: A Structured Approach for DEVS Modeling. In: *Proc. Eur. Simulation Conf.* Eurosis, Belgium. 2010; pp. 94–101.

[14] Özmen Ö, Nutaro J. Activity Diagrams for DEVS models: A Case Study Modeling Health Care Behavior (WIP). In: *TMS/DEVS, SCS Spring Simulation Symposium*. Alexandria. 2015; .

[15] Prähofer H, Pree D. Visual Modeling of DEVS-Based Multiformalism Systems Based on Higraphs. In: *Proc. of the Winter Simulation Conference*. Los Angeles, CA USA. 1993; pp. 595–603.

[16] Harel D. STATECHARTS: A Visual Formalism for Complex Systems. *Science of Computer Programming*. 1987;pp. 231–274.

[17] Freymann B. Task-Based Multi-Robot Controls Based on the SBC Framework and DEVS [dissertation]. Ph.D. thesis, Technical Univ. Clausthal in coop. with Univ. of Appl. Sciences Wismar, Wismar, Germany. 2022. doi: 10.11128/fbs.40.