NSA-DEVSforMatlab

A Matlab-based Tool for Discrete-Event Simulation

A Tutorial Introduction

Peter Junglas

2025-09-25

Contents

1 Introduction

NSA-DEVSforMatlab ("ND4M") is a modeling and simulation tool for discrete-event based simulation. It is based on Matlab and combines the ease of use of graphical tools like Simevents or Arena with the mathematical preciseness of DEVS. For this purpose it uses the DEVS variant NSA-DEVS [1]. It comes with a growing library of components that can be used to build a model graphically using the Simulink editor. A new component can be built either by combining existing components in a hierarchy of subsystems ("coupled model") or by defining it directly as a Matlab class that implements the NSA-DEVS formalism ("atomic model").

The first chapter describes the installation and a short test, using one of the example models coming with ND4M. The following chapters will use simple examples to show, how to build models graphically and how to create own atomic models from scratch. Working knowledge of Matlab and the basics of the Simulink editor are required, but no prior knowledge of DEVS or NSA-DEVS.

All models and scripts that are presented in the following, can be found in ND4M's `Example` directory. It is advisable to rebuild at least the first model from scratch.

Additional information, especially about the inner workings of ND4M and the definitions of NSA-DEVS, is provided in paragraphs marked with a magnifying glass. They can be safely ignored in a first reading.

2 Installation

If you haven't done so already, download NSA-DEVSforMatlab from its Github repository https://github.com/davidjammer/NSA-DEVSforMATLAB and unpack it. You can rename it and/or move it to an arbitrary directory. In the following this directory will be called `NSADEVSHOME`. Furthermore create a directory for your own models, it will be called `MYMODELS`. For concreteness we will use the following structure

```
NSADEVSHOME=/home/testi/nsa-devs/NSA-DEVS
MYMODELS=/home/testi/nsa-devs/mymodels
```

Then, create a subdirectory `tutorial` of `MYMODELS`, which will contain all tutorial examples. Now start Matlab and extend the path to include the NSA-DEVS files:

```
NSADEVSHOME="/home/testi/nsa-devs/NSA-DEVS";
addpath(genpath(NSADEVSHOME+"/Modelbase"));
addpath(genpath(NSADEVSHOME+"/Modelgenerator"));
addpath(genpath(NSADEVSHOME+"/Simulator"));
addpath(genpath(NSADEVSHOME+"/Utilities"));
```

For later sessions save these lines in your `startup.m`.

Test the installation by running one of the example models: First copy the paper examples directory into your directory `MYMODELS`. Open the example model `compswitch.slx` with Simulink to see the structure of the compswitch example. Finally, run the command `runCompswitch`, which should generate the plot shown in Fig. 2.1.

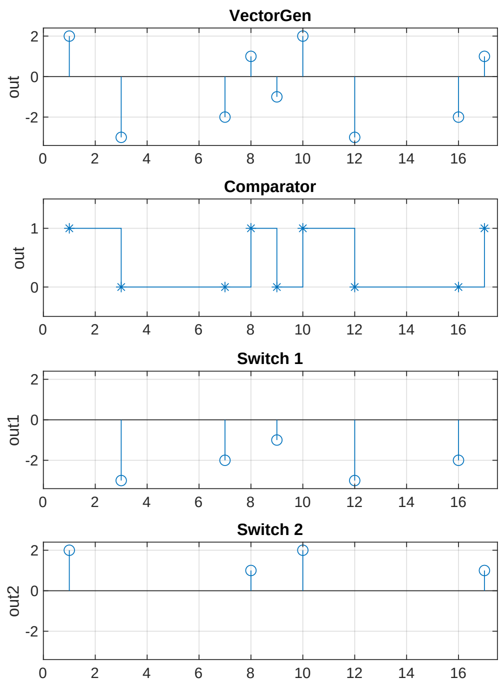

Figure 2.1: Plot generated with `runCompswitch`.

3 Building and running a simple model

The first example will show how to create a simple model using the NSA-DEVS block library and how to run it. The example model adds two streams of incoming values, it displays the incoming streams and the result. The complete Simulink model is shown in Fig. 3.1.

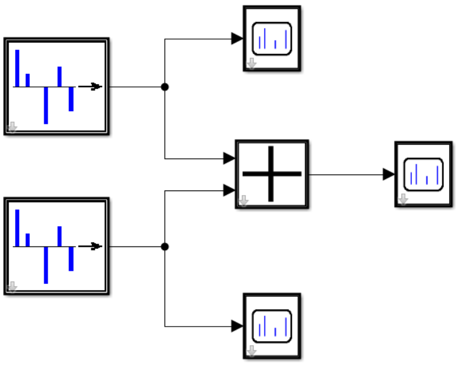

Figure 3.1: Model `tut01`.

To build the model `tut01`, open the Simulink Library Browser and locate the NSA-DEVS library that contains a few sublibraries. In `Sources` you will find the source block `am_vectorgen` that generates given output values at given times. Add two of them to the new model. The sublibrary `Math Operations` contains the block `am_add2` that adds two incoming numbers. Add one to the model. Finally, add three blocks `am_toworkspace` from `Sinks`. They are output blocks that store time and value of incoming events in a global output variable that can be accessed after the simulation. Now connect all blocks according to Fig. 3.1.

All blocks contain parameters, which can be accessed after a double-click on the block. Change the parameter `vector of output times` of the upper generator to `[1, 2, 3, 4, 5]` and its parameter `vector of output values` to `[1, 2, 3, 2, 1]`. Similarly, the lower generator gets the values `[1.3, 2.3, 3, 4, 4.6]` for the output times and `[1, 2, 1, 3, 1]` for its output values. The output blocks need different names for their corresponding output variables. Set the parameter `varname` to `"in1"` and `"in2"` for the upper and lower blocks and to `"out1"` for the block that is connected to the output of the adder. This completes the model `tut01`.

> If you look inside the block `am_add2` – e. g. by clicking at the arrow on the icon –, you'll find that it only contains input and output ports without any functionality. The NSA-DEVS Simulink library contains only the interface of a block. Besides its name and the number and names of the ports, this includes the parameters with their default values, the icon and the documentation. This information is stored in the block mask. The actual definition of a block is provided by a Matlab file in the modelbase. For the `am_add2` component, this is the file `am_add2.m` in `NSADEVSHOME/Modelbase/MathOperations`.

To run the model, create the simple Matlab file `runTut01.m` shown in Listing 3.1. Line 6 runs the model generator, which translates the Simulink model into a set of Matlab files that describe the top-level model and additional coupled models, if necessary. Line 7 starts the simulator, which runs the model for the simulation time given in `tEnd`, cleans up intermediate files and returns all outputs in the variable `out`. Finally, line 8 calls the plot function, which displays the results.

```
function runTut01
% makes and runs the model and plots the results
model = "tut01";
tEnd = 6;

model_generator(model);
out = model_simulator(model, tEnd);
plotResults01(out, tEnd)
end
```

Listing 3.1: Run script for example model `tut01`.

While the run script can be used similarly for all models, the plot function has of course to be adapted to the concrete model and the needs of the modeller. Listing 3.2 presents a simple example that produces the output shown in Fig. 3.2. It uses the output data in the variable `out`, which is a structure that contains a field for each `am_toworkspace` block, using the field name given as parameter in the block (cf. lines 16, 23, 30). Each block returns its data again as a structure containing a field `t` with the vector of output times and a field `y` with the output values.

```
1  function plotResults01(out, tEnd)
2  width = 450;
3  height = 600;
4  screenSize = get(0, "ScreenSize");
5  figureName = "tut01";
6  
7  % open new figure only if necessary
8  hFig = findobj("Type", "figure", "Name", figureName);
9  if isempty(hFig)
10   figure("name", figureName, "NumberTitle", "off", "Position", ...
11        [screenSize(3)-width, screenSize(4)-height, width, height]);
12 end
13 
14 tiledlayout("vertical")
15 nexttile
16 stem(out.in1.t, out.in1.y);
17 grid("on");
18 xlim([0, tEnd])
19 title("in_1");
20 xlabel("t")
21 
22 nexttile
23 stem(out.in2.t, out.in2.y);
24 grid("on");
25 xlim([0, tEnd])
26 title("in_2");
27 xlabel("t")
28 
29 nexttile
30 stem(out.out1.t, out.out1.y);
31 grid("on");
32 xlim([0, tEnd])
33 title("out_1");
34 xlabel("t")
35 end
```

Listing 3.2: Plot function for example model `tut01`.

Though the model looks like a simple Simulink model, this similarity is misleading: Due to the discrete-event structure applied here, outputs don't have values at all times, but only at those instants, when an event occurs. Therefore the add block can't rely on simultaneous input values at both of its inputs, but has to store incoming values, using initial values of 0. The stem plot chosen for Fig. 3.2 emphasizes this behaviour and should make the results comprehensible.

In case you wonder, what is going on after $t = 5$: The `am_vectorgen` repeats the time and output values cyclically, which creates an input at $t = 5 + 1$ for in_1 and at $t = 4.6 + 1.3$ for in_2. To suppress such repetitions, simply add a final very large time at the end of the output times vector and an arbitrary corresponding output value.

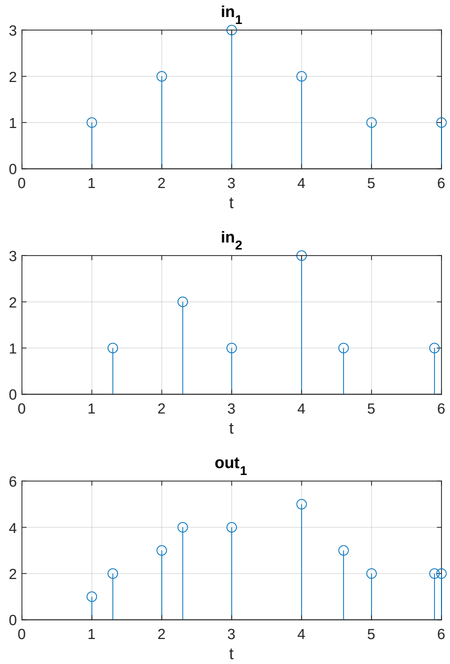

Figure 3.2: Plot generated with `plotResults01`.

More information about the model generator and the semantics of a Simulink-like model in a discrete-event based environment can be found in [4]. The exact description of the simulator is a fundamental part of a DEVS formulation. The NSA-DEVS simulator used here is explained in detail in [2].

4 Working with Queues and Servers

An important application of discrete-event simulation is the modeling of queue-server systems, where people, goods or information are transported through the system. Usually, they are modelled as abstract entities, which carry additional information ("attributes"). The example model will use simple integer numbers ("id's") to represent them.

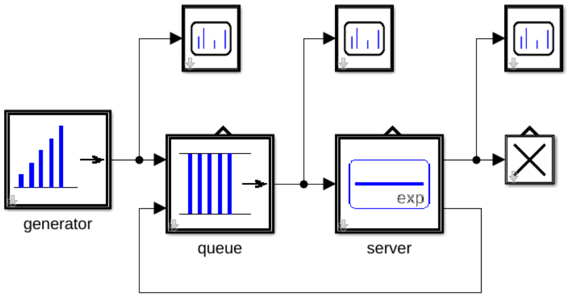

Figure 4.1: Model `tut02`.

Fig. 4.1 shows the basic queue-server model `tut02`. It starts with a simple generator that outputs events with increasing integer values in fixed time intervals (`am_generator`). They reach a queue (`am_queue`), where they wait, until the server (`am_expserver`) is ready for them. After a random, exponentially distributed service time they leave the server and proceed to the terminator (`am_terminator`), where they leave the system. Three output blocks `am_toworkspace` collect the output values of the generator, the queue and the server. The parameter `mean service time` of the server is set to 0.9. As always, the output blocks have different and meaningful names for their `varname`. And just to get nicer looking plots, the parameter `id of the first entity` of the generator is set to 1.

The names of blocks in Simulink can be chosen arbitrarily. In ND4M they have to be proper variable names, therefore they can't contain spaces or other special characters.

An unusual feature is the line running back from the server to the queue. It is needed here due to the semantics of the queue, which outputs entities unless it is blocked. Its blocking status is given by an additional input, which is connected to the output `working` of the server. It is true, when the server is busy.

The exact timing of the signaling loop and the internal behaviour of server and queue are of paramount importance here! Unfortunately, in the usual modeling approach all time delays are zero, the correponding events are concurrent and their exact order is often hard to control. NSA-DEVS uses a different concept, based on infinitesimal delays [1, 2]: All inputs and all "immediate" state changes are delayed, where the delays are defined as block parameters of the form $a + b\,\varepsilon$ with real values a, b and an infinitesimally small ε. In ND4M such values are represented by Matlab vectors `[a,b]`. These parameters are predefined in the library, they are collected in a tab of `Advanced` parameters and generally have the value `[0,1]` $\hat{=}\ \varepsilon$. But to make the queue-server pair work, the queue parameter `delay time of the queuingFree state` has the default value `[0,2]`. In special cases it may be necessary to fine-tune such a parameter to make a model work in the expected way [4]. Exact mathematical definitions and a careful analysis of the queue-server model can be found in [3].

The run function sets the `seed` parameter of the simulator, which guarantees a reproducible outcome of the simulation. Otherweise, one gets a different output with each run. The value of the seed has been chosen carefully to create results that can be easily analyzed (cf. Fig. 4.2): Except for a short period after $t = 3$, the queue is empty before $t = 6$ and after $t = 10$.

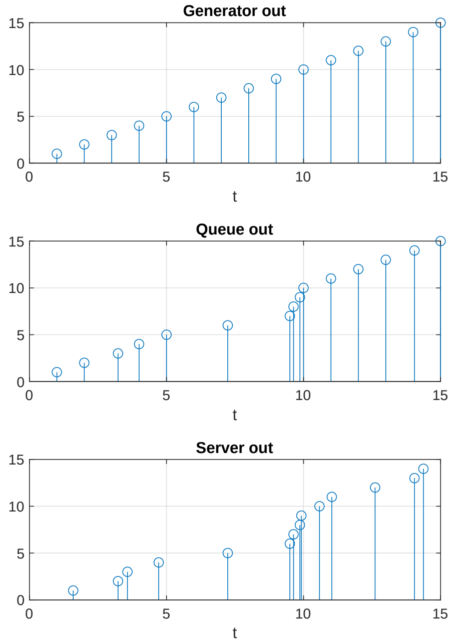

Figure 4.2: Output plot of Model `tut02`.

5 Using Entities with Attributes

Many applications use entities with attributes instead of simple numbers. ND4M contains several components for this purpose [5]. In the next example, they are put to good use to add statistical output to the simple queue-server model `tut02`.

Four atomics are included in the library to handle entity attributes: `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 a 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.

The model `tut03` extends the queue-server model by adding the following statistical outputs:

- the current queue length,
- the current utilization of the server,
- the total throughput time of the entities.

The first value is given explicitely as an output of the queue. For the second value, the number `n` of entities in the server (0 or 1) is sent to the component `am_utilization`, which computes the mean value over time of its input. The throughput time of an entity is the sum of its waiting time in the queue and the processing time in the server. A common way to compute it is to store the creation time of an entity in an attribute and subtract this value from the current time, when the entity leaves the server.

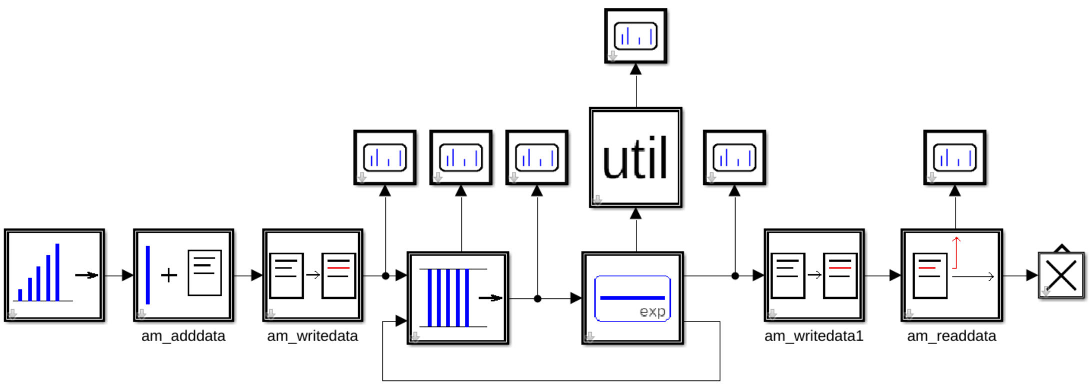

Figure 5.1: Model `tut03`.

This idea is implemented in the model (cf. Fig. 5.1) in the following way: The `am_adddata` component converts the incoming value into an entity with a field `id` to store the incoming number and a field `startTime` with the initial value 0. The `am_writedata` component then computes the current simulation time using the auxiliary function `get_time` and stores it in the `startTime` attribute. More explicitely, its parameter `changing function`, which contains a Matlab command that always stores its result in a variable named `out`, has the value `"out = get_time();"` and the `output field name` is `"startTime"`. After the entity has left the server, another `am_writedata` component changes "startTime" using the command "out = get_time() - in(1);" as its changing function. The variable in(1) refers to the first attribute name given in the parameter input field names, which could be a vector of attributes to use in the computation. Finally the am_readdata outputs the current value of the "startTime" attribute.

The final simulation results are shown in Fig. 5.2. The corresponding plot function has two new features: Firstly, one can't use out.srvOut.y to reference the output vector of the server, because y itself is now a vector of struct variables. The vector of its id fields has to be specified as [out.srvOut.y.id]. The complete plot statement therefore is

```
stem(out.srvOut.t,[out.srvOut.y.id]);
```

Secondly, though the queue length and server utilization are specified by events at special time instants, they are usually displayed using stair plots with

```
stairs(out.qLen.t,out.qLen.y);
```

This corresponds to the idea, that these values stay constant between events, which is true exactly for the queue length, but only approximately for the server utilization.

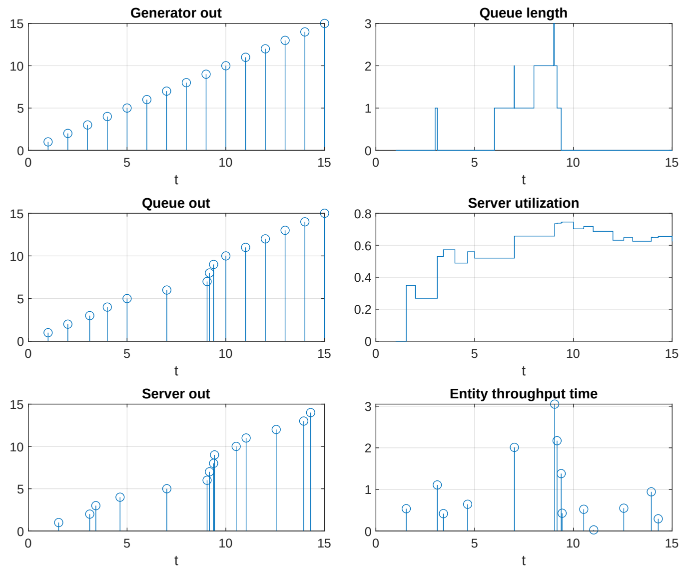

Figure 5.2: Output plot of Model tut03.

6 Creating Coupled Components

Often, entities enter a system not with constant time differences, but in a stochastical manner. An important arrival process is the Poisson process, where the interarrival times are exponentially distributed. Though such a generator is – at the moment of writing – not included in the block library, it can be created easily, as is shown in the next example `tut04a` (cf. Fig. 6.1).

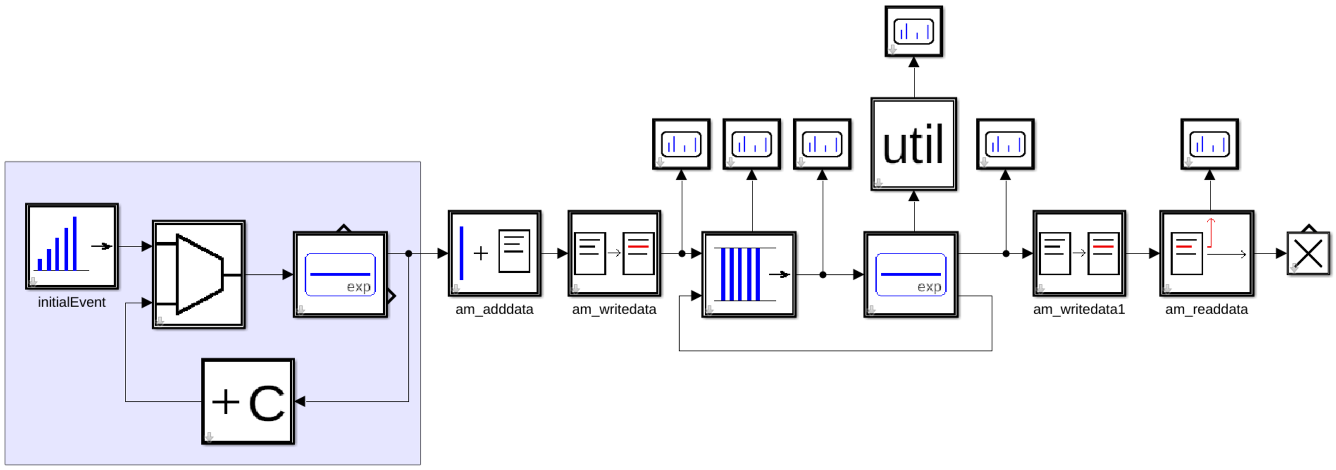

Figure 6.1: Model `tut04a`.

The exponential generator consists of four components: An `am_generator` (named `initialEvent`) creates a single output 1 at time $t = 0$, which is sent through an `am_collect2` block to a server with mean service time $t_S = 1.1$. It is delayed by the `am_expserver` and leaves the generator. A copy is routed through an `am_bias` block, which increments its value, and sent back to the server resulting in a permanent output stream.

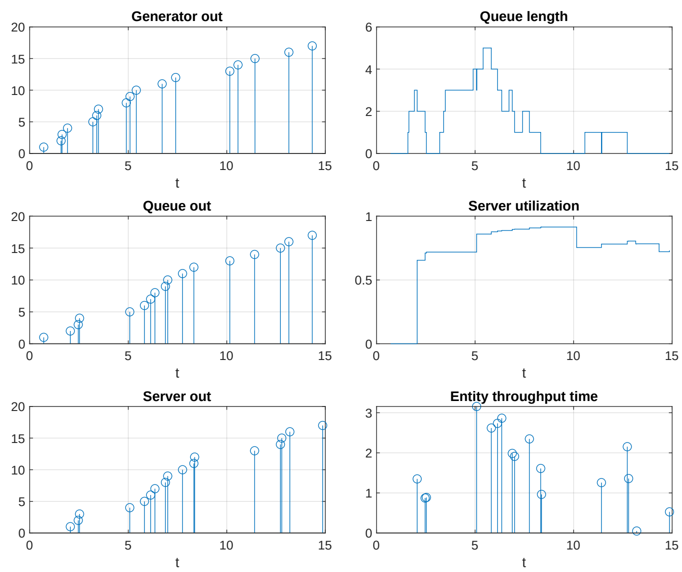

Figure 6.2: Output plot of Model `tut04a`.

Fig. 6.2 displays the simulation results, which show the typical lumpiness of an exponential distribution. To get an impression of the large variability of the results, one should try different values of the seed parameter. Statistical variables such as the mean queue length have a large standard deviation here, therefore one needs a lot of runs (with different seeds) to get good approximations.

In Simulink, the four components of the exponential generator can be combined in a subsystem. To make the model generator work, this subsystem needs a mask. In the simplest case (`tut04b`), it just defines the type of the component in the documentation. Of course, a proper mask contains an icon and a documentation and defines some block parameters. The corresponding model `tut04c` is shown in Fig. 6.3.

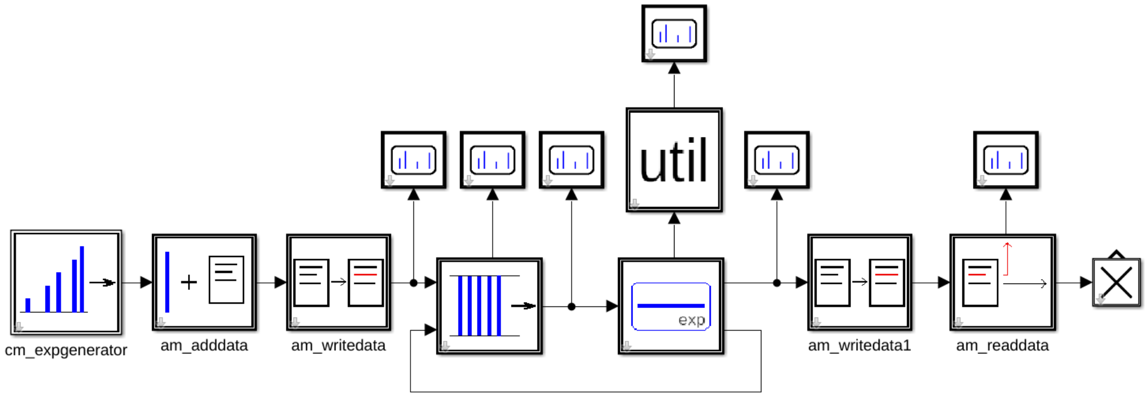

Figure 6.3: Model `tut04c`.

The model generator creates a Matlab description of the main model – and recursively of all subsystems contained in the model – using its Simulink description. To have a look at these files, one can call the model simulator with `out = model_simulator(model, tEnd, "clearFlag", false)`, which stops it from removing them after the simulation. One now finds the directory `tut04c` containing the main Matlab file `build_tut04c`. It consists of five sections:

- create atomics,
- add atomics to simulators,
- create coupled models,
- add simulators and models to coordinator,
- add couplings.

This is the proper description of the coupled model and could have been created manually instead of its Simulink version. The mentioned coordinator and simulators are parts of the model simulator. After use, e. g. for debugging purposes, the directory can be safely deleted, since it will be reconstructed from the Simulink model at the next run.

7 Creating a Simple Atomic Component

To create an atomic component from scratch, one has to write a corresponding class file that implements the definition of an NSA-DEVS atomic model [2]. A simple example `am_max2` that computes the maximal value of its two inputs will illustrate the basic procedure.

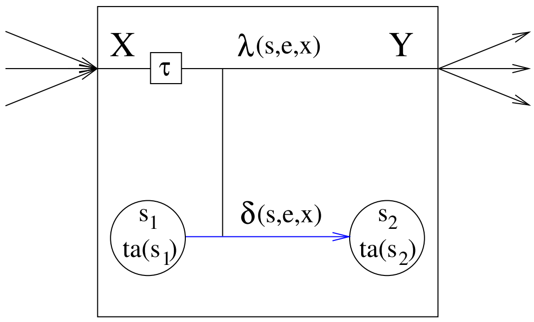

Figure 7.1: Basic structure of an NSA-DEVS atomic model.

The NSA-DEVS specification of an atomic model consists of a set of input ports X and output ports Y, a set of internal states S, an input delay time τ, a function $ta(s)$ that returns the lifetime of a state s, an output function λ and a transition function δ that computes the next state (cf. Fig. 7.1). A state change happens, when the lifetime of the current state is over or when an input event arrives.

Listing 7.1 shows a (slightly simplified version of) the class definition of `am_max2`. As has been noted for the `am_add2` component in section 3, the `am_max2` component has to store incoming values internally. For this purpose, the `properties` section in the class definition contains the two fields `u1` and `u2` (lines 5f). The three additional properties in lines 7–9 are mandatory for ND4M atomics: `name` stores the name of the component, usually defined in the Simulink description of the coupled model containing this atomic, `tau` is the input delay time, usually set to `[0,1]`, and `debug` is a debug flag, usually set to `false`.

```
1  classdef am_max2 < handle
2    %% Description
3    %  outputs maximal value of two inputs (simplified)
4    properties
5      u1
6      u2
7      name
8      tau
9      debug
10   end
11
12   methods
13     function obj = am_max2(name, tau, debug)
14       obj.u1 = -Inf;
15       obj.u2 = -Inf;
16       obj.name = name;
17       obj.debug = debug;
18       obj.tau = tau;
```

```
19      end
20
21      function delta(obj,e,x)
22        if isfield(x, "in1")
23          obj.u1 = x.in1;
24        end
25        if isfield(x, "in2")
26          obj.u2 = x.in2;
27        end
28      end
29
30      function y = lambda(obj,e,x)
31        s1 = obj.u1;
32        s2 = obj.u2;
33        if isfield(x, "in1")
34          s1 = x.in1;
35        end
36        if isfield(x, "in2")
37          s2 = x.in2;
38        end
39        y.out = max(s1, s2);
40      end
41
42      function t = ta(obj)
43        t = [inf, 0];
44      end
45    end
46  end
```

Listing 7.1: Simplified code of `am_max2.m`.

The `methods` section begins with the constructor function (lines 13–19). Its parameter list always starts with a variable for `name` and generally ends with variables for `tau` and `debug`. In between there can be additional external parameters of the component. The constructor defines meaningful initial values for all internal state variables. For an input of the maximum function, this is `-Inf` as the neutral element of the max operation. The lifetime function `ta` (lines 42–44) simply returns the value `[Inf, 0]` ("infinity"), since the state of the component is changed only by incoming events. The `delta` function (lines 21–28) only stores any incoming values, while the `lambda` function (lines 30–40) outputs the maximum of the input values, using stored values, where no current input is available.

Comparing this listing to the actual file `am_max2.m` in the tutorial examples, one finds three differences:

- A state variable `s`, which is used to store a "macroscopic" state ("phase") describing the behaviour of a complex component (cf. section 9). In this simple example it has the fixed value `"running"`.
- A set of output statements for debugging.
- An initial comment describing the ports, (internal) states and (external) parameters of the component.

All of these features are part of a quality component.

To add this new atomic to a Simulink block library, one starts by creating a local Simulink library and adding a subsystem that only contains the input and output ports (Fig. 7.2), using the names given in the class (cf. lines 31f, 39). The library block has the same name as the atomic and a mask with an icon, its documentation and parameters. The parameters are grouped in two tabs: `General`

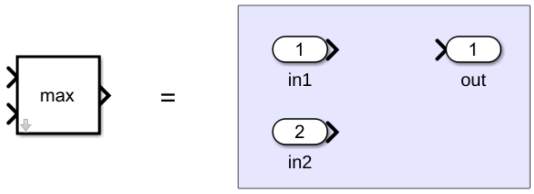

Figure 7.2: Simulink library block for `am_max2`.

for the usual parameters (empty in the current example) and `Advanced` for special values such as `tau` and `debug`.

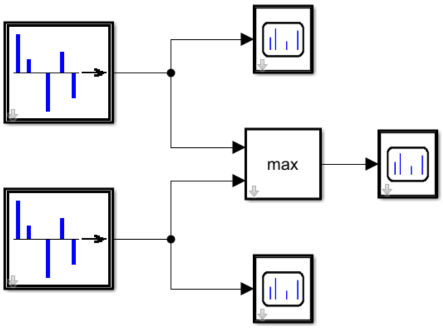

Figure 7.3: Model `tut05`.

To test the component, one creates a small model such as `tut05` (Fig. 7.3). Fig. 7.4 displays the simulation results, if everything is correct. If not: Proceed to section 8.

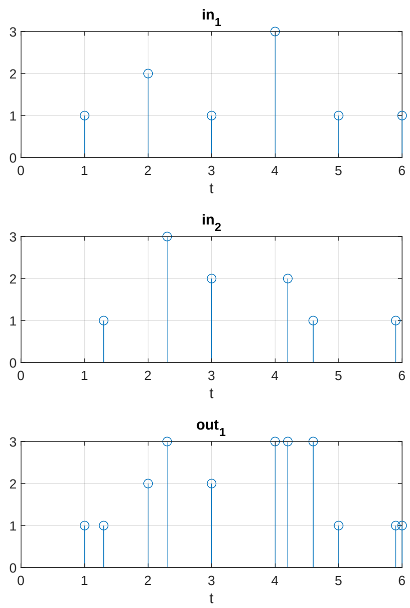

Figure 7.4: Output plot of Model `tut05`.

8 Debugging

The last two sections will dive deeper into the internals of ND4M modeling and simulation. Therefore it is advisable to proceed further only after one has read the "internal information" parts of the previous sections.

Discrete-event models can easily contain very hard to find errors. To support the debugging process, ND4M contains several debugging options at different levels of the simulation process, which will be briefly explained and then applied to solve a problem with the test model `tut06`.

On the highest level, one can use two additional options of the model simulator:

```
out = model_simulator(model, tEnd, "clearFlag", false, "displayFlag", false);
```

If the `clearFlag` is set to `false`, the intermediate Matlab files created by the model generator for coupled models are not deleted after use (cf. section 6). This is helpful to identify errors in the Simulink models. If the `displayFlag` is set to `true`, the simulator outputs time stamps on the fine-grained infinitesimal level during simulation. This can be useful, if the simulator is caught in a loop, or together with block-level debugging.

On the block level, every atomic component has (or should have!) a debug flag as parameter. It can be set to `true` in the Simulink model for each component individually, giving fine grained control to prevent a flood of debug messages. Armed components should output their input and output values and all state changes during the simulation run. This is best used together with `displayFlag = true` to get corresponding time stamps.

Two different methods are especially useful for detecting problems with the order of concurrent events: The model simulator defines a global variable `mu`, which is usually set to 0. If it is larger, it will be used as a small time interval instead of the inifinitesimal ε. This makes the order of formerly concurrent events directly visible in plots. On the other hand, it could lead to a different behaviour of a model, therefore it has to be used with extra care. A non-intrusive way to see concurrent events is to change the value of the `tau` parameter of an `am_toWorkspace` atomic. By default, it is set to `[0,5]`, which usually is larger than the infinitesimal time delays appearing at the block. In this case it gets only the last event of an infinitesimal series of events. Changing `tau` to a value smaller than the appearing delays (usually `[0,0.5]` should work), the block registers all incoming events, which will be plotted at the same time.

This feature of the `am_toWorkspace` atomic is the consequence of the fundamental behaviour of the model simulator: When an event arrives at an input during the delay time of a previous input event at the same port, the previous event will be overwritten completely and the delay time starts afresh.

Finally, the model simulator provides the global variable `DEBUGLEVEL`. If it is set to 1, all internal simulator messages will be collected and displayed graphically (cf. [2]) utilizing the Sequence Diagram tool available from Matlab File Exchange. This feature is mainly used to debug the simulator itself, but it might be useful for very weird timing problems in a model – for *small* models!

The tutorial example `tut06` mainly consists of a standard queue-server model with fixed inter-generation and service times, both equal to 1 (Fig. 8.1). But now the service process, actually a manufacturing process, is assumed to be imperfect: About 50% of the parts have to be reworked. Therefore the entities have a new attribute `outPort`, which is set randomly to 1 or 2

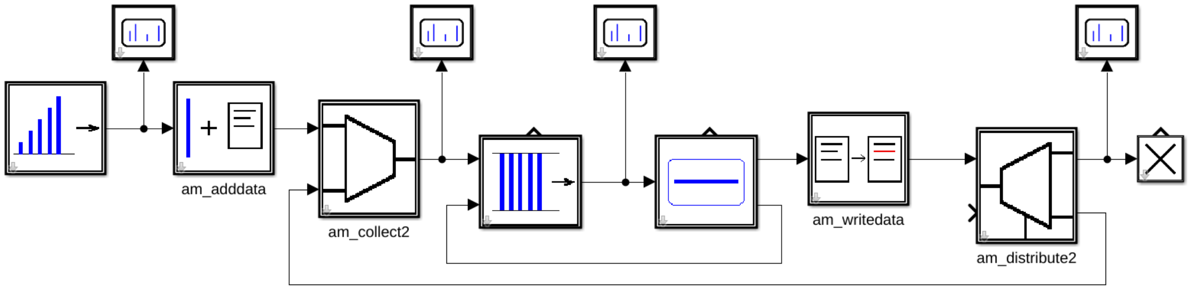

Figure 8.1: Model `tut06a`.

by `am_writedata`. The following `am_distribute2` component uses this attribute to decide, where to route an entity. The imperfect entities are routed back to the queue through an `am_collect2` atomic. We define the behaviour of the model precisely by requiring that parts to be reworked should take precedence, whenever a new and a processed part arrive at the collector simultaneously.

The output of the simulation is shown in Fig. 8.2. While the basic behaviour is ok – all parts are generated correctly and finally leave the system –, it is hard to see, what is going on in detail: The plot `Queue in` never shows parts 5 and 6 and seems to be in conflict with the `Queue out` plot.

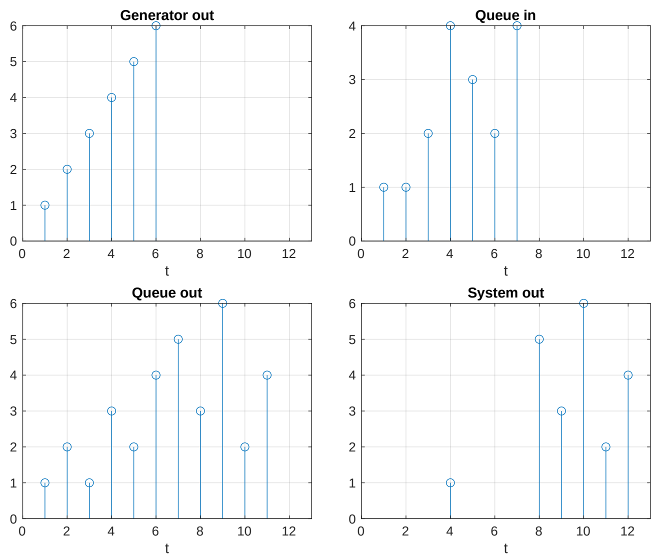

Figure 8.2: Output plot of Model `tut06a`.

Obviously, `Queue in` is missing some events. Since the input of the queue is the only port, where entities can arrive simultaneously, the behaviour is becoming more transparent, when setting the `tau` parameter of the corresponding `am_toWorkspace` block to `[0,0.5]`. The new result is displayed in Fig. 8.3.

Now everything seems to be clear: At $t = 2$ both the processed part 1 and the new part 2 concurrently arrive at the input of the queue, and part 2 appears at the output – in violation of our precedence rule. A fix seems to be easy: To delay the incoming new part, the `tau` parameter of `am_adddata` is enlarged to `[0,2]`. A quick run shows that – against our expectation – the simulation results haven't changed! And increasing `tau` to `[0,3]` doesn't change anything either. Instead of further fiddling around with parameters, we set the `debug` flag of `am_collect2` and set `displayFlag = true` to see precisely, what is going on. A simulation run (with the original `tau` value of `am_adddata`) produces a lot of output in the command window. To precisely understand, what is going, we first have a look at the behaviour, when only a new part arrives:

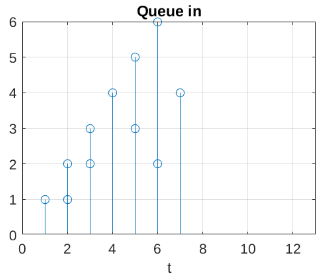

Figure 8.3: `Queue in` plot of Model `tut06b`.

```
----------------------------------------------
t: 1.00 + 2.00 ε
am_collect2 lambda
  in: in1=[ id:1.000 outPort:0.000 ] , out:
am_collect2 entering delta
  phase=idle q=[]
am_collect2 leaving delta
  phase=go q=[ id:1.000 outPort:0.000 ]
----------------------------------------------
t: 1.00 + 4.00 ε
am_collect2 lambda
  in: , out: [ id:1.000 outPort:0.000 ]
am_collect2 entering delta
  phase=go q=[ id:1.000 outPort:0.000 ]
am_collect2 leaving delta
  phase=idle q=[]
----------------------------------------------
```

Due to the delay from the `am_adddata` component, part 1 arrives at the collector at $t = 1 + \varepsilon$, where it is delayed by another ε. It is not output immediately, but stored in the internal queue `q` and the phase changes from `idle` to `go`. After a delay of $\tau_D = 2\varepsilon$ (defined as parameter in `am_collect2`) the part is sent to the `out` port, the internal queue is emptied and the phase returns to `idle`. Let's now proceed to $t = 2$, when the next part arrives.

```
------------------------------------------------------------------------
t: 2.00 + 2.00 ε
am_collect2 lambda
  in: in1=[ id:2.000 outPort:0.000 ] , out:
------------------------------------------------------------------------
t: 2.00 + 3.00 ε
am_collect2 lambda
  in: in1=[ id:2.000 outPort:0.000 ] in2=[ id:1.000 outPort:2.000 ], out:
am_collect2 entering delta
  phase=idle q=[]
am_collect2 leaving delta
  phase=go q=[ id:2.000 outPort:0.000 , id:1.000 outPort:2.000 ]
------------------------------------------------------------------------
t: 2.00 + 5.00 ε
am_collect2 lambda
  in: , out: [ id:2.000 outPort:0.000 ]
```

```
am_collect2 entering delta
  phase=go q=[ id:2.000 outPort:0.000 , id:1.000 outPort:2.000 ]
am_collect2 leaving delta
  phase=go q=[ id:1.000 outPort:2.000 ]
------------------------------------------------------------------------
t: 2.00 + 5.50 ε
------------------------------------------------------------------------
t: 2.00 + 6.00 ε
------------------------------------------------------------------------
t: 2.00 + 7.00 ε
am_collect2 lambda
  in: , out: [ id:1.000 outPort:2.000 ]
am_collect2 entering delta
  phase=go q=[ id:1.000 outPort:2.000 ]
am_collect2 leaving delta
  phase=idle q=[]
------------------------------------------------------------------------
```

Part 2 arrives at the first port of the collector at $t = 2+\varepsilon$ (before the internal delay), while part 1 has been sent back and arrives at the other port of the collector at $t = 2+2\varepsilon$. This is during (ok, at the end of) the waiting time from the first input port, therefore the call of the δ-function is delayed. At $t = 2+3\varepsilon$, both input entities are stored in `q` and the phase is changed to `go`. This change takes 2ε, then part 2 is sent to the `out` port, while the queue is shortened accordingly. Again 2ε later part 1 is output, the queue emptied and the block returns to phase `idle`.

This analysis makes clear, why a delay of the incoming part of `tau = [0,2]` from `am_adddata` doesn't help: In this case part 1 and 2 arrive at the same time $2 + 2\varepsilon$ at the collector. But even `tau = [0,3]` doesn't change the output order: Now part 2 arrives during the waiting time of the incoming part 1, the δ call is delayed and everything proceeds as before. But a larger delay, e. g. `tau = [0,4]` does the trick: The debug output shows that the collector is already in the `go` phase, when part 2 arrives. Therefore, part 1 is output and part 2 is stored in `q` to be output 2ε later, at last coming behind part 1 in the queue component. The output plot (Fig. 8.4) shows the new behaviour with the required ordering of events.

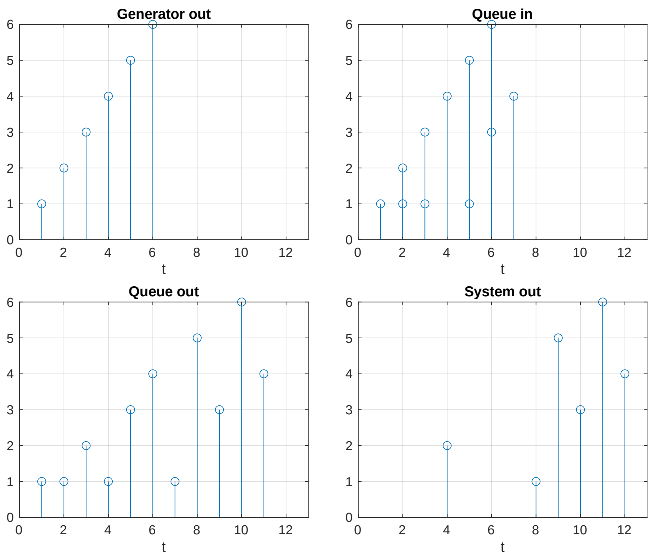

Figure 8.4: Output plot of Model `tut06d`.

9 Creating a Complex Atomic Component

The behaviour of many atomic components is much more complex than the simple component built in section 7. A very typical example is the queue, which can best be described by introducing several phases that combine states with a similar behaviour. The implementation of the basic atomic `am_queue` will be described in some detail, before it is extended to `am_finiteQueue`, a queue with a finite capacity. More background information, especially about the concrete mathematical formulation, can be found in [3].

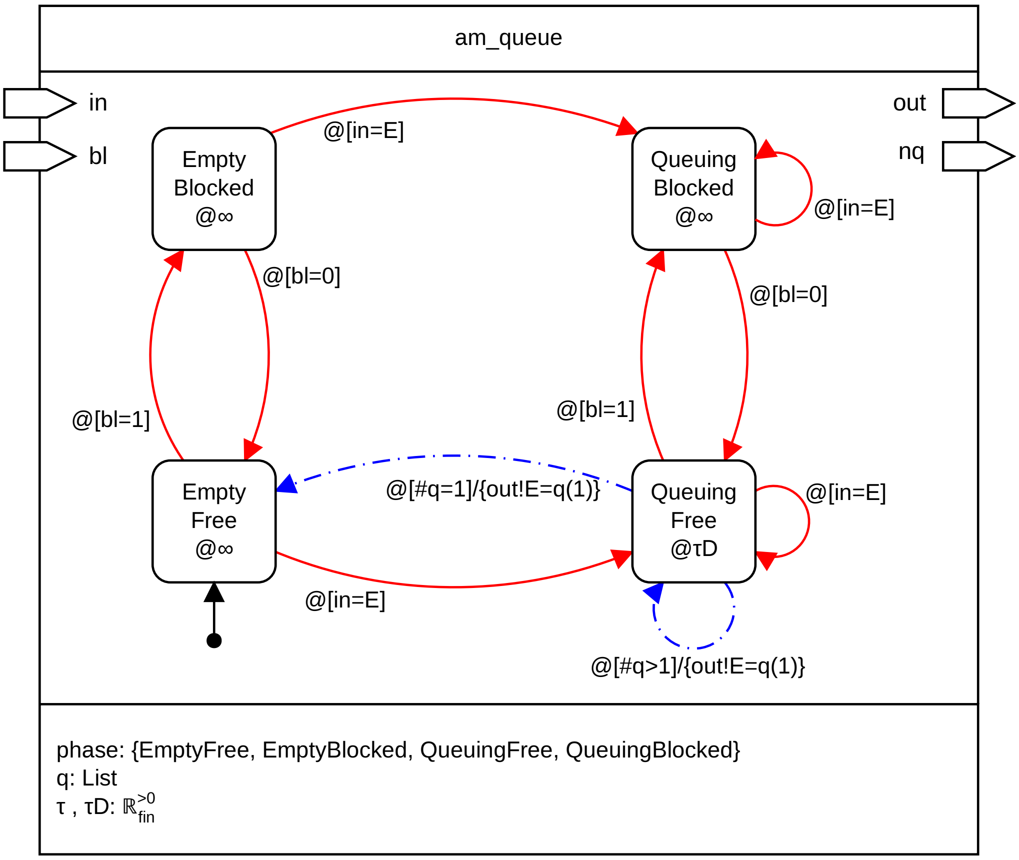

Figure 9.1: State diagram of `am_queue`.

The behaviour of the queue can be described using four phases according to the blocking status and the size of the queue length (empty or not). It is visualized using an NSA-DEVS diagram (cf. Fig. 9.1) as described in [6]. All phases except *QueueingFree* are passive states, i. e. they have infinite lifetime and change only, when inputs arrive. The queue always starts in the phase *EmptyFree*, it changes to *EmptyBlocked*, when the input port `bl` receives a value 1, and to *QueuingFree*, when an entity arrives at input `in`. The most interesting phase is *QueuingFree* being the only one, where the queue outputs entities. It is a transitory state, which means in NSA-DEVS that it has an infinitesimal delay time τ_D. Fig. 9.1 is simplified for clarity, it doesn't show outputs at `nq` or state transitions, when both inputs receive events simultaneously. A complete version is shown in [6].

The implementation of am_queue closely follows the diagram: The lifetime function ta and the output function λ can be read off easily, the more complex state change function δ is structured using the phases at the outer level and the input value at an inner level:

```
function delta(obj,e,x)
  [bl, in] = getInput(obj, x);
  switch obj.s
    case "emptyFree"
      if ~isempty(bl) && bl == "1" && isempty(in)
        obj.s = "emptyBlocked";
      elseif ~isempty(bl) && bl == "1" && ~isempty(in)
        obj.s = "queuingBlocked";
        obj.q = [obj.q, in];
      elseif ~isempty(in)
        obj.s = "queuingFree";
        obj.q = [obj.q, in];
      else  % no entities, bl status remains
      end
    case "emptyBlocked"
      if ~isempty(bl) && bl == "0" && isempty(in)
        obj.s = "emptyFree";
      elseif ~isempty(bl) && bl == "0" && ~isempty(in)
        obj.s = "queuingFree";
        obj.q = [obj.q, in];
      elseif ~isempty(in)
        obj.s = "queuingBlocked";
        obj.q = [obj.q, in];
      else % no entities, bl status remains
      end
    case "queuingFree"
      if isempty(x)     % internal event
        if (isscalar(obj.q))
          obj.s = "emptyFree";
        else
          obj.s = "queuingFree";
        end
        obj.q = obj.q(2:end);
      else               % confluent event
        if ~isempty(bl) && bl == "1"
          % blocking has precedence, no entity leaves!
          obj.s = "queuingBlocked";
          obj.q = [obj.q, in];
        else
          obj.q = [obj.q(2:end), in];
          if isempty(obj.q)
            obj.s = "emptyFree";
          else
            obj.s = "queuingFree";
          end
        end
      end
    case "queuingBlocked"
      obj.q = [obj.q, in];
      if isequal(bl, "0")
        obj.s = "queuingFree";
      end
  end
end
```

Listing 9.1: δ function of am_queue.m.

It is now easy to include a finite capacity: First one introduces the capacity as a new system parameter and state variable and adds a new output `isFull`. Next, the λ-function is extended to send the correct value to `isFull`. Finally, the δ-function checks, whether there is still room for an incoming entity. If not, the entity is lost and a warning displayed. A typical code snippet looks like this:

```
case "queuingBlocked"
  if ~isempty(in)
    if length(obj.q) < obj.capacity
      obj.q = [obj.q, in];
    else
      fprintf("%s, in delta, phase %s - dropping input %s\n", ...
        obj.name, obj.s, getDescription(x.in))
    end
  end
```

Listing 9.2: Part of the δ function of `am_finiteQueue.m`.

One problem with the warning message is the wide range of possible types of income entities: It could be a simple number, a struct variable or even something different. The auxiliary function `getDescription` tries hard to create a string description of its argument.

The example model `tut07` shows the new component in action. Since the loss of entities usually is not an option for a meaningful model – as it isn't in reality! –, one has to make sure that this case doesn't happen. A simple procedure is to stop the generator process as soon as the queue has reached its capacity (cf. Fig. 9.2).

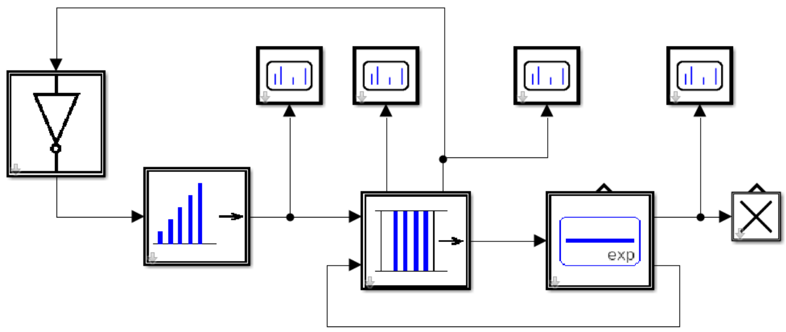

Figure 9.2: Model `tut07`.

The output plot in Fig. 9.3 shows that this approach works: The `isFull` signal stops the `am_enabledGenerator`, until the queue has room for another entity.

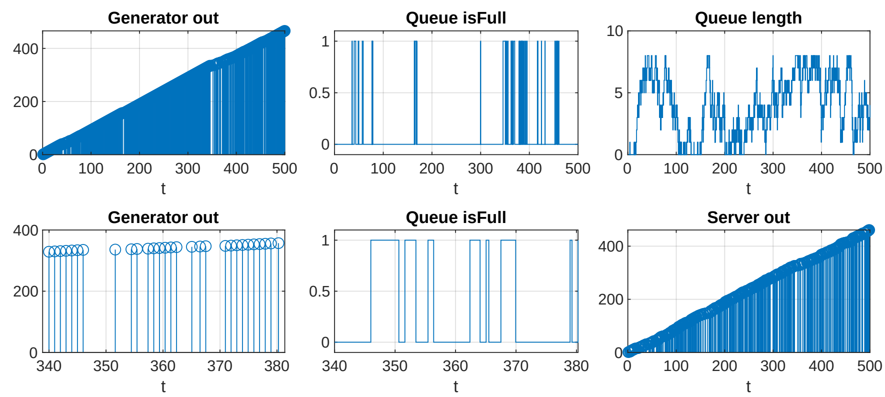

Figure 9.3: Output plot of Model `tut07`.

10 References

The order of the entries follows the logical order of the NSA-DEVS papers. Unfortunately, this does not always correspond to the order of publication date.

[1] Peter Junglas. “NSA-DEVS: Combining Mealy Behaviour and Causality”. In: *SNE Simulation Notes Europe* 31.2 (2021). doi: 10.11128/sne.31.tn.10564, pp. 73–80.

[2] David Jammer et al. “A Simulator for NSA-DEVS in Matlab”. In: *SNE Simulation Notes Europe* 33.4 (2023). doi: 10.11128/sne.33.sw.10661, pp. 141–148.

[3] David Jammer et al. “Implementing Standard Examples with NSA-DEVS”. In: *SNE Simulation Notes Europe* 32.4 (2022). doi: 10.11128/sne.32.tn.10623, pp. 195–202.

[4] David Jammer et al. “Modeling and Simulation of a Real-world Application using NSA-DEVS”. In: *SNE Simulation Notes Europe* 33.4 (2023). doi: 10.11128/sne.33.tn.10652, pp. 149–156.

[5] Peter Junglas et al. “Using component-based discrete-event modeling with NSA-DEVS – an invitation”. In: *Proc. of ASIM SST 2024 – 27. Symposium Simulationstechnik*. doi: 10.11128/arep.47.a4701. München, Germany, 2024, pp. 211–218.

[6] Thorsten Pawletta et al. “Visual NSA-DEVS Modeling Using an Adapted DEVS Diagram”. In: *Proc. of ASIM SST 2024 – 27. Symposium Simulationstechnik*. doi: 10.11128/arep.47.a4727. München, Germany, 2024, pp. 219–226.