Politecnico di Milano

SCUOLA DI INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE (3I)
Corso di Laurea Magistrale in Automation and Control Engineering - Ingegneria dell'Automazione

Tesi di Laurea Magistrale

MODES: a Modelica package for modelling and simulation of discrete event systems

Supervisor
Prof. Gianni Ferretti

Cosupervisor
Ing. Andrea Bartolini

Author
Domenico Nucera
Matr.875900

Anno Accademico 2017–2018

Politecnico di Milano:
https://www.polimi.it/

Ingegneria Industriale e dell'Informazione (3I):
http://www.ingindinf.polimi.it/

Acknowledgements

Before the presentation of this thesis work I feel the necessity of expressing my thankfulness towards a series of people.

At first Professor Gianni Ferretti, who offered me the opportunity to start such a project and followed it with patience, providing a clear direction to follow and helping me in the elaboration of the thesis work. A great acknowledgement belongs also to Ing. Andrea Bartolini, who propelled this work with precious suggestions and resources, which I hope to have translated in a quality library.

I would like to thank the people from I4.0 Lab, especially Ing. Walter Quadrini and Ing. Andrea Luigi Edoardo Caielli, for having hosted me and for having given me much help in understanding and modelling the plant, letting me to obtain everything I needed for my activity.

A special recognition is to my parents. Avete creduto in me, mi avete supportato e soprattutto sopportato in tutti questi anni, un impegno oltremodo notevole. Vi sono infinitamente grato di tutto.

At last, but not least, I would like to thank all the friends whom I shared this years with.

In particular: Alessandra, Ambra, Antonio, Claudia, Fabio, Francesca, Giuseppe, Marina, Marina, Paolo, PierPaolo, Pietro, Shihan, Sonia. Thank you.

Milano, April 2019 Domenico Nucera

Contents

List of Figures

Sommario

Negli ultimi anni l'introduzione di nuove tecnologie digitali nelle industrie ha portato a un nuovo paradigma, quello di Industria 4.0, nel quale l'informatica potrebbe rivoluzionare i processi manifatturieri del domani.

Tra le aree di ricerca avviate in questo scenario, le tecniche di simulazione continuano ad avere un ruolo chiave. Anche se appositi programmi esistono già, certi casi caratterizzati da un elevato grado di complessità richiedono soluzioni all'altezza.

Uno strumento che fornisce già un ampio numero di funzionalità per simulare sistemi complessi è Modelica, usato per implementare modelli in un languaggio basato su equazioni, permettendo di simulare diverse tipologie di processi. Per descrivere alcuni specifici comportamenti, eventi discreti possono essere introdotti per rappresentare cambiamenti repentini in alcune variabili del sistema.

In questa tesi il meccanismo degli eventi in Modelica sarà usato per sviluppare MODES, una libreria orientata all'esecuzione di simulazioni ad eventi discreti.

La simulazione di sistemi a eventi discreti ha un variegato spettro di utilità, e sono spesso applicate con successo in contesti industriali, in modo da analizzare le performance degli impianti di produzione. Poter effettuare simulazioni di questo tipo in Modelica può aprire molte opzioni ai modellatori. Infatti modelli ibridi, in cui interagiscono componenti a tempo continuo con altri ad eventi discreti, potrebbero beneficiare del già ampio nonchè in continua evoluzione insieme di librerie che popolano l'ecosistema di Modelica, in combinazione con dei moduli a eventi discreti caratterizzati da un elevato livello di elasticità.

Nello sviluppo di MODES si è scelto di adottare il compilatore di OpenModelica, che è gratuito, per garantire l'accesso alla libreria ad un più vasto insieme di figure professionali, dal mondo accademico a quello industriale, oltre che per potenziare lo strumento open source di riferimento per eseguire simulazioni in Modelica. Infatti in questo lavoro si è operato unicamente con il linguaggio Modelica, essendo intenzionati a dimostrare le capacità del meccanismo di event iteration implementato nel compilatore OMCompiler.

Parole chiave: PoliMi, Tesi, Modelica, Simulazione ad eventi discreti, DES, DEVS

Abstract

In the recent years new digital technologies applied to the industrial environment have led to a new paradygm, the Industry 4.0, in which computerization is supposed to revolution the manufacturing processes of the future.

Among the many areas of research open in this modern scenario, simulation tools continue to have a key role. Even though many dedicated softwares already exist, application cases characterized by a high level of complexity require solutions up to their demands.

A tool already providing a wide set of features to perform complex simulations is Modelica, which is used to implement models of physical systems in an equation based language, allowig engineers to simulate different types of processes. To describe certain specific behaviours, discrete events can be introduced to represent sudden changes in the system variables.

In this thesis the Modelica's events mechanism will be used to develop MODES, a library aimed at performing discrete event simulations.

Discrete event simulations have a wide range of utilities, and they are usually applied with success to manufacturing systems, in order to gain insights on plants' performances, identifying bottlenecks and more in general to understand the production dynamics. Being able to run simulations of this kind in Modelica can open many options to modelers. In fact hybrid models, in which continous time components interact with discrete event ones, could benefit from the already vast and always evolving set of libraries that populate the Modelica environment, combined with a set of discrete event modules characterized by a high degree of customizability.

It was chosen to develop MODES adopting the OpenModelica compiler, which is free, guaranteeing access to a wider range of professionals, from the academic and the industrial world, while empowering the open source reference tool for executing Modelica simulations. In fact this work uses purely the Modelica language, being driven by the desire of showing the capabilities of the event iteration mechanism implemented in the OMCompiler.

Keywords: PoliMi, Master Thesis, Modelica, Discrete Events Simulation, DES, DEVS

Chapter 1

Object-Oriented Modelling

In the engineering practice it is common to execute simulations of processes and devices we are interested in. Simulations can be used to avoid running expensive or even dangerous experiments, to test controllers or to obtain useful insights.

However, before considering any issue related to the virtual implementation of trials, one has to formulate a model of the system under study: a set of equations that describes its behaviour and allows a mathematical representation of the physical phenomena involved.

This task requires knowledge of the field we are working in, together with the capability to manipulate the equations when needed, either for convenience or because of the setup required by the software we intend to use.

The modeler can choose to decompose the original system into subsystems, which are then modeled individually defining their internal behaviour together with the effect of their interaction with other subsystems, thus obtaining elements that can be mathematically connected to each other.

Considering these elements, Object-Oriented paradigm could be applied on them, declaring them as classes and giving them Object-Oriented features. This procedure is called Object-Oriented Modeling, and it gives us a powerful approach to complex systems' simulation, since decomposing one into a collection of subcomponents going to be joined together can help us see it in a less intricated way.

But also, classes open a vast number of options for the system modeler. They can be reused many times or encapsulated one into another to express various architectures of similar systems.

Inheritance allows us to identify common characteristics, wrap them in a single class, and then expand it into different elements by adding the details that characterize the specific component. This base class could turn very useful in case a new item would be required, letting us only the job of adding its specific attributes without having to redefine it starting by zero.

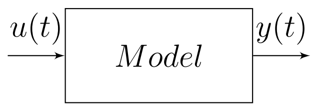
One of the most common examples is the one of electrical components, which obey the same Kirchhoff laws of tension and current at their pins, but have different physical principles acting inside them. In this case, a generic two pin component can be considered, and then expanded into a resistor or a capacity by adding equations to express the voltage difference at its extremities.

1.1 Acausal approach

As we have seen above constructing a model comes down with defining proper equations. Often those equations are of differential nature, meaning that derivative terms are present to take into account dynamic effects, like the acceleration of a mechanical part or the populations' evolution in biology.

Traditionally, systems have been described through a causal approach, made up by Ordinary Differential Equations (ODE) and input-output relationships between subsystems, resulting in a modeling procedure in which signals would flow from one block to another, following a consequantial criterion.

$$\frac{\mathrm{d}\boldsymbol{x}(t)}{\mathrm{d}t} = \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{u}(t), t)$$
$$\boldsymbol{y}(t) = \boldsymbol{g}(\boldsymbol{x}(t), t)$$



Causal Model

In this setup $\boldsymbol{u}(t)$ is a vector input signal while $\boldsymbol{x}(t)$ represents the state variables of the system.

Usually this arrangement is not a straight result of the modeling phase, but needs a reformulation in which the variables to be solved are explicited, so that they can be handled to a numerical solver for the simulation.

Apart from the work required for the equations' manipulation, this technique usually hides the original nature of the system and its components, furthermore it may involve complications when the system's structure changes.

This may turn in a drawback when trying to follow the object-oriented paradigm, since we want to represent every physical item as a class with its variables, behaviour and boundary conditions.

Defining the properties of the physical subcomponents and assembling them according to the system's configuration seems to require a new way to model the system, one allowing to declare an object in the most direct way possible, without having to manipulate the equations up to a point where their interpretability is lost.

The declarative or acausal approach solves this problem: models represent real elements and their interfaces, called connectors, allow us to express their interactions.

So, when representing a model in acausal form, we assert its physical laws as Differential Algebraic Equations (DAE):

$$\boldsymbol{F}(t, \boldsymbol{y}(t), \frac{\mathrm{d}\boldsymbol{y}(t)}{\mathrm{d}t}) = \boldsymbol{0}$$

and then relate the models' blocks to establish the relationships between their interface variables.

This method is highly compliant with how items would be assembled, and it relieves the engineer from the job of assigning a solving equation to every variable, leaving this task to the computational tool.

In fact the vast majority of the integration methods devoloped for simulation work on Ordinary Differential Equations, and even though every ODE can be considered as a special case of a DAE, the problem of converting a DAE into an ODE is not trivial. So, a number of algorithms have been deployed to reduce DAEs or to solve them.

In this situation we are free to concentrate on other aspects of the simulation, letting the software pay the price for rewriting of the explicit form.

1.2 Modelica language

Modelica is an object-oriented and equation based language designed for modeling and simulating complex and heterogenous physical systems, with laws coming from different domains of application.

It supports both a graphical approach, in which components' connections can be visualized and edited, and a textual approach, in which mathematical statements and other characteristics are coded.

The first Modelica design group was held in 1996, in which gathered experts in different languages (like Allan, Dymola, NMF, ObjectMath, Omola, SIDOPS+ and Smile) and application domains. Then, in 1997 it was released, with the capability to model continous systems. Over the course of the years many features were added, and many front-end implementations were distributed. Among these, two of the most famous are Dymola and OpenModelica.

The following paragraphs are not intended to be a complete guide neither a documentation of the language, but just a brief recap about its main functionalities, especially the ones crucial for the develompment of this thesis work.

1.2.1 Introduction

As said above, Modelica is an equation-based language. This means that we can express the model through a series of variables and a sequence of equations relating them.

```
model FirstOrderEquation
    parameter Integer a = 2;
    Real x (start = 1);
equation
    der(x) + a * x = 0;
end FirstOrderEquation;
```

The example above is very simple, it delineates a model with just a first order differential equation (`der(x)` stands for $\mathrm{d}x/\mathrm{d}t$), but in it we can see some basic syntax elements.

Variables that will have to be solved are declared at the beginning of the model, together with parameters, which are going to stay fixed over time. In this case, the value at time $t = 0$ is set next to the declaration, though other options are viable.

Modelica defines different types of variables, together with the possibility to define new ones, but the most important are:

- `Real`: a decimal number.

- `Integer`: a term to express natural numbers.
- `Boolean`: a type that can assume only two values, true or false

Also arrays of any type are allowed.

At line 4 an equation section begins, containing the model's equations. Note that an equation is not intended as an assignment, but really as a mathematical expression. It will be compiler's duty to eventually manipulate it and solve it.

A part from that, there are two main rules when developing a model that must be strictly respected:

- **The number of variables must be equal to the number of equations**
- **The number of equations must be fixed during the simulation**

The second "commandment" doesn't have to be intended as the fact that equations can't change over the course of the simulation, but as a requirement that the number of expressions determining the system's evolution has to be precisely as the number of variables at any time.

As we will see now, the nature of the model is allowed to change at certain instants, as a discontinous behaviour turns out to be useful when modeling many engineering cases.

1.2.2 Events

Together with continous-time oriented formulations, Modelica allows discrete-time expressions. Since many phenomena in the physical world evolve in a really short amount of time, often they are mathematically portrayed as a change that happens instantaneously.

During an event the time freezes, variables and equations are re-evaluated and then the numerical integration restarts.

To fire an event one needs to define logical conditions that can switch from true to false, or vice versa.

A straightforward example could be the one of the ideal diode, whose behaviour is a function of the tension at its pins. To describe it the `if` equation is used. Below its syntax is shown:

```
if expression then
{ equation ";" }
{ elseif expression then
{ equation ";" }
}
[ else
{ equation ";" }
]
end if ";"
```

For a moment it is asked to the reader to ignore many clauses in the code below, as they will be explained later on, and consider intuitively the equation section

```
model Diode "Ideal diode"
    extends TwoPin;
    Real s;
    Boolean off;
equation
    off = s < 0;
    if off then
        v=s;
    else
        v=0;
    end if;
    i = if off then 0 else s;
end Diode;
```

The variables `v` and `i` come from the `TwoPin` line represent the tension across the diode and the current flowing inside it.

For `v`, two equalities are expressed, but since only one acts during each step of the simulation they account as a single equation, whose switching depends on a boolean.

In the current equation the formulation is contracted, but the meaning is nevertheless the same.

The binary term `off` is governed by an equation itself, having in this case the form of a logical expression, which is used to determine which branch of the `if` condition shall be followed.

The kind of logical expression discriminates between two kind of events:

- **Time events** : events activated directly by the reaching of a certain time instant, for example `time > certain_value`
- **State events** : events depending on relations between system's variables, like the ones in the diode example.

While the first type of discontinuity can be scheduled in advance, conditions of the second type have to be checked at every integration step. Everytime one of those triggers an event, a problem is solved to determine with good precision the point in time when the crossing happened.

Another very useful feature is the `when` construct, which allows to activate statements at discrete time. It has the following syntax:

```
when expression then
{    equation ";" }
{ elsewhen expression then
    { equation ";" } }
end when ";"
```

allowing through the `elsewhen` to set a priority and to avoid conflicts in case of simultaneous events.

Another very common example, aimed at presenting this construct is the one of the Bouncing Ball, in which an adimensional body falls from a certain height and hits the ground, to then bounce up again and again.

```
model BouncingBall "The 'classic' bouncing ball model"
  parameter Real e=0.8 "Coefficient of restitution";
  parameter Real h0=10.0 "Initial height";
  Real Height(start = h0) "Height";
  Real Velocity(start=0.0, fixed=true) "Velocity";
equation
  Velocity = der(Height);
  der(Velocity) = -9.81 "Gravitational acceleration";
  when Height < 0 then
    reinit(Velocity, -e*pre(Velocity));
  end when;
end BouncingBall;
```

In the model above `reinit` is a built-in function that reinitializes the value of a certain variable, in this case a speed considered positive going up from the ground, and that acts at a certain instant. The condition inside the `when` is used to set that instant at the point in which the ball is at ground level.

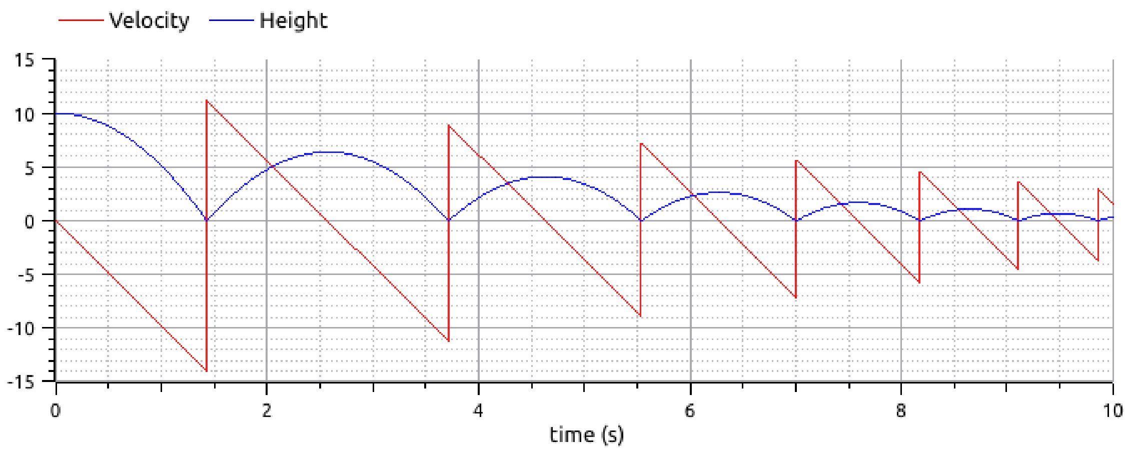


Figure 1.1: Height level and velocity as time progresses in the Bouncing Ball model, the discontious variation at contact is evident

Up to now we have only seen the possibility to declare equations in the apposite code segment, but we can declare another kind of section, the `algorithm` one, where an imperative programming approach can be followed to set values.

In an algorithm section equality declarations are prohibited and assignemnts are carried out by using the syntax `:=`.

Also, one can use an apposite `initial algorithm` section to set initial values for the variables.

```
model BasicAlgorithmModel "Simple model to show
   algorithm feature"
    discrete Real x;
```

```
initial algorithm
    x := 1;
algorithm
    when time > 0.5 then
        x := x - 1;
    end when;
end BasicAlgorithmModel;
```

The code above shows an example in which a time event triggers a modification. Note that the real variable `x` is declared as `discrete`, meaning that it can be updated only at events, and figure 1.2 shows the simulaton result.

Constructs like this one will be exploited many times over the course of this thesis.

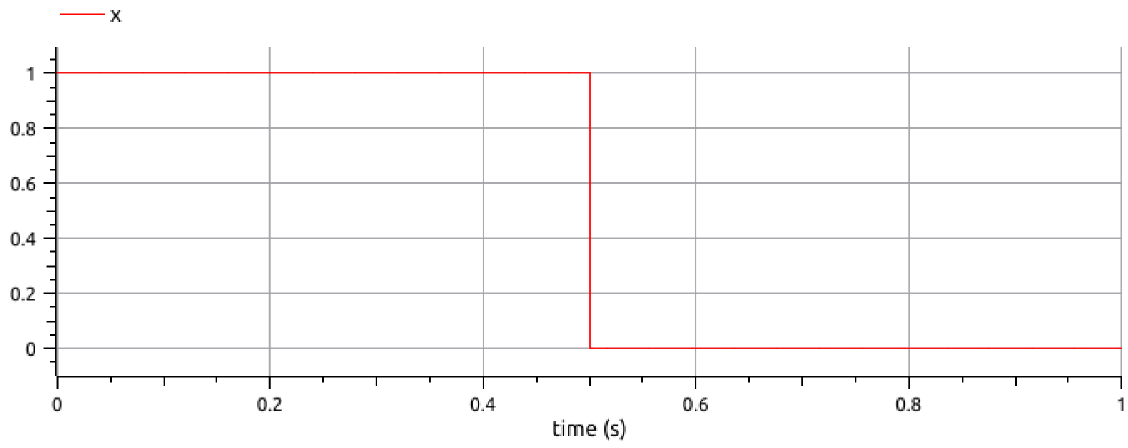


Figure 1.2

When dealing with an algorithm, the corresponding number of equations given by the section corresponds to the number of variables whose value is assigned in that section. A variable cannot be assigned in two different `when` segments, to prevent concurrent assignments. As explained above, in cases like that `elsewhen` should be used.

So far we have seen elementary models whose events had limited consequence. But concatenations of events are possible, and they are handled by a mechanism called event iteration.

Since now we have seen that once an event is fired the simulation time stops and values are then recalculated. Once again, an event check is performed, and if another event happens to be fired the whole procedure is repeated, always at fixed time. Then, when no more events appear to be activated, the simulation restarts.

A notable built-in function in this context is `pre()`, which allows to access the last value the variable had during the previous step of the event iteration.

```
model EventIterationExample "An example in which event
   iteration is applied"
    Integer Int1 (start = 0);
    Integer Int2 (start = 0);
    Integer Int3 (start = 0);
```

```
algorithm
    when time > 0.5 then
        Int1 := 1;
    end when;
    when Int1 > 0 then
        Int2 := pre(Int1) * 2;
    end when;
    when pre(Int1 > 0) then
        Int3 := pre(Int1) * 3;
    end when;
end EventIterationExample;
```

In the example, we have three integers that start with value 0. So when time reaches 0.5 seconds this is conceptually what happens:

1. A time event is triggered, and `Int1` ends up being 1.
2. A check for new events has to be conducted, and we detect one. In fact now `Int1` = 1, so the second `when` is entered. But the value `Int1` had in the last iteration was 0, so `pre(Int1)` will return 0.
3. Now we go in the third `when`, because since `Int1 > 0` was activated the last step, now `pre(Int1 > 0)` is true. Also, since at the previous step `Int1` was equivalent to 1, now also `pre(Int1)` = 1. So `Int3` is effectively augmented.
4. New crossings aren't found, and so the simulation restarts

At the end of the example, only `Int1` and `Int3` have been modified, as can be seen in figure 1.3.

One key detail to be noticed is the placement of a `pre()` to contain the condition of a `when`. Later, when we will talk about the library of this work, this locution will reveal itself more noteworthy than it may now seem.

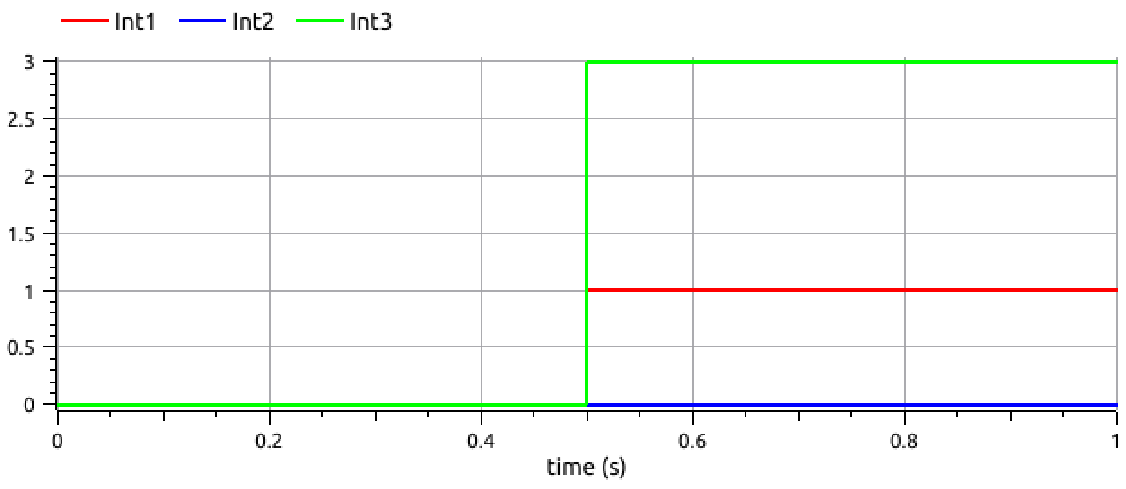
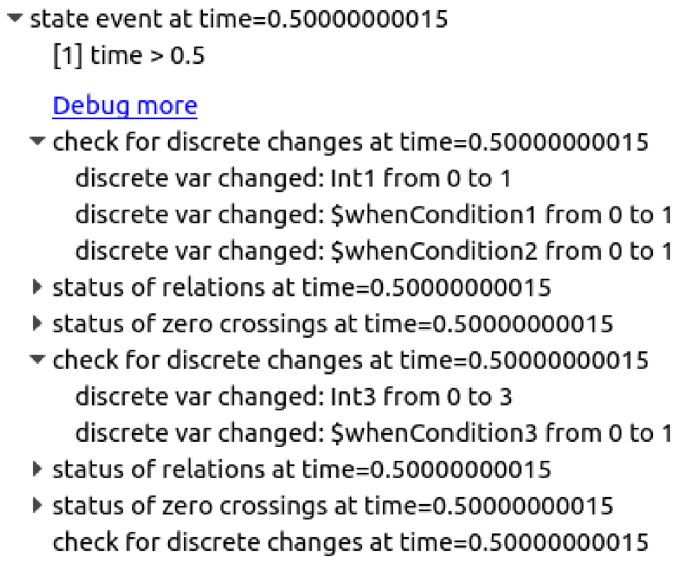


Figure 1.3

All the intermediate steps inside the event iteration are not visible in the final graph. To inspect its proceedings the OpenModelica environment offers an apposite simulation flag called "LOG_EVENTS_V", that displays the sequence of changes when the time is stopped. In Figure 1.4 a portion of the simulation output can be seen. Notice that the condition `pre(Int1 > 0)` is evaluated in a second step of event iteration, due to the delaying action of the `pre()` function. Recall that this delay is not involved with simulation time, but only on the operations carried at frozen time, once events are fired.



```
▾ state event at time=0.50000000015
    [1] time > 0.5

    Debug more
  ▾ check for discrete changes at time=0.50000000015
      discrete var changed: Int1 from 0 to 1
      discrete var changed: $whenCondition1 from 0 to 1
      discrete var changed: $whenCondition2 from 0 to 1
  ▸ status of relations at time=0.50000000015
  ▸ status of zero crossings at time=0.50000000015
  ▾ check for discrete changes at time=0.50000000015
      discrete var changed: Int3 from 0 to 3
      discrete var changed: $whenCondition3 from 0 to 1
  ▸ status of relations at time=0.50000000015
  ▸ status of zero crossings at time=0.50000000015
    check for discrete changes at time=0.50000000015
```

Figure 1.4: Output from LOG_EVENTS_V when simulating EventIterationExample.

To summarize:

- An event is something that happens in 0 time, when the simulation is stopped.
- It is activated when a certain condition is met.
- It can change the behaviour of the model using an `if`, or apply modifications at discrete time with a `when`

1.2.3 Functions

We have already encountered some built-in functions, like `der()` and `pre()`. A Modelica user can define its own functions, which consist of a set of inputs and outputs, some eventual protected variables which do not exist outside of the function, and an algorithm body containing the computations to be made.

```
function FunctionName
    input TypeI1 in1;
    input TypeI2 in2;
    input TypeI3 in3 := default_expr1;
    ...
    output TypeO1 out1;
    output Type02 out2 := default_expr2;
    ...
protected
    < local variables >
    ...
algorithm
    ...
    < statements >
    ...
end FunctionName ;
```

You may notice that default declarations are possible, but most important, a function can return multiple outputs. Assuming a function with 3 inputs and 2 outputs is defined, its call will be like this:

```
(out1, out2) := Fnctn3In2Out(in1, in2, in3);
```

Also, it is possible to wrap code written in C or FORTRAN language inside a function, through the keyword `external`.

```
function ExternalFunction
    input Integer integerinput;
    output Integer integeroutput;

    external "C" integeroutput = C_code_function(
       integerinput);
    annotation(
      Include = "#include \"externalCcode.h\"");
  end ExternalFunction;
```

In this example, a function called `ExternalFunction` provides the interface (an input integer and an output integer) for the function `C_code_function` written in C.

Data types have to be correctly mapped from one language to another: basically, a Modelica `Real` shouldn't be returned as a char C type. Here is a quick table of the main Modelica data types and their "counterpart" in C.

Modelica	*C*
`Real`	double
`Integer`	int
`Boolean`	int
`String`	const char *

More complete tables can be easily found in the Modelica language specification or other books, also because in C return by pointer is allowed.

The bottom annotation is aimed at locating the source code file containing the C function. In this case there is a file `externalCcode.h`, which is a header file, located in the path *Resources*\Include, which should be the same folder of the package.

Through proper annotations one can locate the external source code file in many other ways and in different directories, but right now it is not necessary to go deep into those aspects.

At last, it needs to be specified that functions can't have `equation` sections, can't call many built-in constructs like the already cited `pre()` or `der()`, and that there can be at most one `algorithm` section or one `external` function call, never both.

1.2.4 Connectors

So far we have only seen single models with variables and a mixture of equations or assignments. At this point one may think that more complex models are going to be written by declaring more variables and more equations always in the same chunk of code. Of course this option is always viable, but there can be more practical approaches.

If a physical system can be decomposed into components, an idea could be to model the components and then assemble them into an overall model. To assemble them, their relationships have to be identified and described.

In Modelica models "communicate" using connectors, which define the interface variables of a model. For example, thinking about the rotating shaft of an engine, its interface variables can be its speed and the torque it provides to the system.

In a connector, two kinds of variables are identified:

- **Effort variables**, whose value will have be equal throughout all the connector instances.
- **Flow variables**, whose sum among all the connector instances will have to be 0.

The two categories above are represented in the shaft's example: the speed can be considered as an effort variable, since parts attached to the same beam rotate at the same frequency, while for the Newton's third law the torques are balanced.

So let's consider a connector called Flange that will connect a motor and an inertia:

```
connector Flange "One-dimensional rotational flange of
    a shaft"
   Real phi "Absolute rotation angle of flange";
   flow Real tau "Cut torque in the flange";
end Flange;
```

For an effort variable the declaration is implicit, while in the other case the `flow` keyword is required.

Now, we can declare this connector inside the models we will define to set their interactions.

```
model Engine "Engine providing a constant torque"
    Flange shaft "Connector";
    parameter Real torque_provided = 10.0;
```

```
equation
    shaft.tau = torque_provided;
end Engine;

model Inertia
    Flange shaft;
    parameter Real J = 7.0 "Inertia value";
    Real phi;
    Real w;
equation
    phi = shaft.phi;
    w = der(phi);
    J * der(w) = shaft.tau;
end Inertia;
```

Finally, these two models can be tied at their flanges using the `connect()` statement inside an equation section.

```
model EngineAndInertia "A simple engine connected to an
    inertia"
    Engine engine;
    Inertia inertia;
equation
    connect(engine.shaft, inertia.shaft);
end EngineAndInertia;
```

When the above model is instantiated the `connect()` function sets equations according to how variables were declared in the connector.

```
engine.shaft.tau + inertia.shaft.tau = 0.0;
engine.shaft.phi = inertia.shaft.phi "equations from
    the instantiated model";
```

Every descent in hierarchy is actuated using the dot notation.

One is also allowed to declare connectors or their variables as `input` or `outputs`, letting know the compiler in which way the variables will have to be solved and thus causality in the model.

Connections can also be set in the diagram view. In fact, with graphical annotations one can specify the appearance of models and then linking their connectors, like in Simulink or other graphical programming environments. Figure 1.5 shows the diagram of a system very similar to the one of the example above, realized using the Modelica Standard Library which provides a set of models endowed with icons.

In this case the torque to be provided is given by a signal.

In addiction, also implicit connections are available, by acting on the model's hierarchy. The keyword `inner` provides a way to declare a variable that will be global in all the submodels, while `outer` will specify in the submodels that the variable will come from a model above in the hierarchy.

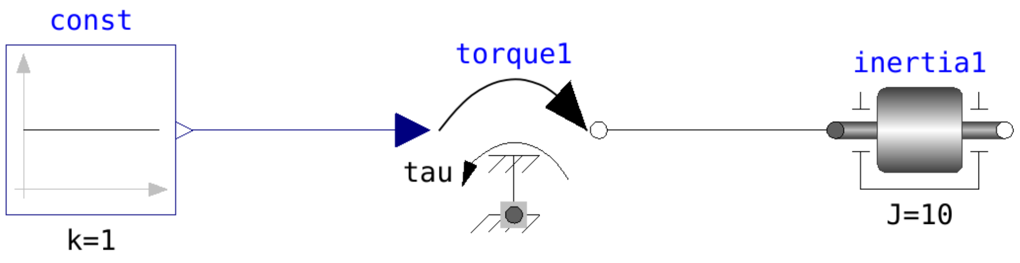


Figure 1.5: Diagram of a Torque and Inertia model. Notice the connectors' ports on the sides of the blocks

The best way to present this feature is to really think a variable that acts globally over the elements of a system, like the temperature in a given ambiance. The example below shows three thermal components sharing the environment temperature.

```
model Component "A common with a temperature T and a
   thermal capacity"
    outer Real T0 "Environment temperature, not known
       in the submodel";
    Real T;
    parameter Real C;
    parameter Real h;
equation
    C*der(T) = h*(T0-T) "Equation governing the
       component's temperature";
end Component;

model Environment
    inner Real T0 "Environment global temperature";
    Component c1, c2, c3;
equation
    T0 = 293 + 10*sin(time/(24*3600)) "Equation
       governing the global temperature";
end Environment;
```

When `Environment` will be instantiated all the variable in the component will be noted with the dot notation, like `c1.T`, apart from `T0`, resulting in flattened equations like the ones below

```
c3.C * der(c3.T) = c3.h * (T0 - c3.T);
T0 = 293.0 + 10.0 * sin(1.15740740740740 7e-05 * time);
```

1.2.5 Classes and Object-Orientation

Up to now we have seen statements that allow us to define models and combine them. This can lead to many techniques for representing physical systems, but there are still language properties to be exploited. In fact, we haven't talked about how Object-Oriented paradigm is implemented in Modelica.

First, it is necessary to introduce a concept that so far we have skipped to let the recap tend towards immediate application: the class.

As the specification says: *"the fundamental structuring unit of modeling in Modelica is the class ... classes provide the structure for objects, also known as instances"*.

A `model` is nothing but a specialized class. The items `function` and `connector` are specialized classes too, but with resctrictions like the impossibility to declare equations inside them.

This new way to see previous formulations doesn't change what we already know on them, but being aware of their underlying nature will be relevant when we will start introducing Object-Oriented features.

Before that, there are other specialized classes worth mentioning:

- `record`: a set of public variables, without the possibility to define neither equations nor algorithms inside. Very similar in its concept to a `struct` for C developers.

- `package`: a class allowed only to contain constants and declarations of other classes.

- `block`: very similar to a model, except for the fact that for every connector we need to specify if its an `input` or an `output`.

In Object-Oriented Programming objects contain data, usually considered as a set of attributes, and methods that can act on them. In Modelica the concept is applied differently, since we have variables whose evolution is defined by equations, but still many features from Object-Oriented Programming have been transposed into this language.

The most commonly used is inheritance, a way to eliminate redundant code by writing it once in a base class and then automatically reuse it in other classes that will inherit from the base one.

Suppose for example to have components with the same characterizing variables but different behaviours. One could define a base model with the variables declarations and then have inheriting models containing only the equations. This is done by using the keyword `extends`. We have already seen an application in the diode example in subsection 1.2.2, which extended `TwoPin` model.

```
partial model TwoPin "Component with two electrical
   pins"
     Real v "Voltage drop between the two pins (= p.v -
       n.v)";
     PositivePin p;
     NegativePin n;
```

```
equation
  v = p.v - n.v;
end TwoPin;
```

p and n are connectors representing the positive and negative pins of an electric component. The partial keyword thwarts the direct instanciation of the model, which can only act as a base class to be extended for example into a resistor or an inductor.

Every kind of Modelica class can be extended, and of course multiple inheritances are possible, either horizontally or vertically. But while inheritance helps avoiding code repetition, an excess can affect negatively the readability of the code.

Another one of the most used language features is the modification: through round parenteses next to an instantiation or an extends clause we can directly modify attributes. We already used many times (start = 0) in the examples to specify initial values, but one can also set other parameters.

```
record RecordWithParameter
    parameter Integer P = 5;
end RecordWithParameter;

model ModificationModel "A model in which modification
   is used to set a parameter"
    RecordWithParameter R(P = 10);
end ModificationModel;
```

In the example, we have a model with a record whose only parameter will end up being 10. We can also extend the record above in a new one with a different parameter value

```
record NewRecordWithParameter
    extends RecordWithParameter(P = 15);
end NewRecordWithParameter;
```

One last very useful possibility is the redeclation, which is applied whenever we want to change the class to which a specific instaciation is related.

Let's say that we have a model with a certain record Record1, and we want an equal model but with Record2 instead of it. Below the records are shown:

```
record Record1
    parameter Integer P1 = 1;
end Record1;

record Record2
    parameter Integer P2 = 2;
end Record2;
```

We can declare Record1 as replaceable inside the model, and then redeclare it with a modification.

```
model ModelWithRecord "A model in which modification is
    used to set a parameter"
    replaceable Record1 R;
end ModelWithRecord;

class RedeclarationExample
    ModelWithRecord M(redeclare Record2 R);
end RedeclarationExample;
```

In the new class model `M` won't have any instantiation of `Record1`. A commonly used pattern is to replace a class with an inheriting one, so that statements previously written do not become incompatible.

To summarize:

- In Modelica everything we define is a class.
- There are defferent types of classes with different purposes.
- All classes can be extended and modified.

Chapter 2

Discrete Events Systems Modelling and Simulation

Discrete event systems can be described through a sequence of ordered events, which happen at precise time instants and modify the system's state.

In many cases describing and simulating a system using this methodology involves a certain degree of abstraction. Usually, processes that can be naturally considered evolving in a continous manner are simplified by the time required for their completion.

Environments in which discrete time simulation results useful can be for example manufacturing plants or supply chains. One of the most popular cases of discrete event system is the Bank Teller, in which we consider a desk with an employee whose job is to serve customers. Customers enter in the office through a door and eventually wait for their turn.

This may seem an elementary example, and probably it is, but we can already notice occurrences that can be modeled as events: the arrival of a customer and its dismission once the desk service is finally erogated.

Taking a block-oriented perspective we can also identify some items and translate them into blocks: the office door can be viewed as a generator that fires events, which are accumulated and processed in a certain amount of time by a server, in this case the bank teller.

In fact, this case is usually modeled as Generator-Queue-Server system. Many simulation softwares offer graphical components to construct behaviours like this.

However, before taking in consideration the simulation aspect, it is worth to discuss a formalism to describe discrete event systems in general. We will cite just a pair of the most important:

- **Finite-State Automata**: also known as finite-state machines, are models of computation in which different states are reached through transitions. As the name suggests, the number of possible states can't be infinite, thus only a limited number of configurations is considered possible. Thus, typically they are specified as a set of states S, a set of inputs I and a transition function δ : S $\times$ I $\rightarrow$ S. Inputs can be activated by events that acted on the system. A usual representation consists in a state diagram containing state slots connected by transitions, as can be seen in image 2.1.

- **Petri Nets**: bipartite graphs in which two different types of nodes, called transitions and places, exchange tokens between places. Everytime a transition is activated, a selected amount of tokens is consumed from places that input the transition while another amount is immitted in the places downstream. A transition can be fired only if enabled, which means that there are enough tokens to be consumed. When multiple transitions are enabled at the same time, which one fires first is not determined. Thus, Petri nets are nondeterministic.
  Figure 2.2 depicts a Petri net with four places, represented by round circles, and two transitions, represented by rectangles.

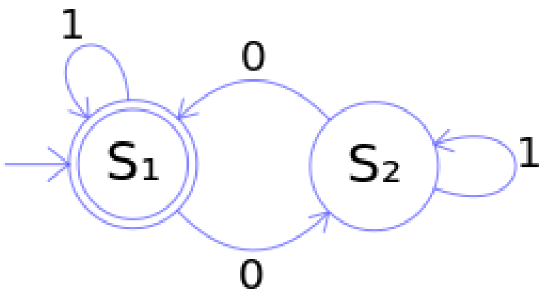


Figure 2.1: Example of a finite state machine consisting in two possible states, which change according to the two inputs 1 and 0

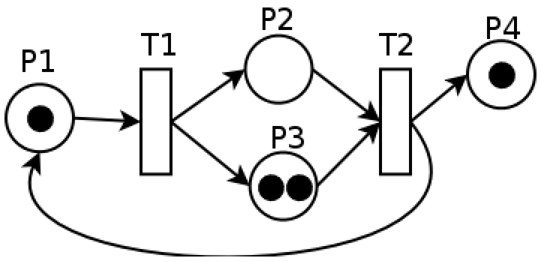


Figure 2.2: Example of a Petri Net

A more general approach, in which certain behaviours could be expressed more directly can be derived. In the following section a new theory for expressing discrete event systems will be introduced.

2.1 DEVS specification

Discrete EVent System Specification, from now on DEVS specification, offers a rigorous way to accurately model discrete event systems and components. It was invented by Bernard P. Zeigler in 1976 [Zeigler et al., 2000]. The original formulation has been extended many times into different versions, but a classical DEVS consists in the following structure:

$$M = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \tag{2.1}$$

where:

- X is the set of input values of the system,
- S is a set of possible states,
- Y is the set of output values,
- $\delta_{int} : S \rightarrow S$ is the internal transition function,
- $\delta_{ext} : Q \times X \rightarrow S$ is the external transition function, having $Q = \{(s, e) \mid s \in S, 0 \leq e \leq ta(s)\}$ as the total state set, in which e is the time elapsed since the last transition,
- $\lambda : S \rightarrow Y$ is the output function,
- $ta : S \rightarrow \Re^{+}_{0,\infty}$ is the time advance function that determines the lifespan of a state.

Basically, according to this formulation, every system is in a state s, which remains unchanged until either its lifespan, given by $ta(s)$, expires or an external event $x \in X$ occurs. In any of these two cases, a transition function is used to obtain the new state, that again will be mantained until an internal or external event occurs. Before an internal transition, the system will output $\lambda(s)$.

As an example, we can consider a generator that only outputs events in a periodic manner. As Bernard Zeigler proposes in his book, a generator system can be expressed in this way:

$X = \{\}$

$Y = \{1\}$

$S = \{\text{"passive"}, \text{"active"}\} \times \Re^{+}$, meaning that the state is composed by two values: one indicates whether or not the generator is active, and another one represents the time that needs to expire until next generation

$\delta_{int}(\text{phase}, \sigma) = (\text{"active"}, \text{period})$, with *period* a parameter belonging to the domain of real numbers

$\lambda(\text{"active"}, \sigma) = 1$

$ta(\text{phase}, \sigma) = \sigma$

with phase = { "active", "passive" }.

The input set is empty, while the only possible output is 1, that is emitted at every internal transition as defined by the λ function. The internal transiton function returns a new state which is always "active" and with a real variable, represented by the *period* value, that represents the desired amount of time to

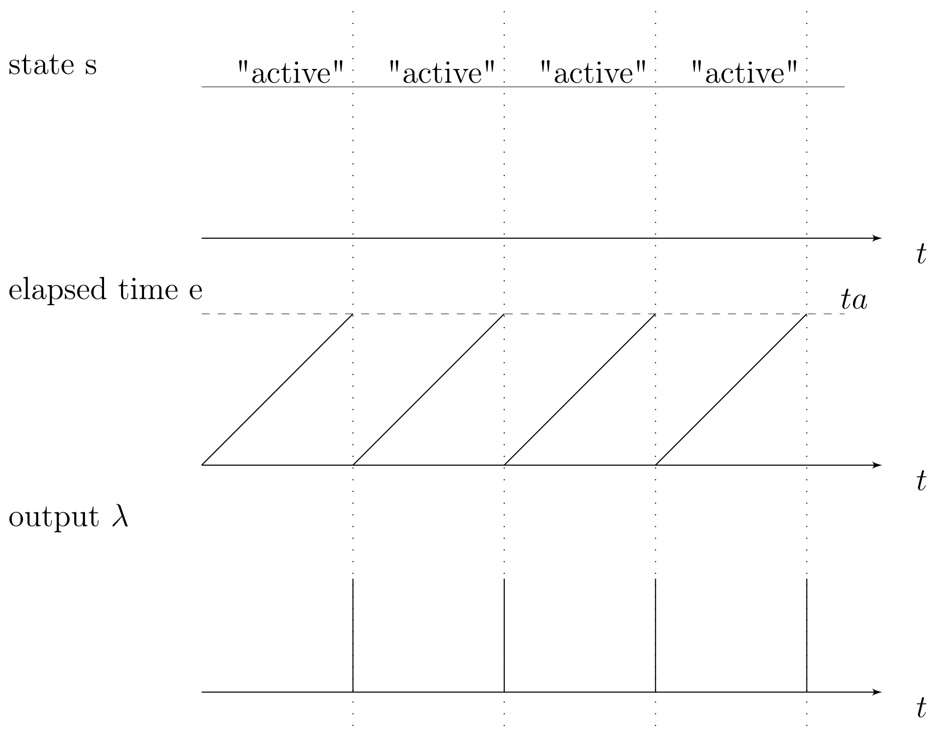


Figure 2.3

spend before the next transition. In fact that is the value returned by ta as the new state's lifespan. The system's variables as a function of time are shown in figure 2.3.

Given the DEVS structure, to define a model it is sufficient to configure its state, the possible inputs and outputs together with a collection of functions that determine its behaviour.

Functions allow also a fair degree of elasticity, having the possibility to contain algorithms and to introduce stochasticity inside their operation.

To make modelling easier the input and output sets can be expanded by introducing ports:

$$X = \{(p, v) \mid p \in InPorts, v \in X_p\}$$
$$Y = \{(p, v) \mid p \in OutPorts, v \in Y_p\}$$

In this way X and Y are sets of input/output ports and values. This results from a natural interpretation in which every discrete component has its own input and output ports. Every port p has its own set of possible values X_p or Y_p.

Still, we haven't talked about two events occurring simultaneously. The case in which a DEVS involved in an internal transition would receive an input involves a decision in which we have to give precedence to a transition function or erase one of the two events.

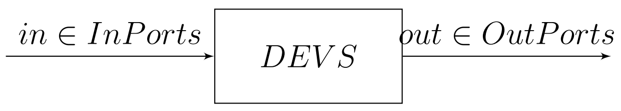


Figure 2.4: a DEVS with an input and an output port

Up to now we have only seen atomic DEVS, which are singular systems to be considered indipendently. In the next paragraph coupled DEVS will be described, and there we will see a possible approach to the question of confliecting events.

2.1.1 Coupled DEVS

A coupled DEVS is a system in which the previously described atomic DEVS are components whose ports are connected. Considering that a coupled DEVS has also its own external ports, its specification is the following:

$$N = \langle X, Y, D, \{M_d \mid d \in D\}, EIC, EOC, IC, Select\rangle \tag{2.2}$$

where:

- X is the set of input ports and values,
- Y is the set of output ports and values,
- D is the set of components' names,
- every M_d is a DEVS model,
- EIC is the set of connections from external inputs to component inputs

$$EIC \subseteq \{((N, ip_N), (d, ip_d)) \mid ip_N \in InPorts, d \in D, ip_d \in InPorts_d\}$$

- EOC is the set of connections from component outputs to external outputs

$$EOC \subseteq \{((d, op_d), (N, op_N)) \mid op_d \in OutPorts_d, d \in D, op_N \in OutPorts\}$$

- IC is the set of connections between components of the coupled DEVS

$$IC \subseteq \{((a, op_a), (b, ip_b)) \mid a, b \in D, a \neq b, op_a \in OutPorts_a, ip_b \in OutPorts_b\}$$

- $Select : 2^D - \{\}$ is the tie-breaking function, to be used in case of conflict between simultaneous events.

Having external ports, a coupled DEVS can be part of another coupled DEVS. Notice that no direct feedback loops are allowed, since no output port of a component can be connected to the same component's input.

In figure 2.5 a basic example is depicted, with a coupled DEVS containing, two DEVS a and b. Having just a pair of ports, EIC will comprehend just the connection between the coupled's input and the first DEVS' one, while the second

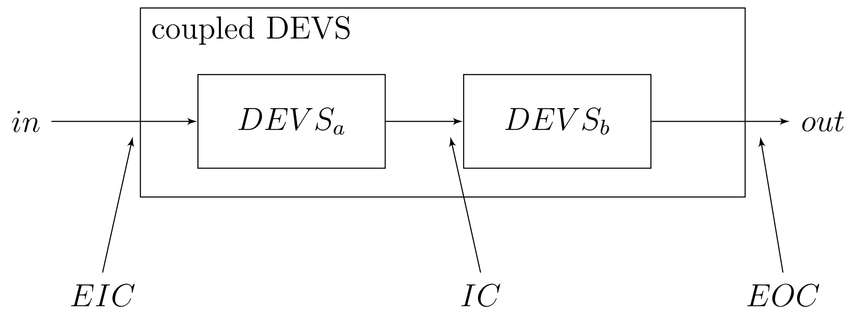


Figure 2.5: a coupled DEVS containing two DEVS elements

DEVS having its port connected with the out will be *EOC* only connection. The connection between the two DEVS will be contained in the *IC* set.

To make sure no conflict rises with simultaneous events, in which a DEVS could have to trigger an internal transition while receiving an external event, the *Select* function is set so as to define precedences.

2.1.2 Parallel DEVS

Parallel DEVS specification is an extension of the classic DEVS one. It was designed so that, when simulating coupled models, parallel processing could be allowed. While in classic DEVS contemporary internal events have to be activated sequentially to avoid conflicts, using the *Select* function, in parallel DEVS they can be executed concurrently. The structure of an atomic parallel DEVS is:

$$pDEVS = \langle X_M, Y_M, S, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta \rangle \tag{2.3}$$

where:

- $X_M = \{(p, v) \mid p \in InPorts, v \in X_p\}$ is the set of input ports and values,
- $Y_M = \{(p, v) \mid p \in OutPorts, v \in Y_p\}$ is the set of output ports and values,
- S is the set states,
- $\delta_{int} : S \rightarrow S$ is the internal state transition function,
- δ_{ext}: $Q \times X_M^b \rightarrow S$ is the external state transition function,
- δ_{con}: $Q \times X_M^b \rightarrow S$ is the confluent transition function,
- λ : S → Y is the output function,
- ta: S → $\Re^+_{0,\infty}$ is the time advance function that determines the lifespan of a state,
- $Q = \{(s, e) \mid s \in S, 0 \leq e \leq ta(s)\}$ is the set of total states.

The biggest addition is the introduction of a third transition function, the confluent one, which acts in case an internal and an external event happen at the same time. Another difference is the presence of the so called bag X_M^b, which is a set with multiple possible occurrences of the elements from X_M, allowing more external events to arrive in the same instant.

Thus, in a coupled DEVS made with parallel components there's no need to specify what may happen in case of event collision, since the formulation itself specifies how to manage them.

In a coupled p-DEVS the time goes on until the first internal transition is encountered. At that point all the components experiencing an internal transition are considered imminent and put into the IMM set. Without a select function, their λ functions will all be actived, and the components receiving an event will considered influenced, and so put into the set INF. The transition functions for every atomic DEVS will be determined with the following operations:

$$\begin{aligned} INT &= IMM - INF \\ EXT &= INF - IMM \\ CONF &= IMM \cap INF \end{aligned}$$

Let's try to conceive the Bank Teller case as a coupled p-DEVS: We can have two atomic DEVS: the door and the Bank Teller. If we consider the port as a generator of signals, which models the frequency at which people arrive, every time a customer enters, the door belongs to the IMM set. So the Bank Teller, having received an output from the door, becomes part of the INF set. Thus, the door will experience an internal transition while the Bank Teller will experience an external one.

In case a customer enters exactly when another is satisfied, both our DEVS would be considered IMM, since they would both trigger an internal event. Then, since the Bank Teller would already receive a new customer at the end of its queue, it would belong to the INF set too, being influenced by the door's DEVS. In the end, the door would be subject to an internal transition while the Bank Teller would execute a confluent one. Then it would go on until the next internal event happens.

Now, from the point of view of the modeler, we are many steps ahead from the two formalisms defined at the begininning of the chapter. Using DEVS one can describe a system by just defining a group of heterogeneous discrete variables aimed at representing its state and determine its evolution through functions.

For example a machine working on a piece can be modeled with a state indicating if the process is active or not. After having completed its job, an internal transition can move the state to idle, while the production of the finished item and the request of another raw one can be realeased as output. When not operating, the next transition would be scheduled at infinite time. Then, the arrival of a new part in input would trigger an external transition, setting the state to active again, and the time advance function would return the time required for the job to be terminated.

In case the processing terminates and simultaneously a raw part arrives, the confluent function would acquire the duty to handle the priorities of those two events. For example, after having thrown the output of a job completed, the confluent transition function could be defined in the following way:

$$\delta_{con} = \delta_{ext}(\delta_{int}(s), e, X_M^b) \tag{2.4}$$

so as to behave like an external transition happened right after the job completion, since δ_{ext} uses as previous state the result of δ_{int}. Thought, the options to handle the priority of simultaneous events are many more, since they can be defined directly inside the confluent function, without having to specify the *Select* function for every possible case.

2.2 DESLib

Given that the aim of this thesis is to provide a package to simulate discrete event systems in OpenModelica, it is worth to describe an already available Modelica library for this task: the DESLib, developed by Victorino Sanz [Victorino Sanz, 2010].

The package was suited for the Dymola environment, which in contrast to OpenModelica is proprietary, and provides features to define parallel DEVS, generate random numbers and to replicate components of the Arena software, one of the most valid tools for simulating industrial plants or other discrete event systems.

It is articulated into 4 main parts:

- RandomLib: aimed at generating random numbers and variates using an algorithm belonging to the family of the Combined Multiple Recursive Generators.
- DEVSLib: defines parallel DEVS formalism implementation.
- SIMANLib: package containing blocks for process-oriented simulation to reproduce the functionalities of SIMAN, a language at the base of the Arena software.
- ARENALib: defining components to emulate the ones of Arena, built with SIMANLib's blocks.

Both SIMANLib and ARENALib components are derived from the DEVS defined inidse DEVSLib.

OpenModelica provides the version 1.6.1 among the free libraries available at installation, but we will take in consideration the version 1.9, which is the most recent available on the author's website.

Since it is an articulated package, instead of presenting the library in its entirety, we will focus on analyzing the most interesting aspects of this work: the messages mechanism and the DEVS implementation, underlining ideas that were taken as an inspiration in this thesis' development.

One of the most important issues when setting up discrete-event components is how they should communicate. In a process-oriented simulation entities, which are abstractions representing discrete items, flow through a series of blocks. They start from a generator, that introduces them into the system, and in the end terminate in a block that displaces them out of it.

As the author points up in his work, many Modelica's inherent characteristics pose some difficulties to the realization of a mechanism for transporting entities' informations. In fact, using connectors as the main communication channel, the amount of data we can transport is limited their variables' number. In fact, in Modelica the absence of variable size data structures does not allow to store an undefined number of entities.

In the author's intentions DEVS should exchange informations if form of messages, described as "impulses of structured data" [Victorino Sanz, 2010, which should be transmitted from one block to the other through connectors. Thus, a message could contain the entity's data, plus eventually other information relevant to the process, like for example the arrival time at the block.

The requirements of the messages passing mechanism are described in the author's thesis work:

- Multiple typologies of connections allowed: 1-to-1, 1-to-many and many-to-1. Thus, for example, there should be the possibility to connect the output of a queue to many generators to accumulate entities from many sources.
- There shouldn't be time delays due to the transmission procedure, that should be instantaneous.
- Eventually, more than one message can be transmitted through the same port.
- A message should be able to contain any kind of information, and those informations could change in every transmission if needed.

To implement this mechanism different options are evaluated.

The first one is to set the message's informations as variables of a given connector. This way, the block at the other end of the connector will be able to directly access the message content.

In case of multiple messages, a communication system based on a protocol similar to the one of TCP would handle the transmission. In this setup, the sender would wait for an ACK from the receiver, which would send it after having acquired the message's informations. The chain would be repeated until all messages are successfully passed.

The procedure would be handled using the Modelica's event iteration, which would set the semaphores to enable and disable the change of connector's variables by the sender.

Naturally, for every different kind of message there would be a different connector. Still, the storage of an undefined number of messages represents an issue, since an excessive amount of data can't be handled by fixed-side arrays.

However, this possible solution is the only one which stores data using Modelica's constructs.

The other two implementation approaches proposed basically externalize the messages' contents outside of the Modelica scope, and then retrieve them when necessary, all with the use of apposite functions.

They are very similar, since they both employ a connector with two variables:

- a reference to the external data structure used to transmit the messages, which is initialized by the receiver at the beginning of every simulation. This reference will be an effort variable, so that it is avalable to all the connected senders, and they can use it to access the external storage to write messages into it

- a flow variable, augmented by the senders anytime they have written a new message in the external data structure. Because of its change, the receiver is aware of the presence of new messages.

What differentiates the two approaches is the external data structure. The options presented to contain the message's informations are:

- Text file storage. The sender writes the message in a text file, whose reference would be set by the receiver. Multiple messages can be easily written in the same file. The drawbacks of this choice are the degradation of performance, due to the frequent I/O operations on files, and the fact that for every different kind of message the writing and parsing functions should be modified.

- Dynamic memory structure: in this implementation every message would be represented by a `struct` written in C programming language. Those structs would be elements of a dynamic data structure, and essentially the transmission would consist in the attachment of the struct to the data structure set by the receiver.
  With respect to the text storage, new typologies of messages would require just to change the original C struct, while the functions could remain unchanged, since practically to manage the messages only their memory addresses are used.

This last approach is the one finally implemented in the DESLib, and it employs a doubly linked list, instantiated by every receiver, who would take the list's physical address and then express it as a Modelica `Integer`. This integer value would be broadcasted through the connector's effort variable to all the possible senders.

Below the connector representing an input port can be seen, with a flow variable whose augmentation announces the transmission of a message and the effort variable. In Figure 2.6 a representation of the communication mechanism is shown, being EQueue the instantiation of the list on which the sender adds messages, implemented as C language structs.

```
connector inPort "Input port for receiving messages"
    flow Real event;
    Integer queue;
end inPort;
```

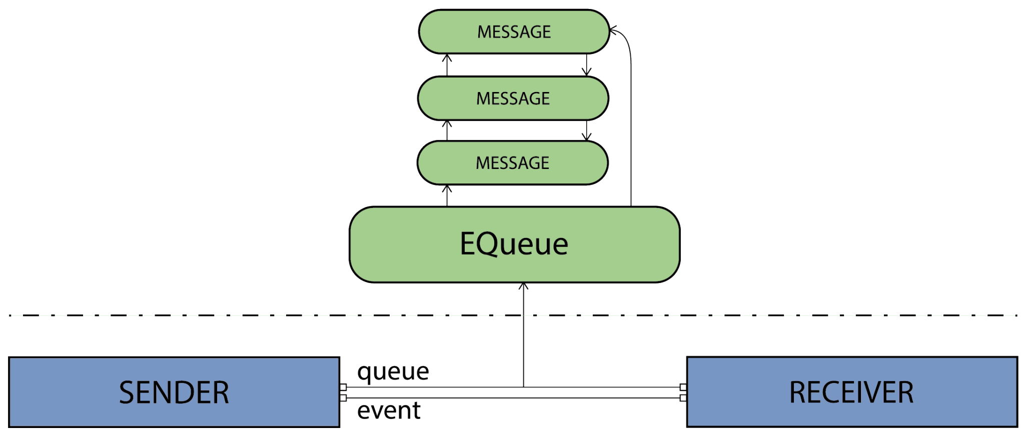


Figure 2.6: Symbolic representation of messages handling underlying data structure. Blue blocks represent messages' sender and receiver, while the C list and the contained messages' structs are in green

To handle the transitions required by the DEVS formalism a partial model representing the atomic parallel DEVS is declared. Every other model from DESLib is an extension of this root class, and thus inherits the event handling system characterizing atomic DEVS models.

The root class has an equation section dedicated to triggering transitions, which apply their changes to state and outputs in apposite algorithm sections.

Conclusions and takeaways

We have seen some of the features that characterize DESLib. The fact that it was developed for Dymola and that it contained many external C functions led me to build a new package, to explore the possibility of deploying discrete-event simulation in pure Modelica using the OMCompiler provided by OpenModelica.

Still, some aspects of this work have been taken into account. Letting the blocks exchange events using connectors is a good idea, even if it may be considered obvious given how many commercial tools allow to attach the ports of their components when modeling the system thorugh a graphical interface.

The use of the DEVS formalism to define the simulation components enables a huge amount of elasticity, since being able to follow the specification in Modelica would allow any user to implement every possible block compliant with the DEVS formalism. Thought, the DEVS atomic model provided in the DEVSLib is tightly bonded with the message communication system. One of the objectives of MODES will be to provide a starting DEVS block that would not restrict future development to a specific communication system.

One last very important matter is the event handling system's nature. Choosing to trigger transitions using equations introduces in the models a degree of acausality, taken in charge by the compiler. In this situation, the determinism of certain series of transitions could be lost. Even if a deterministic order of transitions is still attained, in certain situations, with many events causing one another, it can be very difficult for the modeler to preview what will happen once the event iteration starts.

Together with this, the possibility of algebraic loops arising with certain models is still present. As we will see at the beginning of the next chapter, to run certain simulations it is sometimes required to introduce calculation delays to help the numerical solver, and certain models can be rejected by compilers not provided with specific features to handle algebraic loops.

DESLib provides a specific componenent to break algebraic loops, but there's no guarantee that the delay in event iteration will not affect in some way the sequentiality of a desired event chain.

In this thesis' work a special focus was dedicated to define a causal way of handling events, so that, regardless of the communication mechanism implemented, the different phases of the DEVS models simulation can be explicitly defined and tracked by the modeler.

Chapter 3

MODES: A New Modelica Package for Discrete Events Systems Modeling and Simulation

The purpose of this thesis work is the development of MODES (MOdelica Discrete Events Systems), a Modelica library to support discrete event simulation in the OpenModelica environment. Since every event is fired by one specific element of the system, the nature of discrete event systems is inherently causal. Because of this, the work is essentially based on the causal constructs of the language, such as blocks with input-output connections and algorithm sections.

To describe systems, it has been decided to follow the parallel DEVS formalism. This resulted in the need to guarantee a proper simulation of DEVS systems inside the OpenModelica environment, and then providing a way to express Modelica blocks using the DEVS specification.

Those requirements are satisfied inside the `Atomic_PKG` package, which provides a partial block that can be used to define parallel DEVS atomic models.

Another issue that had to be considered consisted is the representation of the events. We have seen in DESLib a way to express events in form of messages to be exchanged between models. Looking for a simpler solution, in this work events have been represented using changes in value of discrete variables, even though other approaches are left possible.

Along with the `Atomic_PKG` package and the `Template_PKG` package, which provides a block that shows the algorithm sections to be declared to implement new DEVS models, a library named **EventsLib** is present, which allows to perform discrete events simulation of systems in which events go through the blocks following a process-oriented approach.

3.1 DEVS simulation

The first task to consider when developing a library like this is the implementation of a simulation procedure for coupled DEVS systems. For simulating coupled DEVS a common algorithm can be one which, among the atomic components of the system, selects the ones whose internal transition is next to happen, and moves the simulation time to that instant. At that point, their outputs are propagated through their ports to connected DEVS.

Once the output is released, an atomic model can either be about to execute an internal transition, have received external events, both or none of the previous. According to these conditions, the algorithm knows whether or not any DEVS model has to execute a transition function, and also which one depending on the combination of events happened.

After transitions are executed, lifetimes of new states are determined and so the next instant for internal transitions can be chosen.

This algorithm is quite easy, and we can let the Modelica solver detect the time instants for internal transitions using state events. Still, it may be hard and even excessively complicated to design an element that coordinates the transitions.

Instead, a decentralized approach, in which every DEVS model autonomously sends its output values and executes its transitions, seems more easily attainable.

Ideally, the transitions should be synchronized among all the models of the system, so that in case new states with zero lifetime are reached and immediately broadcast their new outputs, the next transitions will be still synchronized. This would make the modeler's job easier when previewing what may happen with many models experiencing instantaneous transitions.

By using algorithm sections, Modelica models are able to perform parallel coupled DEVS simulation. But before presenting the mechanism used in this library, it is essential to discuss a typical problem that may arise when simulating systems: algebraic loops.

Algebraic loops

In the following, we will present a basic problem that can arise when simulating systems, and how to prevent it from happening. This short insight can be useful to understand some characteristics of this library's code, especially the use of the `pre()` function. As we saw in Section 1.2.2, the `pre()` function references the value at the past event iteration, and thus can be used to delay certain logical conditions. Remember, not in time, but in the event serviving sequence.

A very basic algebraic loop may arise when simulating control systems with feedback loops: when the reference variable in input to the block is modified by the output of the same block, an equation is required to solve both for the input and the output.

Some compilers are able to figure out many of these cases, even if it comes with a computational cost, but sometimes this situation is cause of errors. To explicitly avoid them, a usual solution consists in the placement of a numerical delay inside the simulation, so as to use the value calculated at the previous solver step instead of the actual one.

To see how in Modelica a problem like this can be solved, we resort to a really basic and simple model composed by two blocks: an already filled queue and a server asking for events. The server requires a certain amount of time to complete a job and, after having finished it, it queries the queue for a new event to be processed, which the queue should immediately send.

The two blocks' code can be seen below:

```
block Queue "Storage that releases events when queried"
    output Integer event_signal (start = 0);
    input Integer request_signal;
    Integer n_in_queue "number of events the queue is
       storing, begins at 10";
    Boolean request = change(request_signal) "boolean
       that triggers the requirement";
initial algorithm
    event_signal := 0;
    n_in_queue := 10;
algorithm
    when request then
    //If queue is not empty after a request
    //send an event and remove it from queue
        if n_in_queue > 0 then
            event_signal := event_signal + 1;
            n_in_queue := n_in_queue - 1;
        end if;
    end when;
end Queue;

block Server "Processes events and asks for new ones"
    input Integer event_signal "events to be processed
       received by the server";
    output Integer request_signal "integer used to
       query a new event to process";
    parameter Real timeadvance = 2 "time required to
       process an event";
    discrete Real next_TS "timeinstant of next job
       completion";
    Integer elements_processed "elements already
       processed by the server";
    Boolean arrival = change(event_signal);
initial algorithm
    next_TS := 0;
    elements_processed := 0;
    request_signal := 0;
algorithm
    when time > next_TS then
    //when a process is finished
    //augment the elements processed and require a new
       event
        elements_processed := elements_processed + 1;
        request_signal := request_signal + 1;
    end when;
    when arrival then
    //when an event is received set the time for the
       next job completion
        next_TS := time + timeadvance;
    end when;
end Server;
```

In these two blocks events are triggered either by the state event `time > next_TS`, or by a change in value of integer variables. When integers with the same name are connected, `event_signal` and `request_signal`, every augmentation of `request_signal` by the server is interpreted by the queue as a demand for a new event. Respectively, by augmenting `event_signal` the server is informed that a new element ready to be processed has arrived.

While it may seem natural to directly connect boolean variables to represent events, increasing steps in discrete values will be used to represent events in MODES. In fact, by augmenting an integer value not only we can report an event, but also transmit another information equal to the variable's growth. So, even if this is a naive example, we can already see the use of a `change()` function to trigger events.

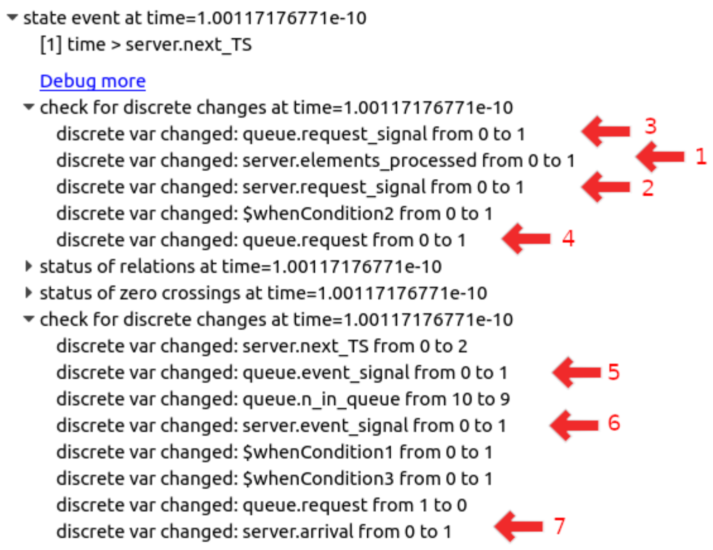
If these two blocks are simulated, on version 1.13.0 $dev - 1619 - gdf67167$, the one used to develop this library, a flattening error occurs. Occasionally in similar situations the flattening procedure is completed, but still other errors arise during the code translation phase.

This because an algebraic loop occurs when in the same iteration step we are trying to solve for the variables that caused the event and the ones that are affected by it.

To avoid this problem and obtain a successful simulation it is sufficient to use a `pre()` inside any of the `when` conditions that are activated by an event external to the block. In our case, we chose to modify the queue's algorithm section in the following way:

```
algorithm
    //request has been substitued with pre(request)
    when pre(request) then
        //If queue is not empty after a request
        //send an event and remove it from queue
        if n_in_queue > 0 then
            event_signal := event_signal + 1;
            n_in_queue := n_in_queue - 1;
        end if;
    end when;
end Queue;
```

In this way, whenever the server finishes its job, `request` augments and the queue is queried. But it is at the next iteration step that the queue's `when` condition is activated, and only at that step the request is satisfied. Figure 3.1 shows the simulation output at the first state event with flag `"LOG_EVENTS_V"` activated, and it can be seen that the system's solution is decomposed in two different steps, one for triggering the request, and one for its satisfaction.



```
▾ state event at time=1.00117176771e-10
    [1] time > server.next_TS

    Debug more
  ▾ check for discrete changes at time=1.00117176771e-10
      discrete var changed: queue.request_signal from 0 to 1
      discrete var changed: server.elements_processed from 0 to 1
      discrete var changed: server.request_signal from 0 to 1
      discrete var changed: $whenCondition2 from 0 to 1
      discrete var changed: queue.request from 0 to 1
  ▸ status of relations at time=1.00117176771e-10
  ▸ status of zero crossings at time=1.00117176771e-10
  ▾ check for discrete changes at time=1.00117176771e-10
      discrete var changed: server.next_TS from 0 to 2
      discrete var changed: queue.event_signal from 0 to 1
      discrete var changed: queue.n_in_queue from 10 to 9
      discrete var changed: server.event_signal from 0 to 1
      discrete var changed: $whenCondition1 from 0 to 1
      discrete var changed: $whenCondition3 from 0 to 1
      discrete var changed: queue.request from 1 to 0
      discrete var changed: server.arrival from 0 to 1
```

Figure 3.1: Fraction of output from the simulation of a model containing the queue and server decribed in the algebraic loop example. Note that during the event iteration the time stays constant. At the first event iteration step the server finishes a lavoration and elaborates a request (1 and 2). Consequently, the queue receives a request (3 and 4). Due to the delay in event iteration, only at the next time step the queue will send the event and the server will receive it (5, 6 and 7).

In other words, it can be said that the `pre()` function can "help" the compiler in the job of solving the discrete equations of the system. In the following we will see that its role will be fundamental to correctly articulate the event chains required to simulate DEVS systems. For a quick recap:

- Algebraic loops can arise in a simulation when the solver finds an equation containing both a variable and its effect on the system.
- Sometimes a delay can be used to help the solver handle the algebraic loop.
- In Modelica's algorithm sections the use of the `pre()` function can prevent flattening errors caused by algebraic loops.

3.1.1 Transition handling of DEVS

The above explanation about algebraic loops suggests that the different phases of the algorithm presented at the beginning of the section may be divided into different steps of an event chain.

In a Modelica simulation the numerical integration goes on until an event is encountered. If every atomic DEVS model holds the value of the next internal transition time, the Modelica simulation can fire a state event everytime the nearest transition time is encountered. When it happens, the procedure to execute the DEVS transitions is divided in five steps of event iteration:

1. At first all the DEVS models about to experience an internal transition activate a boolean variable to keep track of the fact that an internal event happened.

2. Next, they elaborate their output and propagate it using connectors to the DEVS models they are supposed to influence. Thus, also at this step, all the receiving models are aware that they will experience an external transition, and they take note of it in another boolean variable.

3. Then, DEVS atomic models who received an input, acquire it and store it. Now, every activated component of the system determines the kind of transition it will go through. For example, models who experienced an internal event at the first step, but also got external input at step two, will have to execute a confluent transition. Of course models who are not touched by this event chain stay idle and none of their variables changes.

4. At the fourth event iteration step, transitions are executed and states affected by them are changed. Models experiencing external events have already acquired the inputs generated at the second iteration step, and so they can use them to determine the new state.

5. Finally, new lifetimes for states are calculated and also new time instants for next internal transitions can be obtained. Thus, the procedure will repeat itself when the nearest internal transition will be reached.

In Figure 3.2 a Petri Net representing the procedure every atomic DEVS goes through when executing transitions can be found.

By expanding these operations in a series of event iteration steps we prevent possible algebraic loops and still have synchronized transitions between all the atomic DEVS without the need of a coordinating component.

Notice that an atomic DEVS that only has to experience an internal transition will wait two event steps after having acknowledged it: the first to fire outputs and to make sure it is not receiving any external input, and the other to let receiving models acquire their input.

In Figure 3.3 the different phases are graphically represented through a coupled DEVS consisting of three atomic models.

One possibility that has to be taken into account is the one in which a new state has a lifespan of zero duration. This involves that the next internal transition will be fired inside the same event iteration chain, during the same simulation time instant. Sometimes, many confluent transitions may keep being triggered in sequence by a long lasting throughput of external events.

A chance for this circumstance to happen is the one of an empty queue, which has a list of pending requests for events by a group of servers, and that has zero time required to send the events when the possibility to satisfy a request arises. This may be true if we approximate the transit time from queue to server as instantaneous, or because the delaying action is handled by another DEVS model. In this situation, everytime an event arrives to the queue, after its external transition the new state would have zero lifetime, and the next internal transition would output the event to one of the servers. But if in the same phase another input arrives, the subsequent confluent transition would again trigger an immediate internal transition to send the new event. This chain would keep going on until incoming events stop arriving.

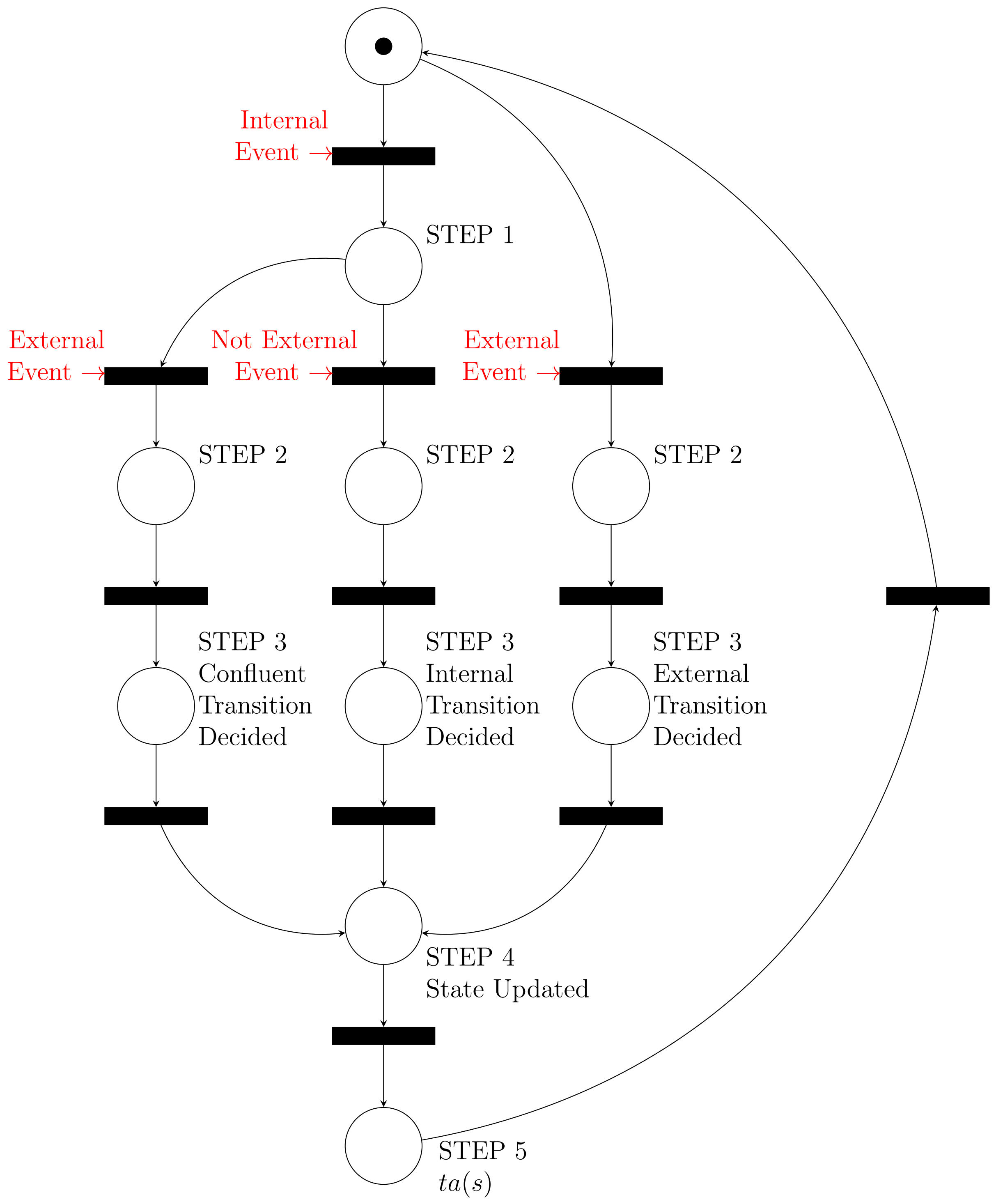


Figure 3.2: Petri Net describing the DEVS simulation event iteration. In red are underlined the possible events happening to a block.

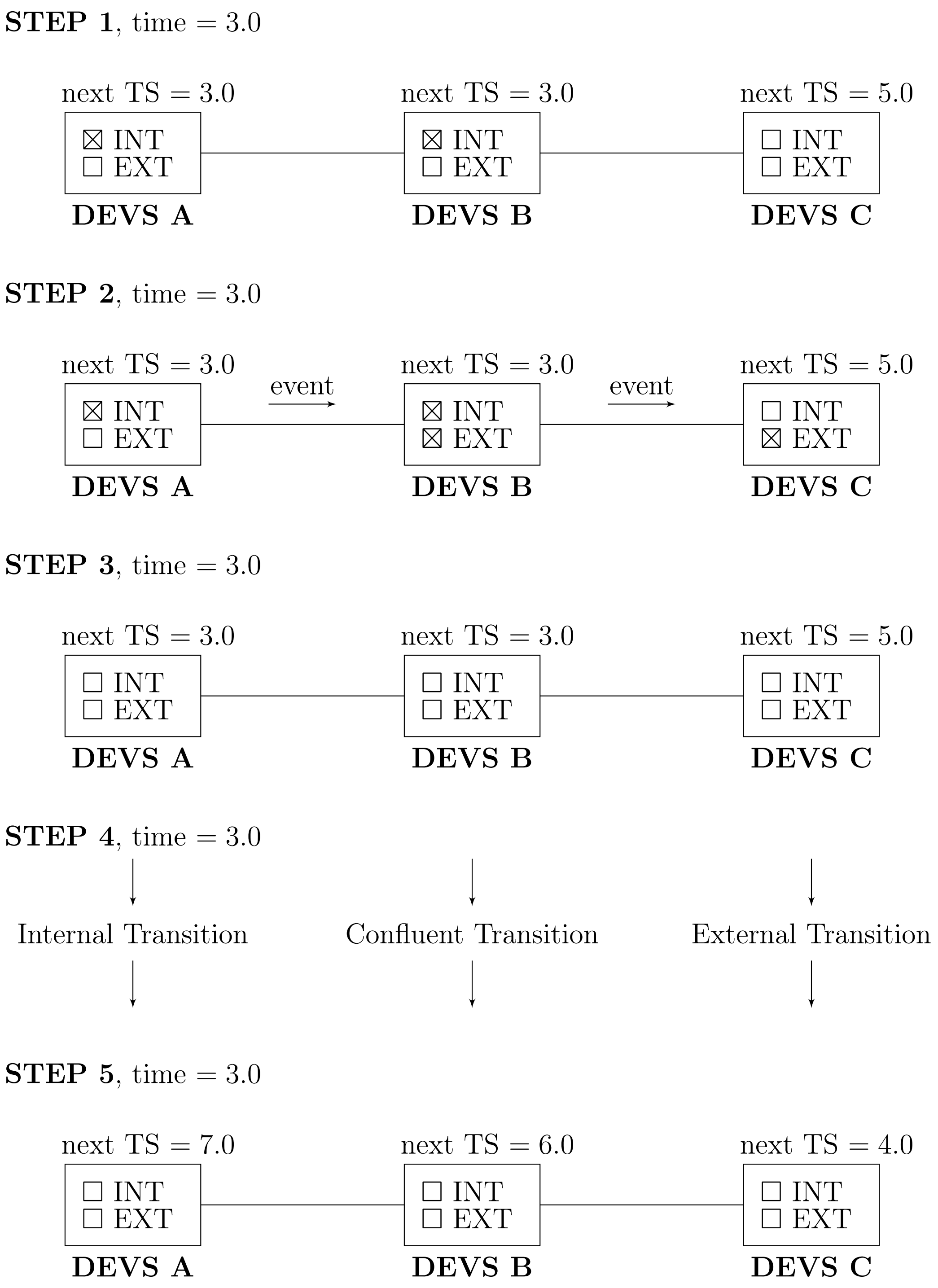


Figure 3.3: Graphic representation of the five steps of DEVS simulation on a coupled DEVS composed by 3 blocks A, B and C. In the example, A and B are about to experience an internal transition and send events to, respectively, B and C. Because of those external events, C takes into account that an external transition needs to be activated, while B will use a confluent transition to handle both internal and external events. Thus, after having determined their transitions at the third step they will execute them at the fourth. At step five they will thus update their next scheduled internal transition time instants according to the new states.

It may look as a strange occurrence, having many inputs coming staggered in different simulation phases, and not simultaneously together. Yet, it has to be taken into account, since complex systems composed by many atomic models can show this kind of behaviour.

Also, one cannot exclude that a model may have to execute a sequence of internal transitions, one after another. This last one is a hazardous option, since if badly planned may lead to infinite loops, which would break the simulation.

Still, the event managing mechanism should account for these possibilities, and guarantee that they are executed properly. It will be up to the DEVS modeler avoiding problems in case he decides to resort to consecutive internal transitions.

Now that we have discussed the procedure to handle the simulation of DEVS models in Modelica, we can concentrate on its implementation. Being decentralized, every atomic model should have its own code segment that controls transitions and other operations, like event sending.

Inside it, the following variables are declared:

- Elements that effectively track events. `next_TS` indicates, in seconds, the time instant at which the next internal transition is scheduled. A simple expression like `time > next_TS` would let the simulation trigger a state event when that moment is reached. This condition is evaluated only once inside the block, and its value is stored inside the boolean `internal_event` . `external_event` is an array of boolean conditions that represents the arrival of inputs. Their acquiral and processing are not related to this variable, which only has to account for reporting that an external event is happening. Its dimension is undefined, every inheriting atomic model will have to explicitly declare it accounting for its number of ports.

```
discrete Real next_TS;
Boolean internal_event = time > next_TS;
Boolean external_event[:];
```

- A set of variables to keep track of the kinds of events arrived. At the third step of the procedure shown at the beginning, these variables will be evaluated to choose the transition to be executed.
  `internal_transition_planned` is an array of two elements. This because, after an internal event happened, before triggering the internal or confluent transition two iteration steps have to pass. So the first element of the array, `internal_transition_planned[1]`, will be marked `true` when `time > next_TS`, and then at the second iteration step `internal_transition_planned[2]` will be marked too. If an external event is detected, `external_transition_planned` is used to take note of it.

```
Boolean internal_transition_planned[2] (start = {
   false, false});
Boolean external_transition_planned (start = false)
   ;
```

- Three variables to define the transition that will occur and trigger it. They are set at the third iteration step according to the value of variable `internal_transition_planned[2]` and `external_transition_planned`. At the fourth iteration step, the selected boolean will trigger the transition procedure to update the state.

```
Boolean internal_transition (start = false);
Boolean external_transition (start = false);
Boolean confluent_transition (start = false);
```

- A variable that needs to be true right after a transition, to trigger the updating of `next_TS` based on the new state at the fifth and last step of the simulation chain. That will be `pre(transition_happened)` and it is directly handled by an equation which delays its rise by one step with respect to one of the transition decision. The use of the `pre()` will make this variables rise at the fifth step of event iteration.

```
Boolean transition_happened = pre(
   confluent_transition) or pre(external_transition
   ) or pre(internal_transition);
```

- In the end the main elements used to manage the transitions are:
  - Variables `internal_event` and `external_event` to detect the happening of an event. External events are triggered in an array, since input messages could arrive from more than one port.
  - The two different typologies of events occurred are recorded inside `internal_transition_planned` and `external_transition_planned`. `internal_transition_planned` is an array of two elements to delay the happening of an internal transition after an internal event.
  - Three transition triggers are present, one for every possible transition.

The variable `next_TS` is determined by an algorithm section that uses the state variables to calculate it after every transition, exactly as the $ta(s)$ function expressed in the formalism. The elements of the array `external_event` are handled in a different manner by every atomic DEVS model according to its possible inputs.

The `change()` function will be applied on its input connectors' variables to assign `external_event` values. Still other options are available, like the use of an `edge()` function on some booleans. What is required is that whenever an external input arrives to the DEVS atomic block, a corresponding element of `external_event` array turns to `true`.

Having properly set these variables, the algorithm section that manages the three transition triggers is composed by a `when` condition with two additional elsewhen branches. The first one is used to set the planned transitions according to the evens happened, and its condition is an array of booleans [1]. It is shown below:

[1]cat is a Modelica function used to concatenate arrays on a specified dimension

```
when cat(1, {change(next_TS), internal_event, pre(
    internal_transition_planned[1])}, external_event)
    then //Whenever an event happens or there was an
    internal event at the previous time step this when
    is entered
        if pre(internal_transition_planned[1]) then
            internal_transition_planned[2] := true "At
               first, if the internal event was at the
               previous time step update the second
               flag";
        elseif internal_event then
            internal_transition_planned[1] := true "
               Otherwise, in case of internal event,
               fire the first flag";
        end if;
        if max(external_event) then
            external_transition_planned := true "In
               presence of an external event";
        end if;
        internal_transition := false;
        external_transition := false;
        confluent_transition := false "In case an event
             is triggered but a transition occurred at
           the previous iteration step, set to false
           all transition triggers";
```

Whenever the time crosses `next_TS`, the when condition is entered since `internal_event` rises and `internal_transition_planned[1]` turns to `true`.

This happens during the first step of the event iteration. At the second one, outputs are calculated and sent by another algorithm section whose `when` condition is now triggered by `pre(internal_transition_planned[1])`. So, having `external_event` array properly set, atomic DEVS that receive an input enter the same condition and modify `external_transition_planned` value. This because the function `max(external_event)` returns `true` if anyone of the array's elements is, and so the `if` branch devoted to assigning `external_transition_planned` is entered.

Also, still in the second phase, any model who previously experienced an internal event, now enters again in this `when` because of the `pre(internal_transition_planned[1])` condition present, so that `internal_transition_planned[2]` can be activated.

To make sure that transitions can be correctly triggered again in case there has already been a transition at the previous step, this `when` condition flattens to `false` the booleans `internal_transition`, `external_transition` and `confluent_transition`. A situation like this should never present itself, since events are supposed to happen and be detected at steps one and two, not four.

In case a component not following the five steps is inserted in the system, the mechanism pulls down the three transition triggers and starts again the rise of the planning booleans. Recall also that, differently from equations, whenever you set to true a variable in an algorithm, that variable won't come back to false unless you command it explicitly in another assignment.

Previously we cited the possibility of multiple consecutive internal events. In this case, the `when` wouldn't be reentered by using only the `time > next_TS` condition, since it would have been already activated many iteration steps ago. Thus, to virtually let multiple internal events happen the condition `change(next_TS)` is present at the beginning of the `when`. To cause an immediate internal event it is sufficient to modify `next_TS` while mantaining it below the current `time` value.

Of course everytime `next_TS` is updated this `when` condition is entered, but since normally the new value is set to be ahead of time, no change in `internal_transition_planned` variables happens.

This additional condition could seem like a forcing, and seeing `next_TS` values unrelated from reality may cause confusion when reading the intermediate iterations after a simulation. To make things clearer, when needing for an immediate internal transition, I used to set the `next_TS` value with decreasing negative numbers, which are clearly unnatural, since negative time instants hardly make sense inside a simulation, but give the possibility to recognize at a first glance that the occurring transition was required to be immediate.

At the beginning of the third iteration step, if outputs have been propagated and received synchronously by all atomic models, we shouldn't enter again in that `when` condition.

Now, the following `elsewhen` can be entered:

```
elsewhen not max(external_event) and (pre(
  internal_transition_planned[2]) or pre(
  external_transition_planned)) then
   if external_transition_planned and
      internal_transition_planned[2] then
       confluent_transition := true "in case both an
         internal and an external event happened use
         the confluent transition";
   elseif external_transition_planned then
       external_transition := true;
   elseif internal_transition_planned[2] then
       internal_transition := true;
   end if "An if else tree decides which transition
      should be triggered";
   internal_transition_planned := { false, false};
   external_transition_planned := false "At the end,
      all transition planning variables are put to
      false";
```

Without new events arriving in input and with a new transition planned, the above section can be activated. Depending on the events happened the proper transition is triggered, being it confluent, external or internal.

Especially, internal transitions are fired only when (pre(internal_transition_planned[2]) happens to be true, so that every atomic DEVS is delayed enough to let the receiving models acknowledge their condition.

The code segment is not executed in case an external event is coming again from the same input port. If all the connected components share the same nature of DEVS models this won't happen, but in case any uncompliant block is inserted somewhere inside the model, an input may arrive at a different iteration step from the one planned. In this scenario, transitions are not triggered as long as events are coming. In this case synchronicity between transitions would be lost, but the simulation would still go on.

In other words, every atomic DEVS triggers and receives events indipendently. When no more events are received, it fires its transition. If all the connections are among uniform DEVS models sharing the same transition routine, then synchronicity will be achieved.

So, if DEVS models are connected to other DEVS models only, this when condition is entered at the third step of the DEVS simulation. Now, every model that had received or generated events has determined which kind of transition it will go through. Also, planned transition booleans are all resetted at the end of the code section.

In the next iteration step all transitions will be executed.

In case no new external events arrive and the first when condition is not entered, which is supposed to be the block's behaviour in normal working conditions, one last elsewhen segment will make sure the three transition triggers are resetted. That should happen at the fifth event iteration step, while the next internal event time instant is being calculated.

```
elsewhen pre(internal_transition) or pre(
   external_transition) or pre(confluent_transition)
   then
     internal_transition := false;
     external_transition := false;
     confluent_transition := false "In case nothing
        happened, but at the previous step there has
        been a transition, reset the triggers";
end when;
```

By having all the atomic DEVS models share this well defined routine, we can ensure that the simulation of DEVS systems is carried inside the OpenModelica environment. Before analyzing other elements that compose the fundamentals of how DEVS are defined in this library, let's have a quick review of the transition handling characteristics:

- The operations to simulate DEVS systems are divided into five steps of event iteration.
- Every atomic DEVS model carries his own simulation steps, and if more events keep arriving sequentially, it acquires them and fires its transition once they occur no more.

- However, when in a simulation DEVS components are connected only between themselves, it can be said that their algorithm sections run at the same frequency, and so they propagate outputs and execute transitions in a synchronous manner.

- Consecutive internal events are allowed for modeling systems who go through a series of states with zero lifespan. To achieve that, it is sufficient to change the time instant of the next internal transition while mantaining it below the actual simulation time value.

3.1.2 DEVS atomic models structure

As we said above, all DEVS components in this library have to share a common code section that manages their transitions. Having a partial atomic model from which to start for defining every new model now seems to be a natural choice.

So, an `Atomic_PKG` package is used to store this base model and its components. It contains four Modelica classes:

- A record called `State`, which stores the variables representing the state of the atomic DEVS.

- A record `InputVariables_X`, which contains a set of variables updated at every external event, to store values from input ports of the system and that are used by confluent and external transitions to update the state. It serves as the *Bag* of parallel DEVS specification, storing the informations of messages arrived in input.

- A record `OutputVariables_Y` that contains all the possible variables that the system can output. Before every internal transition, this record is modified by an apposite algorithm section, representing the $\lambda(s)$ function specified in the formalism. Those variables are then used to set the output connectors of the system.

- A partial block named `Atomic` which is the core of this library and that is extended everytime a new atomic DEVS model has to be defined. It contains the code explained above to simulate DEVS systems.

Up to now we have talked about atomic DEVS models, but since the nature of discrete event systems is inherently causal, all the DEVS models are implemented using Modelica `block` classes. This because a block by definition can have only input and output connections, removing acausality from the process of event sending and receival.

The objective of this package is to give the possibility to define atomic systems following the DEVS formalism, while not restraining the modeler to any specified event communication system. The connectors used to transport events are left to the choice and the necessities of the modeler, which can set `InputVariables_X` and `OutputVariables_Y` according to the values that represent the nature of the system's events he wants to describe.

The `Atomic` block will contain a `replaceable` instantiation of everyone of those records:

```
replaceable State S;
replaceable InputVariables_X X;
replaceable OutputVariables_Y Y;
```

Every new DEVS block may redeclare `S`, `X`, or `Y` depending on the new records it will use to define itself. In the block are present parameters `n_inputs` and `n_outputs`, declared as protected.

One last variable that needs to be calculated is the time elapsed since last transition e, which is used by the external and confluent transition functions to obtain the new state. The following variables are thus declared:

```
discrete Real last_transition_time (start = 0);
Real elapsed_time = time - last_transition_time;
Boolean transition_happened = change(
   S.transition_number);
```

As we can see, `elapsed_time` is constantly assigned by an equation that compares the simulation time with `last_transition_time`, a discrete number that records the time instant of the last transition, and that is assigned by the following algorithm section:

```
algorithm
    when pre(transition_happened) then
        last_transition_time := time;
    end when;
```

so that after every transition the `elapsed_time` variable can go to 0 and start increasing again. `transtion_happened`, as explained above, is a boolean condition that can be used to trigger when statements right after a new state has been generated.

In addition, three variables accounting for the number of every transition type executions are present:

```
Integer n_of_internal_transitions_happened(start = 0);
Integer n_of_external_transitions_happened(start = 0);
Integer n_of_confluent_transitions_happened(start = 0);
```

They have no utility in running the simulation, but can be used to inspect the kind of transitions the system went through at the end of the simulation.

The partial block `Atomic` can be extended to define new atomic blocks. To properly work, those blocks should have the following characteristics:

- Define and redeclare new records `S`, `X`, and `Y` according to the represented systems. It is highly advised to let new redeclarations be an inheriting class from the base records.

- Use an `initial algorithm` to set initial values and initialize records.

- Define an algorithm section whose `when` is activated by the variable `transition_happened` to determine a new `next_TS` after the state has changed, at the fifth an final step of the DEVS simulation chain.

  ```
  algorithm
  //UPDATING next_TS
      when pre(transition_happened) then
          next_TS := S.some_state_variable;
      end when;
  ```

  Notice that while in the DEVS specification the $ta(s)$ function returns the lifespan of the new state, this algorithm section has to output directly the time instant of the next internal transition. This because there's no software that will automatically select the blocks with the nearest scheduled transition, but every block has to autonomously schedule its own transition time instants. Nothing prevents us from calculating the lifetime of the new state and then add to it the actual simulation time. What is required is that every atomic block provides for determining when its next internal transition is going to happen.

- Use an algorithm section to update output record `Y` values from the state ones. Recalling the original five steps in which the DEVS simulation takes place, every atomic DEVS model generates outputs at the second event iteration step, one step after `internal_transition_planned[1]` was flagged. So the algorithm section takes the following form:

  ```
  algorithm
      when pre(internal_transition_planned[1]) then
          Y.some_output_variable :=
             S.some_state_variable;
      end when;
  ```

  `Y` values can be then equated to connectors' variables to send events through ports.

- Contain a code section to acquire the incoming events and store their informations inside the `X` record. To avoid problems arising with algebraic loops, and to allow for an event transmission system very similar to the one suggested by the messages graphs showed in the DEVS formalism specification, the `X` variables have to be assigned with the `pre()` of the actual connectors' values.

  ```
  algorithm
      when pre(external_event) then
          X.some_input_variables := pre(
             CONNECTOR.some_connector_variable);
      end when;
  ```

  Notice also that the `when` condition is activated by delayed values of `external_event` array. Thus, even if input is set to arrive at the second step of the event iteration, it is acquired in the X a step later, the one in which type of transition the atomic will go through is decided. Just a step before X is used to determine the new state.

- Redeclare external_event array with a specified dimension, and also set its values according to changes in connector's ports. Also, parameters' n_inputs and n_outputs values should be set.
- Finally, an algorithm section to execute the transition functions should be defined. Its when branch should be triggered one step after transitions are determined, so it should be written in this way:

```
algorithm
    when pre(confluent_transition) then
    //CONFLUENT TRANSITION CODE
        S.some_variable := S.some_variable + 1;
    elsewhen pre(external_transition) then
    //EXTERNAL TRANSITION CODE
        S.some_variable := S.some_variable + 1;
    elsewhen pre(internal_transition) then
    //INTERNAL TRANSITION CODE
        S.some_variable := S.some_variable + 1;
    end when;
```

It may seem reasonable to insert many of these procedures inside functions, but functions may introduce a problem when returning records containing parameters, which are required to define array sizes. Since arrays with undefined size turned to be troublesome without recurring to constant variables in redeclarations, algorithm sections have been preferred to handle atomic DEVS models' operations. As we will see in EventsLib, being able to define the number of input and output ports can be advantageous when modeling systems using the OpenModelica graphical environment.

3.1.3 DEVS example: a simple server

To better understand how a DEVS can be defined using Atomic_PKG, we will go through a complete example of atomic DEVS development. We will choose a very simple server model, with only one input port and one output port, that can either receive or send events. No blocking on output ports is assumed to happen, meaning that the model at its output is always ready to receive new events.

What this DEVS block is asked to do is receive an event, and then output it after a precise amount of time, together with the request for a new input event.

Many features of this basic model preview the characteristics of blocks contained in this library, and that are going to be explained in the next section.

An event is viewed as an augmentation in value of one of the input ports' variables. So, in case an input port is augmented by one, a change() function applied on it assigns a value to the related external_event array element, while the corresponding X variables will be updated by the same value.

It is a simple yet robust solution, that allows to inspect inputs and outputs sent directly in the OpenModelica plotting perspective.

Now that we have a way to represent elements, the connectors to be used are the ones used in **EventsLib** and presented in section 3.3.

```
connector In_Port
    input Integer event_signal "Event arrival signal";
    output Integer event_request_signal "Channel to let
        the block require events";
end In_Port;

connector Out_Port
    output Integer event_signal "Signal used to send
       signals";
    input Integer event_request_signal "Events
       communicated from blocks connected in output";
end Out_Port;
```

They are a couple of connectors representing the same port type in input and output mode. Still, every port has an explicit input and output variable:

- `event_signal` is the integer blocks are supposed to use to send events in output to other blocks
- `event_request_signal` represents a way for blocks to communicate with blocks feeding its input. In this case it will be used to forward requests to a queue, but it can also be used to share informations about the port, for example if it is closed and the block is not able to receive events.

Our server block will need two ports: one to receive elements in input and one to send them.

It's state can be a record `StateSer`, which inherits from the original `State` record from `Atomic_PKG`:

```
record StateSer
    extends Atomic_PKG.State;
    discrete Real next_TS "time of next scheduled
       internal transition";
    Integer elements_processed "Elements successfully
       processed by the server";
    Integer request "Requests sent by the server";
    Boolean active "true if the server is working on a
       part or not";
    Integer acquired_inputs "Input events already used
       since last transition";
end StateSer;
```

- `next_TS` is the time instant of next internal transition, that will be assigned to the respective block's variable after every transition.

- `elements_processed` is the amount of elements processed in output. Remember that this value will be used to set the output value before the transition happens, so it as augmented already when the process is supposed to begin.
- `request` represents the amount of requests sent by the server. It should start at one at the beginning of the simulation, so that at the beginning the queue connected in input is advised that it has to forward an event as soon as possible.
- `active` lets us know if the server is currently working or not.
- `acquired_inputs` is the amount of arrived events that have been used to start a process. When a job is launched, this value is incremented, and if the input is bigger than this value we know that we have an event at disposal.

Some of this variables are redundant. We know that the server is active or not just by checking if `next_TS` is different from infinite. `elements_processed`, `request` and `acquired_inputs` all increase with the same rate. But they are used to gain a more explicit idea of the block's condition, nearer to the physical processing unit it shall model.

A similar procedure can be used to set input and output records:

```
record InputVariables_XSer
    extends Atomic_PKG.InputVariables_X;
    Integer events_arrived "Events arrived at this
       event_iteration";
end InputVariables_XSer;

record OutputVariables_YSer
    extends Atomic_PKG.OutputVariables_Y;
    Integer elements_processed "Elements successfully
       processed by the server";
    Integer request;
end OutputVariables_YSer;
```

Now that records are set, we can start to build our block. We have to extend the base atomic block, and in doing this many modifications have to be applied:

- We have to redeclare `S`, `X` and `Y` with the new records we defined.
- Even if for this case it is not necessary, `n_inputs` and `n_outputs` parameters can be set to 1
- `external_event` shall be redeclared with a boolean array of dimension 1, since the only kind of event we plan to receive will be the one from the input port.

We can thus declare a parameter `timeadvance` to specify the time required to process an event, together with the two ports. The beginning of our block will look like this:

```
block Server
    extends Atomic_PKG.Atomic(redeclare StateSer S,
       redeclare OutputVariables_YSer Y, redeclare
       InputVariables_XSer X, n_inputs = 1, n_outputs =
        1, redeclare Boolean external_event[n_inputs]);
    parameter Real timeadvance = 3.0;
    //Block's ports
    In_Port IN;
    Out_Port OUT;
```

We can skip the state's initialization, just recall that `S.request` shall be initialized with value 1, so as to launch a request right at the first internal transition, scheduled at time 0.

`next_TS` will be updated every time with the value of `S.next_TS`:

```
when pre(transition_happened) then
    next_TS := S.next_TS "After every transition
       next_TS equals the value set into the state";
end when;
```

While the output will be set equal to corresponding state variables, the input record `X` will use the `IN` port's `event_signal` to update its value. An equation section is used to set the connectors output variables and the `external_event` trigger

```
algorithm
    when pre(internal_transition_planned[1]) then
        Y.elements_processed := S.elements_processed;
        Y.request := S.request;
    end when;
algorithm
    when pre(external_event) then
        X.events_arrived := pre(IN.event_signal);
    end when;
equation
    OUT.event_signal = Y.elements_processed;
    IN.event_request_signal = Y.request;
    external_event[1] = change(IN.event_signal);
```

Having settled everything in the correct way, we can finally focus on the transitions. The confluent one can be fired only right after an event has been sent. It will check if there are incoming events that have not been already used, and in that case it will launch the process, otherwise the block will stay idle by setting the next internal transition time at infinite.

```
when pre(confluent_transition) then
//CONFLUENT TRANSITION CODE
    if  X.events_arrived > S.acquired_inputs then
```

```
          S.elements_processed := S.elements_processed +
             1;
          S.request := S.request + 1;
          S.active := true;
          S.acquired_inputs := S.acquired_inputs + 1;
          S.next_TS := time + timeadvance;
      else
          S.next_TS := Modelica.Constants.inf;
          S.active := false;
      end if;
```

When instead an external event arrives we should assume that the server is idle and is waiting for a response to its request. Even with that, whenever an external transition happens we check that the server is not already active before using the incoming event to start a job.

```
elsewhen pre(external_transition) then
//EXTERNAL TRANSITION CODE
    if not S.active and X.events_arrived >
        S.acquired_inputs then
          S.elements_processed := S.elements_processed +
             1;
          S.request := S.request + 1;
          S.active := true;
          S.acquired_inputs := S.acquired_inputs + 1;
          S.next_TS := time + timeadvance;
    end if;
```

After an internal transition the server is just left in an idle state.

```
elsewhen pre(internal_transition) then
//INTERNAL TRANSITION CODE
    S.next_TS := Modelica.Constants.inf;
    S.active := false;
end when;
```

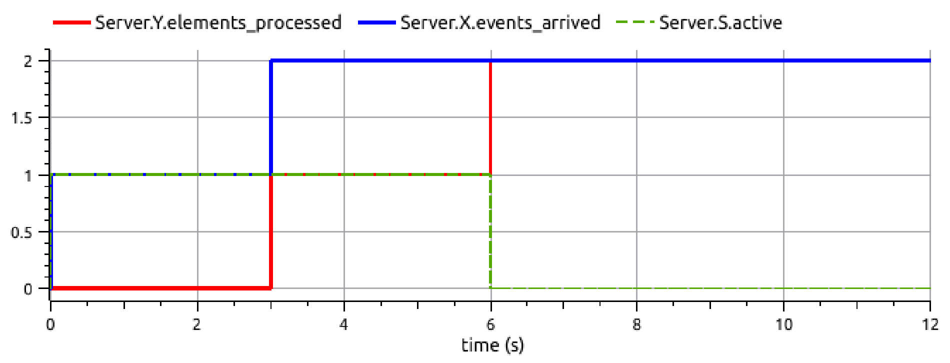


Figure 3.4: Simulation result of the Server block when connected in input with a queue already storing two events

In figure 3.4 we can see a simulation result of this block with a `timeadvance` of 3.0 value, connected with a queue DEVS block, properly defined to send a stored event when requested, already filled with 2 events. The simulation runs for 12 seconds.

As can be seen the block receives two events and processes in the established time. After the queue is emptied, the server stays in idle mode without processing units, waiting for a new one.

In the process, it goes through a total of three internal transitions, one to send the initial request and the other two sending an output too, and two external transitions, one for every input event received.

3.2 Event transmission

Before describing the EventsLib package, containing basic modules to perform simulations, it is worth discussing the ways in which messages can be passed from one block to the other.

We have already seen in DESLib a mechanism to transmit messages based on external C functions and dynamic data structures. This approach was reimplemented and ran successfully using OMCompiler, and nothing prevents us from declaring in the `X` record a set of variables representing pointers to linked lists.

Still, a solution in pure Modelica language was considered preferable. Not only it wouldn't involve the presence of additional source files, but it would also show the language's completeness and universality, along with the capability to express systems coming from various domains.

DEVS models are supposed to exchange messages, which are couples of port-value pairs (p, v), with $p \in InPorts$ and $v \in X_p$. In the most basic examples the value is intended as a single-dimension component, but many applications require a structured message with more than one variable.

To accomplish the simulation procedure above described, the messages sending process should be contained inside one event iteration step to attain synchronicity. Imagine a scenario in which two models are employing a given amount of iterations to exchange messages, while concurrently another sector of the system keeps going through transitions. The case in which these two subsystems get in touch may invalidate our simulation, even if the blocks are robust enough to handle the momentaneous asynchronization.

To implement ports, as in the code examples seen up to now, connectors are best suited, since they provide an interface that can be used to let models share and transmit informations.

However, the circumstance in which multiple messages need to be sent out of the same port, represents a problem in a language with fixed size data structures. Even though one can provide a connector with a certain capacity to transfer multiple messages at the same time, their number would be still constrained by the number of variables declared in the connector. Thus, in a simulation case with no upperbound available on the possible maximum number of messages transfered through one port, the requirement of simultaneous transmission of multiple messages can't be satisfied using pure Modelica language.

Still, in many cases the messages sent can be allocate in batches to be distributed in different transitions by different executions of $\lambda(s)$ functions with no consequence for the correctness of the simulation. Receiving models can block their ports when their space for accomodation is full to avoid the arrival of messages that can't be retained and that otherwise would risk to be lost.

Many concrete cases for discrete event simulation intrinsically involve some limits on the amount of messages that can be sent: industrial parts occupy some space, so stockpiling capacity is a possible source of bottlenecks that simulations ought to point out, and we can easily assume in our modeling hypotheses that the amount of people arriving at the Bank Teller is limited by the bank's door's width.

Given that many cases of application involve natural constraints on the amount of entities able to simultaneously circulate inside the system, entities whose nature should be represented by messages, and that Modelica is oriented towards the modeling and the simulation of physical systems, it follows that connectors with fixed size transfering capacity satisfy the requirements to describe most systems of interest.

During this thesis work two possible ways to send messages have been elaborated, one which identifies the arrival of a message with an augmentation in front of the connector's previous value, and one that embraces the impulsive nature of the messages described in the DEVS specification.

- To implement the first approach it is sufficient to increase or decrease the value of a connector's variable to represent a message. The receiver will detect this variation using the `change()` function, and thus will update the input record `X` with the new connector variable. If necessary, it will be receiver's duty to quantify the connector's variation with regard to previous values.

- To represent messages like impulses the connectors' variables should mantain a default value for the whole simulation, and assume the message value v just for a step of the event iteration, sufficient to trigger an external event condition and then let the receiver retreive the message.
  This solution can be performed using the `Atomic` block by just adding a when condition to let the output `Y` values return to the default value at the third step of the DEVS simulation event iteration. This operation can easily take the form:

```
algorithm
    when pre(internal_transition_planned[1]) then
        Y.some_output_variable :=
           S.some_state_variable;
    elsewhen pre(internal_transition_planned[2])
       then
        Y.some_output_variable := default_value;
    end when;
```

  Given that with this approach a change in the input variables happens twice, `external_event` array should be assigned in an appropriate way, so as to ignore the restoration to the default value when triggering the transtions. In this case the `edge()` function or a default value comparison are suggested to the modeler.

Between the two, the first technique has been chosen to represent events in the models developed for this work. Even if it may seem unorthodox to actually increase a variable to represent a new incoming message rather than sending the desired value, having a connector's whose variable's evolution can be tracked over the course of time can be very useful when visualizing the results of a simulation.

Let's not forget that Modelica plots represent the variables over the course of the simulation time, so event iteration occurrences are hidden and can be inspected only using the apposite simulation flag. Thus, when visualizing the connectors, employing the second approach we would be able to see only the default value for all the duration of the experiment, which can be impractical when interpreting the final result.

Instead, as already seen in Figure 3.4, an approach in which the output is augmented at every event guarantees a direct visualization of the simulation outcome.

3.3 EventsLib

EventsLib is a package providing blocks to perform discrete events simulation. Its functionalities are inspired by the SimEvents library, provided in the SIMULINK environment.

All the active blocks are developed starting from the Atomic partial block, and they have been modeled and designed following the DEVS specification formalism.

The basic unit that is transmitted from one block to the other is the event, which can represent discrete uniform entities or other components flowing inside the system, depending on the needs of the user.

Every augmentation of value 1 in the blocks' connectors indicates that an event moves from output to input port. An augmentation with value $n \in N$ corresponds the transfer of n events from output to input port.

A block can have multiple input and output ports, but all the possible connections must be one-to-one. All the atomic models have the same interface, consisting in the two connectors **In_Port** and **Out_Port** already seen in the DEVS server example.

```
connector In_Port
    input Integer event_signal "Event arrival signal";
    output Integer event_request_signal "Channel to let
        the block require events";
end In_Port;

connector Out_Port
    output Integer event_signal "Signal used to send
       signals";
    input Integer event_request_signal "Events
       communicated from blocks connected in output";
end Out_Port;
```

In a process-oriented approach, events circulate from one block to the other, carried by the `event_signal` connector's variable. Still, at times it can be necessary to manage the traffic of events, by blocking ports or in the server's case require explicitly a queue to send events. To achieve this, the variable `event_request_signal` is used. Counterintuitively, it constitutes an output variable inside **In_Port** and viceversa. The connector's nature of two way signal has been chosen to be visually hidden, to show the modeler components in which events go through the system from beginning to end. In image 3.5 it is possible to see a sender block is supposed to transmit events to a receiving one.

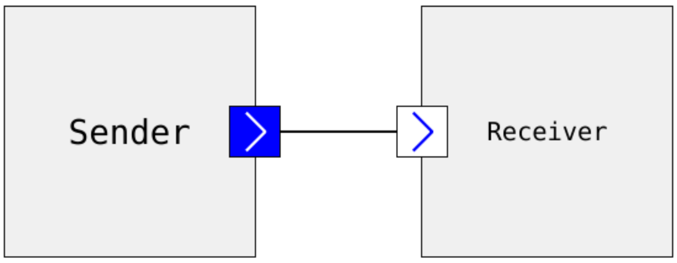


Figure 3.5: A pair blocks from EventsLib, connected using In_Port and Out_Port

Even if we are interested in the throughput of events from the left block to the right block, the `event_request_signal` variable allows to handle the events flux without having to instantiate a second explicit connection.

To require states with 0 lifetime a function **ImmediateInternalTransition** can be used to set the `next_TS` variable to decreasing negative numbers.

To allow for stochastic behaviour of models a package **RNG** is included, containing a block that provides constant, uniform, exponential and gaussian distributions.

A package named **Blocks** contains a shortcut to the library's components, that can be connected and assembled to create coupled parallel DEVS, and thus model and simulate discrete event systems. Many atomic blocks use arrays of connectors, so multiple one-to-one connections can be performed. When possible, the user can act on the parameters **inputsNumber** and **outputsNumber** to set the connector array's dimensions.

For example, a queue can receive and store events from multiple sources. Still, it is necessary to specify the number of input and output ports of the systems, otherwise a discrepancy between the number of variables and equations of the model will lead to a flattening error.

Many models can block their input ports by using the request variable in the connector. When this situation occurs the block outputs a -1 value through its ports to its upstream blocks, otherwise the value 0 is set to indicate that events are free to flow. This triggers an external transition to all the tributary blocks, which thus update their information on output ports statuses.

This possibility can be used to reproduce the space availability issue of certain queues, together with the related bottlenecks. Still, the blockage of ports requires a transition, during which the arrival of new events can't be excluded. All the sending blocks are set to let an entire transition cycle pass between one transmission and the other, to give the receiver time to block its ports if necessary. In case two senders have a batch of events to output and their waiting transitions are not paired, the receiver capacity may be exceeded. The user can choose with a parameter if the exceeding events can be retained or discarded.

So, these are the basic principles of EventsLib:

- EventsLib's blocks exchange events in form of augmentations of their connectors' variables.
- Only one-to-one connections are allowed, but certain components can have multiple one-to-one connections.
- Blockage of ports can be used to model bottlenecks of our systems.

In the following, a detailed description of the blocks properties and features can be found. For a more exhaustive explanation of their operation, in Appendix A the transition routines of certain blocks are described together with their code sections commented.

Generator

This block cyclically introduces events in the system. It is possible to send multiple batches of events simultaneously by using more than one output port.

In case any of its ports happen to be blocked, it continues to prepare the generation, but doesn't send events once generated. In case a port is blocked and unlocked between two generation time instants, no visible effect can be seen in the inputs and outputs of the block.

It starts a new generation only once all the already generated events have been successfully sent.

To tune it, it is possible to act on its random distribution for selecting the timeadvance and the following parameters:

- **outputBatchDimension**: amount of events that are going to be created at every new generation.
- **sendEventsSingularly**: if true, batches of events will be sent one by one, to let the receiver some transition steps to close its ports if necessary. If the system being modeled has no serious constraints on capacity, this parameter can be set to false so that all generated events will be sent together in the space of a unique transition. Having less transitions to inspect can be more convenient when simulation events are analyzed.
- **generateOnStart** can be used to determine if the first event generation will happen at the beginning of the simulation or after a certain amount of time.

Queue

A queue receives events from inputs and stores them. At the same time, it receives requests from server blocks through its output ports, and will send events to the querying port if it has enough of them stored to satisfy the request. The priority with which requests will be satisfied is given by the index of the connector.

The maximum amount of events that a queue can be stored can be defined by its capacity. If a queue has a set of pending requests, it will first satisfy them with the events arrived, and only after will check if the storing limit has been reached. In case, it blocks its output ports. Its behaviour can be defined with the following parameters.

- **capacity** defines the maximum amount of events that the queue is supposed to store. If no unlimited capacity is desired, this value can be set to -1.

- **fragmentedService** allows to choose between two policies for pending requests satisfaction. If true, the queue is allowed to partially satisfy pending requests. For example, if a server required two events, when an event arrives it is directly sent to that server, even if it doesn't satisfy completely the request. When false, a queue will satisfy a pending request only if it has enough events to serve it entirely.

- **allowEventsInExcess** can be used to specify whether or not events surpassing the capacity limit will be kept or discarded.

- **startingEventsInQueue** specifies the amount of events which the queue initially possesses, usually 0.

Server

A server block queries for new events and then processes them taking up some time. Once an batch of events is processed, it queries for new events and sends the ones it has just completed. A new job won't start until all the processed events have been successfully sent.

All arriving events are collected and conserved, so practically it is possible to connect a server directly to the output of a generator or another block, which will ignore the server's requests and will transmit events at its own rate.

To restrict the amount of requests to only one queried block, a server is allowed to have only one input port, but multiple batches of events can be simultaneosly sent through its outport ports if desired. If any of the output ports is blocked, the server keeps the job active but doesn't send events when completed. The following parameters can be used to set its behaviour, together with the parameters used to define the time required to complete its process:

- **outputBatchDimension** is the amount of events generated after every process is finished.

- **requestDimension** is the amount of required events to begin a process.

- **sendEventsSingularly** if true batches of events will be sent one by one, to let the receiver some transition steps to close its ports if necessary. If the system being modeled has no serious constraints on capacity, this parameter can be set to false so that all events will be sent together in the space of a unique transition. Having less transitions to inspect can be more convenient when simulation events are analyzed.

Delay

A delay block receives events and outputs them after a certain amount of time, delaying their route inside the system. It can take events from many input ports and it can send simultaneously multiple events through its outport ports. If any of the output ports is blocked, the delay block doesn't send events.

Every delay block has a defined and limited capacity of events that can be simultaneously delayed, and when that capacity is reached input ports are blocked. Nonetheless, an additional storing space is always kept outside of the nominal value, to account for an excess of events.

The delay imposed by the block can be of stochastic nature, in this case the order with which events will exit the block is not guaranteed to be the same of entering.

With a specific setup, the delay block can be used to model the service time need by a queue to transport an event to a server, by inserting a delay block with one input and one output in the middle of the queue's and server's ports. The delay block will transfer requests from the server to the queue, and thus will receive events that will be transmitted after the required amount of time. Remember that it is necessary to employ a delay with enough capacity to contain all the events requested by the server, and that the delay can't take requests from more than one server, since managing many different pending request from different ports is not one of its properties.

Together with the delay time, the following parameters define the block's behaviour:

- **capacity** defines the maximum amount of events that the block is supposed to simultaneously store.

- **sendEventsSingularly** if true batches of events will be sent one by one, to let the receiver a transition step to close its ports if necessary. If the system being modeled has no serious constraints on capacity, this parameter can be set to false so that all events will be sent together in the space of a unique transition. Having less transitions to inspect can be more convenient when simulation events are analyzed.

- **allowCapacityExceeding** can be used to specify whether or not events surpassing the capacity limit will be kept or discarded. Every delay block has an additional space for events equal to twice the number of input ports. When even this additional space finishes, the block automatically discards exceeding events indipendently from this parameter.

RandomSwitch

A random switch receives events from possibly more than one input port and randomly redirects them towards one of its two output ports.

- **sendEventsSingularly** if true batches of events will be sent one by one, to let the receiver some transition steps to close its ports if necessary. If the system being modeled has no serious constraints on capacity, this parameter can be set to false so that all events will be sent together in the space of a unique transition. Having less transitions to inspect can be more convenient when simulation events are analyzed.

- **allowDeterministicChoice** lets the user choose between two policies in case one output port becomes blocked. When the parameter is set to true, the block will automatically output incoming events towards the open port. If set to false, the random switch will behave like both output ports are blocked, thus it won't send anything and will close its input ports. Events already arrived will be retained with no capacity limit.

- **firstPortProbability** is the probability that an incoming event will be directed to the first output port. The second's port probability will be complementary to this parameter's value.

Combiner

The combiner block receives events from two different input ports and merges them into a unique batch of events. Multiple batches of events can be eventually sent through multiple output ports.

When any of its output ports is blocked it blocks its two inputs. The parameters below can be used to define its behaviour:

- **eventsRequired1** is the amount of events coming from the first port required to generate a batch of combined events.

- **eventsRequired2** is the amount of events coming from the second port required to generate a batch of combined events.

- **outputBatchDimension** is the amount of events generated for every successful combination.

- **sendEventsSingularly** if true batches of events will be sent one by one, to let the receiver a transition step to close its ports if necessary. If the system being modeled has no serious constraints on capacity, this parameter can be set to false so that all events will be sent together in the space of a unique transition. Having less transitions to inspect can be more convenient when simulation events are analyzed.

Displacer

The displacer receives events from its inputs and remove them from the system, that is, it doesn't transmit them anymore. It can be placed at the end of a working line to collect events which completed their lifecycle, eventually from more input ports.

AuxiliaryBlocks

A set of auxiliary blocks is provided to eventually manage connections between components. Even though many blocks provide for multiple connections, the user can choose to pick a block to merge a pair of ports or other simple operations. They are not modeled following the DEVS formalism, thus they do not require any event iteration step to accomplish their operations, mantaining the DEVS synchronized.

- **EventMultiplier**: takes events from input and output and multiplies by a constant. If we desire a block to output an event from its first port and concurrently two events out of the second port, an EventMultiplier can be placed at the end of this last port to multiply its output.
  The user can also choose to multiply the request value.

- **EventDuplicate**: for every event received transmits an event through both its two output ports. The user if the request value the input will set is going the minimum or maximum of the two output ports.

- **EventSum**: acquires events from two input ports and redirects them through one input port by summing them. The request value from the output ports is directly propagated at its input ports.

RNG package

To account for stochastic behaviour of models, a mechanism to generate random numbers had to be implemented. In this library, the implementation of the Xorshift1024* algorithm contained in the Modelica Standard Library is used to generate random numbers for selecting statuses lifetimes.

Its operations have been wrapped inside a block named **RandomGenerator**, that uses this generator to obtain uniform and exponential distributions for different models. Blocks entitled to follow stochastic behaviour have these parameters to select the desired behaviour:

- **localSeed** and **globalSeed** are the seeds employed to generate the RNG's state, and will have to be set by the user.

- **distributionSelected** is an integer that can be used to select the desired distribution. A constant distribution is related to the value 0, uniform distribution to value 1, exponential distribution to value 2 and a gaussian to value 3. If another integer is set, by default the constant distribution is selected.

- **distributionParameter1** and **distributionParameter2** are the parameters used to characterize the distribution.
  In case constant distribution is chosen, the block will deterministically return **distributionParameter1**, otherwise the distributions will take the following forms:

$$Uniform : \mathcal{U}(a, b)$$

$$Exponential : \mathcal{E}(\lambda = a)$$

$$Gaussiam : \mathcal{N}(\mu = a, \sigma = b)$$

  with

$$a = \textbf{distributionParameter1}, \;\; b = \textbf{distributionParameter2}$$

  In case a constant distribution is required and **distributionParameter1** < **distributionParameter2** the numbers generated will be constant and equal to **distributionParameter1**.

3.3.1 A Bank Teller simulation

To see a quick example of how EventsLib blocks can be used for a simulation, let's consider again the Bank Teller system.

To model it, it is sufficient to connect in series a generator, a queue, a server and a displacer, as in figure 3.6. The generator will account for representing the arrival of customers at the Bank, they will enter the queue and will stay there until the Bank Teller calls for them. After that, served customers will exit the system and a displacer will count their number.

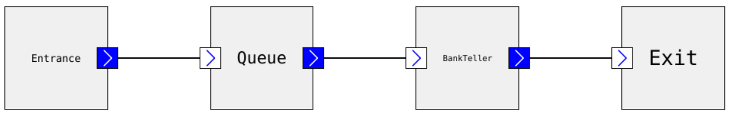


Figure 3.6

To run a simulation experiment, we will set the generator to let customers enter following an exponential distribution with $\lambda = 0.25$, while the bank teller will employ exactly 10 seconds to serve each customer.

In this condition we can suppose that, on average, every ten seconds at least two customers will enter the system, while only one will be removed from it. So we can expect that the queue will be filled going on with the time, and as we can see in figure 3.7 that is what an experiment run for 1000 seconds shows.

The same experiment run on SimEvents led to a similar result, with a queue constantly filled, as visible in figure 3.8. Since both servers work all the time without idle times and their service time is deterministic, both simulations output 99 served customers at the end of the simulation.

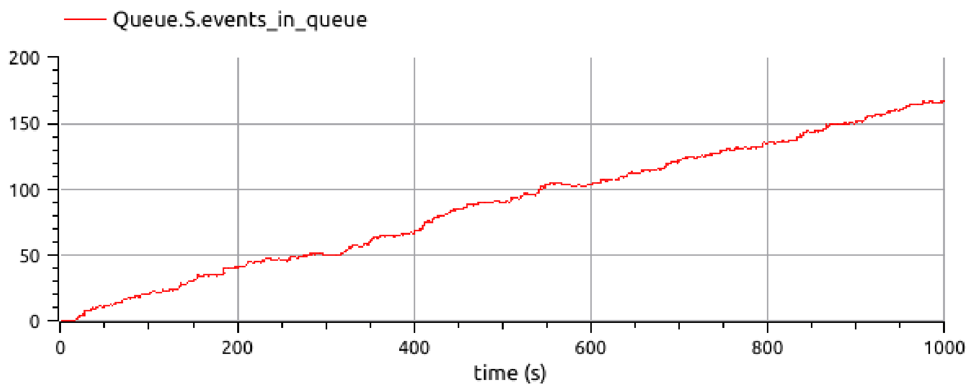


Figure 3.7

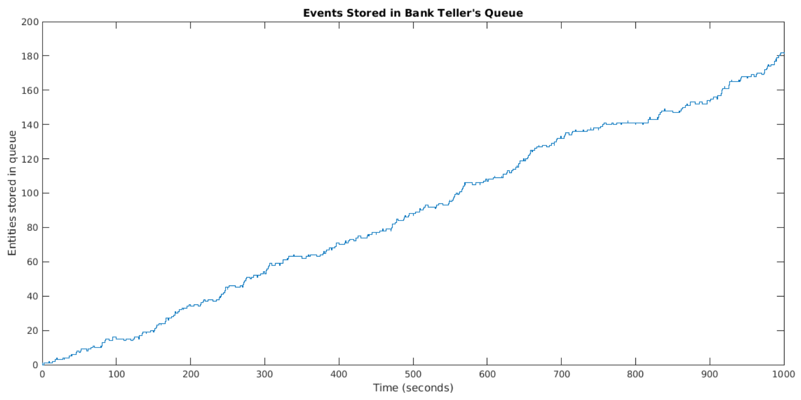


Figure 3.8

For a little variation, we can upgrade our bank and add two additional Bank Tellers, so that the system will take the form seen in figure 3.9. For this model to work properly, the number of output and input ports respectively of the queue and the displacer has to be explicitly set to 3 using the apposite parameters **outputsNumber** and **inputsNumber**. Then every server's input with index one will have to be connected to one of the queue's outputs in the following manner:

```
connect(Queue.OUT[1], BankTeller1.IN[1]);
connect(Queue.OUT[2], BankTeller2.IN[1]);
connect(Queue.OUT[3], BankTeller3.IN[1]);
```

The same will have to be done with the displacer's inputs.

Now if we run the experiment with the same blocks' parameters as before, for 10000 seconds, we will see that in both simulation engines the amount of elements in the queue will oscillate but will never start to increase, as shown in figure 3.10.

For a final comparison we can arrange our system to behave deterministically, by setting the generators to immit a customer in the system every 4 seconds. With this setup in the same time both queues are always empty, while the three Bank Tellers serve a total of exactly 2498 customers.

Many more complex models can be defined using **EventsLib**'s blocks. Multiple production stations can be connected in series and in parallel, while the combiner

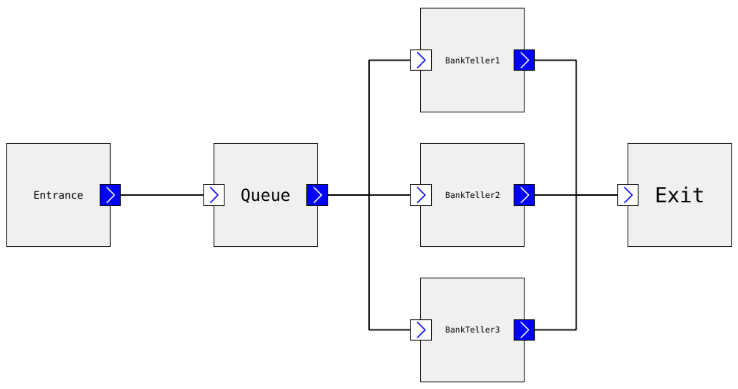


Figure 3.9: figure showing the model representing the 3 Bank Tellers in parallel configuration. The system is composed a generator in line with a queue that supplies 3 servers, whose events end in the final displacer

block can be used to merge different working lines and the random switch can model the different routes entities can take in case of defect.

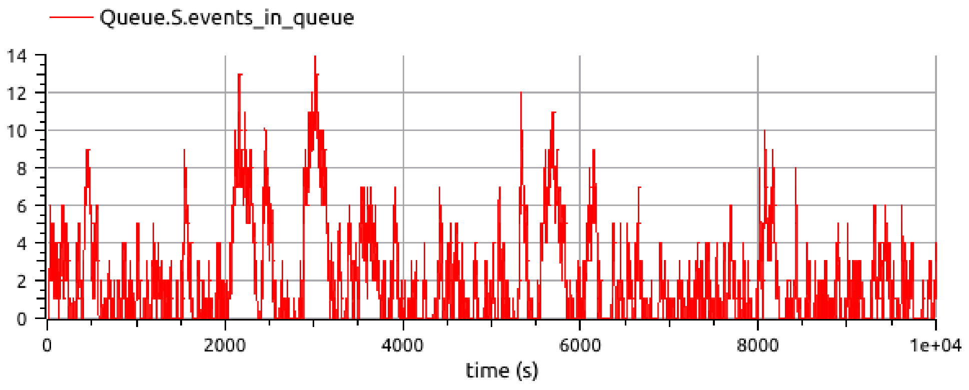
3.4 Conclusions

We have seen the elements that compose the MODES library. It allows to run simulations of coupled DEVS by defining atomic DEVS models and connecting them. This result has been achieved using only the Modelica language, proving the OMCompiler to be able to sustain pure discrete event simulations.

The **EventsLib** package provides blocks to perform simulations of plants or other systems of interest, in which multiple events go through several processes. In its development, the possibilities offered by the Modelica language to define complex DEVS models have been explored, showing that with well defined algorithm sections it is possible to handle a massive amount of events being exchanged.

The main advantage of having a well defined partial Atomic block from which to start when defining new atomic models, is the possibility to easily create new components. Adopting the above defined procedure to handle events, new blocks can be rapidly implemented without the need to arrange ad hoc solutions aimed at avoiding the algebraic loops that arise when defining assignments in algorithm sections. In this situation, the modeler is left to concentrate on algorithms that characterize the transitions, instead of worrying on the event management system.

In contrast to many commercial softwares available on the market, a modeling environment like the one offered by OpenModelica allows users to customize and define atomic models according to their necessities. As we will see in the next chapter, this can be a powerful choice when dealing with the various configurations that can be found in industrial plants.


Queue.S.events_in_queue
time (s)
0
2000
4000
6000
8000
1e+04
0
2
4
6
8
10
12
14

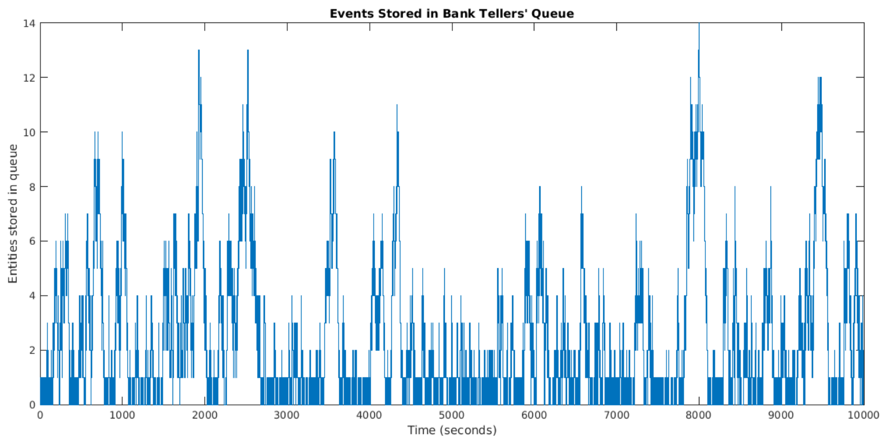

Events Stored in Bank Tellers' Queue
Entities stored in queue
Time (seconds)
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
0
2
4
6
8
10
12
14

Figure 3.10

Chapter 4

I4.0 Lab Simulation

After having developed a way to simulate parallel DEVS models in the Modelica environment, the next step was applying the customization of the DEVS specification to a real case of discrete events system. In fact, the biggest advantage of having a parallel atomic DEVS partial block from which to start is that, given a system with any degree of complexity, it becomes possible to subdivide it in many atomic models and then couple them. If the necessity to model particular behaviours arises, new blocks can be defined.

In this chapter we will see how the particular configuration of a production plant was sucessfully modeled defining a set of blocks to represent its components.

The facility of interest, which can be seen in Figure 4.1, is the production line installed in the I4.0 Lab at the Manufacturing Group at the Department of Management, Economics and Industrial Engineering of Politecnico di Milano [Cimino, 2018], developed by Festo[1] and Siemens[2].

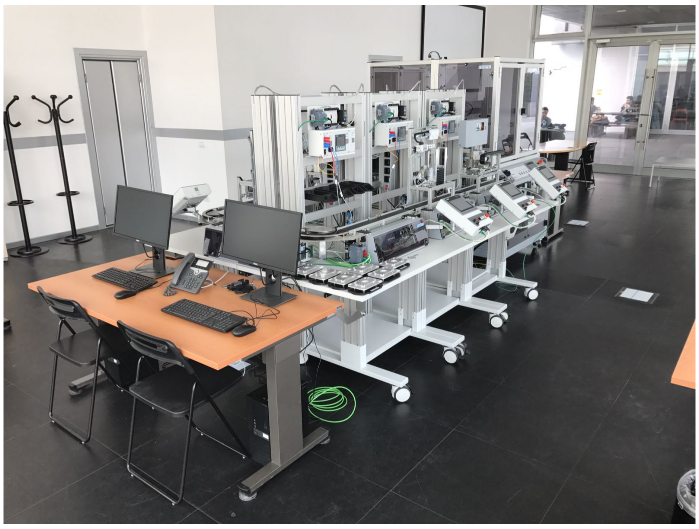


Figure 4.1: Photo of the I4.0 Lab

[1]https://www.festo.com/cms/it_it/index.htm
[2]https://www.siemens.com/it/it/home.html

4.1 The I4.0 plant

The plant consists in a production line that reproduces the assembly process of mobile phones. It is composed by an aggregation af modules called CP-Lab stations, whose structure can be seen in Figure 4.2, and a CP-Factory station, shown in figure 4.3.

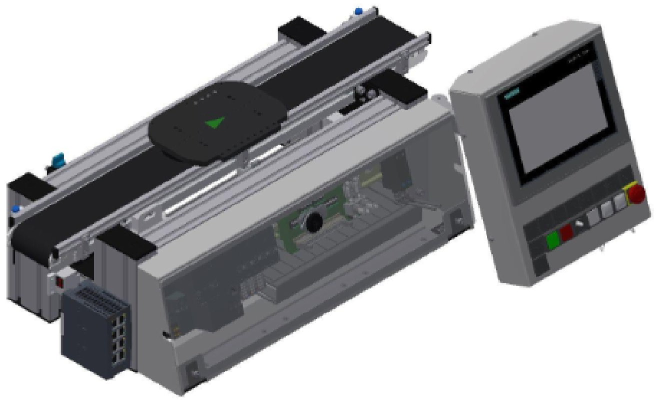


Figure 4.2: CP-Lab station

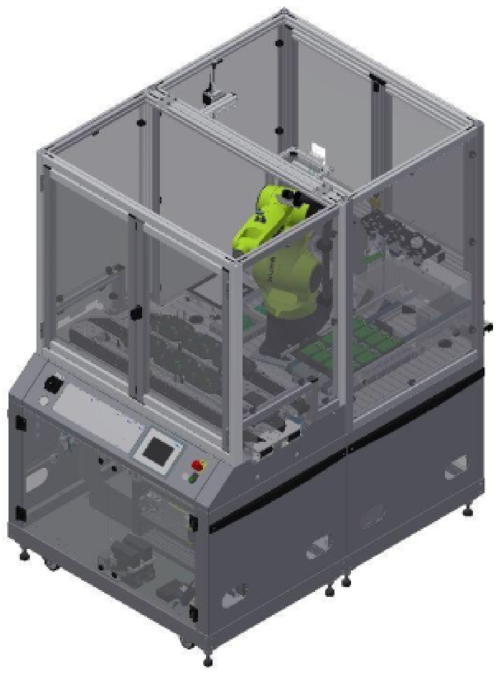


Figure 4.3: CP-Factory station

The plant's structure is represented in Figure 4.4. As can be seen, the different stations are connected by an automated conveyor belt. Working parts travel on this belt and thus through all the stations. More precisely, every WIP (Work In Progress) is put on a carrier, which is carried by the conveyor throgh the plant and that is stopped and kept blocked by a stopper whenever a working station is reached, as shown in Figures 4.5 and 4.6. In this way the working part is processed while the conveyor can still transport other carriers.

Once the process is completed, the stopper is released and the carrier can travel towards the next destination, while a carrier waiting to be manufactured will be

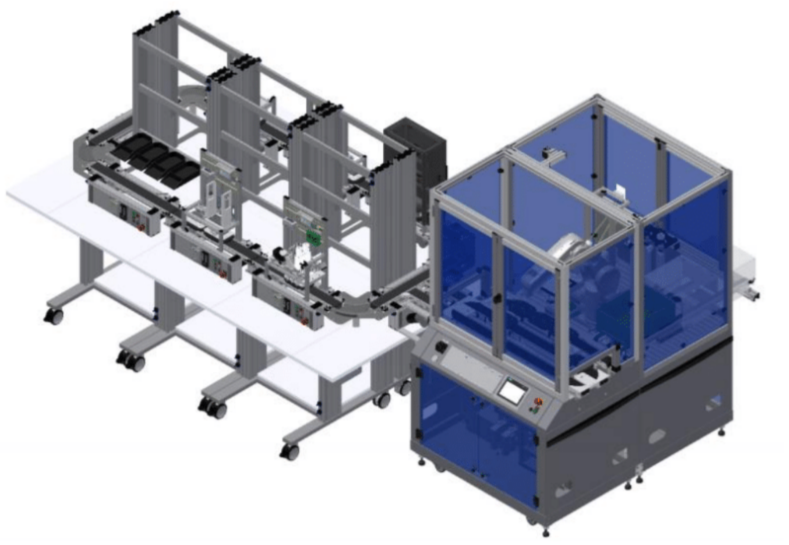


Figure 4.4: Representation of the I4.0 Lab assembly line

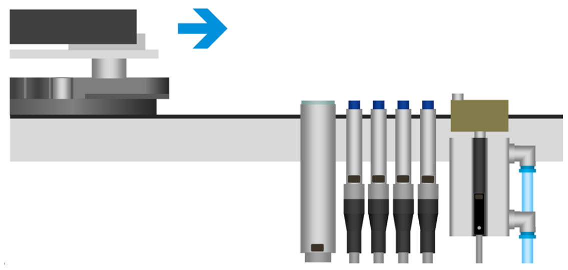


Figure 4.5: Carrier travelling towards station

finally able to reach the working station. In this configuration, in fact, there are no explicit buffers. Accumulations of pieces waiting to be processed will just have their carriers attached one to the next, like the carriages of a train.

The product which the line is supposed to work is made up by four components: Front cover, PCB, Fuses and Back cover, which are shown in Figure 4.7.

So, the total process is divided into seven phases, one for each station, which in order are:

1. **Manual Station**: station in which the operator can interact with the machine, introduce the carrier and withdraw the finished piece.

2. **Front cover Magazine**: station in which the front cover is placed in an empty carrier.

3. **Drilling Machine**: station in which the drilling operation is executed on the cover positioned on the carrier.

4. **Robot Cell**: station in which the PCB (Printed Circuit Board) is placed in the front cover of the Phone, and the fuses are embedded in the PCB.

5. **Visual Inspection Module**: station in which the visual inspection checks if the PCB and the fuses are well positioned inside the cover.

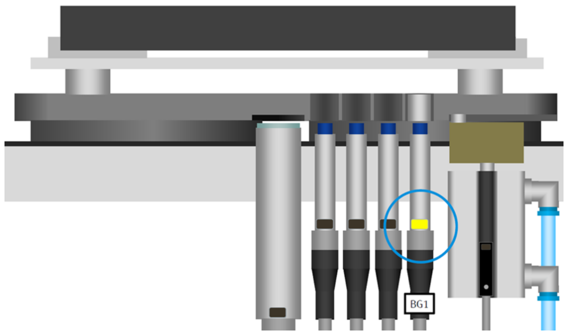


Figure 4.6: Carrier being blocked at station

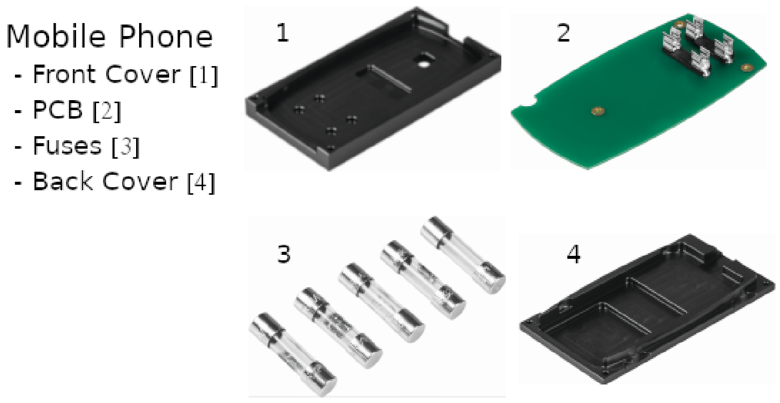


Figure 4.7: Parts assembled to compose the phone's prototype

6. **Back cover Magazine**: station in which the back cover is positioned on top of the product.

7. **Press module**: station in which the back cover is pressed to close the piece.

The plant's layout with stations' positions associated with the numbers already used is shown in Figure 4.8. Number 8 in the figure indicates a bridge placed to connect the Robot Cell with the rest of the plant.

All stations but the Manual Station and the Robot Cell are CP-Labs, being the Robot Cell a CP-Factory. In particular, the CP-Factory presents a peculiar characteristic: when working on a carrier, it allows a maximum of two other working parts to be in queue behind it. The excessive carriers are deviated directly towards the next working station, the Visual Inspection Module, which will notice that no Printed Board Circuit is present, and will register in the production plan that the piece is not following the working plan, since a working step is missing.

At this point, this incomplete part will travel through all the stations, which will not process it. In fact placing the Back Cover on a phone without the Printed Board Circuit inside would create a difective product. So this carrier will reach the Human Station, in which the operator will let the carrier continue its travel without removing the WIP and the stations 2 and 3 won't again execute any operation on the carrier, since they already completed their jobs on it, and will keep it only the time necessary to read its RFID-tag. At this point, our carrier will reach another time the Robot Cell, hoping this time to be accepted.

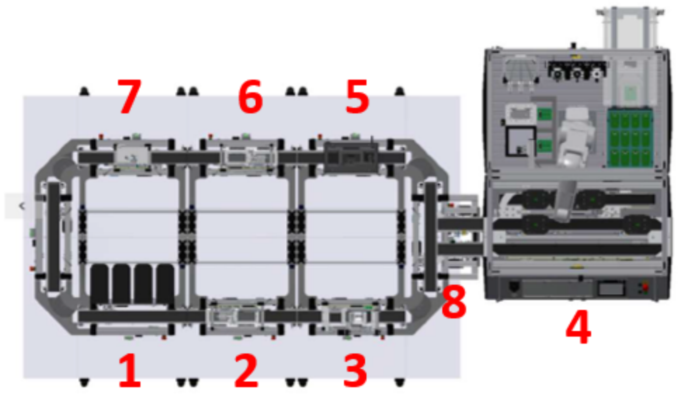


Figure 4.8: Representation of the I4.0 Lab assembly line's layout

If the queue has enough space, the WIP now will be manufactured by the robot after having travelled through the whole plant, otherwise it will be deviated again and the described process will repeat itself.

At this point we can summarize the main features of this plant, especially the ones relevant to our discrete event modeling task:

- The I4.0 Lab presents a plant in which carriers travel in circle through a series of stations, to then be emptied of their finished part when coming back in the first station, the one with the human element. Empty carriers will thus start the first working step of a new product when reaching the next station.
- Every station blocks its carrier's journey for a certain amount of time, depending on the operation it has to accomplish.
- There is no buffer in which carriers are stored before a station. Carriers just wait their turn by getting in line on the conveyor, one in contact with the end of the other.
- To travel from one station to the other, carriers spend some time, according to conveyor's speed and the distance to cover.
- A particular queue, the one preceding the Robot Cell can decide to deviate carriers in excess directly to the next station. Deviated carriers won't have their WIP processed by any station until they're finally able to access the Robot Cell and receive the electronic components which must be present inside the product.
- Finished products are withdrawn at the Manual Station by the human operator, who lets the carrier go to start the lavoration of a new mobile phone.

4.2 Modeling

To model this system, a different approach to define events has been adopted in opposition to **EventsLib**. In **EventsLib** every augmentation of value 1 in a connector's signal represented a single event and thus the arrival of an entity. Augmentations of $n \in N$ represented n events, which could be seen as n entities arriving inside a block.

To model the above described plant, it shall be noticed that it is impossible for two carriers to arrive simultaneously from the same conveyor. One has to follow the other, and in doing this a time delay is involved. So an approach in which an augmentation, indipendently from its value, would represent a singular carrier arrival was adopted.

In case the necessity to transmit more information would have arisen, it would have been enough to add variables to the original connector, allowing the transmission of structured data. In this scenario though, reusing the interface from **EventsLib** proved sufficient.

In this setup the amount of the increase would account no more for the number of carriers, but for the type of WIP present in the carrier. In fact, two different kinds of WIPs are identified:

- One needing to be processed by every station it arrives in, and that at end of the cycle counts as a finished product. The arrival of a carrier of this kind will be identified by an increase of value 1 in the connector's signal variable.
- Another WIP that has been refused by the Robot's queue and thus will pass through all the other stations without being manufactured. When arriving to the Manual Station it will not account for a finished product. When this kind of WIP will eventually reach the Robot, it will finally change its nature into a type 1 WIP. Until then, it will be recognized by an augmentation of value 2.

So, we can begin to presume that our queues will need to introduce some kind of delay in the avalability of the pieces arrived at their inputs, and that they will also have to recall the type of elements contained together with their order. In fact, there's no possibility for a carrier to surpass another one.

Stations can be easily modeled by servers, with the particularity that two different timeadvances will have to be expressed, one for every type of WIP.

Finally, we will need a specific queue before the Robot Cell, and our generator will need to introduce only a finite number of carriers in the system, since once immitted the carriers keep circulating in the plant.

With these modeling ideas in mind, using the **Atomic_PKG** to define new blocks a new package had been created, named **I40Lab**, which uses the same interface connectors and the same **RNGBlock** from **EventsLib**. The blocking mechanism is identical too, with a value of the request's signal equal to -1 to indicate the blockage of an input port.

To describe the system the following atomic models have been defined:

Generator

An identical block to the one of **EventsLib**, with the difference that it outputs only WIPs of type 1 and only in a limited quantity, determined with an apposite parameter, since the amount of carriers is limited.

Server

Another block very similar to the namesake one declared in **EventsLib**, with the difference that two different processing times can be chosen for different kinds of WIPs and that a parameter allows to choose the Robot's station as able to convert WIPs of type 2 into type 1.

DelayedQueue

An apposite queue, which receives carriers but makes them available only after a certain amount of time, required to model the travelling time along the conveyor. This block's capacity will reflect the length of the conveyor before the station, since only a limited amount of carriers can be stationed on a belt due to space issues.

As a modeling hypothesis, it is assumed that no fractional portions of a carrier can be contained inside this queue, thus input ports will be open only with enough space to contain at least an entire carrier.

TwoWayQueue

An atomic block aimed at modeling the special characteristic of the queue preceding the Robot's station. It is identical to the **DelayedQueue** block, except for the fact that instead of blocking ports when the capacity is reached, this block redirects the incoming carriers in excess through a relief port as WIPs of type 2.

ReceivalDelayer

A block aimed at modeling the time required for a carrier, stationed in queue waiting for a server's lavoration to finish, to reach the station's stopper and thus begin its required process. A block of this type should be placed between a queue and its server.

At an implementation level this block transmits the received request from the server immediately to its queue, while the events incoming from the queue are delayed.

Delay

A block delaying inputs and blocking ports when full, sending carriers directly when available and, instead of **DelayedQueue**, not when required. Will be used to represent the travelling time in some portions of the conveyor belt.

Displacer

A block used to model the finished products removal by the human operator. Every time a carrier with WIP of type 1 exits the Manual Station, it means that a finished product has been removed from the carrier, and thus a variable counting the finished pieces is increased.

In practice it just directly transmits the received carriers while taking into account the type 1 passed through it.

Now that the components of this system have been defined and described, we can assemble them to obtain our simulation.

To represent all the stations, the connection in series of a **DelayedQueue**, a **ReceivalDelayer** and a **Server**, which can be seen in Figure 4.9, was used. In this configuration, the queue represents the presence of carriers before the working station, and the time employed to reach it. Imagining a situation in which a carrier is being processed and another one is right behind it, the queue's space finishes exactly at the final extremity of the second carrier.

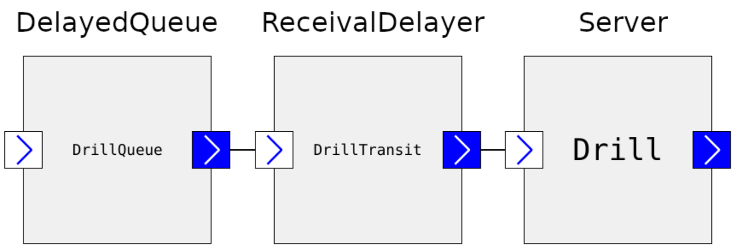


Figure 4.9: Drill station model

When a server completes its job and outputs its WIP, the time required for the next available carrier to reach the stopper is represented by the **ReceivalDelayer**, which forwards requests from the server to the queue immediately, but waits a certain amount of time to let its inputs reach the server.

Every queue's capacity is defined by the length of the conveyor belt it represents. For the Robot Cell station, the queue is represented using a **TwoWayQueue** block.

To represent the Human Operator activities, in addition to the series of blocks already described, a **Displacer** is present right at the end of the server, to account for the number of finished products collected during the Operator's job. In parallel to it, a **Generator** models the initial introduction of the carriers inside the system. This entire configuration can be seen in Figure 4.10.

To account for portions of conveyor belt before certain queues, two **Delay** blocks have been used:

- A first **Delay** block to model the transit from the end of the Drill station to the beginning of the fork preceding the Robot.
- The second delay models the time required for carriers deviated to immit in the Visual Inspection's queue.

The final model can be seen in Figure 4.11. It can be considered as divided in 7 different areas, five representing CP-Lab stations, one for the robot and one for the Manual Station. Since this has to account as an example of modelization, it has been decided not to aggregate these areas into coupled DEVS. Still, consider that for more complex models it can be very convenient to aggregate atomic models into coupled ones to then reuse them.

Having a well defined model, it becomes necessary to tune its parameters to reflect the behaviour of the real system.

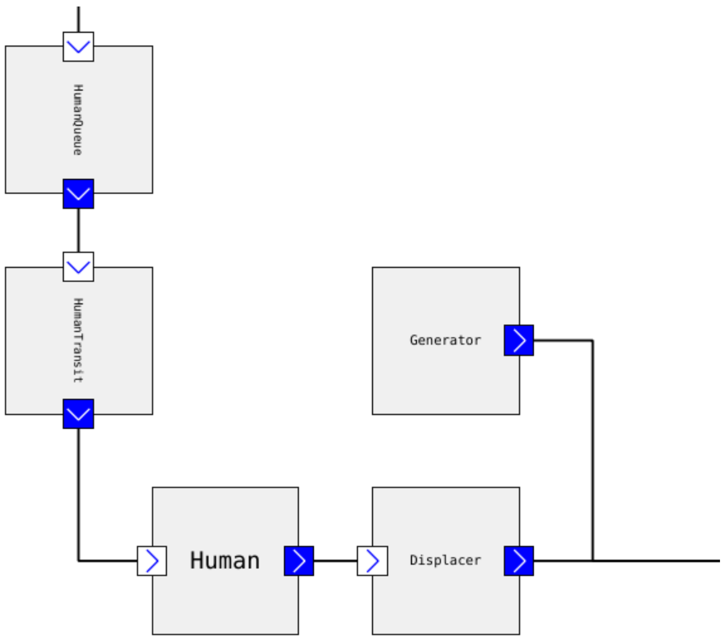


Figure 4.10: Manual Station model

At first, capacities for all the blocks representing conveyor sections have been defined, measuring their length and considering that a carrier is no longer than $18cm$. In this way the following capacities for every portion of belt have been derived.

Belt Section	Carriers Capacity	Traveling Times [s]
Human's Queue	9	21.5
Front Cover Queue	2	7.16
Drill Queue	2	7.16
Drill to Robot Queue Fork Transit	14	35.15
Robot Queue	2	5.85
Deviation to Visual Inspection	3	6.6
Visual Inspection Queue	7	20
Back Cover Queue	2	7.16
Press Queue	2	7.16

Next, travel durations needed to be defined, together with the processing time intervals. To obtain those times the production monitoring measurements were taken as a reference. Combining these data with stopper sensor rise time measurements, a time table was identified for a process of assembling with no fuses in the product's PCB. As a modeling choice, it has been decided to leave the travel time as a deterministic variable, while modeling station's processes as gaussians. For the manual station an exponential distribution is instead chosen, to model the fact that the operator may be momentaneously absent from his position to supervise a sector of the plant.

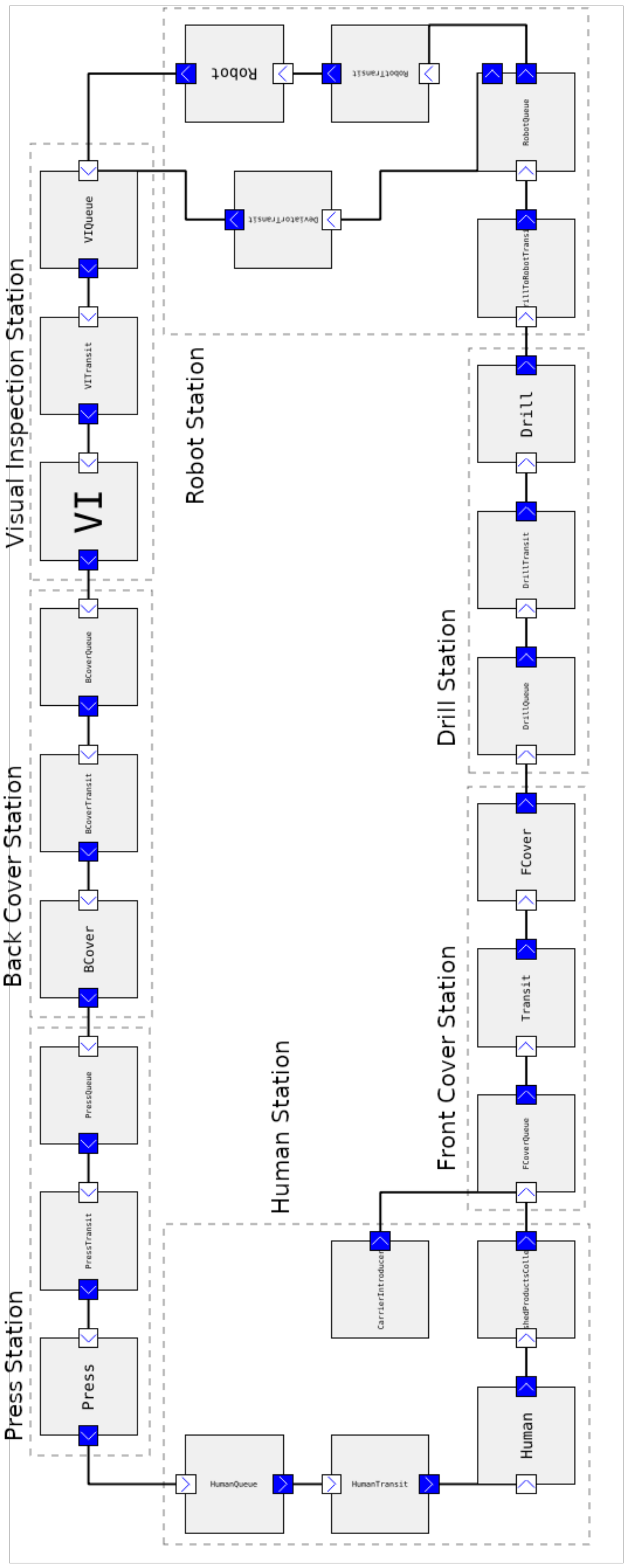

Visual Inspection Station
VIQueue
VITransit
VI
Robot Station
Robot
RobotTransit
DeviatorTransit
RobotQueue
rillToRobotTrans
Back Cover Station
BCoverQueue
BCoverTransit
BCover
Drill Station
Drill
DrillTransit
DrillQueue
Press Station
PressQueue
PressTransit
Press
Human Station
Front Cover Station
FCover
Transit
FCoverQueue
CarrierIntroducer
shedProductsColle
Human
HumanQueue
HumanTransit

Figure 4.11

Station	Mean Process Time [s]	Process Time Standard Deviation [s]
Manual Station	8.71	
Front Cover	9.14	0.35
Drill	12.14	0.35
Robot	81.14	0.35
Visual Inspection	6.16	0.37
Back Cover	9.14	0.35
Press	12.93	0.27

While the time spent by a WIP of type 2 inside every station has a mean of 3.56 s and a standard deviation of 0.52 s.

By running an experiment and a simulation with a single carrier, the experimental data tells us that 241 seconds were employed to complete a full working cycle, starting from the arrival to the first station and finishing with the product collection by the operator , while the simulation employed 241.88 seconds.

In the next section we will see the results of more than one simulation and the insights we can gain on the plant, but before talking about the simulation results, it is worth mentioning an important aspect that has to be taken into account when simulating a system of this complexity in Modelica.

To prevent possible infinite loops of event iteration, the Modelica simulation stops at runtime when a certain amount of consecutive event iterations is reached. If it is true that this problem arises naturally when a modeler accidentally creates a DEVS block that ideally runs an infinite number of consecutive internal transitions, it is also true that a simulation with many transitions triggering one another can be interrupted prematurely.

At the time in which this thesis is written, the default number of maximum event iterations allowed by the OMCompiler is 20. A model like the one above described can experience some problems with such a low value. To run the simulation, a number of 40 maximum event iterations was considered appropriate.

This value can be set using the "-mei" compiler flag. It can be set through the simulation setup window directly in the OMEdit environment, or by manually writing a code line like the one below in the model:

```
annotation( __OpenModelica_simulationFlags(lv = "
   LOG_EVENTS_V,LOG_STATS", mei = "40"));
```

As can be seen, this instruction enables the `LOG_EVENTS_V` too, to obtain a very practical setup for a simulation with a significant amount of events.

4.3 Simulation and results

One of the first decisions that our simulation can help us take is the number of carriers that need to be introduced inside the system. The laboratory owns 8 carriers, so a maximum of 8 WIPs can be manufactured simultaneously. By running multiple simulations we can try to inspect if the number of carriers affects the final number of products over a precise time span of activity.

It was decided to carry simulations of half an hour to see the final plant productivity for each possible quantity of carriers in the plant:

Carriers	Finished Prouducts
1	7
2	13
3	19
4	19
5	19
6	19
7	19
8	19

After a certain point, increasing the number of carriers doesn't improve the throughput in terms of finished products. This because, as we will see with further analysis, the biggest limit to the production is the Robot station, whose effect of bottleneck on the plant's performance could have been already foreseen when looking at the processes' durations for every station.

Still, the plant is large enough to let all the carriers flow without experiencing a relevant time loss for WIPs waiting in queues before a station.

So, it was decided to carry a simulation with 8 carriers, but with larger duration, to gain insights on the systems' behaviour.

In addition to the records `S`, `X` and `Y`, a fourth record named `STATS` was created and instantiated inside certain blocks, to calculate useful statistics from the plant. Average statistics were elaborated dynamically during the simulation by having a weighted mean between the variable of interest over the time elapsed since last transition and the already calculated average value over the total time before that transition. For example, to calculate the percentage of active time of a server the following formula was used:

$$activityPercentage = \frac{100 \ * \ activeTime}{totalSimulationTime} \tag{4.1}$$

which, as explained above, was not computed at the end of the simulation, but during it, and so the formula has been modified taking into account the weighted mean between the statistic value up to the birth of the new state and one related to the latest state. Since the statistic over a state's lifetime can be either 100 or 0, since a state conceives the server either as active or not, the assignment rule for the calculation of the activity percentage was synthesized in the following way:

$$\begin{cases} activityPercentage \leftarrow \frac{activityPercentage * (simulationTime - e) + 100 * e}{simulationTime}, & \text{if } ACTIVE \\ activityPercentage \leftarrow \frac{activityPercentage * (simulationTime - e)}{simulationTime}, & \text{otherwise} \end{cases}$$

with $simulationTime$ corresponding to the global variable `time` of the Modelica environment, and e the elasped time from last transition, found in MODES under the name `elapsed_time`, already seen when presenting the total state set $Q = \{(s, e) \mid s \in S, 0 \leq e \leq ta(s)\}$ from the DEVS specification, in 2.1.

The code lines accomplishing this operation are shown below.

```
if S.active then
    STATS.activityPercentage := (
       STATS.activityPercentage * (time - elapsed_time)
        + elapsed_time * 100) / time;
else
    STATS.activityPercentage := (
       STATS.activityPercentage * (time - elapsed_time)
       ) / time;
end if;
```

In this way, in case the server was actively manufacturing a piece the average percentage of working time would increase. In Figure 4.12 we can see the evolution over time of this statistic.

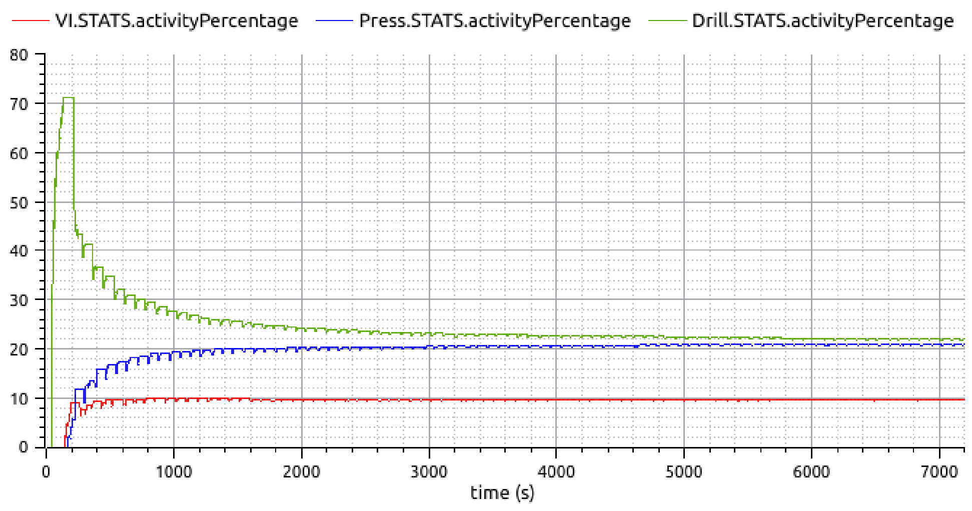


Figure 4.12

So, it was finally decided to simulate the plant for 2 hours, a total of 7200 seconds, and see what we could derive by the simulation about the plant.

The first aspect that we wanted to inquire was the activity of the stations. Since the travel time is relevant in this system, we can suppose that many stations will spend most of their time waiting for a piece to process.

In fact checking at the `activityPercentage` plot in Figure 4.13 we can easily see that the only station active almost all of the time is the Robot's one. The histogram indicates for every station the percentage of time spent with a working part stopped inside it. The robot is busy for almost all of the simulation time, since its process is the longest.

Still, we should consider that for the other stations a certain portion of time may be spent on WIPs of type 2, which would be stopped inside the station without an effective lavoration being carried. So, for major clarity, also the percentage of activity spent specifically on WIPs of type 1 was monitored. This represents the sole portion of time in which a station would be manufacturing a piece.

The resume of this simulation's aspect can be seen in Figure 4.14, which shows for every server, aside from the Robot, the percentages of time being idle, working on

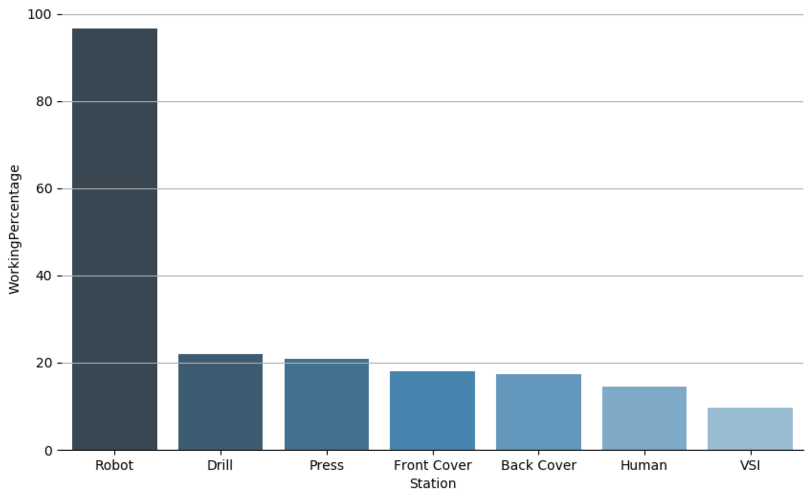


Figure 4.13

a WIP of the first kind and with a WIP rejected by the Robot's queue stopped. We can observe that on average every station is idle for almost 80 % of the time, while only a small interval concretely contributes to the production of mobile phones.

The next very important issue to be investigated is the possibility of a time loss inside the queues. Even though most of the time in the queue is naturally spent traveling, it may be possible that a carrier can be blocked right behind another one being processed by a server, or even worse, behind another carrier already waiting.

The state variable accounting on the number of elements inside every queue can already give us an idea on the amount of carriers being stacked inside a queue block. We can see an example in Figure 4.15, which shows the number of carriers contained in the conveyor section before the drilling station over a span of 1000 seconds.

It is natural that every queue contains some carriers, since they are passing through it, nonetheless this graph can already let us gain a first impression on the possible time losses due to latency inside every queue.

For a more precise quantification, we can take the difference between the availability time of every carrier sent by the queue and the real time instant in which the part leaves the block. This because when an event representing a carrier enters an atomic model representing the time spent moving along the belt's path, the time instant of arrival at the end of the queue is determined. If an event left the block after that time, it means that a time loss occurred since the carrier's journey was blocked by another carrier being processed.

We can thus build a statistic regarding the average difference between planned and effective exit time instants for every carrier sent in output by every queue, and thus know the average interval spent waiting behind another process.

Again, the biggest delay is caused by the queue serving the Robot's station, which confirms its nature of bottleneck in a plant otherwise made up by stations fast enough to let the carriers flow with few or no interruptions. In fact, a carrier spends on average 150 seconds stationing before arrivining to the Robot's stopper.

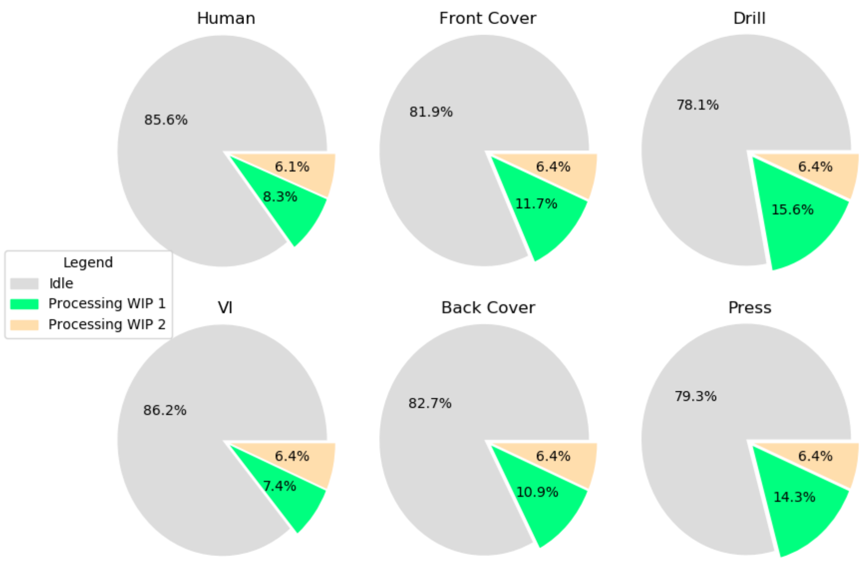


Figure 4.14

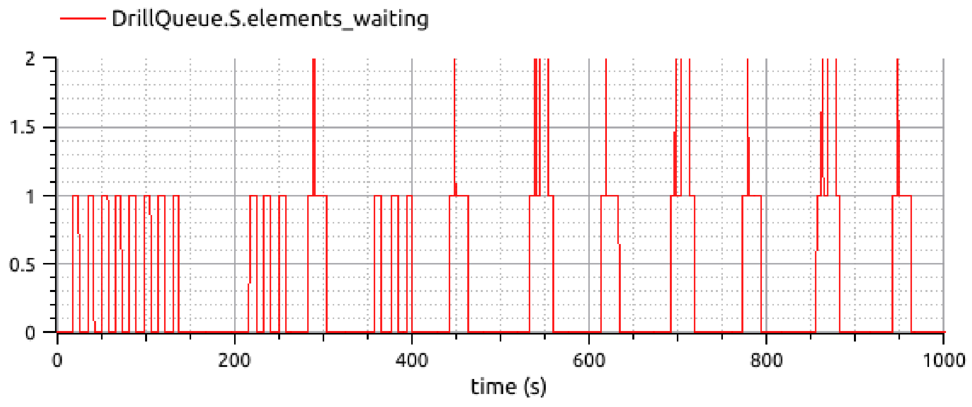


Figure 4.15

The average waiting times before other stations can be seen in Figure 4.16. Pay attention, these quantities are not related to the duration of the travel from one station to the next. They express the average interval of time that a carrier spends being stopped in the queue before arriving to its server.

Finally, we can inspect the proportion between WIPs of type 1 and 2. Since only one sector of the plant is entitled to let the nature of a WIP change, the robot's one, to know how many WIPs of every kind are circulating inside the system it is sufficient to set a dedicated statistical variable inside the Robot's queue model. We recall that it is a special block named `TwoWayQueue`, which takes WIPs in input and, instead of blocking its ports, redirects carriers in excess towards an alternative output port with a specific connector's variable augmentation of value 2.

Whenever a WIP of type 1 has to be deviated we know that in the system there will be a new type 2 WIP, while respectively we can say that a WIP of type 2 will

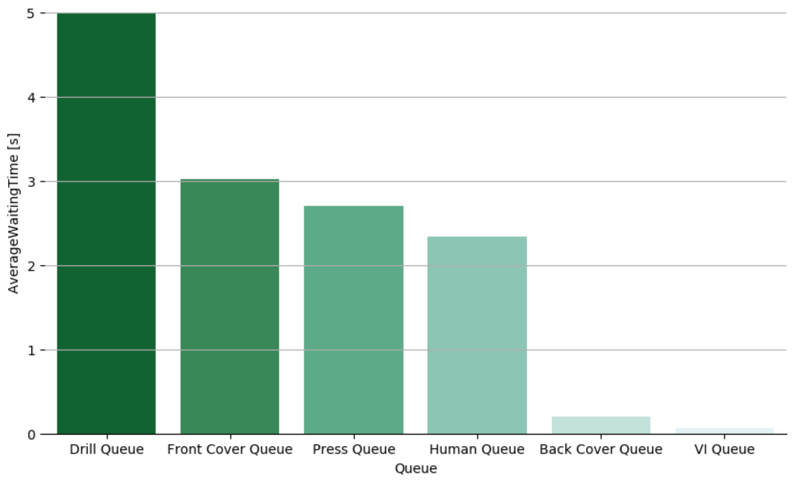


Figure 4.16

transform itself when succesfully acquired by the queue and set to be in line for being processed by the Robot.

By using this information we are allowed to know at any time how many items of both types are circulating inside the plant. On average, almost 3 WIPs of type 2 circulate inside the plant, which sounds acceptable considering that following the above presented idea usually in the system there are already 3 type 1 WIPs inside the CP-Factory.

In the end we can conclude that this plant's performance is greatly influenced by the above cited CP-Factory, whose process takes the majority of the working cycle time. Probably if it wasn't for the characteristic deviation system of its queue we would just experience a great column of carriers preceding the Robot, which at times would release manufactured products that would be finally processed by the remaining CP-Labs.

With respect to the CP-Factory, CP-Labs show a uniform degree of performance, and the human operator seems to have a considerable amount of time to spend away from his station. In fact, usually the real operator is a student who is not only entitled to accomplish the tasks required by the Manual Station, but has also to supervise the plant while checking the experiment being carried on the lab's instrumentation.

4.4 Final considerations

We have seen how a real industrial plant can be modeled and simulated using a set of blocks developed in Modelica language. The great advantage of having an atomic model that allows us to implement components compliant with the parallel DEVS specification is that it becomes possible to approach a great variety of situations, relying on the great amount of customizability of our simulation blocks.

In this situation we found a system very distant from the Bank Teller and similar examples, with physical entities having a defined length and volume moving on a mechanical conveyor belt, thus with travel times which couldn't be ignored. Plus, a particular kind of queue characterized the plant, with different kind of WIPs sharing the same working line simultaneously.

Still, given the relative simplicity of the system with respect to larger industrial plants, it was easy to come up with different blocks, in some case readapt them from `EventsLib`, change some of their features and obtain a new model representing precisely the test case encountered.

With the declaration of a dedicated record to store statistics it was possible to obtain useful insights on many aspects of the plant's lavorations. For convenience this particular set of variables was calculated inside the transitions' code. Even though the use of the variable `elapsed_time` inside internal transition is not contemplated by the DEVS specification, the statistical variables have no effect in determining the evolution of the system's state. They just monitor certain characteristics of the model.

With this test case it is possible to consider the Modelica language fully able to run reliable discrete event simulations. Recall that the model was coded purely in Modelica language, and that the event iteration mechanism proved apt to handling multiple consecutive transitions of DEVS models.

Conclusions

The starting aim of this thesis was to explore the possibilities offered by the Modelica's event iteration mechanism to run discrete events simulations. At the end of this work we can conclude that Modelica is fully capable to simulate discrete events systems.

The possible algebraic loops arising when models trigger events back to the original source can be solved using delays in the event chain. With the adoption of the parallel DEVS formalism it resulted practical to handle the algebraic loops directly in the approach developed to simulate DEVS systems.

The combination of the DEVS formalism and the Modelica simulation environment guarantees to simulate coupled DEVS by defining blocks and connecting them with an interface of choice. It has been shown that a set of one-to-one connections with a pair of integers can be used to model a real industrial plant, but it is possible to pick up connectors with more variables to send elaborated messages, if it helps in dealing with the representation of more complex systems.

At the time in which this thesis is written the simulation of discrete event systems in Modelica can be considered an open field of work.

The most natural evolution for this kind of activity is the modeling and simulation of hybrid systems, which combine continous time and discrete event charactiristics. A straightforward case can be one in which a server doesn't have a specified timeadvance to select the end instant of a process, but resorts to a continous time model to determine the progress of its job.

Modelica is well suited towards continous time systems, its Standard Library is wide and covers a variety of models coming from different domains, and being able to simulate discrete event systems opens a vast set of options to introduce hybrid modeling among the practices adopted by many industrial realities. In contrast to SimEvents, which already provides hybrid modeling capabilities using the totality of the SIMULINK enironment, Modelica offers the possibility to model systems with acausal equations, and with MODES a modeler can customize already existing discrete event models or create new ones applying an already broadly used formalism.

Also defining a practical way to transmit messages with structured data, representing for example entities features, can be an interesting line of development. The possibilities offered by Modelica's object orientation and the connector's class can be applied to design an extendable interface to forward records informations, allowing a user to define entities and then using their variables to characterize the simulation.

When developing a tool with these features, it seems obvious to try comparing it with already existing commercial softwares leading the sector. Arena is one of the most well-known, and we have seen Victorino Sanz used it as reference to design its DEVS in DESLib.

To match an environment like that many features still need to be implemented, but having a well defined foundation based on the DEVS formalism is a good point to start. Other future possible developments can be the definition a of random number generation system providing a broad choice of distributions.

We have also seen in section 4.3, when showing the results of the I4.0 Lab simulation, how many insights can be gained with variables representing statistics of interest for a figure designated to evaluate a system's performances. Identifying and developing statistics of interest for the different blocks can be very important when applying Modelica simulations to real cases of industrial plants.

In the end it can be said that Modelica is a language apt to model systems coming from a broad range of domains. The discrete events add a new dimension, potentially allowing to define standard ways to merge the simulation of complex physical processes with the oversight of the production line they belong to.

It is an area of research ready to be explored.

Appendix A

EventsLib blocks

In this appendix the algorithm sections that constitute some of the blocks contained in EventsLib are going to be described in detail. This analysis may turn useful to anyone interested in using the library, but could also provide a possible start for whoever intends to implement new atomic DEVS models. Real world scenarios present a considerable variety of situations, while the development of discrete event simulation tools in Open Modelica is an open field of work. Thus, a possible modeler may at times consider practical to define new transition functions, or modify the ones already existing, to obtain blocks suited for his necessities.

The following content is not intended to define a standard procedure of how DEVS functions should be coded in Modelica, neither wants to influence any future development in this area. It should just be considered a little perspective on the author's personal experience when deploying DEVS models into Modelica.

As already explained over the course of this thesis, a DEVS model outputs messages right before every internal transition, which is triggered when the residual lifetime of a status reaches zero value. We have seen that in the implementation of the `Atomic` block, which constitutes a root class for all these models, the time instant in which this situation is going to happen is held inside a variable named `next_TS`, which is compared to the effective simulation time to trigger Modelica time events.

When designing atomic DEVS models, it is asked to the modeler to adopt a mentality in which the state evolution has to be planned accordingly to the output that is going to be propagated at the next internal event.

Also, we have seen in 3.2 that in EventsLib an approach has been taken to consider messages as augmentations in front of the previous connectors' value. As a consequence, over the course of a simulation one will see the connectors' variable as an increasing series of steps. This is very useful when graphically inspecting the results, since the amount of events exchanged can be directly checked without the need to look at the event iteration internal procedure. Still, every DEVS model has to execute a subtraction to obtain the number of new events.

A.1 Generator

A generator is supposed to create new events and introduce them in the system at certain time instants. Those time instants can be distantiated in a determinisic or stochastic manner. In our case the interval between generations will be either constant or set by numbers coming from a random distribution.

Thus, it will have a certain number of output ports, whose quantity is going to be declared by a parameter, while a constant will set the value of `n_inputs` parameter to 0. Notice that in this work `n_inputs` and `n_outputs` will refer to the number of input and output ports, which both give rise to external events in the block, respectively by event arrival or from the request signal variation.

So, the declaration of our generator block will begin in the following way:

```
block Generator
    extends Atomic_PKG.Atomic(redeclare StateGen S,
      redeclare InputVariables_XGen X(n_outputs =
      n_outputs), redeclare OutputVariables_YGen Y,
      n_inputs = inputsNumber, n_outputs =
      outputsNumber, redeclare Boolean external_event[
      n_outputs]);
    constant Integer inputsNumber = 0 "number of input
      ports of the system";
    parameter Integer outputsNumber = 1 "number of
      output ports of the system";
    Out_Port OUT[n_outputs];
```

Remember that `n_inputs` and `n_outputs` are protected variables of the atomic block, and that the modeler cannot access them when opening the properties tab of the block in graphical environment. In **EventsLib**, to allow for a quick manipulation of the required number of ports, parameters `inputsNumber` and `outputsNumber` are used. To restrict any modification of the number of input ports, its number has been declared as a constant.

As seen in the basic Server example at 3.1.3, new records extending the base ones need to be used to redeclare the instantiations of `S`, `X` and `Y`, respectively the state, the input bag and the output result of our system. `external_event` is declared to have a boolean for every output port, so that an eventual blockage of ports can be detected.

Now we can concentrate on the behaviour our component is required to model. Potentially this block should be able to generate simultaneously more than one event, and thus it will start a new generation once all the already generated events have been sent.

Also a generator needs to be stopped from sending more events when one of its output ports is blocked. So external events coming from output ports are supposed to be used to update a state variable, that will be used to choose if the next internal transition will be scheduled normally or at infinite time.

Also, once generated, events may have to be sent one by one or in a single batch, depending on a parameter set by the user. In this case, an idle internal transition shall pass between transmissions, so as to give the possibility for a receiving model to block its ports, like depicted in figure A.1.

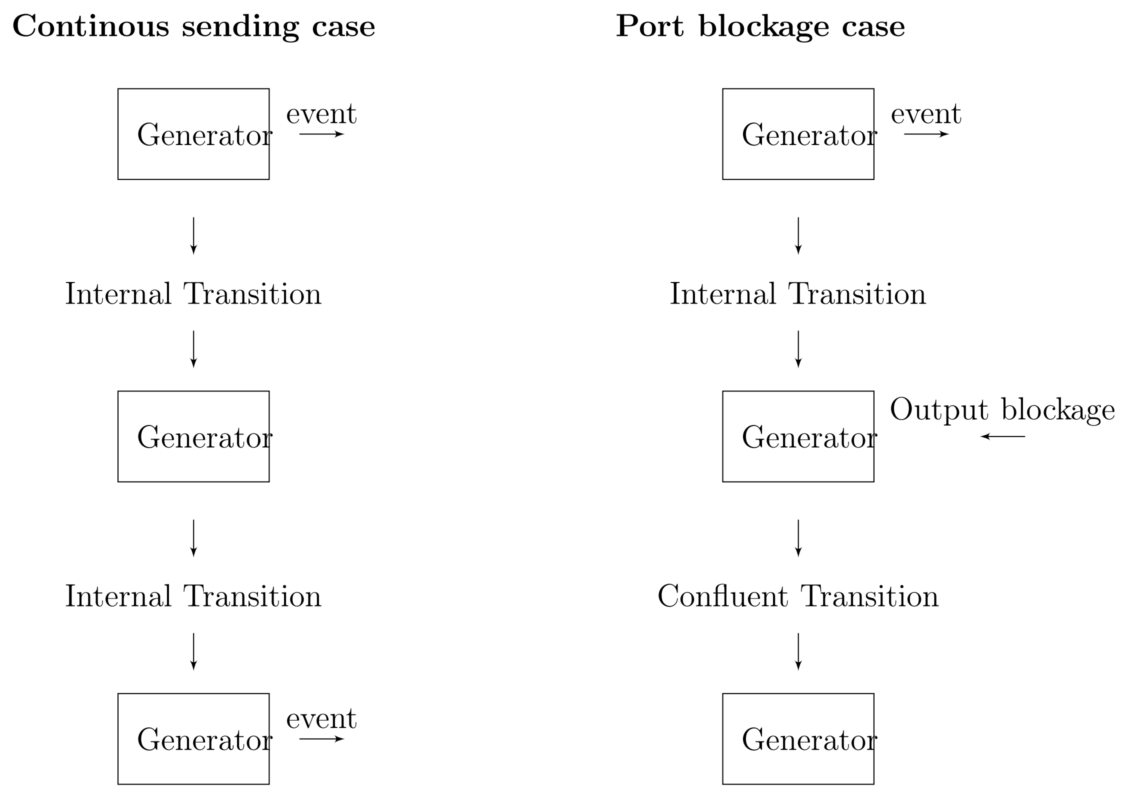


Figure A.1: Figure representing the generator's transmission procedure to send events through multiple transitions. On the left, it can be seen that the block will employ a transition time after every transmission to give receivers the possibility to block their ports in case the last event made them reach their capacity limit. On the right, when a receiver blocks a port the generator goes through a confluent transition and after that stops the transmission until the port si opened again

This allows our block to generate even a huge amount of events and then send them gradually without risking to overflow the receiver. Remember that this entire process happens at time stopped, requiring five event iteration steps for every transition. That is why, when the receiver has no capacity issues, it may be more convenient to send a unique batch of n events, augmenting by n the connector's variable in a single transition.

Thus, the possible operations required can be delineated:

Internal transtion:

1. In case an event had been sent after the last transition, launch a new immediate transition to eventually send the next event in line between two transitions.
2. In case no event had been sent, send one.
3. In case all events have been sent, schedule a new generation advanced in time.

4. Determine the next internal transition time depending on previous evaluations.

External transtion:

1. Check if output ports are blocked or open.
2. Determine the next transition based on the necessity of a transmission or the blockage of ports.

Confluent transtion:

1. Check if ports are blocked or open.
2. In case an event had been sent after the last transition and, if output ports are open, launch a new immediate transition to eventually send the next event in line after two transitions.
3. In case no event had been sent, send one if ports are open.
4. In case all events have been sent, schedule a new generation advanced in time.
5. Determine the next internal transition time depending on previous evaluations.

As we can already see, the introduction of a port being blocked and possibly multiple consecutive transmissions brings our atomic model to a point where each internal event will no more coincide with a generation.

Before presenting the state, it is convenient to show the outside elements that will take part in the state updating procedure, which are the parameters mentioned in 3.3 plus the external input variables:

- `sendEventsSingularly`: is the boolean entitled to determine the two aforementioned transmission modalities, one in which singular event transmissions are activated like in image A.1 and the one in which the connector's `event_signal` increases directly once by the number of events generated.

- `outputBatchDimension`: represents effectively the number of events created at each generation.

- `X.portBlocked[n_outputs]`: is the variable from the instantiation of record `InputVariables_X`, which is updated using connectors `event_request_signal` for every output port in the following algorithm section:

```
when pre(external_event) then
    for i in 1:n_outputs loop
        X.portBlocked[i] := pre(OUT[i].
           event_request_signal) == -1 "If the
           output port's request value is -1, the
           port is considered blocked";
    end for;
end when;
```

It tells us the condition of everyone of the output ports' statuses by comparing `event_request_signal` with the selected value for representing the blockage of a port, that is -1. `OUT` is the array of output ports of the system.

Now, we can look at the state of the system and its variables:

```
record StateGen
    extends Atomic_PKG.State;
    discrete Real next_TS "Effective time instant of
       next internal transition";
    discrete Real next_scheduled_generation "Since
       next_TS is affected by blocking of ports, this
       variable stores the completion time instant";
    Integer events_already_sent "Events that the block
       had already sent through its output ports";
    Boolean blocked_output "true when
       OUT.event_request_signal differs from zero, thus
        output is blocked";
    Integer events_in_batch "Events belonging to the
       last generated batch still to be sent";
    Integer events_being_sent "Events that are going to
        be sent at the next internal event";
    Boolean immediate_transition_required "boolean set
       to true everytime that the next status will have
        0 lifetime";
end StateGen;
```

- `next_TS` and `next_scheduled_generation` are the state variables used to set the internal transition time instants. While `next_scheduled_generation` indicates the time instant supposed to trigger the next generation, not necessarily it coincides with the next internal transition effective time. In case of ports being closed, the generator is required anyway to remember when new events would be ready to be introduced in the system. In this way, if ports are reopened before the scheduled generation time instant, the effective behaviour of the block would remain unchanged. So, by using a variable to represent the real internal transition time and another one to generation completion effective time instant we can model the fact that our generator works independently from the ports condition, but just sends events when ports are free to be used.

- `blocked_output` is updated using variables coming from `X`. If any of the possibly multiple output ports results blocked, this state variable is used to store this information inside the state and stop sending operations.

- `events_in_batch`, `events_already_sent` and `events_being_sent` are the variables used to handle the procedure of events sending through output ports. `events_in_batch` represents the number of events that the generator awaits to send. So, when this variable reaches zero value, it will be time to schedule a new event generation. `events_already_sent` and `events_being_sent` are used to handle the sending mechanism. Every $\lambda(s)$ function sums these two variables to set the connectors' event signals. While `events_already_sent` stores the amount of events already sent in the connector, `events_being_sent` indicates the specific number of new events that will be transmitted at the next activation of $\lambda(s)$ algorithm section. After every transmission `events_already_sent` will have to be updated by adding to it `events_being_sent`, while we will have to recall to put `events_being_sent` to 0 to avoid an excessive and wrong amount of sendings.

- `immediate_transition_required` is a flag variable used to explicitly command a state with zero lifetime and, probably, propagate an output. It will be set to false at the beginning of every transition and then set to true whenever necessary over the course of the various evaluations.

Now, we can finally check the transitions code, to gain an insight on the algorithms used to determine this DEVS model's behaviour.

Confluent transition

Whenever a generator receives an external event, it surely means that one of its output ports' condition changed. Thus, after setting to false the immediate transition flag, a for loop will be used to update the state variable entitled to consider the blockage of ports.

```
S.immediate_transition_required := false "at the
   beginning of every transition we assume no immediate
    transition will be needed, to then check if it is
   over the course of the function";
S.blocked_output := false "At the beginning of every
   transition it is assumed that the output is not
   blocked, to then check for every port";
for i in 1:n_outputs loop
    if X.portBlocked[i] then
        S.blocked_output := true "If any port is found
           blocked the whole system considers the
           output blocked";
    end if;
end for "loop aimed at checking if any of the output
   ports is considered blocked";
```

In this moment, we know exactly the port statuses, and we will use this information to decide if triggering another transition is necessary or not.

Now the confluent transition function needs to evaluate how to change the state considering if at the last transition a new event had been sent in output or not. Recalling figure A.1, this condition determines if at the second next transition an event will be sent.

Thus, we evaluate the two possibilities using an if else statement, whose first branch is shown below, in which the case where our output has been just augmented is handled:

```
if S.events_being_sent > 0 then
    S.events_already_sent := S.events_already_sent +
       S.events_being_sent "if an event had been sent
       during the last transition, add it to the
       already sent events";
    if S.events_in_batch > 0 then
        S.events_being_sent := 0;
        if not S.blocked_output then
            S.immediate_transition_required := true "if
                an event has been sent but there are
               still and the ports are open schedule a
               transition with no outputs";
        end if;
    else //if no other events are left to be sent,
       schedule a new generation
        S.next_scheduled_generation := time +
           RNGBlock.randomValue "when a new generation
           is started, the next internal event is
           scheduled to be in advance by an amount of
           time set by the RNG";
        RNGBlock.trigger := RNGBlock.trigger + 1 "
           augmenting the RNG trigger to require the
           generation of a new random number";
        if sendEventsSingularly then
            S.events_in_batch := outputBatchDimension -
                1;
            S.events_being_sent := 1;
        else
            S.events_in_batch := 0;
            S.events_being_sent := outputBatchDimension
               ;
        end if "if events have to be sent one by one,
           send one, otherwise send entire batch";
    end if "if there are still events left to send,
       schedule an idle transition, otherwise start a
       new generation";
```

The first instruction to be given in this case is to update the number of events already sent. Then, an evaluation on what still remains to be sent is carried, being this value stored inside `events_in_batch`.

- If there are still events left, at first set to 0 the next sending, and then if the port is unlocked, trigger a consecutive internal transition to wait for an eventual blockage by the receiver.

- If there are no more events left to be sent, it means that a new generation needs to be scheduled. The timeadvance is given by the random number provided by **RNGBlock**, a block declared inside the generator which provides for random numbers. Its variable `trigger` needs to be increased to let the block know that a new random value needs to be prepared for the next transition.
  Once the new generation is scheduled, all the events inside the batch and the ones to be sent are prepared depending on the user's need to send event one by one or together in a single message.

In the other case, we should take in consideration the option to directly send an event if there is any left and the output ports are open. Again the sending is influenced by the ports' condition and by the modeler's intention to stagger or not the transmission.

If an event had not been sent at the last transition the number of `events_in_batch` has to be bigger than 0. Still, for completeness of the behaviour's definition and for clarity, the case of a new generation schedulation is considered. Notice that a generation is scheduled indipendently from an eventual ports' blockage. This because the generator is required to provide for new events, if their future transmission may be blocked.

```
else //if no events have been sent before this
   transition
    if S.events_in_batch > 0 then
        if not S.blocked_output then
            S.immediate_transition_required := true "if
               the last transition did not output
              anything, there are still elements to be
               sent and the outputs are open, send
              immediately an event";
            if sendEventsSingularly then
                S.events_in_batch := S.events_in_batch
                   - 1;
                S.events_being_sent := 1;
            else
                S.events_being_sent :=
                   S.events_in_batch;
                S.events_in_batch := 0 ;
            end if "if events have to be sent one by
              one, send one, otherwise send entire
              batch";
        end if;
    else //if no events have been sent but the batch is
        somehow empty, schedule a new generation. This
       case is almost impossible, but still it is
       treated
        S.next_scheduled_generation := time +
           RNGBlock.randomValue "next generation is
```

```
            scheduled at a timedvance in advance to the
            actual time";
        RNGBlock.trigger := RNGBlock.trigger + 1 "
            augmenting the RNG trigger to require the
            generation of a new random number";
        if sendEventsSingularly then
            S.events_in_batch := outputBatchDimension -
                1;
            S.events_being_sent := 1;
        else
            S.events_in_batch := 0;
            S.events_being_sent := outputBatchDimension
                ;
        end if "if events have to be sent one by one,
            send one, otherwise send entire batch";
    end if;
end if "if branch used to decide what to do right after
    an internal event depending on the events sent";
```

Finally we can decide the next internal transition time depending on the other state's variables. At first the most relevant condition is the ports' status: until they are blocked the next internal event will be scheduled at infinite.

Otherwise, if an immediate transition had been required a function will change `S.next_TS` to the smallest integer negative number among the values already smaller than `S.next_TS`, to assign the new state 0 lifetime and thus let a new consecutive transition happen inside the same event iteration chain.

In a normal generation situation instead, the next internal transition will correspond with the next generation scheduled, whose timeadvance has been already defined when programming the event.

```
if S.blocked_output then
    S.next_TS := Modelica.Constants.inf "until output
        is blocked next transition is scheduled at
        infinite time";
elseif S.immediate_transition_required then
    S.next_TS := Functions.ImmediateInternalTransition(
        S.next_TS) "if an immediate transition is
        required, this function changes next_TS to a
        negative number lesser than the actual one";
else
    S.next_TS := S.next_scheduled_generation "otherwise
         the next internal event will be when normally
        scheduled";
end if;
```

Notice that `S.next_TS` is not the `next_TS` variable used to trigger the transitions. The value of `next_TS` will have to be assigned in the apposite algorithm section representing the $ta(s)$ function.

External transition

An external transition is only entitled to check if the ports' blockage condition changes, as at the beginning of the already seen confluent transition. Next, it determines like the confluent transition the next internal transition time. In case a port had been blocked for a certain period of time and then it is opened after the generation was scheduled, the `next_TS` will end up taking the value of `next_scheduled_generation`, which being `< time` will immediately set the status lifetime to 0 and trigger a new internal transition to send the first generated event.

Internal transition

The internal transition can be identical to the confluent one excluding the update of the `blocked_output` state variable. Recall that as the DEVS specification states, the incoming messages are not argument of the internal transition function, so `X` record should not be used when updating the state with an internal transition.

We have seen how the transition functions of an atomic model can be written using an imperative programming style. For completeness, the algorithm section updating `Y` record and the equations assigning value to output ports and external event boolean array are shown below:

```
algorithm
    when pre(internal_transition_planned[1]) then
        Y.eventsSent := S.events_already_sent +
           S.events_being_sent "At transition time the
           number of outputs sent is equal to the value
            of events already sent plus the events set
           at this new lambda output. The transition
           will thus update the state number of
           elements already sent";
    end when;
equation
    for i in 1:n_outputs loop
        OUT[i].event_signal = Y.eventsSent;
        external_event[i] = change(OUT[i].
           event_request_signal);
    end for;
```

A.2 Queue

A queue is supposed to receive events, store them, and finally send them whenever required. Eventually, events and requests may come from different ports. Still, if the origin of an arrived event doesn't constitute a relevant information, it is important to account for the port of origin of a request to know where to send events.

The arrival of both events and requests is identified by an increase in the connectors' signals variables. The possibility of an output port being blocked is not even considered, since queues should be connected in output only to servers, which have no reason to block their ports since they just ask for and process events.

Instead, if a queue is required by the modeler to have a certain capacity, at a certain point our atomic model may be required to block its input ports, and then maybe reopen them once the amount of stored events comes again below the limit level imposed by the modeler.

In case a number of events superior the storing possibilities of the queue is present, the user can choose through a parameter if they can be retained or discarded. When analyzing the code, an alternative way in which events may not be lost while never surpassing the capacity imposed by the user will be explained. For conceptual reasons this solution was not applied, but will be presented when showing the code section dealing with the update of the state variable counting the number of events stored.

We have seen that a queue has two different possible typologies of answers it can transmit through its connectors: the input ports' status, and events sent to servers which required them. Both these outputs will have to propagated using immediate internal transitions, since the queue may be seen as a "passive" component, which only reacts to external factors but doesn't generate anything on its own.

The case of a queue needing a service time to send events has not been treated, because of the complications that may arise when different requests have to be satisfied at different time instants over the span of a first request satisfaction. To model this possibility the addition of a **Delay** block between a queue's output port and a server may be used, as explained in 3.3. In this case, resorting to a coupled DEVS model instead of having a more complex singular atomic one has been preferred.

As a modeling choice, it has been decided that a queue first stores the received events, then removes the ones it sends to satisfy requests, and only in the end checks if the capacity has been reached. Thus, the queue can act as a transitory place for events passing through a system.

At this point, we can start to think about the operations required for every transition function:

Internal transtion:

1. If an internal event happened it means that an output needed to be sent. Being the queue a reactive block, without external events its state doesn't need to change and the next internal event can be scheduled at infinite.

External transtion:

1. In case of external event, a queue has to update the amount of events it stores if any has arrived.

2. Then, it checks if new requests have arrived, and accounts for the number of events every output port currently needs.
3. If there are requests that can be satisfied with events stored, then the queue sends them.
4. At this point, the amount of events stored inside the queue can be checked. If it turns out that input ports statuses must be changed, then the queue changes them.
5. Finally, decide the new status lifetime, 0 if some output needs to be propagated, otherwise infinite.

Confluent transtion:
At the beginning of a confluent transition, we are aware that we have sent something in output, but at the same time an external input arrived. Since having sent an output doesn't influence in any way our new state, the confluent transition function will simply execute the same procedure of the external transition function, that is:

1. Update the amount of events it stores if any has arrived.
2. Then, it checks if new requests have arrived, and accounts for the number of events every output port currently needs.
3. If there are requests that can be satisfied with events stored, then the queue sends them.
4. At this point, the amount of events stored inside the queue can be checked. If it turns out that input ports statuses must be changed, then the queue changes them.
5. Finally, decide the new status lifetime, 0 if some output needs to be propagated, otherwise infinite.

Now, we can see the variables involved in the transition process while recalling the parameters of the block.

Input Variables

The input variables contained inside the X record are the following:

```
Integer eventsArrived[n_inputs] "number of elements
   arrived in input";
Integer requestsArrived[n_outputs] "number of requests
   arrived from outputs";
```

The first variable indicates the total amount of events received at input ports, while the second one counts the requests. Both these variables are arrays, and this fact is very important to let the model know from which output port requests came.

Output Variables

```
Integer eventsSent[n_outputs] "elements sent as output
   to requesting blocks";
Integer inputPortsRequestValue "Value used to
   eventually block input ports";
```

The first array will contain the amount of events sent to every output port. Notice that while the generator only needed to know the amount of events to be sent, to then send them equally through its output ports, the queue has a variable for every requirer.

`inputPortsRequestValue` is the value to be given to the `event_request_signal` connector variable for all the input ports. Normally it stays at zero, but if a queue blocks its inputs this value is set to -1.

Block's parameters

We have already cited the queue's parameters in 3.3, but we will recall them here to help the reader:

- `capacity` defines the maximum amount of events that the queue is supposed to store. If no unlimited capacity is desired, this value will be set to -1, and that will be interpreted as the fact that the queue can hold an infinite amount of events.
- `fragmentedService` specifies between two policies for pending requests satisfaction. If true, the queue is allowed to partially satisfy pending requests. For example, if a server required two events, when an event arrives it is directly sent to that server, even if it doesn't satisfy completely the request. When false, a queue will satisfy a pending request only if it has enough events to liquidate it entirely.
- `allowEventsInExcess` will be used to specify whether or not events surpassing the capacity limit will be kept or discarded.
- `startingEventsInQueue` specifies the amount of events which the queue initially possesses, and it is used in the initial algorithm section of the block.

State Variables

Here are shown variables defining the state of our queue:

```
parameter Integer n_inputs;
parameter Integer n_outputs;
Boolean immediate_transition_required "boolean set to
   true everytime that the next status will have 0
   lifetime";
discrete Real next_TS "time instant of next internal
   transition";
```

```
Integer events_in_queue "number of elements in queue";
Integer acquired_events[n_inputs] "value of elements
   arrived at the last transition from input ports";
Integer requests_acquired[n_outputs] "value of requests
    arrived at the last transition from input ports";
Integer pending_requests[n_outputs] "requests awaiting
   to be satisfied for every input port";
Integer events_sent[n_outputs] "elements to be sent out
    of output ports";
Boolean blocking_input_ports "If true then the queue
   reached maximum capacity and so input ports need to
   be blocked";
Integer eventsDiscarded "amount of events arrived that
   could not be retained because of capacity issues and
    that will be discarded from the system";
```

- immediate_transition_required and next_TS have the same role as the namesake terms in the generator's state. immediate_transition_required is used inside the algorithm section to explicitly require a state with zero lifetime and thus output something. next_TS contains the time instant for the next internal event.

- events_in_queue counts the number of events the queue stores.

- acquired_events[n_inputs] indicates, for every input port, how many of the total events arrived have been collected. Since events are sent by augmentation, practically it becomes necessary for the receiver to recall the past input value, so that through a subtraction the events newly arrived can be obtained.

- requests_acquired[n_outputs], pending_requests[n_outputs] and events_sent[n_outputs] are the terms in charge of acquiring, registering and satisfying the requests from servers connected in output. Like acquired_events, requests_acquired is used to store the value of input variables at previous transitions, so as to obtain by subtraction the recent requests.
  pending_requests stores the current need for events of every server connected in output. Whenever we receive a request and satisfy it, this value shall be 0. But if we receive a request and we are not able to satisfy it because our queue is empty, then this value can stay at 1 to recall through the various transitions that the server at that port is still waiting for an event. Whenever we will be able to send an event through that port, we will put that variable to 0, and then augment it again at the next request. With this array the queue knows where to redirect the events it stores.
  events_sent is the array of variables that we augment to transmit events through output ports. This variable will be used to set the values Y.eventsSent in the $\lambda(s)$ algorithm section.

- `blocking_input_ports` is the variable used to indicate whether or not input ports have to be blocked depending on the relation between the number of events stored and the capacity of the queue. Note that setting this variable to `true` does not automatically block our ports. An internal transition needs to be scheduled instantly too to let the $\lambda(s)$ routine output the closure condition.

- `eventsDiscarded` is the variable that counts the events eliminated from the system due to an excess in capacity. Just recall that if `allowEventsInExcess` parameter is set to true this variable will always stay at 0.

Now that we have described the state variables, we are ready to look at this block's transitions.

Confluent transition

As above explained, nothing changes in a queue that has gone through an internal event. So, a confluent transition just has to handle the external influences on the block.

At the very beginning the first operation to be carried is the acquiral of new events in input:

```
for i in 1:n_inputs loop
    S.events_in_queue := S.events_in_queue +
      X.eventsArrived[i] - S.acquired_events[i]"update
       the number of elements in the queue considering
       the difference between actual and previous
      inputs";
    S.acquired_events[i] := X.eventsArrived[i] "at the
      next transition the previous inputs will be
      equal to the actual ones";
end for "Cicle aimed at updating the events_in_queue
   value";
```

For every possible source of events the storage variable is increased by the difference between the input's value and the value at the previous transition, which is finally updated.

Notice that in this way all events arrived are acquired, without accounting for capacity issues. If one may turn interested in avoiding the problem of capacity excess, it is just possible to acquire a sufficient amount of events so as to stay below the capacity limit.

As said at the beginning of the queue's paragraph, this approach was discarded, since basically it would imply that some events have been sent but they are not acquired, and thus they would conceptually be somewhere in a connector's space, whose physical counterpart in certain applications may be ambigous or inexistent.

As a modeling paradygm, it has been chosen that events sent have to arrive and be retained instantaneously by another block, without leaving in the connector something waiting to be acquired successfully. If someone decides not to acquired them, input ports may be blocked anyways, stopping the transmission of new events, while the unacquired events may be collected with time while the queue empties along the simulation.

Still, the same behaviour can be achieved by just letting the queue store events above its capacity with the apposite parameter. In this case, the state variables directly reflect the connectors influence on the block, and conceptually it was considered a better solution.

Next, a similar loop updates the requests.

```
for i in 1:n_outputs loop
    if X.requestsArrived[i] - S.requests_acquired[i] >
       0 then
         S.pending_requests[i] := S.pending_requests[i]
           + X.requestsArrived[i] - S.requests_acquired
           [i] "number of previous requests is updated
           according to the difference between actual
           and previous requests";
         S.requests_acquired[i] := X.requestsArrived[i]
           "at the next transition the previous
           requests will be equal to the actual ones";
    end if;
end for "Cicle aimed at updating the pending_requests";
```

Notice that while for the events contained in the queue there was only one variable, here requests are updated differently for every output port. In this way our model knows the needs for every server, and thus will use this information to redirect its stored events.

In addition, a check on the arrival of new requests is conducted with an if before proceeding to the update. In this way, if for example a fault in modeling occurs and an output connector is attached to the input of another queue, which at a certain point may need to close its ports due to arrivals from other inputs, no negative pending requests happen in this queue. Of course the starting value for every element of `S.requests_acquired` is 0.

At this point the queue has precise informations about the stored events and the needs of servers at its outputs. The next task consists in satisfying all the possible pending requests. For every output port, in case of a pending request it is checked whether or not there are enough events to satisfy it.

```
S.immediate_transition_required := false "At the
   beginning of requests evaluations we assume no
   response is needed";
for i in 1:n_outputs loop //check requests from every
   output port
    if S.pending_requests[i] > 0 then //in case a
       pending request from port i is present
         if S.events_in_queue >= S.pending_requests[i]
           then
```

```
            S.events_sent[i] := S.events_sent[i] +
               S.pending_requests[i] "elements sent at
               output i is augmented enough to satisfy
               the request";
            S.events_in_queue := S.events_in_queue -
               S.pending_requests[i] "elements are
               effectively prelevated from the queue";
            S.pending_requests[i] := 0 "No more
               requests are left pending after
               satisfaction";
            S.immediate_transition_required := true "we
                flag that we will need an internal
               transition to output the elements to be
               sent";
        elseif fragmentedService and S.events_in_queue
           > 0 then //In case the queue doesn't have
           enough elements to satisfy the request, in
           case of unitary service the request is
           fulfilled partially
            S.events_sent[i] := S.events_sent[i] +
               S.events_in_queue "All elements in the
               queue are sent in output";
            S.pending_requests[i] := S.pending_requests
               [i] - S.events_in_queue "the pending
               request is reduced by the number of
               elements the queue contained";
            S.events_in_queue := 0 "the queue is
               emptied";
            S.immediate_transition_required := true "we
                flag that we will need an internal
               transition";
        end if;
    end if;
end for;
```

In case `fragmentedService` is allowed, events are sent also for requests which can't be fully satisfied. Every time a new event is scheduled to be sent, `S.immediate_transition_required` is flagged true, since there is a state modification with consequences on the output.

Once the servicing phase is completed, the capacity issues need to be checked, to verify if our queue is full and act consequently.

```
if S.events_in_queue > capacity and capacity > -1 and
   not allowEventsInExcess then
    S.eventsDiscarded := S.eventsDiscarded +
       S.events_in_queue - capacity;
    S.events_in_queue := capacity;
end if "If the queue has a certain capacity, after
```

```
   having satisfied requests, events in excess are
   removed";
```

After that, it is determined if a variation in the input ports blockage must happen.

```
if capacity > -1 and S.events_in_queue >= capacity and
   not S.blocking_input_ports then //In case the queue
   contains more elements than allowed and the input
   ports are not blocked, then block them immediately
   by setting immediate_transition_required to true
    S.blocking_input_ports := true;
    S.immediate_transition_required := true;
elseif S.events_in_queue < capacity and
   S.blocking_input_ports then //In case the port was
   locked but there is room for elements, unlock the
   inputs immediately by setting
   immediate_transition_required to true
    S.blocking_input_ports := false;
    S.immediate_transition_required := true;
end if "if branch used to handle the blocking of input
   ports";
```

In case a variation is necessary, to be sure it will be reflected in output the flag `S.immediate_transition_required` is again set to true. So, even if no events are sent to satisfy requests an eventual blockage of ports can be achieved at the next transition.

Finally, depeding on if an output response is necessary or not the new state's lifetime can be set to either 0 or infinite.

```
if S.immediate_transition_required then
    S.next_TS := Functions.ImmediateInternalTransition(
       S.next_TS) "In case the queue has to send
       elements or inputs need to blocked, next
       internal transition needs to be immediate";
else
    S.next_TS := Modelica.Constants.inf "in case no
       elements need to be sent and ports are unlocked,
        the queue becomes passive and sets to infity
       its next internal event";
end if;
```

External transition

As said above, the external transition is equal to the confluent one.

Internal transition

An internal transition happens after having sent outputs without having received any external event. In this case, the queue is left idle, waiting for new events to happen.

```
S.immediate_transition_required := false "at the
   beginning of every transition we assume no immediate
    transition will be needed, to then check if it is
   over the course of the function";
S.next_TS := Modelica.Constants.inf;
```

A.3 Server

In 3.1.3 we already saw an implementation of a server supposed to represent the processing action that characterizes many physical cases, from the Bank Teller of the famous example to machines and stations inside industrial plants.

The server block `EventsLib`'s server block is supposed to have the same features, but with the possibility to require and send more than one event, having a sending mechanism identical to the generator's one, exposed in figure A.1, while monitoring if any of its output ports happens to be blocked.

As for the generator, in case at least one output port results closed, it was decided that the server block should not send anything. However, independently from the output closure condition, a server should complete an already started process in the planned time. In this way its effective activity can be monitored without considering eventual bottlenecks. Otherwise, one may experience a simulated process lasting for a long time, while in reality the real server would have been idle, waiting to send its outputs before querying its input and beginning another operation.

Every server is restricted to have only one input port, so that its requests are served by only one queue. If a modeler is willing to let the server access to multiple sources of events, these sources should be connected to a queue which eventually would deliver events to more servers according to their necessities.

So there are two possible outputs that the server block will be able to send:

- Events through its output ports, which would either come to a displacer or another block that would again delay or condition in any other way their journey inside the system.
- Explicit requests to its queue. A different block, for a example a generator, would simply ignore the queries and send its events at his own rythm, and the server would simply acquire them and store them until it processes them. The system would keep on going in the same way.

At the same time a server can receive two different kind of input signals:

- Events coming through an input port, which in normal conditions would be the ones demanded and provided by a queue.

- The blockage or opening of its output ports, which can dictate the sending action of events.

Now we can already delineate the operations every transition will have to execute.

Internal transtion:

1. After an internal event, two possibilities arise:
   - A process is finished and the server obtains some new events to send.
   - The server is not actively working on a job and it's sending the completed events. In this case it can either be in one of the two conditions described in the generator's section, that are having already transmitted an event or having spent a transition being idle to check if the receiver didn't block its ports.
2. In the first case, the server adds the new events to a sending variable and considers their transmission depending on the blockage of its outputs.
3. Otherwise, it keeps going on with its sending transitions, eventually sending a request together with the last event.
4. In case no events are left to send, the server awaits to obtain enough input events to start a new process.

External transtion:

1. If an external event arrived, at first the state variable regarding output ports blockage is updated.
2. Next, eventual events coming in input are acquired.
3. In case there are still events finished waiting to be sent, a transmission is prepared if no closure of output is detected.
4. If the server was not active and no processed events were remaining, it means that the block was waiting for new events in input to begin a new process. If enough have arrived, it shall use them to start a new process.
5. If instead the server is active, then it should continue until completion without changing anything.
6. Finally, depending on the three previous evaluations, the lifetime of the new state can be determined.

Confluent transtion:

1. In case of confluent transition the effect of external events is evaluated like in the first two steps of the external transition.
2. After that, the same evaluations characterizing the internal transition are carried.

Input Variables

The record X contains variables to reflect the two possible inputs that the block can receive

```
parameter Integer n_inputs;
parameter Integer n_outputs;
Integer eventsArrived[n_inputs] "Events arrived from
   input ports";
Boolean portBlocked[n_outputs] "Array that stores
   information about blockage of output ports (an
   output port is considered blocked when its request
   value -1)";
```

portBlocked acts in the exact way as the generator's namesake variable. eventsArrived is an array with one element, since n_inputs is restricted by a constant of value 1. Still, its sequential nature is conserved to make easier an eventual development of a version with more input ports.

Output Variables

The record Y instead contains singular variables, since the request and the transmission are supposed to be uniform among all connectors.

```
Integer eventsSent "Events sent in output by the server
   ";
Integer request "requests for events emitted by the
   server";
```

Block's parameters

Server's parameters have been already described at 3.3, and here they are recalled:

- outputBatchDimension is the amount of events generated after every process is finished.

- requestDimension is the amount of required events to begin a process, and so it is the value by which Y.request is augmented every time all the finished events are successfully transmitted.

- sendEventsSingularly acts in the same way as the generator's namesake in controlling the way in which output events are forwarded.
  If true batches of event will be sent one by one, to let the receiver some transition steps to close its ports if necessary. If the system being modeled has no serious constraints on capacity, this parameter can be set to false so that all events will be sent together in the space of a unique transition. Having less transitions can be very useful when simulation events are analyzed.
  Notice that this variable acts only on the transmission of events in output. Requests are always forward entirely, since no ports blockage ever happens between queue-server connections.

State Variables

```
parameter Integer n_inputs;
discrete Real next_TS "time of next scheduled internal
   transition";
discrete Real next_scheduled_generation "Since next_TS
   is affected by blocking of ports, this variable
   stores the real completion time instant";
Integer events_acquired[n_inputs] "number of overall
   events acquired at input ports, effectively they are
    input events at the end of the previous transition"
   ;
Integer events_in_buffer "number of events currently
   ready to be processed but not being actually
   processed. Used to store events if request is served
    partially";
Integer events_processed "Elements successfully
   processed by the server";
Integer request "Requests sent by the server";
Boolean active "true if the server is precessing events
    or not";
Boolean blocked_output "true when
   IN.event_request_signal differs from zero, thus
   output is blocked";
Integer events_already_sent "Events that the block had
   already sent through its output ports";
Integer events_in_batch "Events belonging to the last
   generated batch still to be sent";
Integer events_being_sent "Events that are going to be
   sent at the next internal event";
Boolean immediate_transition_required "boolean set to
   true everytime that the next status will have 0
   lifetime";
```

- `next_TS` and `next_scheduled_generation` are the variables used to determine the lifetime of the new state by setting the time instant of the next internal event.

- `events_acquired[n_inputs]` and `events_in_buffer` are used as in the queue to acquire and store incoming events. The server needs its own variable to collect arrivals in case the queue is set with parameter `fragmentedService`, and thus answers to the original query may come spaced in time.

- `events_processed` counts the total number of events successfully processed by the block. It can be considered more an informative variable rather than a term influencing the behaviour of the model.

- `request` is the number of requests inoltrated by the block.
- `active` represents whether or not the server is processing a unit.
- `blocked_output` is updated using variables coming from `X`. If any of the possibly multiple output ports results blocked, this state variable is used to store this information inside the state and stop sending operations.
- `events_already_sent`, `events_in_batch` and `events_being_sent` act as the generator's namesake variables, and allow the server to follow the same transmission procedure of the generator.
- `immediate_transition_required` is the variable used to explicitly require a state with zero lifetime, so as to trigger a new output propagation.

Confluent transition

Like we said, whenever an external event influences the server, `blocked_output` and `events_in_buffer` need to be updated.

```
S.immediate_transition_required := false "at the
   beginning of every transition we assume no immediate
    transition will be needed, to then check if it is
   over the course of the function";

S.blocked_output := false "At the beginning of every
   transition it is assumed that the output is not
   blocked, to then check for every port";
for i in 1:n_outputs loop
    if X.portBlocked[i] then
        S.blocked_output := true "If any port is found
           blocked the whole system considers the
           output blocked";
    end if;
end for "loop aimed at checking if any of the output
   ports is considered blocked";

for i in 1:n_inputs loop
    S.events_in_buffer := S.events_in_buffer +
       X.eventsArrived[i] - S.events_acquired[i] "
       number of events ready to be processed are
       updated by the difference between actual and
       previous inputs";
    S.events_acquired[i] := X.eventsArrived[i] "at the
       next transition previous input elements will be
       equal to the actual ones";
end for "for loop to acquire arrived events";
```

After that, it is time to evaluate how to react to an eventual output transmitted at the last invocation of $\lambda(s)$.

If the server was active, an internal event indicates that our job is completed, and so we will have a new set of events to send. Notice that when a server finishes its job it doesn't propagate anything in output. If the output isn't blocked, we can already order a transmission.

```
if S.active then
    S.active := false "if server was active after an
       internal event it needs to be deactivated";
    S.next_scheduled_generation :=
       Modelica.Constants.inf "next process ending is
       scheduled at infinite";
    S.events_processed := S.events_processed +
       outputBatchDimension "events processed are
       augmented";
    S.events_in_batch := outputBatchDimension "events
       to be sent are put into batch";
    if not S.blocked_output then
        S.immediate_transition_required := true;
        if sendEventsSingularly then
            S.events_in_batch := S.events_in_batch - 1;
            S.events_being_sent := 1;
        else
            S.events_being_sent := S.events_in_batch;
            S.events_in_batch := 0;
        end if "if events have to be sent one by one,
           send one, otherwise send entire batch";
        if S.events_in_batch == 0 then
            S.request := S.request + requestDimension "
               a request is elaborated";
        end if "if no more events would remain to be
           sent, send a request too";
    end if "if output ports are open events can be sent
        too";
```

The next elseif branch deals with the situation in which the server is not processing anything and so it's in its sending phase. In this case the operations are similar to the ones of the generator, to articulate the transmission in different transitions.

In case the event we are sending is already the last of the batch, we propagate a request together with it. In case no more events are left and `events_in_buffer` has the necessary amount of events, we start a new lavoration.

```
elseif not S.active then
    if S.events_being_sent > 0 then
        S.events_already_sent := S.events_already_sent
           + S.events_being_sent "if an event had been
           sent during the last transition, add it to
           the already sent events";
        S.events_being_sent := 0;
        if S.events_in_batch > 0 then
            if not S.blocked_output then
                S.immediate_transition_required := true
                   "if an event has been sent but
                  there are still and the ports are
                  open schedule a transition with no
                  outputs";
            end if;
        elseif S.events_in_batch <= 0 then
            if S.events_in_buffer >= requestDimension
               then //if there are enough events to
               start the process
                S.next_scheduled_generation := time +
                  RNGBlock.randomValue "the next
                  process is scheduled to be finished
                  at the actual time plus the
                  parameter time advance";
                RNGBlock.trigger := RNGBlock.trigger +
                  1 "augmenting the RNG trigger to
                  require the generation of a new
                  random number";
                S.active := true "starting a process
                  the server turns to active";
                S.events_in_buffer :=
                  S.events_in_buffer -
                  requestDimension "events ready to be
                   processed are reduced";
            end if;
        end if;
    elseif S.events_being_sent <= 0 then
        if S.events_in_batch > 0 then
            if not S.blocked_output then
                S.immediate_transition_required := true
                   "if no events have been sent at the
                   last transition, but still there
                  are and outputs are open, trigger a
                  new transition to send new ones";
                if sendEventsSingularly then
                    S.events_in_batch :=
                       S.events_in_batch - 1;
                    S.events_being_sent := 1;
                else
                    S.events_being_sent :=
                       S.events_in_batch;
                    S.events_in_batch := 0;
                end if "if events have to be sent one
                   by one, send one, otherwise send
                   entire batch";
                if S.events_in_batch == 0 then
                    S.request := S.request +
                       requestDimension "a request is
                       elaborated";
                end if "if no more events would remain
                   to be sent, send a request too";
            end if;
        elseif S.events_in_batch <= 0 then
            if S.events_in_buffer >= requestDimension
               then //if there are enough events to
               start the process
                S.next_scheduled_generation := time +
                   RNGBlock.randomValue "the next
                   process is scheduled to be finished
                   at the actual time plus the
                   parameter time advance";
                RNGBlock.trigger := RNGBlock.trigger +
                   1 "augmenting the RNG trigger to
                   require the generation of a new
                   random number";
                S.active := true "starting a process
                   the server turns to active";
                S.events_in_buffer :=
                   S.events_in_buffer -
                   requestDimension "events ready to be
                    processed are reduced";
            end if "if no events are left to send and
               there are enough events ready to
               process, start one";
        end if;
    end if;
end if "if before this internal event server was
   active, it means that a process has been completed
   and that the server needs to be deactivated.
   Otherwise sending of events needs to be managed";
```

After having followed a procedure similar to the one for the generator transmission, the lifetime of the new state is determined. Notice that the completion of a task has precedence over the closure of outputs, to let boolean `active` be raised for an amount of time representing the effective interval in which the process was carried.

```
if S.immediate_transition_required then
    S.next_TS := Functions.ImmediateInternalTransition(
        S.next_TS) "if an immediate transition is
        required, this function changes next_TS to a
        negative number lesser than the actual one";
elseif S.active then
    S.next_TS := S.next_scheduled_generation;
elseif S.blocked_output then
    S.next_TS := Modelica.Constants.inf "in any case,
        if output ports are blocked next internal
        transition is scheduled at infinite time";
else
    S.next_TS := S.next_scheduled_generation "otherwise
         the next internal event will be when normally
        scheduled";
end if "if to evaluate next_TS in relation to the
    output being blocked";
```

External transition

In the external transition, after having updated `blocked_output` and `events_in_buffer` like at the beginning of the confluent transition, two possibilities are considered: one in which the server is not active and has received enough events to start a process, and another one in which there are events needing to be sent and the output ports aren't blocked.

```
if S.events_in_buffer >= requestDimension and not
   S.active and S.events_in_batch == 0 then //if there
   are enough events to start the process
    S.next_scheduled_generation := time +
        RNGBlock.randomValue "the next process is
        scheduled to be finished at the actual time plus
         the parameter time advance";
    RNGBlock.trigger := RNGBlock.trigger + 1 "
        augmenting the RNG trigger to require the
        generation of a new random number";
    S.active := true "starting a process the server
        turns to active";
    S.events_in_buffer := S.events_in_buffer -
        requestDimension "events ready to be processed
        are reduced";
elseif S.events_in_batch > 0 and not S.blocked_output
   then
    S.immediate_transition_required := true "if after
        an external event there are still events to send
         and output ports happen to be open, trigger an
        immediate internal event to send events";
    if sendEventsSingularly then
        S.events_in_batch := S.events_in_batch - 1;
```

```
            S.events_being_sent := 1;
      else
            S.events_being_sent := S.events_in_batch;
            S.events_in_batch := 0;
      end if "if events have to be sent one by one, send
          one, otherwise send entire batch";
      if S.events_in_batch == 0 then
            S.request := S.request + requestDimension "a
                request is elaborated";
      end if "if no more events would remain to be sent,
          send a request too";
end if;
```

At the end `next_TS` is determined like in the transition above described.

Internal transition

The internal transition is identical to the confluent one except for the fact that no state variable is updated from inputs.

Bibliography

References cited in the text

Cimino, Chiara

2018 *CPS and Digital Twin in the Industry 4.0 era: two application cases*, Politecnico di Milano. (Cit. on p. 63.)

Sanz, Victorino

2010 *Hybrid System Modeling Using the Parallel DEVS Formalism and the Modelica Language*, Dpto. de Informática y Automática, UNED. (Cit. on pp. 24, 25.)

Zeigler, Bernard P., Herbert Praehofer, and Tag Gon Kim

2000 *Theory of Modeling and Simulation*, ed. by Academic Press, ISBN: 978-1936007103. (Cit. on p. 18.)

Additional material consulted

Fritzson, Peter

2010 *Principles of object-oriented modeling and simulation with Modelica 2.1*, John Wiley & Sons.

Kofránek, J, M Mateják, P Privitzer, and M Tribula

2008 "Causal or acausal modeling: labour for humans or labour for machines", *Technical computing prague*, pp. 1-16.

Modelica® - A Unified Object-Oriented Language for Systems Modeling - Language Specification

2017 version 3.4.

Sanz, Victoria, Alfonso Urquia, and Sebastian Dormido

2008 "Introducing messages in Modelica for facilitating discrete-event system modeling", in *Proceedings of the 2nd international workshop on equation-based object-oriented languages and tools*, 029, Linkoping University Electronic Press, pp. 83-93.

Sanz, Victorino, Alfonso Urquia, François E Cellier, and Sebastian Dormido
2010 "System modeling using the Parallel DEVS formalism and the Modelica language", *Simulation Modelling Practice and Theory*, 18, 7, pp. 998-1018.

Sanz, Victorino, Alfonso Urquia, and Sebastian Dormido
2009 "Parallel DEVS and process-oriented modeling in Modelica", in *Proceedings of the 7th International Modelica Conference; Como; Italy; 20-22 September 2009*, 043, Linkoping University Electronic Press, pp. 96-107.

Tiller, Michael
2012 *Introduction to physical modeling with Modelica*, Springer Science & Business Media, vol. 615.