INFORMATION TECHNOLOGY FOR EUROPEAN ADVANCEMENT

"D.7.4.1 – Library for electrical and air conditioning components with failure modes "

"D7 : Model component Library"

MODRIO (11004)

Version Final

Date 20/04/2016

Authors

Philippe CARER	EDF R&D / MIRE
Xavier DE BOSSOREILLE	Ecole Supérieur Ingénieur SUPELEC (Internship Student)
With the contribution of Martin OTTER	(DLR Germany)

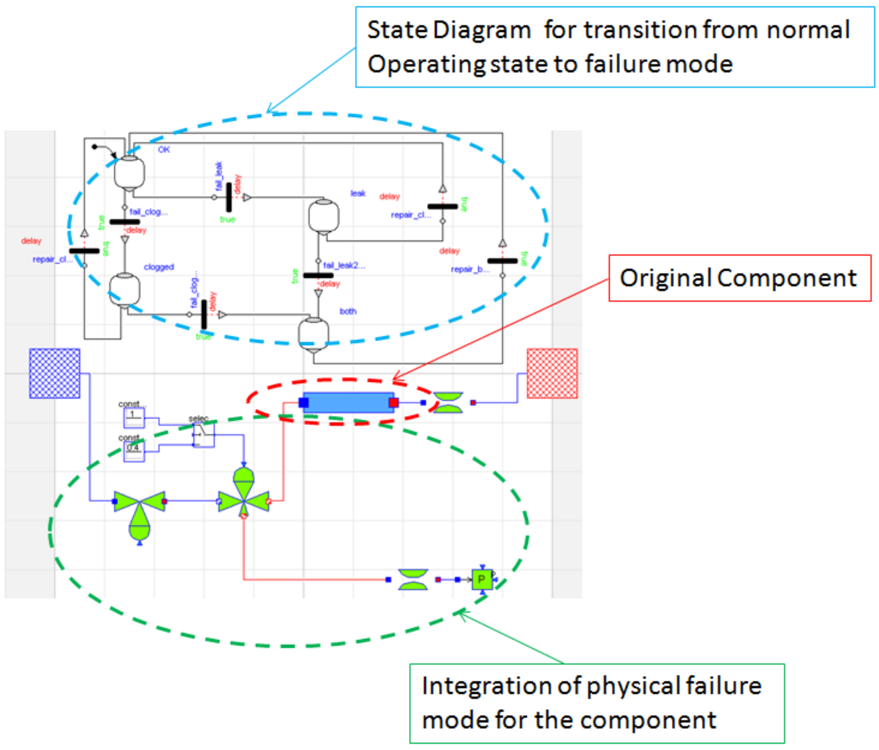
Executive summary

Description of the Library of component for the Data Center Demonstrator with failure mode

This library has been built in the context of the Data center Demonstrator (D8.6.1) for the MODRIO Project.

The Idea is to use some already existing component in the library of electrical or thermohydraulic component and to add two layers to be able to treat the reliability of component

- The first layer consists of elements used to create operating physical failure modes of the component studied
- The second layer is the graph diagram which represents the transition between operating state of the component and derated state or failure mode of the component. The transition between the different state are provoked by stochastic transition (obtained with Monte Carlo simulation)


State Diagram for transition from normal Operating state to failure mode
Original Component
Integration of physical failure mode for the component
OK
leak
clogged
both
fail_leak
fail_clog
fail_leak2
repair_cl
repair_b
delay
true
const
selec

Example of the "pipe" component from the "Thermosypro Library" with failure mode and state diagram transition

Some component has been integrated in the library :

- From THERMOSYS PRO library : components "Pipe" , "Pump", "switch" (see file **MODRIO WP741_Help_Thermosys_pro_Component with failure 20140922.pdf)**
- The component used for the electrical diagram of the data center : Circuit-breaker, protection relay, UPS (see file "**D741_M25_Help_of_Library for electrical components with failure modes for Data center. HTML**")

Summary

1. Presentation of the document

The different component built during the MODRIO project for the demonstrator Data Center are presented in this document. The principle adopted to obtain component with failure mode is to use component in library already existing for the different component such as thermosysPro for hydraulic component or "modelica.electrical.analog" library for electrical component. This library has been built in the context of the Data center Demonstrator (D8.6.1) for the MODRIO Project.

Two files [1], [2] associated with this deliverable present in more detail the description of the components used in this document.

[1] MODRIO WP741_Help_Thermosys_pro_Component with failure 20140922.pdf

[2] "D741_M25_Help_of_Library for electrical components with failure modes for Data center. HTML

The Idea is to use some already existing component in the library of electrical or thermohydraulic component and to add two layers to be able to treat the reliability of component

- The first layer consists of elements used to create operating physical failure modes of the component studied
- The second layer is the graph diagram which represents the transition between operating state of the component and derated state or failure mode of the component. The transition between the different state are provoked by stochastic transition (obtained with Monte Carlo simulation)

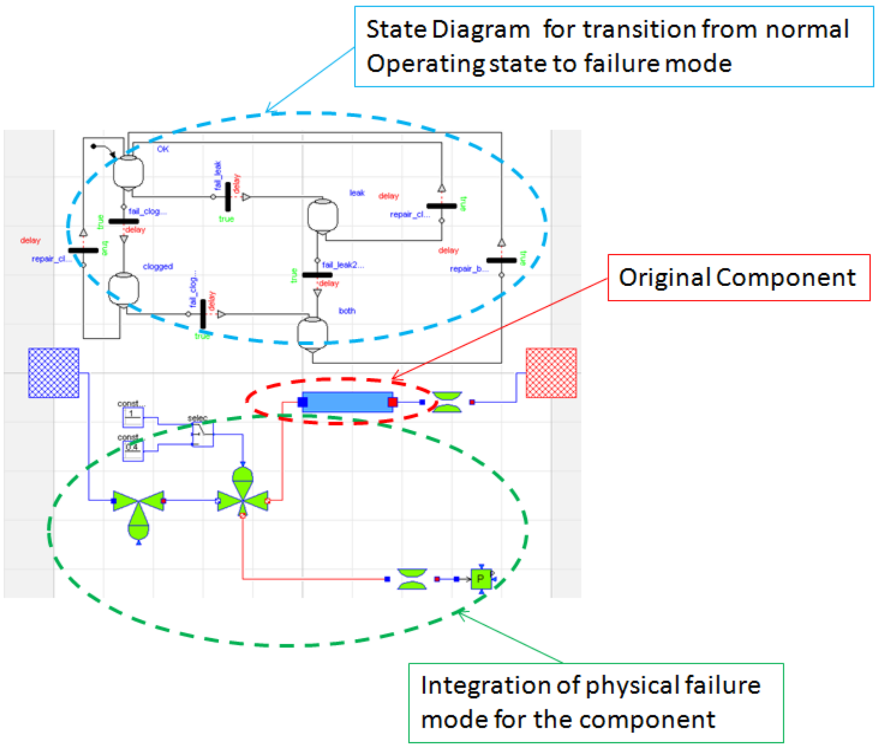

State Diagram for transition from normal Operating state to failure mode
Original Component
Integration of physical failure mode for the component
OK
fail_leak
delay
true
leak
repair_cl...
fail_clog...
clogged
fail_leak2...
repair_b...
both
const
selec

Figure 1 : component "pipe" from the Thermosyspro library with failure mode and state diagram

Two MODELICA libraries has been used to add the transition between nominal mode and failures mode for the component:

- Firstly with the "state graph machine" MODELICA library
- Secondly with the "Continuous time state machine" [3] MODELICA library

The example used at least for the data center used the second library. These second library treats more easily the Monte Carlo simulation then the first "state graph machine" MODELICA library.

The result obtained with the state graph library is presented first in this document and secondly the result obtained with the state machine library.

2. Description failure mode for component with the "state graph " MODELICA library

The development with the Modelica_StateGraph library is presented in this chapter.

2.1. Developing the transitions

After different experiments, we choose firstly to represent the different states of a component with state-graph models. The states of the state-graph will obviously correspond to the different possible states of the component (OK 1, OK 2, ..., failure 1, failure 2, ...). The transitions will represent the failings, repairings, or "normal" transitions of the component. For example, an electric switch will have the "normal" states *open* and *close*, and the failed states *stuck open* and *stuck close*. The transitions will be the failures like *getting stuck open*, the repairs, and the "normal" transitions *opening* and *closing*. The physical functions of the switch will then depend of the presence in a state or in another one.

There already exists a Modelica library, called Modelica_StateGraph2, to represent state-graphs. However, in this library, the transitions can be fired only via Booleans sent to them or via fixed tempos. This doesn't correspond to our need of transitions following probability laws. To avoid this, we decided to create our own transition, inspired by this already-existing transition.

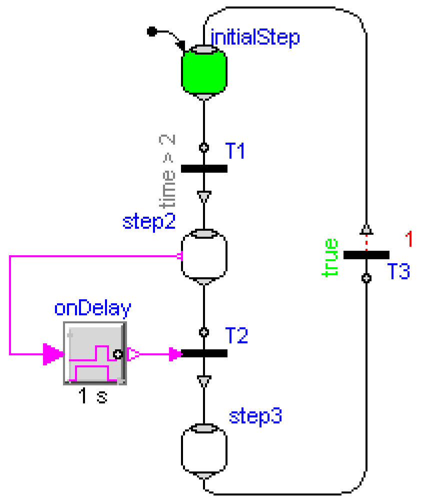


Figure 2 : example of a state-graph with the Modelica_StateGraph2 library

First of all, we implemented a function to obtain pseudo-random numbers uniformly distributed between 0 and 1. This is not something natural in Modelica due to its deterministic nature, but this technique was found several years ago so we won't develop it here. We then need to use these random numbers in our transitions. The basis for this part has been developed by M. Otter, from DLR, one of the project partners, in the article *Efficient Monte Carlo simulation of stochastic hybrid systems [4]*. We adapted his work for our case.

The spirit of this mechanism is that we control the Boolean that fires the transition. Supposing the state before the transition is called state A, the strategy is as follows :

```
When arriving in state A
    Get a random number r
End when
If in state A
    Compute the cumulative distribution function
    Compare the cumulative distribution function with the number r to fire the transition
End if
```

The exact code in Modelica can be found in the Appendix A

By using the cumulative distribution function, we avoid having to calculate in advance the moment of the next transition. Since this time depends on the failure rate, we can have varying failure rates which will directly influence the time of the next failure. This technical trick is explained in detail in *Efficient Monte Carlo simulation of stochastic hybrid systems [4]*.

We then add a criterion to determine which kind of probability law we want to use (uniform, exponential, Weibull...), leading to different cumulative distribution functions.

For the failure rates, we can input mathematical functions depending of other variables of the models. This way, we can have our varying failure rates. This trick is done by declaring the failure rate as a variable (not a *parameter* or a *constant*) and by adding the following annotation : *annotation(Dialog)*. This way, a field to input the equation will be present in the interface.

2.2. Failures on solicitations

In certain cases, we have to consider failures that occur only when we solicit our component. For example, a light bulb can break down on average every P hours. This is what we considered in the previous part. But a light bulb can also break down every N time you switch it on. For this case, we developed another transition with two possible outputs.

On solicitation, we reach a virtual step (here called *step1*) where we compute a random number (still the uniform distribution between 0 and 1). We then compare this random number to the failure rate and activate the corresponding "classic" transition.

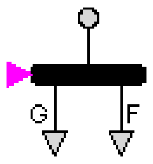


Figure 3 : Icon of a transition on solicitation

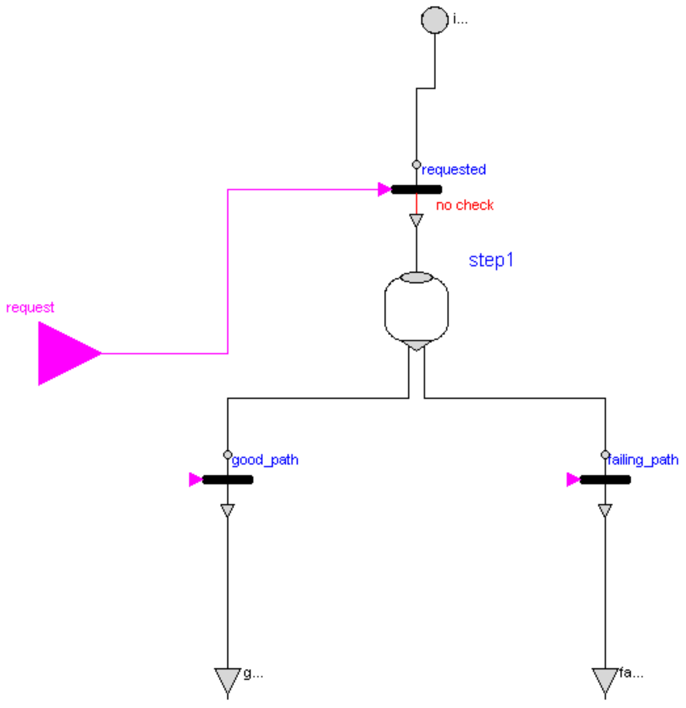


Figure 4 : Modelica model of the transition on solicitation

2.3. Illustration on the simple test-case

For our test-case, we take a circuit with two lines with transformers. We decided to model the failures of the lines as a varying resistance. When the line is working, we have a very small resistance. When the line breaks down, we have a very high resistance (so almost no current is going through it). It means that the current into a line can vary from I/2 to I.

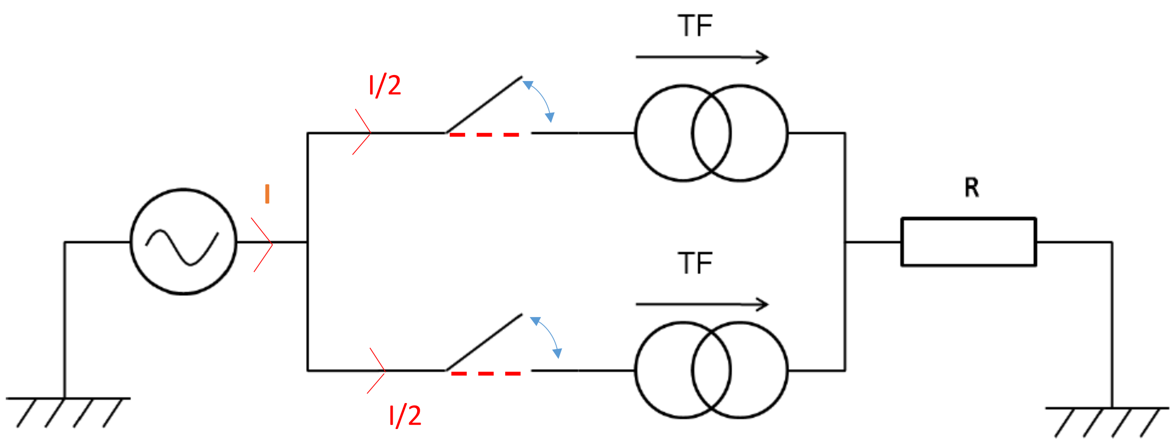


Figure 5 : example of circuit with failure rate of two lines varying in function of the current flow

This corresponds to the following models :

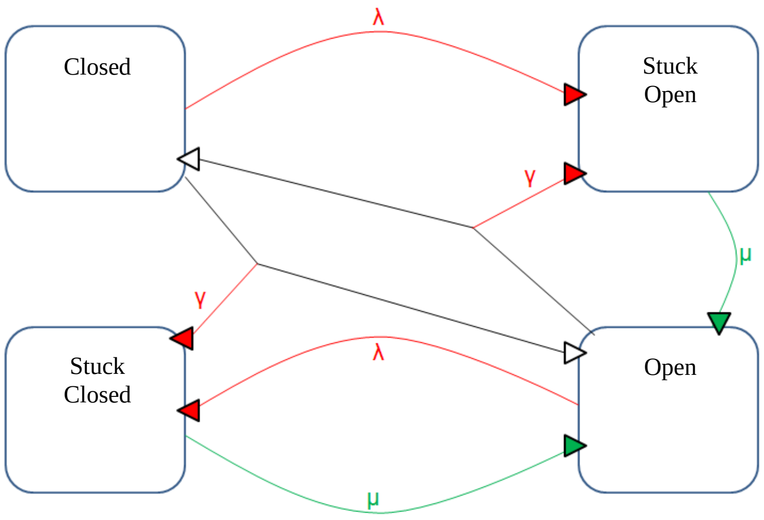


Figure 6 : state-graph for a virtual switch on the line

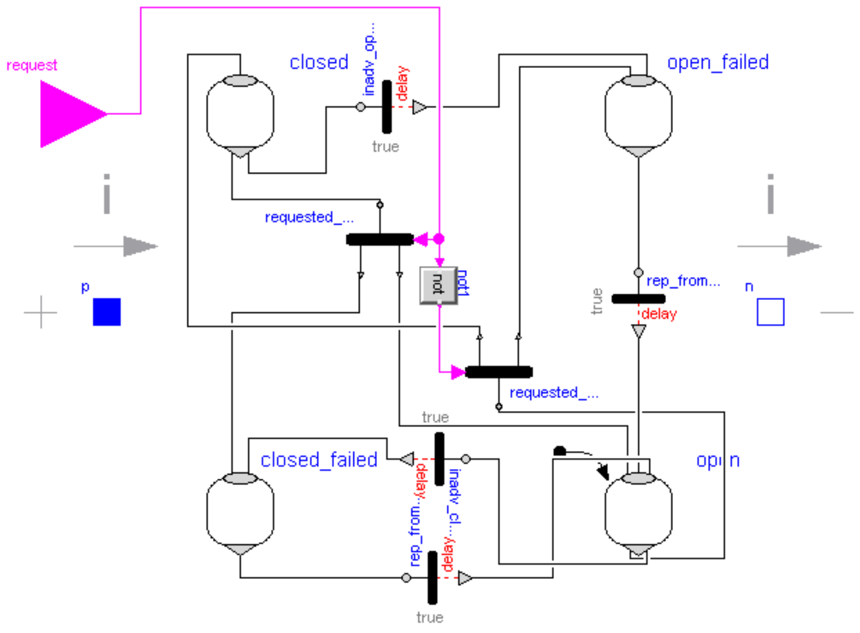


Figure 7 : Modelica model of the virtual switch

For the simulation, we put the same basic failure rates (1 failure/month) and the same repair rates (10 repairs/month) on each circuit-breaker. The failure rates are influenced by the current in the branch in this manner :

- If both branch are working, each failure rate is multiplied by 1
- If a branch is down, the failure rate in the other one is multiplied by a factor between 1 and 2 depending on the current going through the working branch

For the solicitation signals, we used constants. Failures on solicitation are not allowed in this example.

In figure 8, we can compare the state of the system (alimentation of the load) depending on the influence of the current for one simulation. An identical seed for the random generator was used in both experiments.

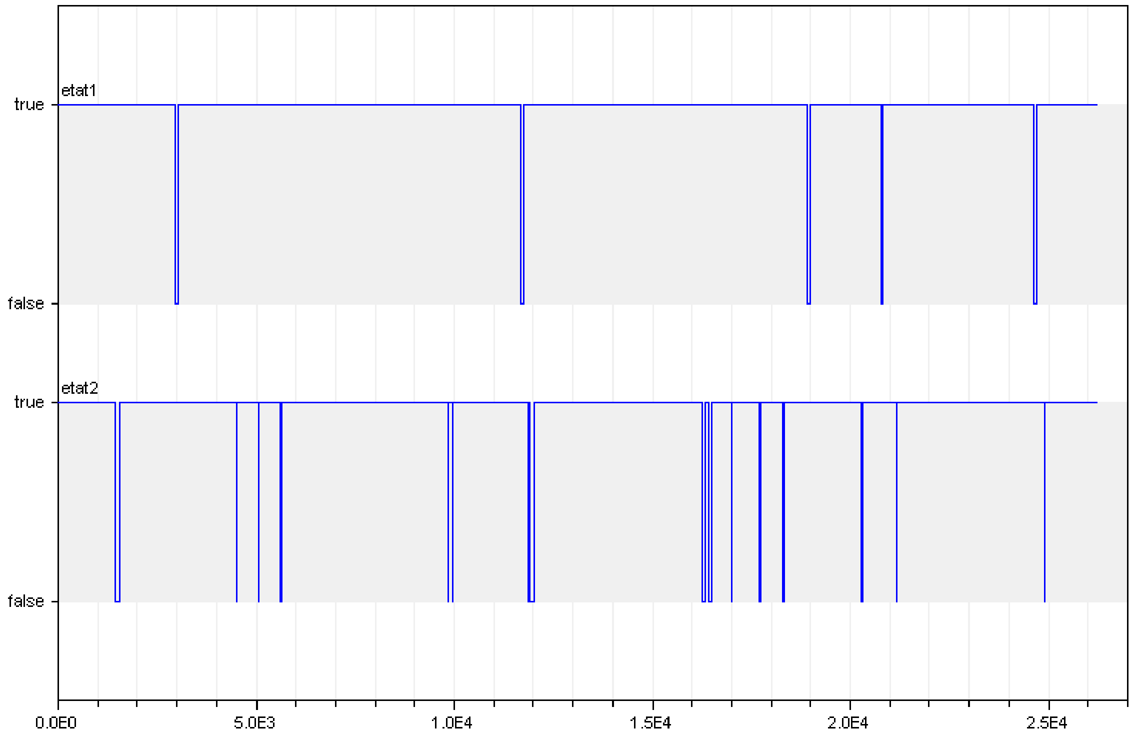


Figure 8 : state of the system without influence(top) and with current influence (bottom)

We can also in figure 9 take a look at the availability of the system (in blue without the current and in red with the current influence).

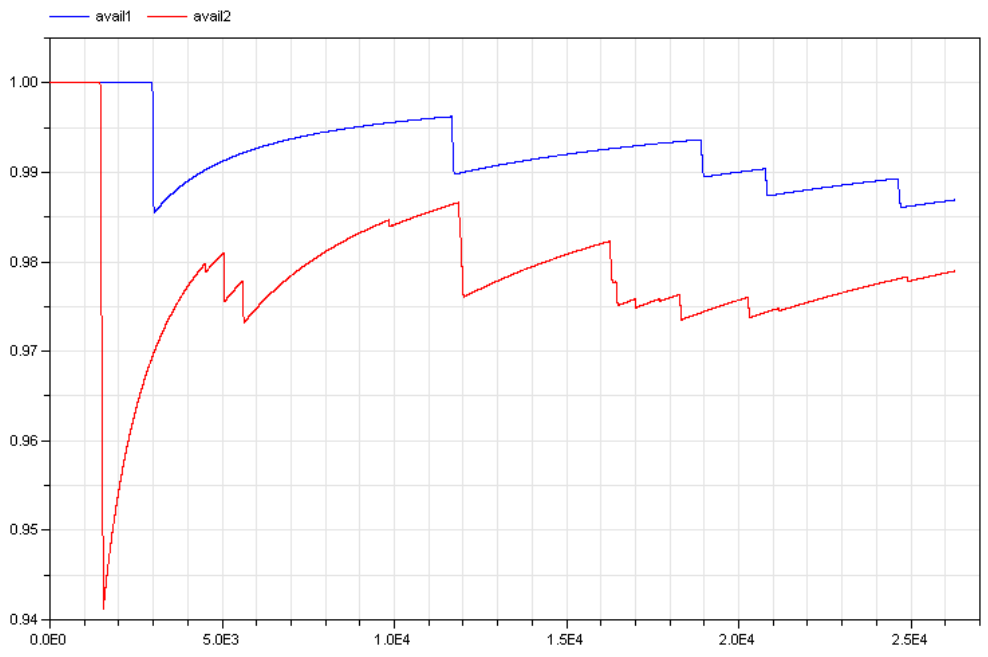


Figure 9 : availability of the system (blue : independence; red : current influence)

And the resulting reliability of the system (failure/s) with the same colors in figure 10.

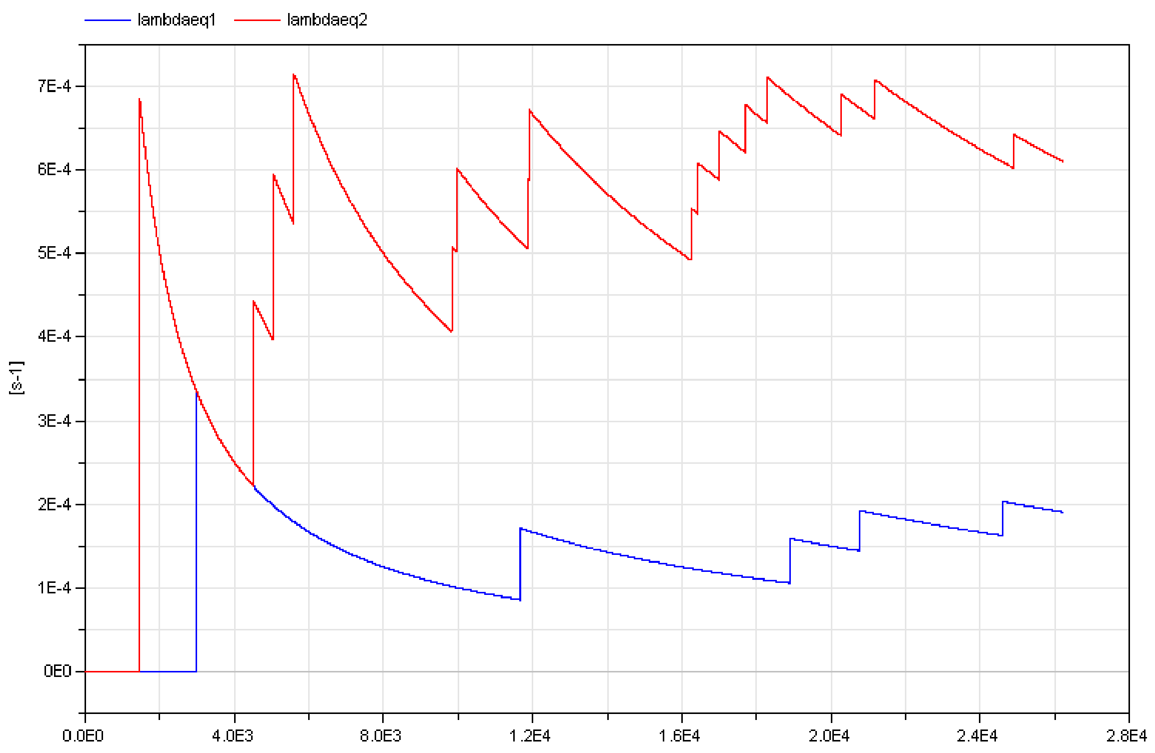


Figure 10 : reliability of the system (blue : independence; red : current influence)

Taking into account the current can change the availability of the system. Even with two independent components, the failures can’t be considered as completely independent since the failure of one of them redrafts the circuit, changing the energy going through the other one.

Of course, to obtain the real availability and reliability of the system, we need to perform a great number of simulations, with different seeds for the random number generator, meaning doing Monte-Carlo simulations. This poses no problem in theory (and was done for this example), but the Monte-Carlo part will be discussed later in this paper.

3. Building the component with failure mode

3.1. Method of the layer

In the previous part, we managed to model different states of a component, with the associated transitions. However, in the shown example, we kind of recreated a switch with a very raw physical functioning. Having to recreate every kind of component for our study would be very long and would require some expertise in these fields (here electric, hydraulic and thermic). Moreover, these components (usually without their failures) already exist in dedicated libraries or in the Modelica Standard Library, developed by experts.

In order to reuse these already-existing libraries, we decided to implement our failures by adding a layer on the already-existing components. This way, we don't need to focus on how the component is implemented, but solely on which failure(s) we want to represent. The method can be summed up by the following graph :

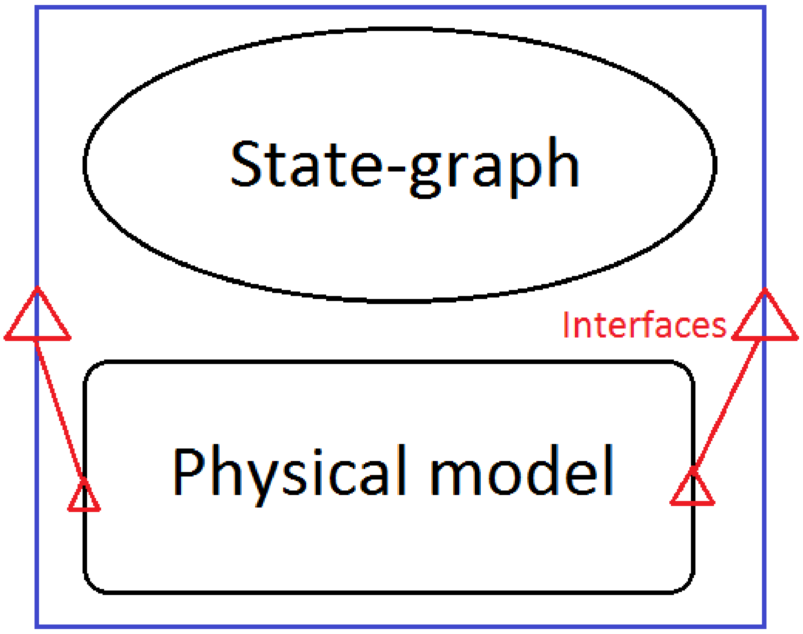


Figure 11 : adding a layer

For the user, the interfaces are the same than if using the "classical" component. At the top of the graph, the state-graph part is simulated. Based on the activated state, different instructions can be sent to the physical part (bottom of the graph). The physical part, which includes the "classical" component, simulates all the physical variables, as implemented by the creators of the "classical" library. These variables feed the equations of the failure rates in the state-graph.

To sum it up, the developer does not need to recreate the physical characteristics of the component, and the user can use this component as he would the classical one, without losing all the expertise contained in it.

3.2. Examples on hydraulic components

The method just described has been tested on some components of the ThermoSysPro library, developed by the department STEP at EDF R&D. The components that were tested are a pipe, a switch valve and two pumps. We will present the pipe and a pump to illustrate the strategy.

3.2.1. Pipe

In the ThermoSysPro library we used the LumpedStraightPipe model of a pipe. Even if one might think a pipe is something very simple, it corresponds to more than fifteen equations, despite being the simplest pipe model in this library. As a consequence, we don't want to plunge into the code and modify anything.

Figure 12 : icon of a "pipe" from the "Thermosys-pro" Library

We chose to represent two kinds of failures in this pipe : a leak, and a partial clogging. This lead to the following state-graph :

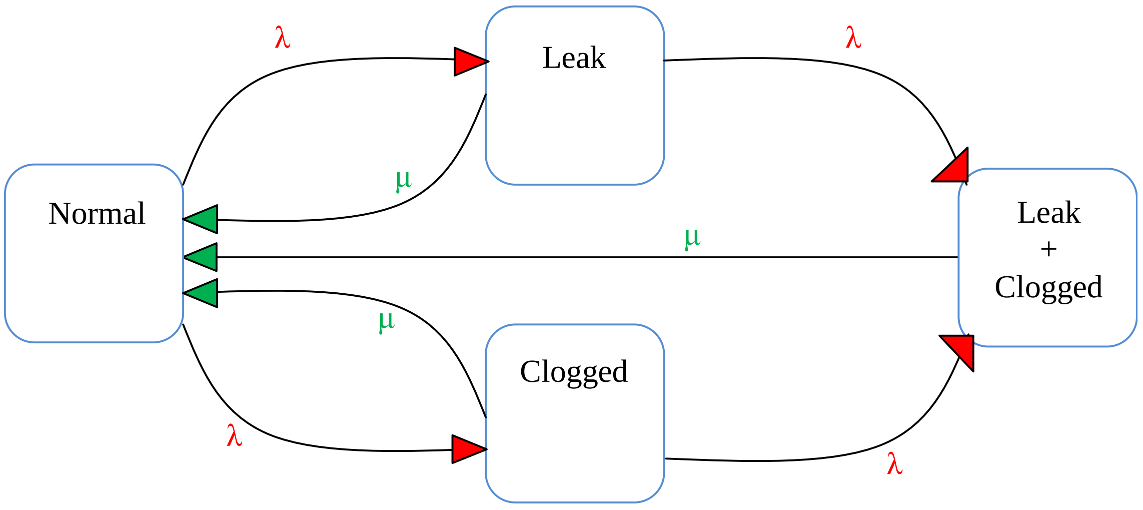


Figure 13 : state-graph for the pipe

A simple reasoning leads to considering that a leak corresponds to rerouting part of the flow towards a virtual sink, thanks to a three-way valve, and a partial-clogging corresponds to partially closing a virtual valve. This leads to the following physical model :

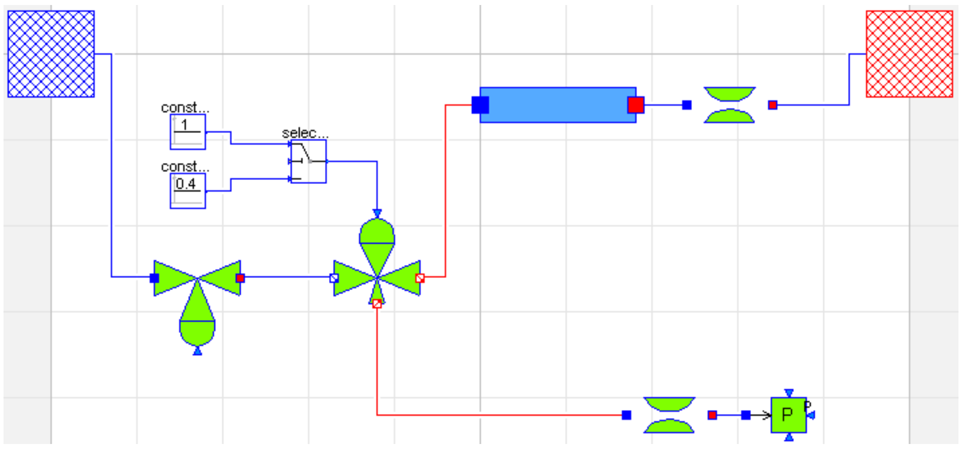


Figure 14 : physical model for the failures in the pipe

Even if it seems to be a more complicated model, in fact the needed valves and sink are already usual components of the ThermoSysPro library. As a consequence, we never had to go check the physical equations, or even to wonder what the variables are.

The combination of the state-graph and of the physical part leads to the final model :

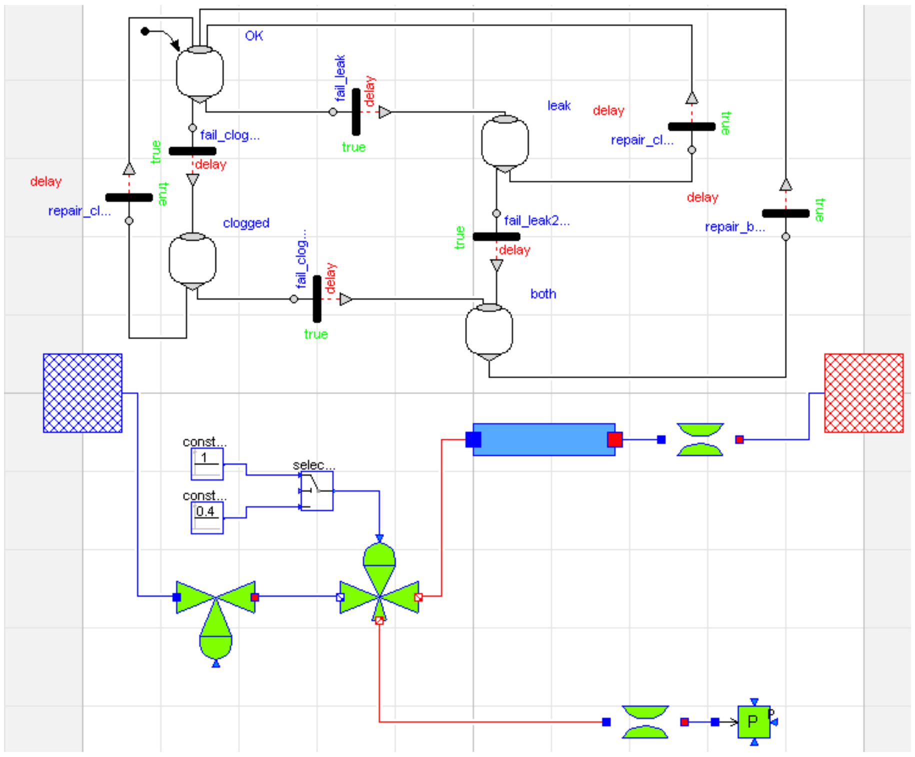


Figure 15 : global model for the faulty pipe

The only two things that had to be done in code were :

- copy/paste the parameters of the "classical" pipe so that the user can access them
- give the relations between the states of the state-graph and the commands of the valves

3.2.2. Pump

We quickly present the case of the *StaticCentrifugalPump* model of a pump. Here we decided to consider the possibilities of a partial or complete break-down of the pump, a leak before the pump or a leak after the pump (due to bad seals for example). The interesting thing is that the leaks are not directly linked to the functioning of the pump. The failures are considered to be independent. The only influence they have on each other is via for example the flow or the pressure in the component. As a consequence, we can implement separate state-graphs in the model. This leads to the following Modelica model :

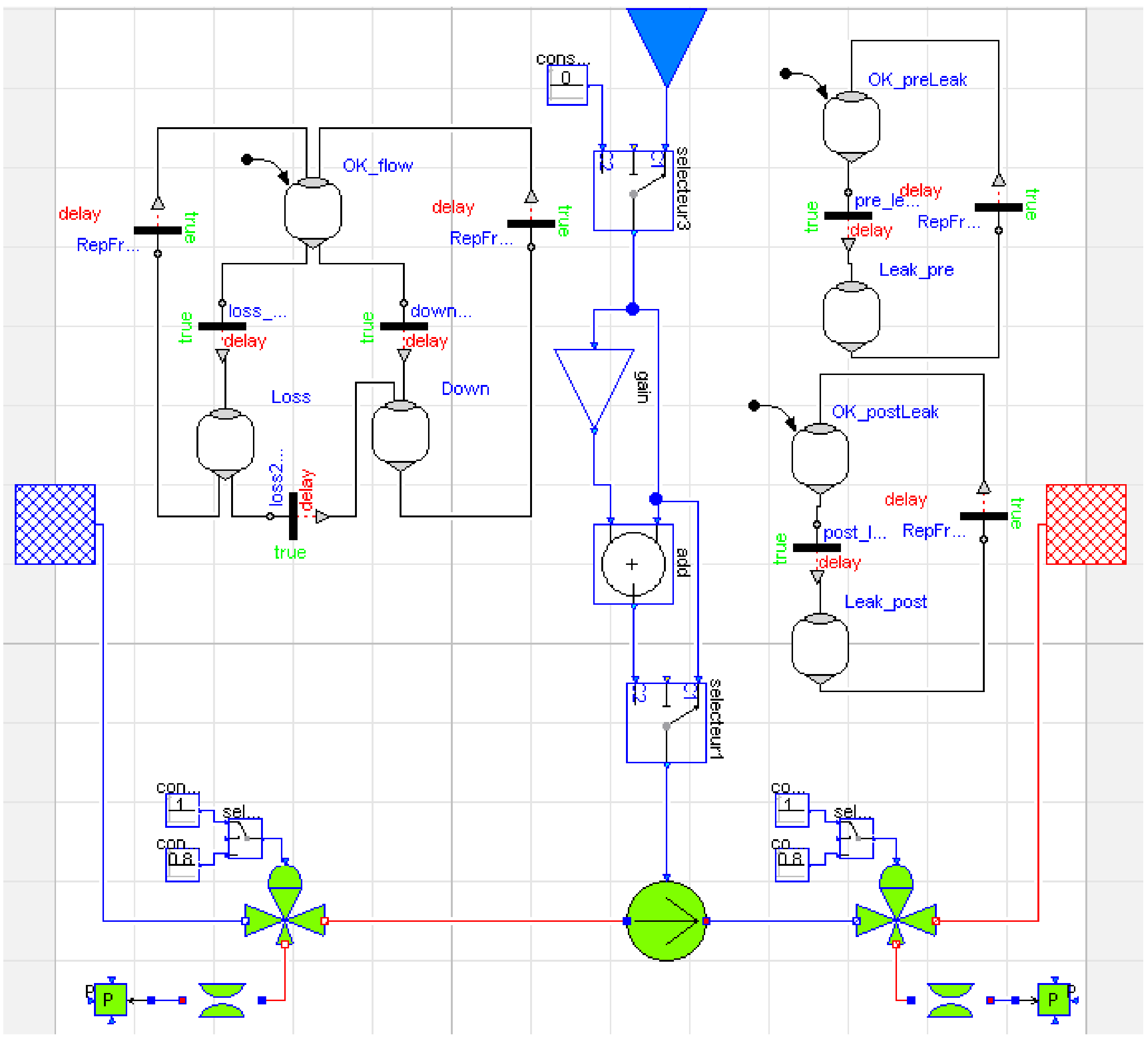


Figure 16 : global model for a pump

Modelica has no problems having different state-graphs in the same model. Thus, we can easily in any component add or remove a failure, since we then give a failure rate that can depend on the physical parameters. This saves us from having one big state-graph when we have several possible and non-exclusive failures.

3.3. Example on a circuit-breaker

For the electric part of our study, we chose to use the QuasiStationnary part of the Standard Modelica Library. The aim was to reduce the simulating times by taking advantage of the fact that the failures will only rarely appear in the data-centers, so most of the time "nothing" will be happening.

A first approach for a circuit-breaker leads to the same state-graph than for the simple example in part 2.3.

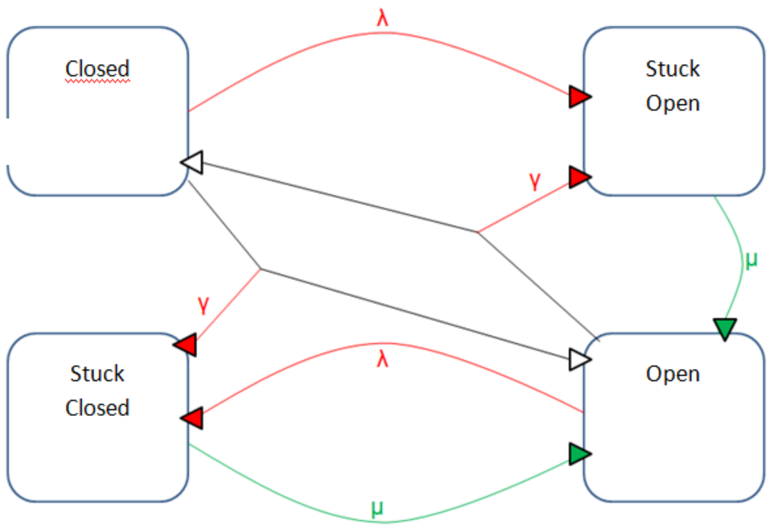


Figure 17 : state-graph for a circuit-breaker

However, we can be more precise and divide it between three distinct parts :

- the circuit-breaker itself, which can refuse to respond (failures on solicitations)
- the captors, which can give a wrong measure
- the protection, which can have digital errors or glitches for example

This leads to the following trio :

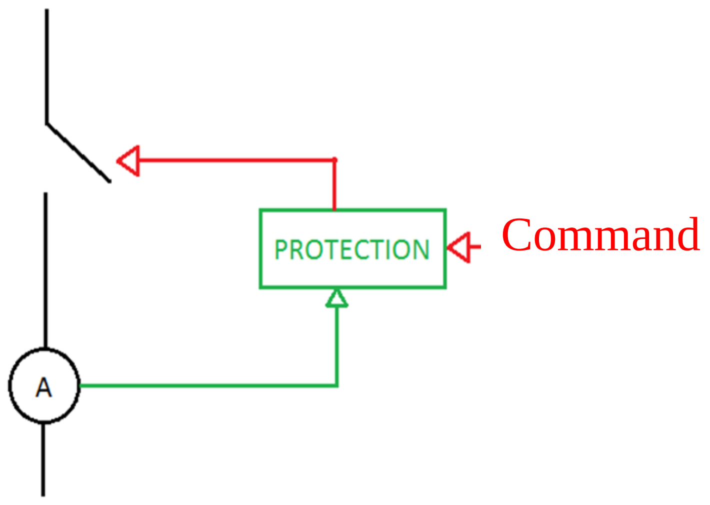


Figure 18 : trio circuit-breaker, sensor, protection

Each element will then have its own state-graph. This way, we can easily compare the influence of different sort of captors for example when planning a new data-center or improving one.

The examples for the circuit-breaker and the protection are shown on the next page. Note that for clarity, we represented here only one state-graph for the circuit-breaker, but in the real model we have one state-graph per phase.

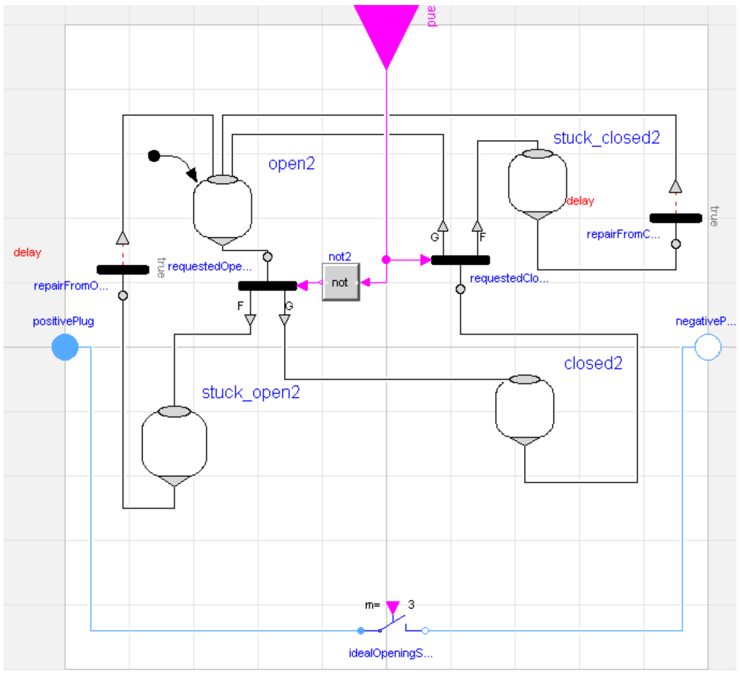


Figure 19 : model for the circuit-breaker part (state-graph for only one phase)

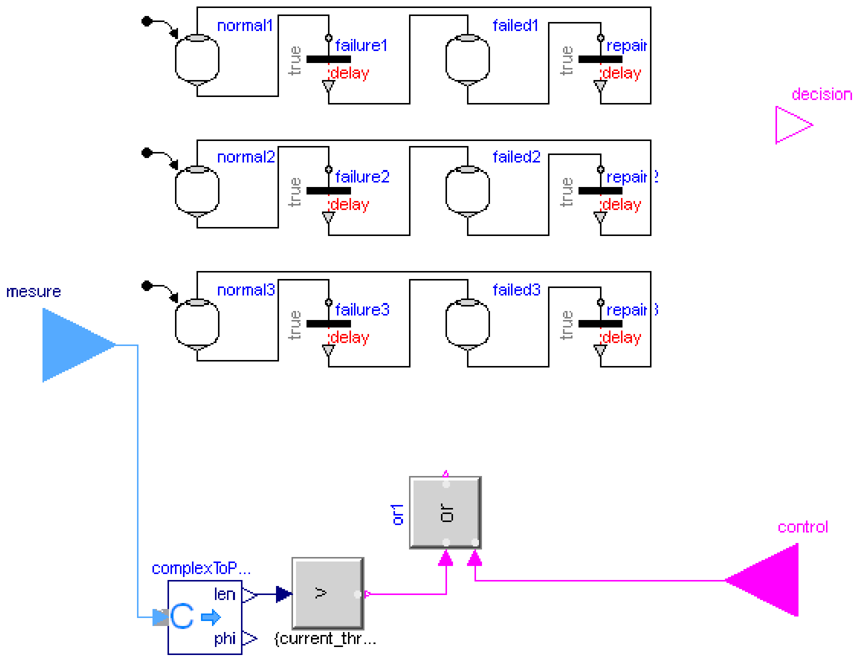


Figure 20 : model for the protection for three phases system (one state-graph per phase)

3.4. Conclusion on the components

To conclude, here is the strategy we decided to adopt in order to have something as universal as possible :

- choose a component considered to be "interesting"
- select the different possible modes (failures included) for this component
- make a state-graph
- make the physical model (upper layer)
- determine the different varying failure rates

This way, we can adapt already-existing libraries to include the failure modes, no matter the studied domains.

4. Monte Carlo simulation into MODELICA

The principle of the MONTE CARLO simulation into MODELICA [4] is indicated in appendix.

5. Different component built with the "state machine" MODELICA library for the Data Center demonstrator

The different components build secondly with the "state machine [3]" library are presented in this chapter.

This second choice of the "state machine" library (after the "state graph" library) is due to the need to perform a lot simulation, with the Monte Carlo method, that is more easier to do with the "state machine" library (the Monte Carlo method is described in the Appendix A and B).

5.1. Indication of the assumption to model the data center with MODELICA

The figure below includes the different component and assumptions taking into account in the model of the data center with MODELICA :

- The control command to switch from the utility power supply to the back-up generator
- The back-up generator
- The circuit breaker where are includes different the failure mode; One has consider the assumption that the failure occurred mainly on this component which integrates a sensor, a protection relay and the mechanical part of the circuit breaker
- The UPS (Uninterruptible power supply)
- And the Load

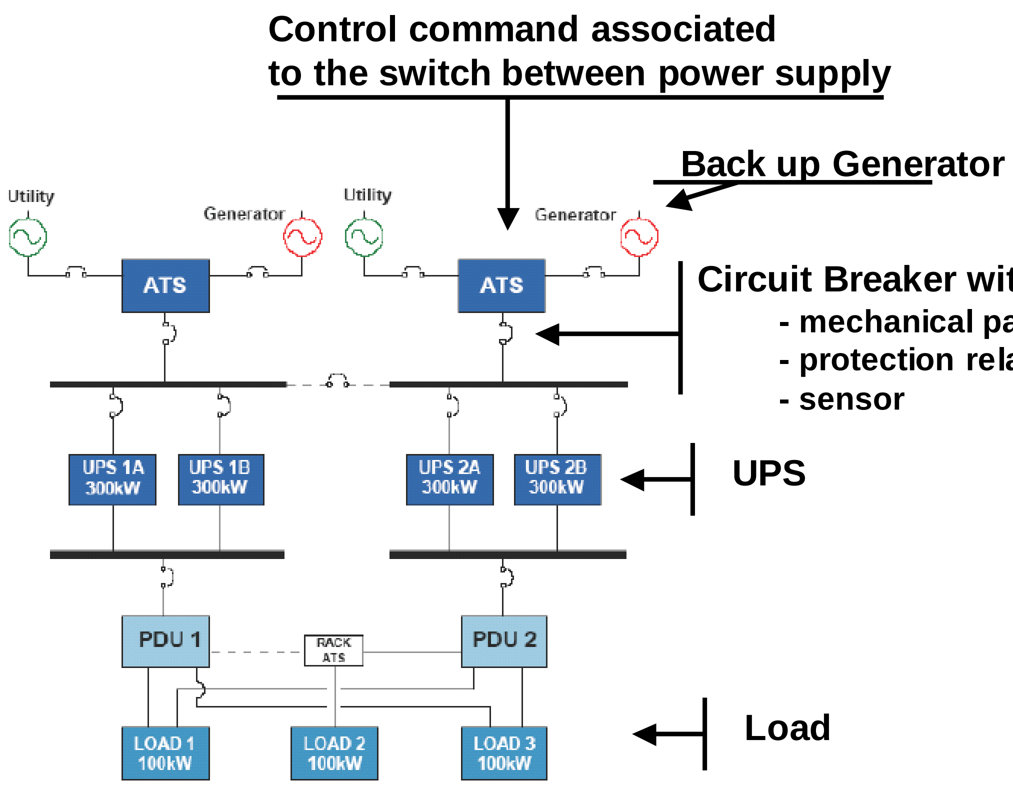


Figure 21 : Diagram of Electrical network of the Data Center

5.2. Presentation of the MODELICA model for the component of the data center

The component described in detail in the files [2] are presented. The failure mode are mainly integrated into the component circuit- breaker.

5.2.1. Reliability model for the Circuit breaker

The circuit breaker is composed of three element : mechanical part current sensor and protection relay

Package Content

Name	Description
MechanicalCircuitBreaker	"mechanical part of the circuit-breaker"
CurrentSensor	current sensor for the circuit-breaker
Protection	protection of the circuit-breaker

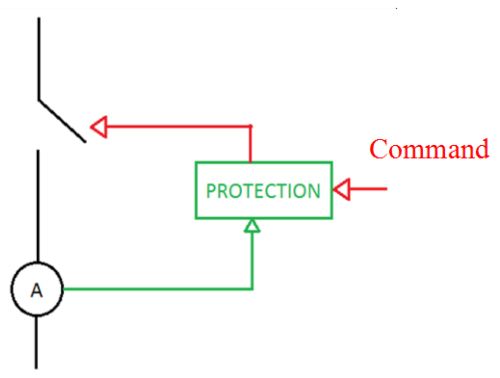


Fig 22 : General Diagram for the Circuit-Breaker with mechanical part, protection relay, sensor

➢ **"mechanical part of the circuit-breaker"**

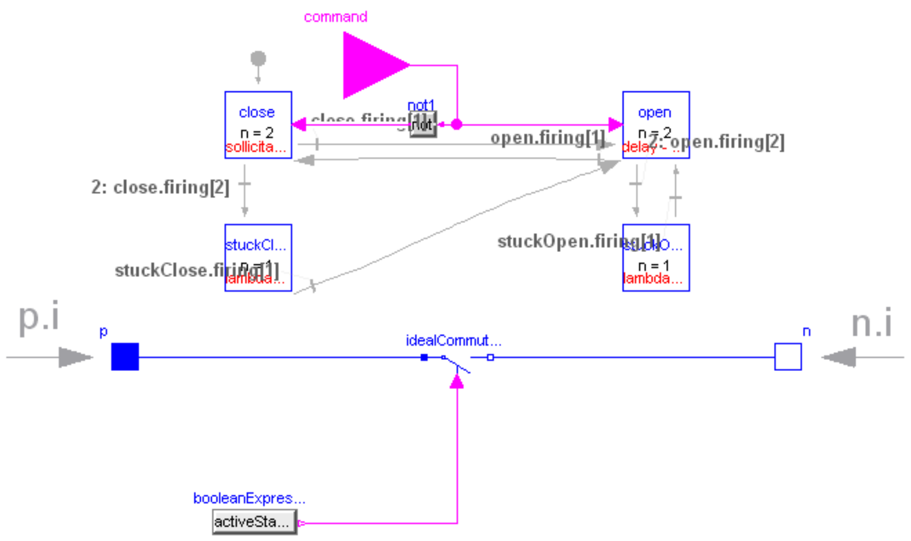


Fig 23 : State machine diagram for the mechanical part of the Circuit - Breaker

The table below give the parameter of the reliability parameter for the mechanical part of the circuit breaker

Parameters

Type	Name	Default	Description
Real	gamma_stuckClosed	1/10	probability of not opening on sollicitation
Real	mu_fromStuckClosed	5	repair rate from stuck closed
Real	gamma_stuckOpen	1/10	probability of not closing on sollicitation
Real	mu_fromStuckOpen	5	repair rate from stuck open

- **current sensor for the circuit-breaker**

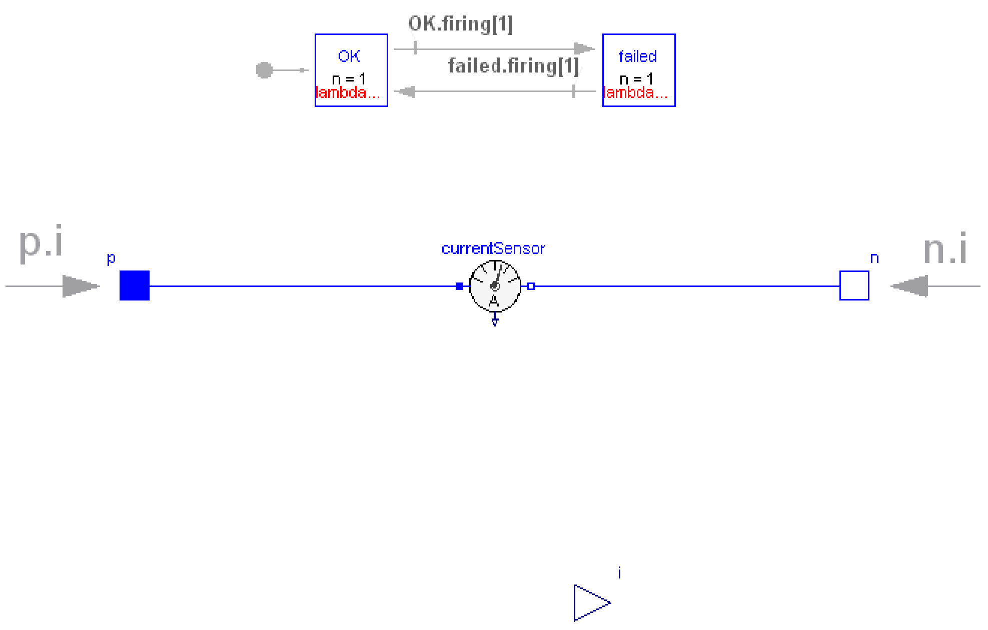


Fig 24 : Diagram for the sensor associated with the circuit breaker

Information

Extends from Modelica.Electrical.Analog.Interfaces.TwoPin (Component with two electrical pins).

Parameters

Type	Name	Default	Description
Real	I_max		maximum current measured by the sensor
Real	failureRate	1	failure rate for the sensor
Real	repairRate	10	repair rate for the sensor

➢ protection relay of the circuit-breaker

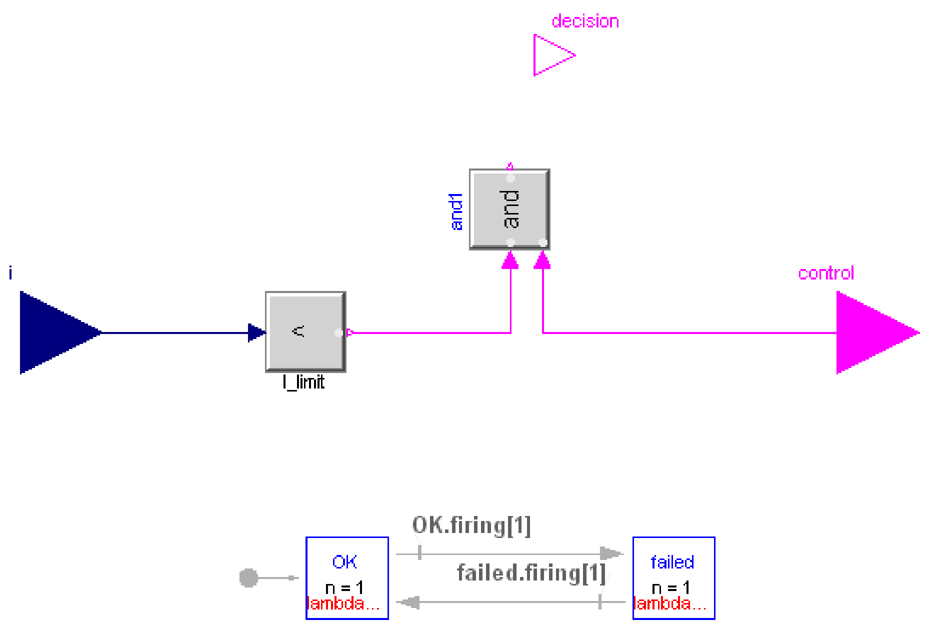


Fig 25 : Diagram for the protection relay associated with the circuit breaker

Parameters

Type	Name	Default	Description
Real	I_limit		maximum current before opening the circuit-breaker
Real	failureRate	1	failure rate from the protection
Real	repairRate	10	repair rate for the protection

5.2.2. Global component for the circuit breaker

FinalModelsDLR.TestCases.DataCenterCase.Components.GlobalCircuitBreaker

circuit-breaker

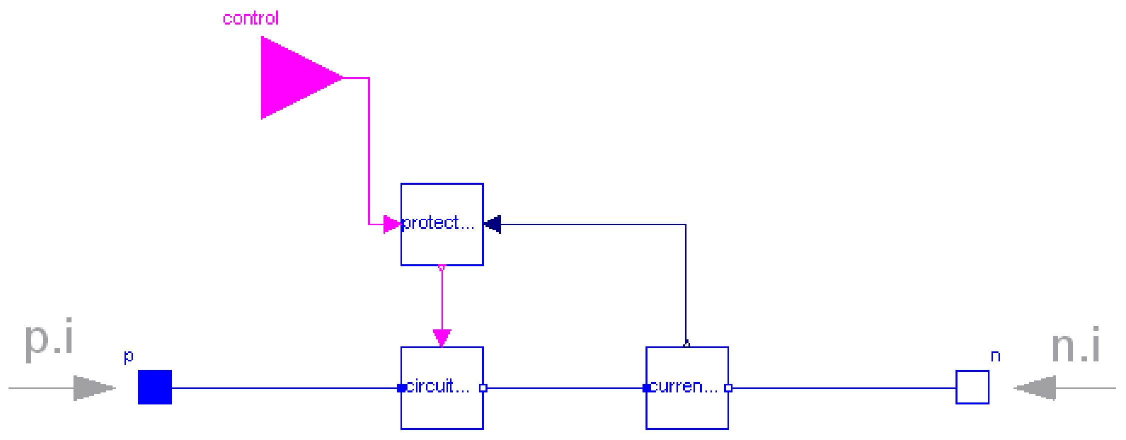


Fig 26 : Modelica Diagram for the global circuit-breaker

Information

Extends from Modelica.Electrical.Analog.Interfaces.TwoPin (Component with two electrical pins).

Parameters

Type	Name	Default	Description
Real	I_limit		maximum current before opening the circuit-breaker
Real	I_max		maximum current mesured by the sensor
Real	protection_failureRate	1	failure rate for the protection
Real	protection_repairRate	10	repair rate for the protection
Real	break_gammaStuckClosed	0.1	probability of not opening on sollicitation
Real	break_muStuckClosed	10	repair rate from stuck closed
Real	break_gammaStuckOpen	0.1	probability of not closing on sollicitation
Real	break_muStuckOpen	10	repair rate from stuck open
Real	sensor_failureRate	1	failure rate for the sensor
Real	sensor_repairRate	10	repair rate for the sensor

5.2.3. Back-up generator

Integrator is integrate to the the back-up generator in order to represent that it is need some time to operate this component.

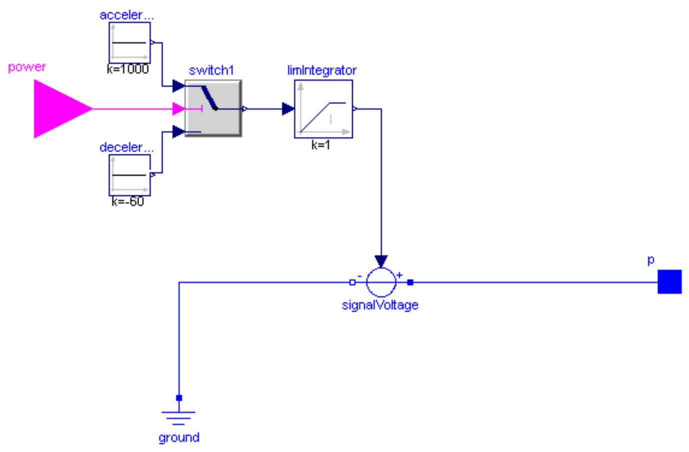


Fig 27 : Modelica Diagram for the Back-up generator

Parameters

Type	Name	Default	Description
Real	amplitude		tension of the generator when at cruise speed

5.2.4. Uninterruptible power supply

Some “hysteresis box” has been added to obtain the stabilization of the model

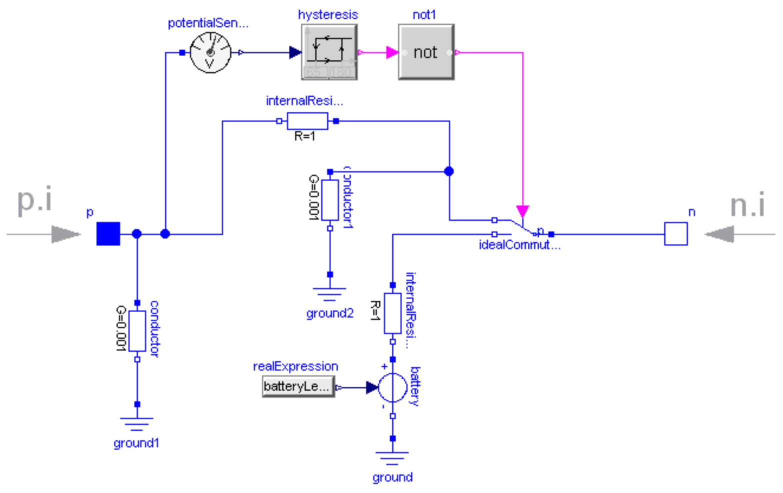


Fig 28 : Modelica Diagram for the UPS

Information

Extends from Modelica.Electrical.Analog.Interfaces.TwoPin (Component with two electrical pins).

Parameters

Type	Name	Default	Description
Real	maxBattery	110	maximum possible charge of the battery, limited by the technology
Boolean	initial_loaded	true	true if already loaded at the beginning
Real	trigger_potential	1	potential threshold for switching between normal supply and battery supply

5.2.5. The data load

This component represents the two redundant parts of the data center to supply the computer which are represent par a resistor

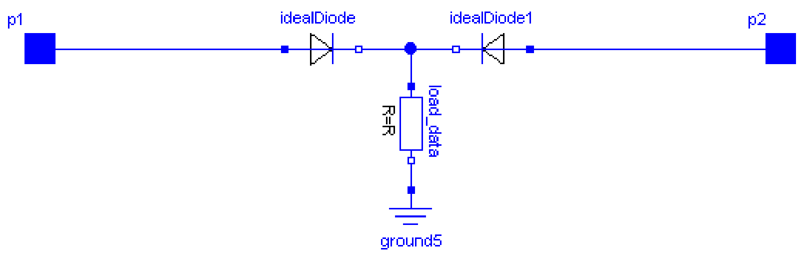


Fig 29 : Modelica Diagram for the UPS

Parameters

Type	Name	Default	Description
Real	R		load

5.2.6. The Control generator

The control generator represent the switch between the utility and the back-up generator

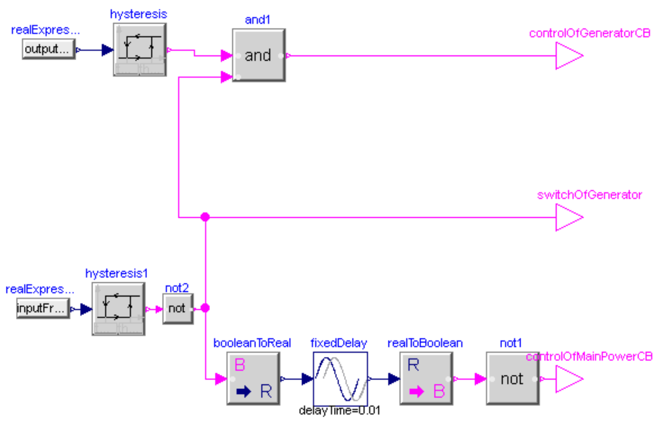


Fig 30 : Modelica Diagram for the Control Generator (back-up)

Parameters

Type	Name	Default	Description
Real	outputOfGenerator	1	value of the potential from the back-up generator
Real	thresholdOutputOfGenerator		minimum potential to switch on the back-up generator
Real	inputFromMainAlimentation	1	value of the potential from the main alimentation
Real	thresholdInputFromMainAlimentation		minimum required potential from the main alimentation

6. References

[1] MODRIO WP741_Help_Thermosys_pro_Component with failure 20140922.pdf

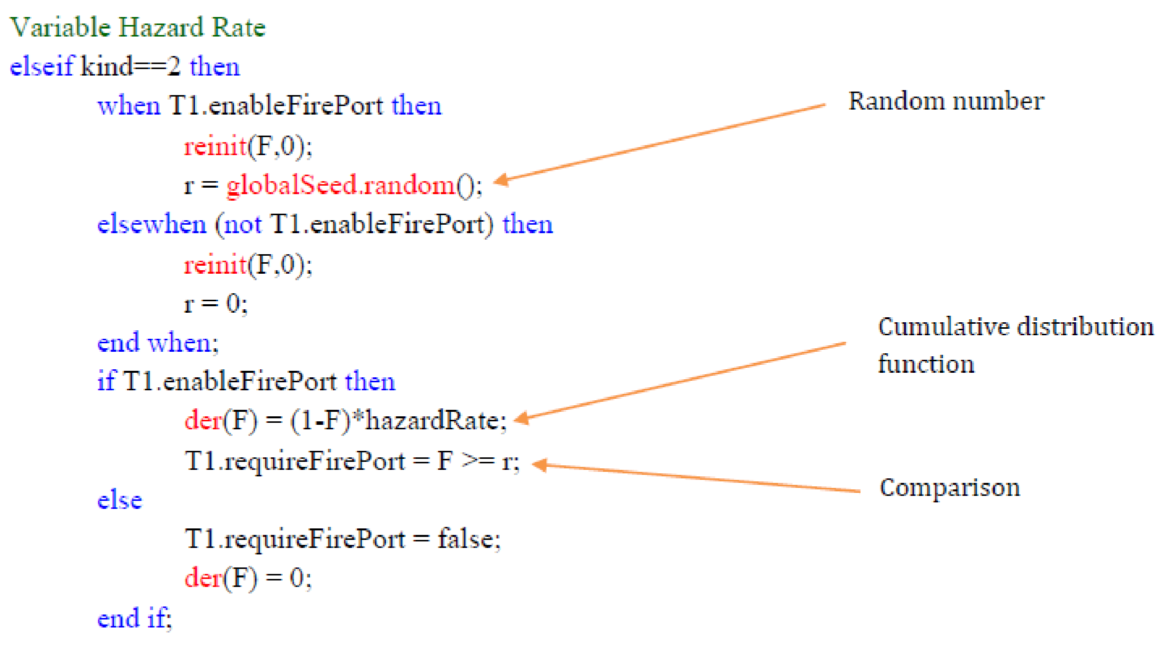
[2] "D741_M25_Help_of_Library for electrical components with failure modes for Data center. HTML

[3] Elqmvist H. Gaucher F., Mattsson S.E., Dupont F. (2012) : **State machines in Dymola**. Modelica 2012 Conference Munich Germany, Sept 3-5, 2012 Download: http://www.ep.liu.se/ecp/076/003/ecp12076003.pdf

[4] Bouissou M., Elqmvist H. Otter M. and Benveniste A. (2014) Efficient Monte Carlo simulation of stochastic hybrid system. MODELICA'2014 Conference Lund, Sweden March 10-12

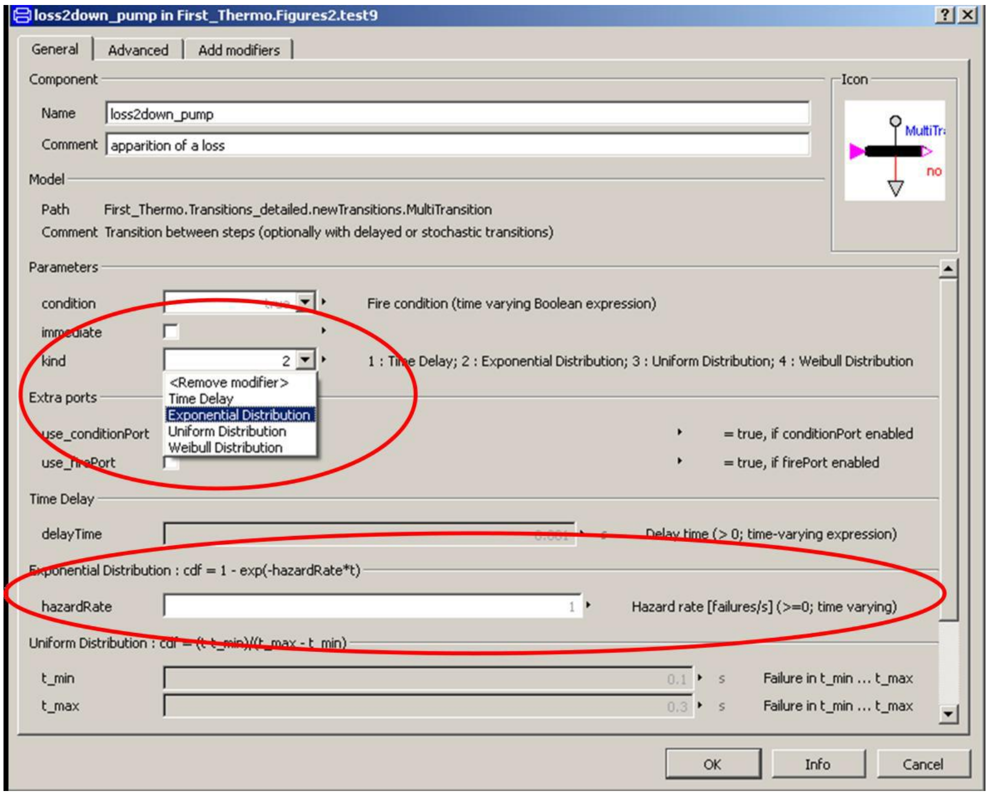
APPENDIX A : Monte Carlo Simulation with MODELICA

This is the part for an exponential law.



```
Variable Hazard Rate
elseif kind==2 then
        when T1.enableFirePort then
                reinit(F,0);
                r = globalSeed.random();
        elsewhen (not T1.enableFirePort) then
                reinit(F,0);
                r = 0;
        end when;
        if T1.enableFirePort then
                der(F) = (1-F)*hazardRate;
                T1.requireFirePort = F >= r;
        else
                T1.requireFirePort = false;
                der(F) = 0;
        end if;
```

Interface for the transition. The developer can choose which kind of probability law he wants to use, and then input mathematical expressions for the parameters of the law.



loss2down_pump in First_Thermo.Figures2.test9

General | Advanced | Add modifiers

Component
Name: loss2down_pump
Comment: apparition of a loss

Model
Path: First_Thermo.Transitions_detailed.newTransitions.MultiTransition
Comment: Transition between steps (optionally with delayed or stochastic transitions)

Parameters
condition: true — Fire condition (time varying Boolean expression)
immediate: ☐
kind: 2 — 1 : Time Delay; 2 : Exponential Distribution; 3 : Uniform Distribution; 4 : Weibull Distribution

Extra ports
use_conditionPort — = true, if conditionPort enabled
use_firePort: ☐ — = true, if firePort enabled

Time Delay
delayTime: 0.001 s — Delay time (> 0; time-varying expression)

Exponential Distribution : cdf = 1 - exp(-hazardRate*t)
hazardRate: 1 — Hazard rate [failures/s] (>=0; time varying)

Uniform Distribution : cdf = (t-t_min)/(t_max - t_min)
t_min: 0.1 s — Failure in t_min … t_max
t_max: 0.3 s — Failure in t_min … t_max

APPENDIX B : Building model with the continuous –time state machine in Dymola

"Continous-time state –machine" library in Dymola

In Modelica 3.3 exists the possibility for synchronous state-machines. Here is an extract from their specification: Any Modelica block instance without continuous-time equations or algorithms can potentially be a state of a state machine. A cluster of instances which are coupled by **transition** statements makes a state machine. All parts of a state machine must have the same clock. All transitions leaving one state must have different priorities. One and only one instance in each state machine must be marked as initial by appearing in an **initialState** statement.

In our case we don't want to rely on a clock, but instead have continuous-time state machines, containing continuous-time equations. This is a generalization of the Modelica 3.3 to continuous-time. This is enabled in Dymola by activating some Hidden flags. (This was studied by H Elmqvist [3]). The principle of the MONTE CARLO simulation has been presented in the paper [4] (Bouissou et al.)

Given a random time T whose *cumulative distribution function (cdf)* F is defined as

$$F(t) = \Pr(T < t) \qquad (5)$$

the corresponding *hazard rate*, $\lambda(t)$, is defined as:

$$\lambda(t) = \lim_{\Delta t \to 0} \frac{\Pr(T < t + \Delta t \mid T > t)}{\Delta t} \qquad (6)$$

The hazard rate can then be expressed as

$$\lambda(t) = \frac{F'(t)}{1 - F(t)} \qquad (7)$$

that is

$$\frac{dF(t)}{dt} = (1 - F(t))\lambda(t) \qquad (8)$$

For Monte Carlo simulation, the time to the next event, T, is determined by drawing a uniform random number, r in $[0,1]$, and solving:

$$F(T) = r$$

When λ is constant, the solution to the differential equation (7) is:

$$F(t) = 1 - e^{-\lambda t}$$

and

$$T = -\frac{\ln(1-r)}{\lambda} \qquad (9)$$

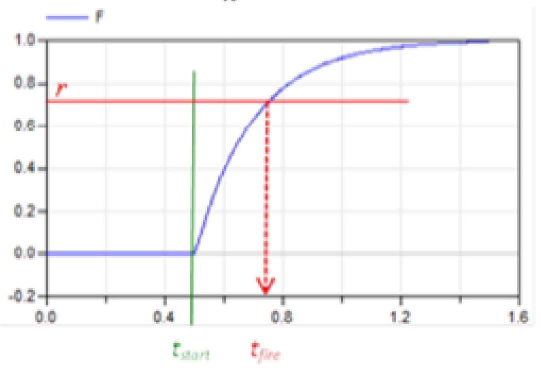


Figure 1: The "inverse cdf" technique for drawing a random number according to a given distribution

First Version

```
    outer GlobalSeed globalSeed;
    Real r, t_next;
    parameter Real hazardRate;

  equation
    when enableFire then
      r = globalSeed.random();
      t_next = time - log(1-r)/hazardRate;
    end when;


    if enableFire then
       fire = time >= t_next;
    else
       fire = false;
    end if;
```

Second version

```
    outer GlobalSeed globalSeed;
    Real r;
    input Real hazardRate(min=0);
  equation
    when enableFire then
      r = globalSeed.random();
      reinit(F,0);    // start at F=0
    end when;

    der(F)  =  (1-F)*hazardRate;
    if enableFire then
       fire = F >= r;
    else
       fire = false;
    end if
```

Introducing the random transitions

As explained in 3.1, we now have access to continuous-time state-machines. However, the Modelica structure of these state-machines is conceived for deterministic comportments. In the transition object, we can only indicate a Boolean condition. This means we can't directly indicate stochastic transitions, and we can't have several transitions linked to one another, as it would be the case for a random choice for example. As a consequence, we decided to create some generic kind of states that will enable us to have our stochastic comportment.

The main idea is to have the states contain all the stochastic parts. The transitions contain only a check of a Boolean in the from-state. We will here illustrate with the case of a transition which random time before activation follows an exponential law of parameter lambda (figure A1 : two states, one transition). We suppose already having a random number generator globalSeed that gives uniform random numbers between 0 and 1.

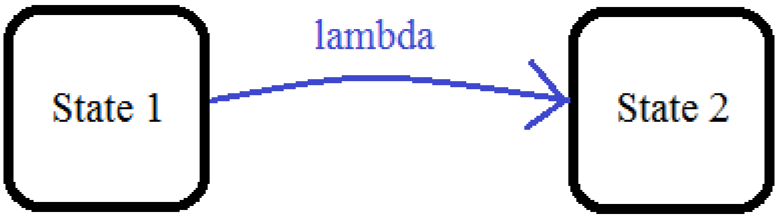


Figure A1

We create a model with the corresponding parameters and variables:

```
model StateExpo "state with exponential stochastic exit(s)"
        Real r "random number";
        Boolean firing "firing of the failure transition";
        Real F "cumulative distribution function";
        parameter Real hazardRate "hazard rate";
```

We then add the equations corresponding to the exponential distribution and the time comparison.

```
initial equation
  F =0;
equation
        der(F) = (1-F)*hazardRate;
        edge.u = F>r;
        firing = edge.y;
```

The use of an edge block is to prevent some Boolean loops that could otherwise appear.

And lastly, we draw the random number r each time the state is entered.

```
if enteringState() then
  r = globalSeed.random();
       else
         r=pre(r);
```

```
                              end if;
```

With this structure for the states, if we want to have a stochastic transition leaving a state State1, we just need to put in the transition the Boolean condition State1.firing to model it.

Now that we obtained a way to model a stochastic transition, we can refine our state model to have time-varying hazard rates. In Dymola, this can be easily done by declaring the hazard rate as a variable (instead of a parameter) and adding a simple annotation :

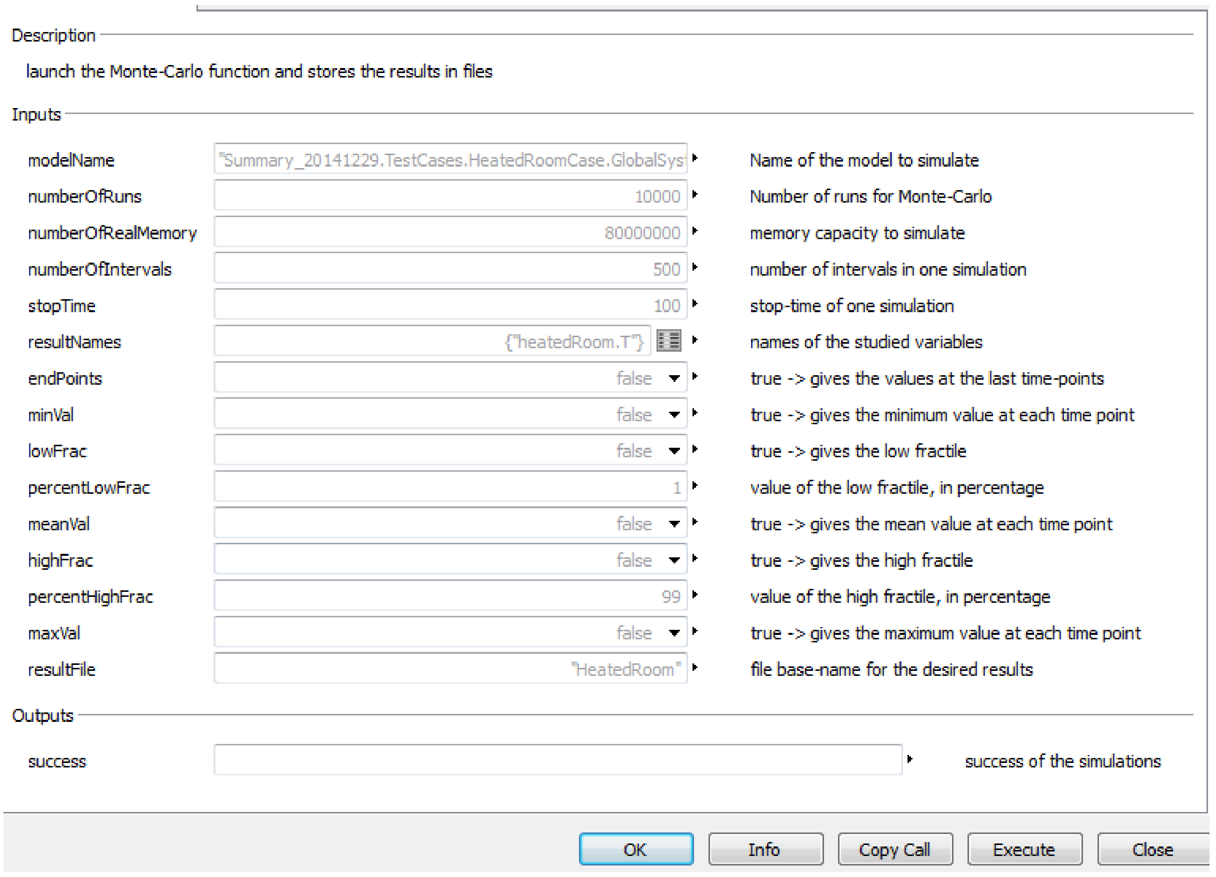
```
Real hazardRate = 1 "time-varying hazard rate" annotation(Dialog);
```

This way, when building our system, we can put a time-varying expression for the hazard-rate.

Monte Carlo simulation

In order to simulate models with stochastic comportments, we choose to use the Monte-Carlo method. This means we will launch a great number of times the “same” simulation, only changing the seed of our random number generator. To do this great number of simulations with some efficiency, we took advantage of the recent SimulateMultiResultsModel() function in Dymola. This function was originally designed to launch several times the same simulation while modifying the value of a parameter, and store the corresponding trajectories of the asked variables. In our case, since the seed for our random number generator is automatically changing each time, we have no parameter to modify between each simulation.

The user gives in parameters to our function MC_function() the path of the model to simulate, the number of requested simulations, the simulation parameters, the name of the studied variable(s) and the requested indicators. For the indicators, we choose to allow the computation of the mean trajectory, the minimum values, the maximum values, two fractiles (customizable percentages) and the end-value. These indicators are enabled with Booleans.


Description

launch the Monte-Carlo function and stores the results in files

Inputs

modelName	"Summary_20141229.TestCases.HeatedRoomCase.GlobalSys	Name of the model to simulate
numberOfRuns	10000	Number of runs for Monte-Carlo
numberOfRealMemory	80000000	memory capacity to simulate
numberOfIntervals	500	number of intervals in one simulation
stopTime	100	stop-time of one simulation
resultNames	{"heatedRoom.T"}	names of the studied variables
endPoints	false	true -> gives the values at the last time-points
minVal	false	true -> gives the minimum value at each time point
lowFrac	false	true -> gives the low fractile
percentLowFrac	1	value of the low fractile, in percentage
meanVal	false	true -> gives the mean value at each time point
highFrac	false	true -> gives the high fractile
percentHighFrac	99	value of the high fractile, in percentage
maxVal	false	true -> gives the maximum value at each time point
resultFile	"HeatedRoom"	file base-name for the desired results

Outputs

success		success of the simulations

Computing the exact fractiles would require a too great number of operations. Indeed, to compute it on N simulations, you would need a magnitude of N^2 comparisons. Since the Monte-Carlo method is based on a great number of simulations, this would not be acceptable. To avoid this, we decided to compute an estimation of the fractile. For this, we used the work presented in *Quantile Optimization for Heavy-Tailed Distributions Using Asymmetric Signum Functions* from Jae Ho Kim and Warren B. Powell, Princeton. According to this paper, the α -fractile of X is the limit Y of the sequence:

$$Y_n = Y_{n-1} - \gamma_{n-1} * sgn_\alpha(Y_{n-1} - X_n)$$

$$where\ sgn_\alpha(u) = \begin{cases} 1 - \alpha\ if\ u \geq 0 \\ -\alpha\ if\ u < 0 \end{cases}$$

$$and\ \gamma_n\ verifies:$$

$$\sum^{\infty} \gamma_n = \infty$$

$$\sum^{\infty} (\gamma_n)^2 < \infty$$

For the initialization Y0 and the sequence γ_n, we chose to first sort the 100 first values. We then take the value of rank α for Y0, we put sigma = 25value+75value/2*75value as a scaling factor, and we put γ_n, = sigma/n^(2/3).

Using this algorithm, we managed to have a number of operations proportional to the number N of simulations, instead of N^2.

In our function, we first do the simulations (with the simulateMultiResultsModel() function), then we compute the indicators with the results of the simulation, depending of the requested indicators. In case the combination size of the model and number of simulations would be too big for the physical memory, the user can also input some memory usage limitations in the parameter of the function. The simulations and indicator computations are then separated into batches of acceptable sizes, to be run one after another and thus prevent any memory saturation.

Heated-room case [4]

Our first example is a very simple one. However, its simplicity enabled experimenting and developing our new method efficiently. This test-case has been solved in several studies for tool-comparisons (Bouissou and Jankovic 2012, Bouissou et al. 2013) and served for our first drafts (Bouissou et al. Modelica Conf 2014).

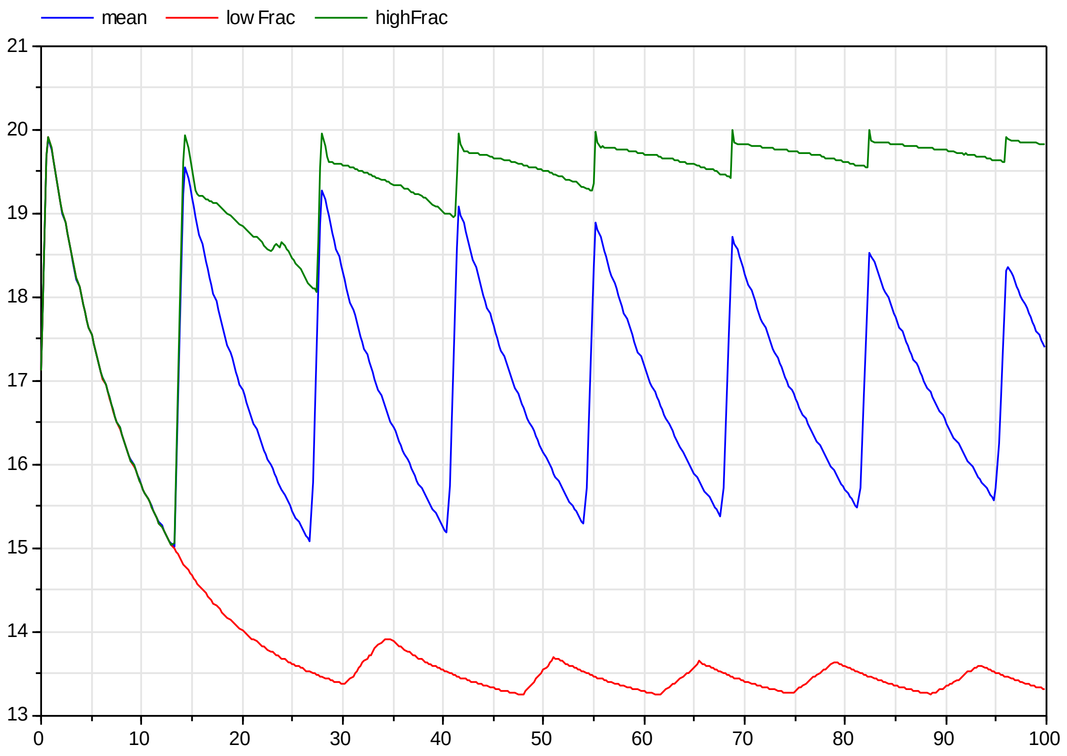
The system consists of a room containing a heater. The heater is controlled by a hysteresis on the ambient temperature. If the ambient temperature falls below 15°C, the heater is switched on. If the ambient temperature reaches 20°C, the heater is switched off. The outside temperature is constant at 13°C. We suppose the ambient temperature follows the differential equation:

$$\frac{dT}{dt} = 0.1 \times \left(Outside_{temperature} - T\right) + 5 \times \left(Heater_{is_{on}}\right)$$

With Heater_is_on an indicator function equal to 1 if the heater delivers power, 0 otherwise. The time t is in hours and the ambient temperature T is in °C.

In nominal circumstances, the ambient temperature should be a deterministic succession of convex and concave exponentials between 15°C and 20°C. However, we give to the heater a constant failure rate lambda=0.01/h and a constant repair rate mu=0.1/h. The aim is to study the influence of the random failures on the ambient temperature.

We model the heater as a state-machine with two states (OK+failed). With our Monte-Carlo simulations, we compute the mean value, the 1% fractile and the 99% fractile of the ambient temperature, for 1,000,000 simulations. This takes us about 1h30min.



Complement for the different kind of developed states

The example of a state shown in the previous part can now be customized for the different kind of transitions we want to model : probabilistic transitions, Markov transitions, on solicitation transitions and on solicitation with a minimum time spent transitions.

Probabilistic transitions

These transitions are very close to the example from 3.2. The only difference is that we want to potentially have several transitions leaving the same state. The solution is to add a parameter, an Integer nbOfExits giving the number of exiting transitions. We then replace each variable by a vector of variable of size nbOfExits. We obtain the following Modelica code for the state :

```
model probabilistic "state with stochastic exit(s)"


 parameter Integer nbOfExits=2   "number of possible stochastic exits from the state";

 Real r[nbOfExits] "random numbers";

 Boolean firing[nbOfExits] "firings of the failure transitions";

 Real F[nbOfExits] "cumulative distribution functions";

 Real hazardRate[nbOfExits] = ones(nbOfExits) "time-varying hazard rates" annotation(Dialog);


 outer GlobalSeed globalSeed "random number generator";


 Modelica.Blocks.Logical.Edge edge[nbOfExits];


                    initial equation
```

```
        F = zeros(nbOfExits);

      equation
        der(F) = (ones(nbOfExits)-F).*hazardRate;
        firing = edge.y;
         for p in 1:nbOfExits loop
           edge[p].u = F[p]>r[p];
         end for;
        if enteringState() then
          for n in 1:nbOfExits loop
            r[n] = globalSeed.random();
          end for;
        else
          r=pre(r);
        end if;

end
```

Markov transitions

We are here in the case of Markov chains, where the time spent in a state is deterministic, but the choice between the different exiting transitions is stochastic. As in the probabilistic transitions, we add the parameter nbOfExits. We also add a Real parameter for the duration that should be spent in the state before exiting it. We replace the cumulative distribution function F by a simple measure of the time spent in the state. Lastly, we compare the random number r with the probabilities for each exiting transitions. We obtain the following Modelica code:

```
model StateMarkov "state with Markov stochastic exit(s) name.firing[i]"
      parameter Integer nbOfExits=2 "number of possible stochastic exits from the state";
      parameter Real stateDuration "time to be spent in the state";
      Real r "random number";
      Boolean firing[nbOfExits] "firings of the failure transitions";
      Real timeSpent(start=0) "time spent in the state";
      Real hazardRate[nbOfExits] = ones(nbOfExits) "time-varying hazard rates, the sum
    must be equal to 1"  annotation(Dialog);
      Boolean firing_time "true -> spent enough time in the state";
      outer GlobalSeed globalSeed "random number generator";

      Modelica.Blocks.Logical.Edge edge;

    equation
      der(timeSpent)=1;
```

```
        firing_time = edge.y;
        timeSpent>stateDuration = edge.u;
        if enteringState() then
          r = globalSeed.random();
        else
          r = pre(r);
        end if;
        firing[1] = firing_time and r<hazardRate[1];
        for n in 2:nbOfExits loop
          firing[n] = firing_time and r<sum(hazardRate[1:n]) and r>=sum(hazardRate[1:n-1]);
        end for;

end StateMarkov;
```

On solicitation transitions

This case is almost the same as the Markov transitions, the difference being that instead of activating the transitions on a timer, we do it upon the activation of a Boolean. This happens for example in the case of a failure on solicitation. As a consequence, we can simply reuse the previous code and just replace the time measurements by a Boolean input sollicitation. This gives the following Modelica code :

```
model StateSollicitation "state with sollicitation stochastic exit(s) name.firing[i]"


        parameter Integer nbOfExits=2 "number of possible stochastic exits from the state";
        Real r "random number";
        Boolean firing[nbOfExits] "firings of the failure transitions";
        Real hazardRate[nbOfExits] = ones(nbOfExits) "time-varying hazard rates, the sum
      must be equal to 1" annotation(Dialog);
        outer GlobalSeed globalSeed "random number generator";


        Modelica.Blocks.Interfaces.BooleanInput sollicitation;
      equation
        if enteringState() then
          r = globalSeed.random();
        else
          r = pre(r);
        end if;
        firing[1] = sollicitation and r<hazardRate[1];
        for n in 2:nbOfExits loop
          firing[n] = sollicitation and r<sum(hazardRate[1:n]) and r>=sum(hazardRate[1:n-1]);
        end for;
end StateSollicitation;
```

On solicitation with a minimum time spent transitions

This case is the same as the previous one, except after having entered the state, you can't leave it before a minimum time. This is the case for example for security switches. To model it, we can just reuse the previous model and re-add to it the time measurements. This gives the following Modelica code :

```
model StateSollicitation2 "state with temporisation and sollicitation stochastic exit(s) name.firing[i]"


  parameter Integer nbOfExits=2 "number of possible stochastic exits from the state";
  Real r "random number";
  Boolean firing[nbOfExits] "firings of the failure transitions";
  Real hazardRate[nbOfExits] = ones(nbOfExits) "time-varying hazard rates, the sum must be equal to 1" annotation(Dialog);
  outer GlobalSeed globalSeed "random number generator";
  parameter Real delay "minimum time spent in the state before being able to leave it";
  Real t "time spent in the state";


Modelica.Blocks.Interfaces.BooleanInput solicitation;
Modelica.Blocks.Logical.Edge edge;
equation
        der(t)=1;
       when enteringState() then
         reinit(t,0);
       end when;
       if enteringState() then
         r = globalSeed.random();
       else
         r = pre(r);
       end if;
       edge.u = sollicitation and t>delay;
       firing[1] = r<hazardRate[1] and edge.y;
       for n in 2:nbOfExits loop
         firing[n] = r<sum(hazardRate[1:n]) and r>=sum(hazardRate[1:n-1]) and edge.y;
       end for;
end StateSollicitation2;
```