EDF R&D - Paris Saclay Lab

Getting started with PyCATSHOO V1.2.2.8

Document version V1.1

Hassane CHRAIBI hassane.chraibi@edf.fr

pycatshoo.org - August 2018

Contents

1 Introduction **3**

2 Not exhaustive review of studies and test cases carried out with PyCATSHOO **3**
2.1 Heated Room . . . 3
2.2 Heated Tank . . . 6
2.3 Operating simulation of cascaded spillways . . . 7
2.4 RLC Circuit modeling and simulation . . . 9
2.5 Cooling system of spent fuel pool . . . 9

3 Basic concepts **11**
3.1 Implementing process overview . . . 12
3.1.1 Generic modeling . . . 12
3.1.2 Specific system modeling . . . 17
3.1.3 Creation of a quantification model . . . 18

4 Generic modeling **18**
4.1 Variables . . . 18
4.1.1 addVariable() . . . 18
4.1.2 setXValue() . . . 19
4.1.3 value() . . . 19
4.2 References . . . 19
4.2.1 addReference() . . . 19
4.2.2 cnctCount(), value(), xValue()) . . . 20
4.2.3 sumValue() . . . 20
4.2.4 productValue() . . . 20
4.2.5 Reference.andValue . . . 21
4.2.6 Reference.orValue . . . 21
4.3 Message boxes . . . 21
4.3.1 addMessageBox() . . . 21
4.3.2 addMessageBoxImport(), addImport() . . . 21
4.3.3 addMessageBoxExport(), addExport() . . . 21
4.4 Automata, states and transitions . . . 22
4.4.1 addAutomaton() . . . 22
4.4.2 addState() . . . 22
4.4.3 setInitState() . . . 22
4.4.4 currentState() . . . 23
4.4.5 addTransition() . . . 23
4.4.6 setDistLaw() . . . 23
4.4.7 addTarget() . . . 24
4.4.8 setCondition() . . . 24
4.4.9 setInterruptible() . . . 25
4.4.10 setModifiable() . . . 26
4.5 Events and sensitive methods . . . 26
4.5.1 addStartMethod() . . . 26
4.5.2 addSensitiveMethod() for states . . . 27
4.5.3 addSensitiveMethod() for automata . . . 27

4.5.4 addSensitiveMethod() for references . . . 27
4.5.5 addSensitiveMethod() for transitions . . . 28
4.6 PDMP Manager . . . 28
4.6.1 addPDMPManager() . . . 30
4.6.2 addPDMPODEVariable(), addODEVariable() . . . 30
4.6.3 addPDMPExplicitVariable(), addExplicitVariable() . . . 30
4.6.4 addPDMPEquationMethod(), addEquationMethod() . . . 30
4.6.5 setDvdtODE() . . . 31
4.6.6 setDValue . . . 31
4.6.7 addPDMPBoundaryCheckerMethod(), addBoundaryCheckerMethod() . . . 32
4.6.8 addPDMPWatchedTransition() . . . 33

5 System construction **33**

6 Quantification **34**
6.1 Setting parameters' and initial values' files . . . 34
6.1.1 File of parameters . . . 34
6.1.2 File of initial values . . . 35
6.2 Call to the system construction method . . . 39
6.3 Construction of prior simulation indicators . . . 40
6.4 Setting sequence filters . . . 42
6.5 Setting simulation parameters . . . 43
6.6 Launching simulation . . . 45
6.7 Setting post-simulation indicators . . . 45
6.8 Post-simulation sequences' filtering and printing . . . 47
6.9 Loading a results' file . . . 47

7 Implementation of a test case **48**
7.1 Step 1: Simple Heaters . . . 49
7.1.1 Heater state variables . . . 49
7.1.2 The heater behavior . . . 49
7.1.3 Step 1 - testing . . . 54
7.2 Step 2: Heaters with master / slave operating . . . 57
7.2.1 Heater reference . . . 58
7.2.2 The heater behavior . . . 58
7.2.3 The heaters communication with the rest of the system . . . 58
7.2.4 Step 2 - testing . . . 63
7.3 Step 3: Introduction of the heated room . . . 66
7.3.1 Room state variables . . . 66
7.3.2 Room references . . . 67
7.3.3 The room behavior . . . 67
7.3.4 The room communication with the rest of the system . . . 67
7.3.5 The room PDMP Manager part . . . 67
7.3.6 Step 3 - testing . . . 77
7.4 Step 4: Drawing sequences . . . 82

1 Introduction

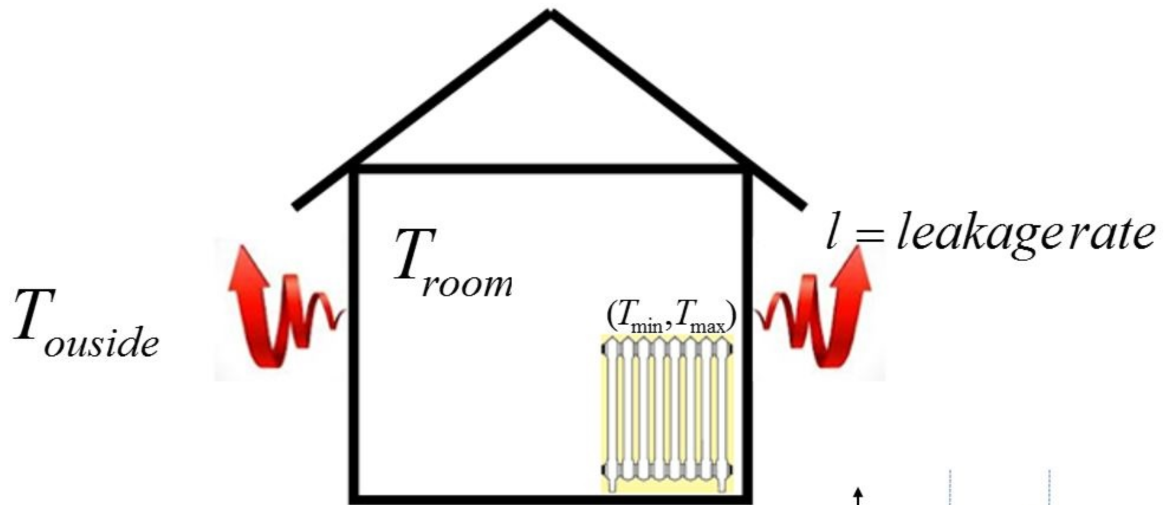
Classic methods of probabilistic safety assessments (PSA), often used in nuclear domain, even when they go beyond the static approaches, remain, for historical reasons, confined in discrete events framework and quickly reach their limits when finer dependability assessment is needed. These methods neglect a large part of information about our systems. In particular, physical phenomena modeling, are not explicitly present in PSA and are often replaced by so called conservative assumptions.
Yet several attempts have been made to remedy this. But their implementations are sometimes based on a tool dedicated to the discrete events modeling into which some continuous modeling functionalities are intruded. Conversely, the starting point may be a tool dedicated to the 0D physical modeling patched with some discrete events modeling functionalities. In both cases the results are not satisfactory because their use for the assessment of even simple systems often requires some convoluted modeling which may be so difficult to maintain..
PyCATSHOO brings an original solution in that it offers a method and a tool where the two paradigms, continuous deterministic on the one hand and discrete stochastic on the other hand, are natively integrated. It also provides a lot of convenient tools which make user feel at ease with the increase of modeling complexity due to this integration.
To ease the understanding of what PyCATSHOO is, we begin this document by a non exhaustive list of some studies carried out with PyCATSHOO.

2 Not exhaustive review of studies and test cases carried out with PyCATSHOO

2.1 Heated Room

In this example the studied system is a room whose temperature is to be kept between two thresholds values using one or several heating devices equipped with thermostats. When the heaters are stopped, the temperature in the room drops because of thermal leakage and the low external temperature. As the heaters may fail, there is a chance that the room temperature will not be maintained in the appropriate range. Several performance indicators of this room heating system can be assessed with PyCATSHOO:

- The evolution over time of the mean temperature in the room
- The quantiles of the temperature (e.g. values of the 1% highest temperatures and 1% lowest)
- The mean time spent by the room under a given temperature value
- The cumulative probability distribution function of a chill event occurrence (e.g. Temperature < 14.°C)



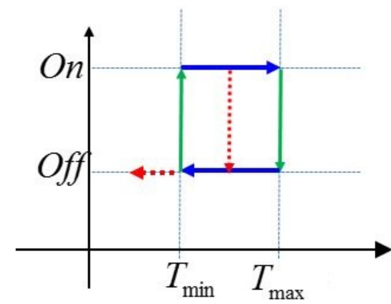


Figure 1
Overview of the test case Heated room

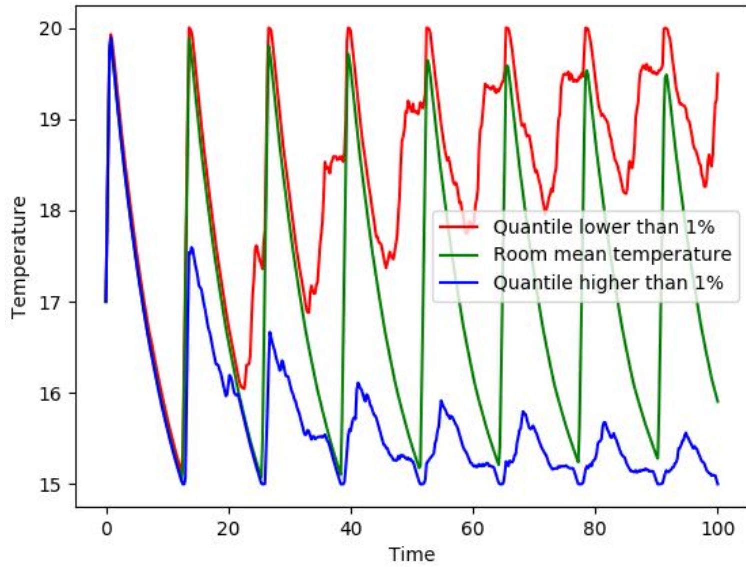


Figure 2

An example of the heated room test case outputs - Means and quantiles

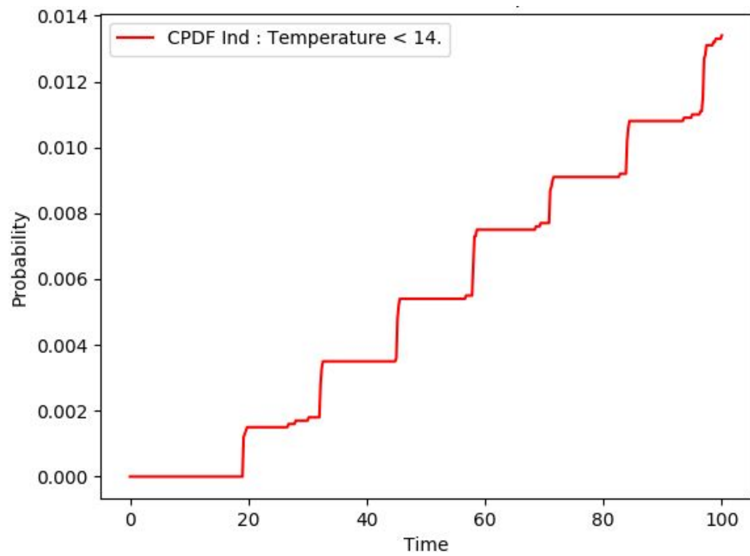


Figure 3

An example of the heated room test case outputs
Cumulative distribution of the undesired event probability

2.2 Heated Tank

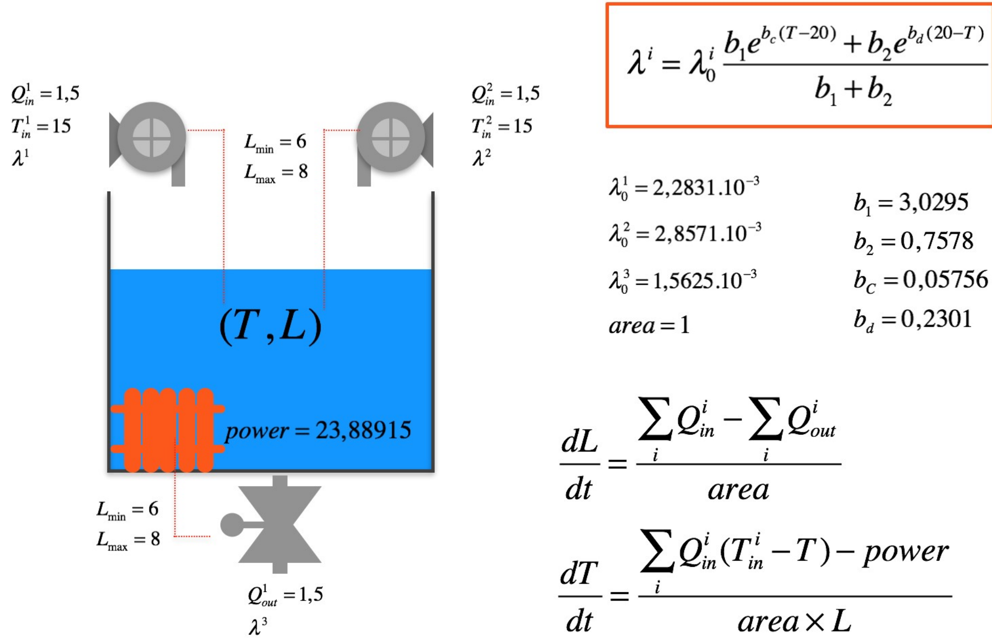


Figure 4

Overview of the test case Heated tank

This example comes from a well known heated tank benchmark. In this case the system is composed of five components:

- One water containing tank
- A source of heat (fuel) that brings a constant amount of thermal power to the water contained in the tank
- Two pumps that supply the tank with cold water
- A valve which allows the water contained in the reservoir to be evacuated

The two pumps and the valve are actuated according to the level of water in the tank in order to prevent the overflow and the dryout. The objective here is to evaluate the probability that, at a given time, the system has experienced one of the three following undesired events.

1. Dryout: The level of water in the tank goes under 4m
2. Overflow: The level of water in the tank exceeds 10m
3. Overheating: The temperature of water in the tank reaches 100°C

The figure 4 reminds the benchmark data. In particular, it gives the system of ordinary differential equations (ODE) which governs the temperature and the level inside the tank. This ODE system is deduced form the material and energy balances. The figure 4 also gives the failure rate of the pumps and the valves as a function of the temperature.

Despite its simplicity, this example fully falls in the scope of the dynamic reliability. Indeed, its discrete behavior depends on the continuous state variables in two ways. The first one is deterministic since the system configuration changes when some limits are exceeded by the continuous variables. The second way is stochastic since the system configuration changes after component failures whose occurrence rate depends on the continuous variables.
The figure 5 gives the cumulative occurrence probability distribution of the three undesired events.

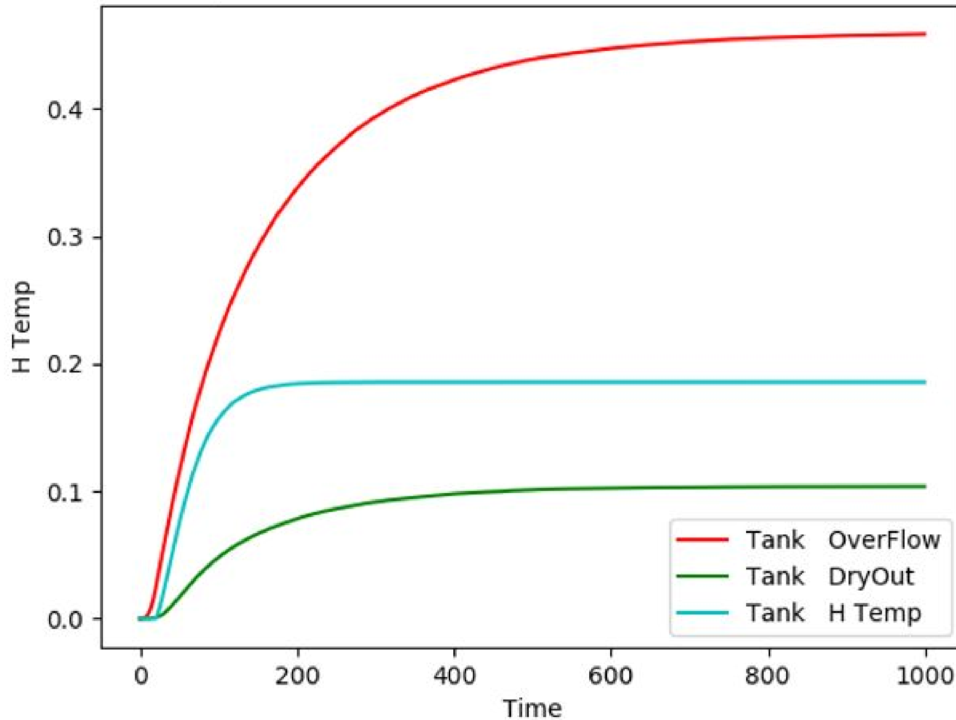


Figure 5
An example of the heated tank test case outputs Cumulative distribution of the undesired events probabilities

2.3 Operating simulation of cascaded spillways

This example is taken from a real study carried out at EDF. It represents the modeling of gated spillways of a cascaded dams with two objectives: the validation of the spill control rules and the calculation of the return period distribution of the incoming water flow rate and of the water levels in the dams' upstream basins.
A simplified representation of this system model is given in the figure 6 below. This model processes several hundreds of thousands of hydrographs weighted by their probabilities of occurrence.
The stochastic behavior currently taken into account is only related to the floods' hydrographs. But

this study is called to extend the stochastic behavior to the potential failures of the components of the entire spillway system.

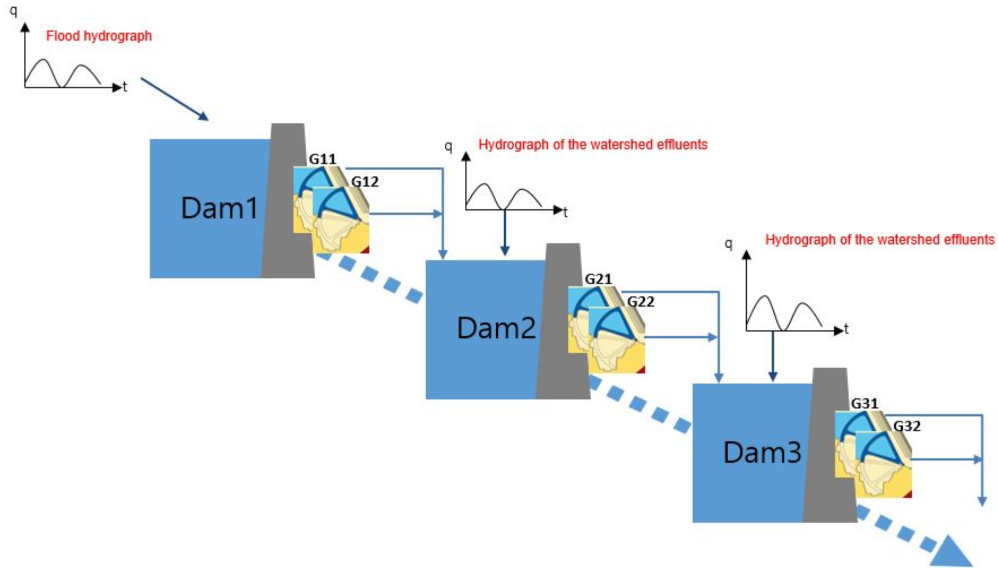


Figure 6

Overview of the dam cascade system

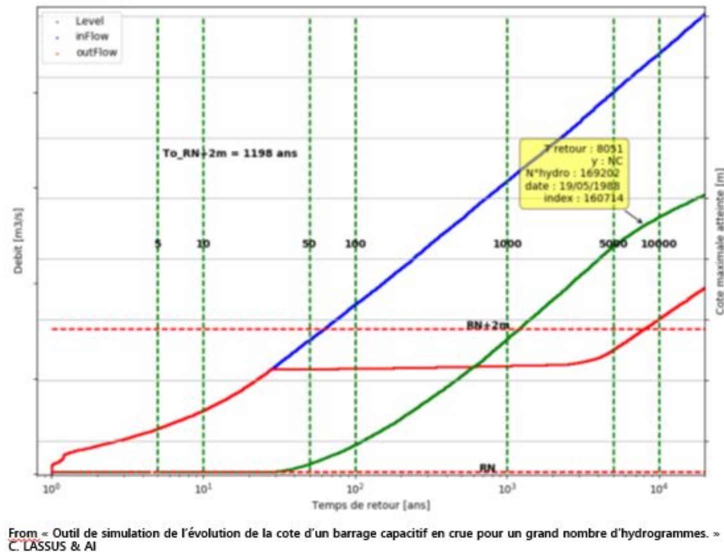


Figure 7

An example of the dam cascade system assessment results

2.4 RLC Circuit modeling and simulation

Figure 8 gives the output of the simulation model of a simple RLC circuit. For training purpose, we have developed a library of several deterministic models of simple electrical components. We have then used this library to model and simulate circuits such as the one presented in the figure 8.

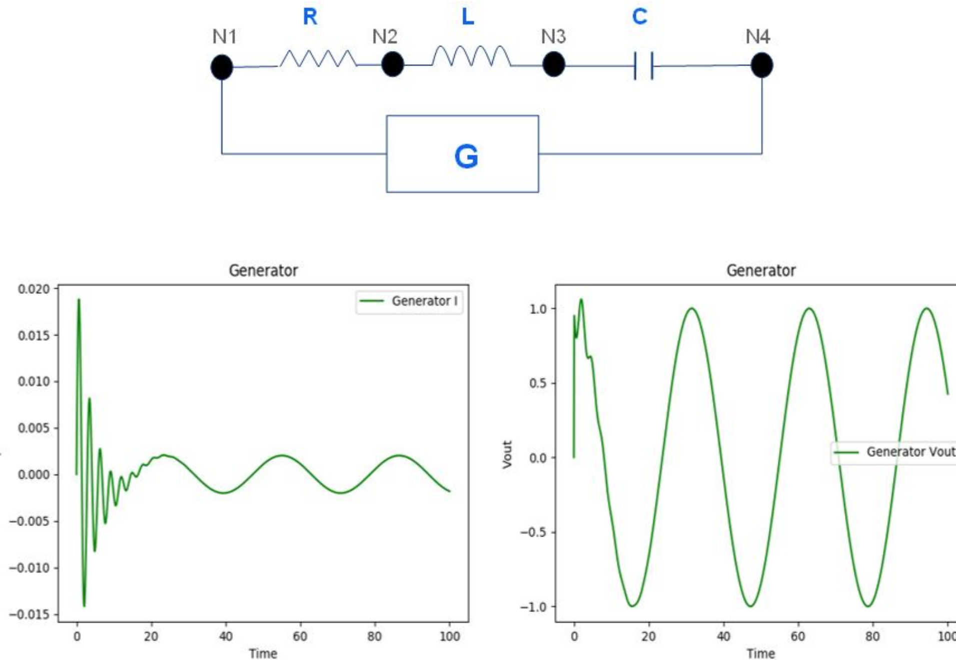


Figure 8

A simple RLC circuit & its simulation outputs

2.5 Cooling system of spent fuel pool

The assessment of this system has been reported in a paper presented at PSAM13.
The system consists of three trains: two identical main trains and a third fully independent train (or "stand-by train"). In the simplified design adopted for the present study (as shown in following figure 9), each train is composed of three water loops connected to two heat-exchangers. Each loop is composed of a valve, a check-valve and pump. A train is supplied by electrical power through a dedicated switch board division which is itself fed by the external electricity network and backed-up by a dedicated emergency diesel generator. The three trains are operated in passive redundancy way in order to maintain the water temperature in the pool under 80°C. Three undesired events have been assessed in this study:

1. Temperature exceeding 80°C
2. Temperature reaching boiling threshold (100°C)
3. Fuel uncovering (water level lesser than 16m)

These three events are expressed in terms of thresholds exceedance by deterministic continuous variables which result from an ordinary differential equation system.
Because the conventional methods are limited to discrete events paradigms, they cannot be used, without conservative assumptions, to assess the aforementioned undesired events.
The assessment with PyCATSHOO of these three undesired events have been compared to the evaluation, by a conventional method, of an event equivalent to the simultaneous loss of the three cooling trains. The figure 10 below gathers the results of the assessment of the four undesired events. It shows how high the safety margins loss can be when such a study is carried out with conventional approaches.

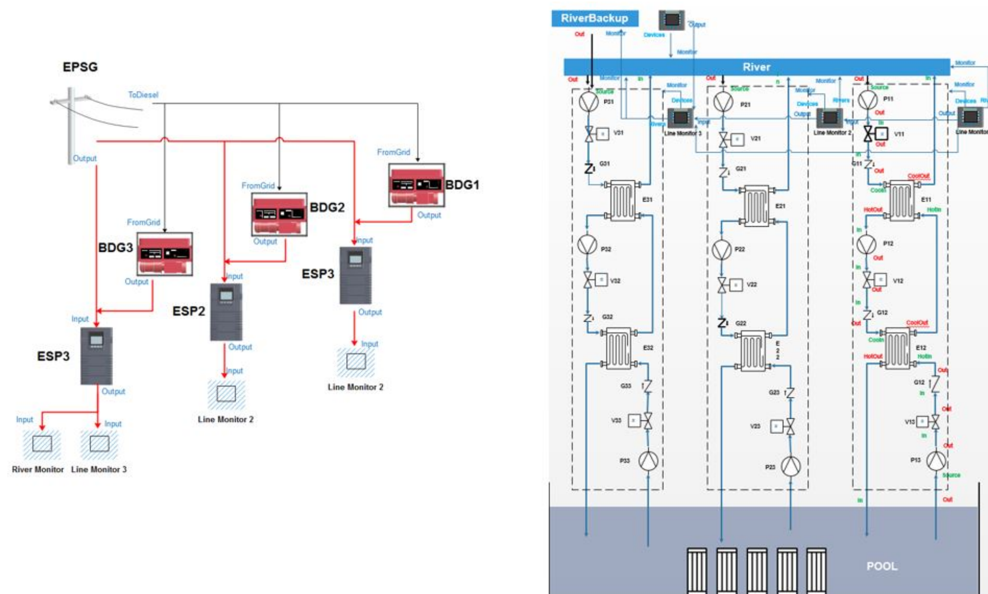


Figure 9

Overview of the PyCATSHOO simplified model of the Spent Fuel Pool Cooling system

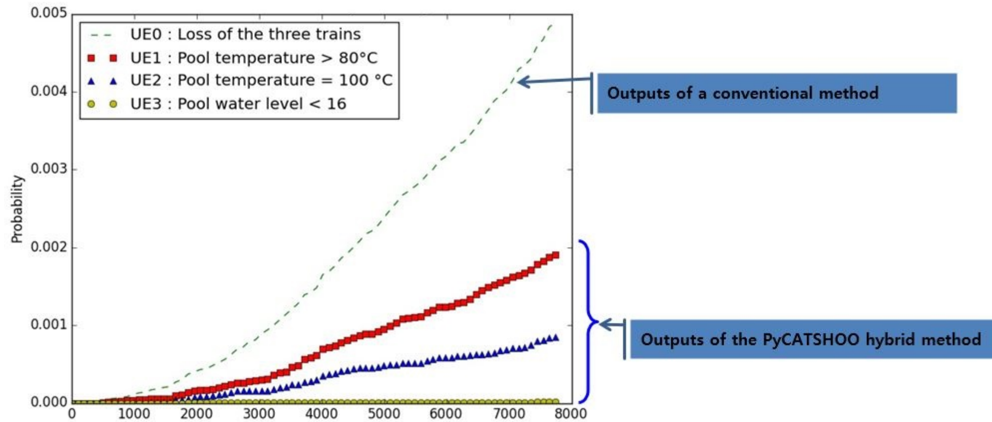


Figure 10

Excerpt from the assessment results of the Spent Fuel Pool Cooling system assessment

3 Basic concepts

PyCATSHOO can be used to model and assess the performances of complex hybrid dynamic systems. Here, performance, most of the time, means dependability and "hybrid" qualifier denotes the coexistence and interaction, in the same model, of discrete and stochastic behavior in the one hand with deterministic and continuous physical phenomena in the other hand.
PyCATSHOO is a dynamic library released in several versions. These different versions allow the use of PyCATSHOO in C ++ language or in Python language, on Windows 32/64, Linux 32/64 or Mac OS platforms. The library offers three kinds of tools:

- Tools that help in generic modeling i.e. building libraries of classes that stand for the components of a given system category. These libraries are also called ***knowledge bases***.
- Tools dedicated to the modeling of a specific system based on the instantiation of the classes imported from a knowledge base. These tools also allow to customize and start parallel simulations.
- Post processing tools to help with the analyzing of the results.

PyCATSHOO is based on the notion of component abstractions which are classes, derived from the ***CComponent***, that belong to a knowledge base. The PyCATSHOO model of a system must be included in a class derived from ***CSystem*** where:

- The instantiations of the knowledge base classes are realized according to the system composition
- The components of the system are linked to each other according to its architecture
- The system quantification is carried out via Monte Carlo simulation
- The equations of the physical phenomena are solved

The restitutions of quantification results and their post processing may be done in two ways according to the goals of the study.
The first way requires the definition of performance indicators before the simulation run. This is the recommended approach as it requires less memory although it slightly slows the simulations. The second way consists in monitoring some system states and variables during the simulation. This means that all the values taken by these elements during the simulated sequences are memorized. At the end of the simulation, one can use an instance of the ***CAnalyser*** class. This class provides statistical computational routines. It helps requesting indicators such as mean values, standard deviation, sojourn times, etc.

3.1 Implementing process overview

The PyCATSHOO implementation process can be summed up in three steps illustrated in the following figure 11.

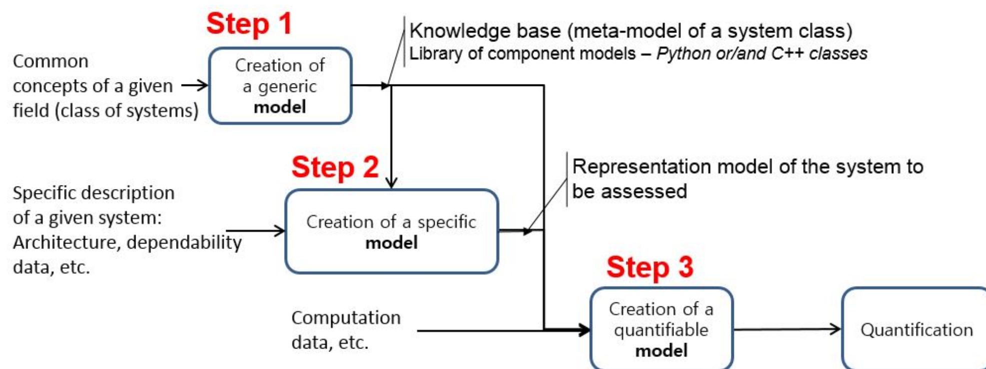


Figure 11

PyCATSHOO Implementing process

3.1.1 Generic modeling

This first step consists in creating a model of every component which makes up the class of the systems to be assessed. Such models must cover functional and dysfunctional behavior of a component, define its mission, the information which must be available to the rest of the system and the information it is susceptible to receive from the rest of the system.

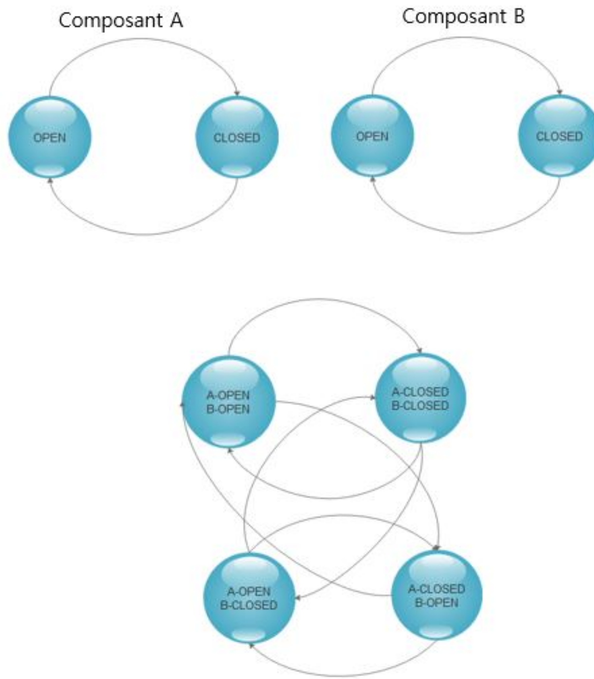


Figure 12

System and Component states

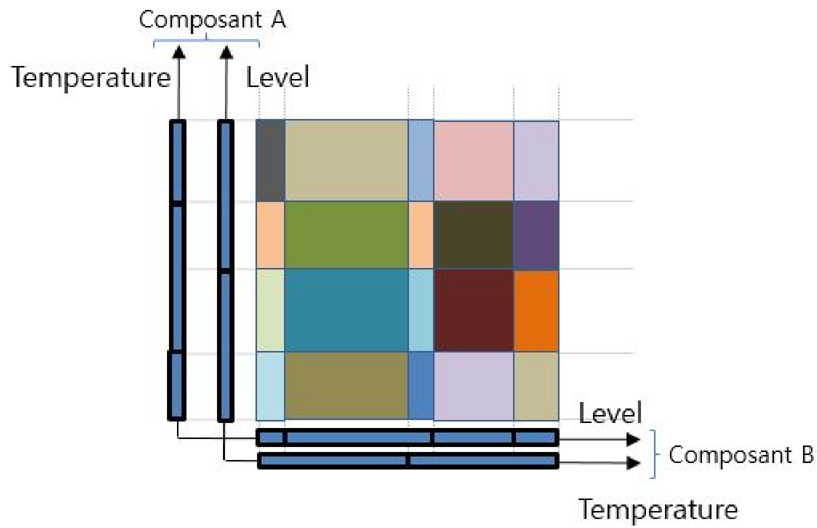


Figure 13

Regions of the system and a component continuous state variables

So this modeling consists in creating a representing **class** for every system class of components or concepts. This class has to be derived form ***CSystem***, a PyCATSHOO builtin class. It holds several kinds of items:

- Variables: They are the actual intrinsic state variables. They may be discrete or continuous and they contribute to characterizing the state of a component to which they belong.
- References: Designated below by the term **"references"**, they are references to external variables. They play a role of a receptacle for the information coming from the outside of the component and hold its perception of the state of the system surrounding components.
- The message boxes: The object oriented principle of encapsulation requires that only the owner component has a direct access to its variables. So the raison d'être of the message boxes is to ensure the communication between a component and the rest of the system while respecting this principle. A message box holds several channels or slots. Each channel is connected to one variable or one reference. When connected to a variable a slot exposes the value of the latter to the component's outside. Conversely, when connected to a reference, the slot feeds the latter by incoming values originated from variables of other components.
- Automata: They define the behavior of the component. They are composed of a set of states and transitions. The latter are characterized by a probability distribution and a condition. The condition is a boolean expression or function which involves the state variables and the references. This means that when the variables change or when new values are received by references, a transition may be triggered since its condition may become true. The automata in PyCATSHOO are defined at the component level for obvious reasons: avoiding the combinatorial explosion and preserving the understanding of the system behavior and its modeling. Indeed, as shown in the two figures 12 & 13, on the one hand the state space of the entire system is the Cartesian product of the states of its component and, on the other hand, the zoning of continuous state variables of the entire system is also a Cartesian product of regions where the component can evolves when it is in a particular state. These Cartesian products are not actually constructed that is why the combinatorial explosion is avoided.
- The PDMPs (Piecewise Deterministic Markov Processes): PyCATSHOO is based on the theoretical framework of the piecewise deterministic Markov processes (PDMP) which can be roughly described as follows:

- The system may be in one of a finite set of modes M and, for each mode $m \in M$ the continuous state variables are confined in a specific region $\Omega_m \subset \mathbb{R}^d$
  This means that the global state of the system is :

$$E = \bigcup_{m \in M} E_m \quad where \quad E_m = m \times \Omega_m$$

- For every mode m, $x(t+s) = \Phi_m(s, x(t))$
  In general Φ_m is a solution of an ordinary differential equation system : $\frac{dx(t)}{dt} = f(t, x(t))$

- When in a mode m the system may be subject to a spontaneous jump into another mode m'. Otherwise it evolves until it hits the boundary $\partial\Omega_m$, if it exists, of the current region Ω_m in which case the system is forced to jump into a new mode m'' which is compatible with the new position in the continuous state space. The probability of the jump outside the current mode is given by the following equation:

$$P(T < t | Z = (x,m)) = \begin{cases} 1 - e^{-\int_0^t \lambda_m(x(u))\,du} & \text{if} \quad t < t^*(x,m) \\ 1 & \text{if} \quad t \geqslant t^*(x,m) \end{cases}$$

where

$$t^*(x,m) = \begin{cases} inf\{t > 0, \Phi_m(t,x) \in \partial\Omega_m\} & if \quad \partial\Omega_m \quad exists \\ \\ \infty & if \quad \partial\Omega_m \quad doesn't exist \end{cases}$$

In case of a spontaneous jump, the probability to reach a particular mode m' is the given by the following equation:

$$K_{x,m}(m') = \frac{\lambda_{m\to m'}(x)}{\lambda_m(x) = \sum\limits_{n\in M, n\neq m} \lambda_{m\to n}(x)}$$

Where $\lambda_{m\to n}(x)$ is the transition rate from the mode m to n given the position in the continuous state space.
In case of a forced jump when the boundary is hit, the probability to reach a particular mode m'' is given by the following probability:

$$K_m(m'') = \gamma_{m\to m''}$$

The transition rate $\lambda_{m\to n}$ and the probability $\gamma_{m\to m''}$ are an input data for systems' assessment.

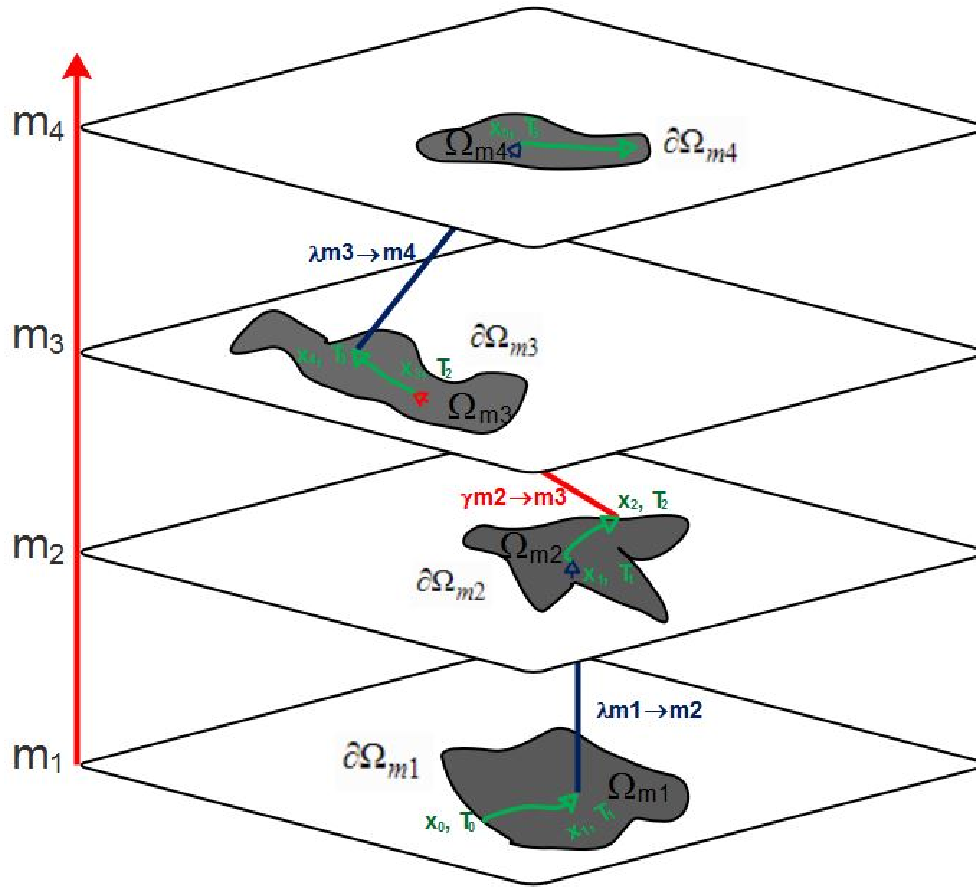


Figure 14

A PDMP trajectory

The figure 14 above gives an example of a PDMP's trajectory. Green curves stand for deterministic evolutions which may be interrupted by spontaneous jumps represented by blue arrows. Forced jumps after a boundary reach are pictured by red arrows.
Hence, even if the PDMP is a global entity shared by the entire system, it is defined in a distributed way. Indeed, every component contributes to its definition by giving a part of the global ordinary differential equation system, a part of the global system of linear equation or by giving a part of the global linear programing system which addresses integer and mixed models.

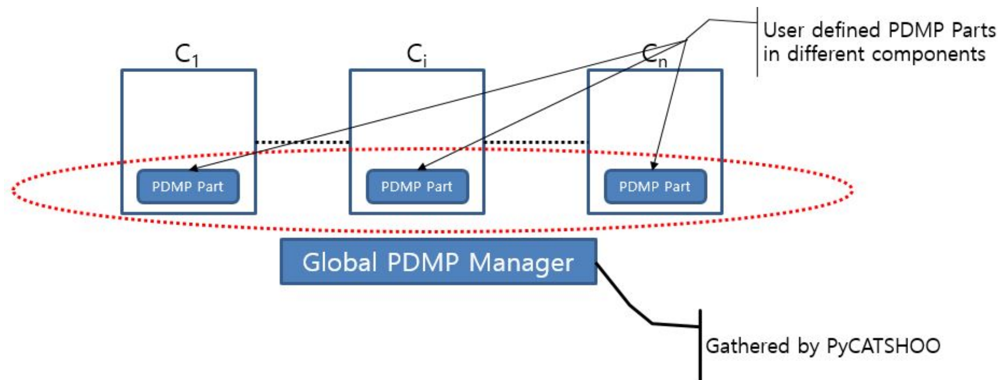


Figure 15

Distributed PDMP Definition

3.1.2 Specific system modeling

The second step aims at modeling a specific system. It uses our knowledge about this system in terms of composition, architecture and different data. This modeling is guided by the knowledge base used as a meta-model. As illustrated in figure 16, it consists on the one hand in instantiating the classes of the knowledge base according to the composition of the system and, on the other hand, in connecting the message boxes of these instances according to the architecture of the system.

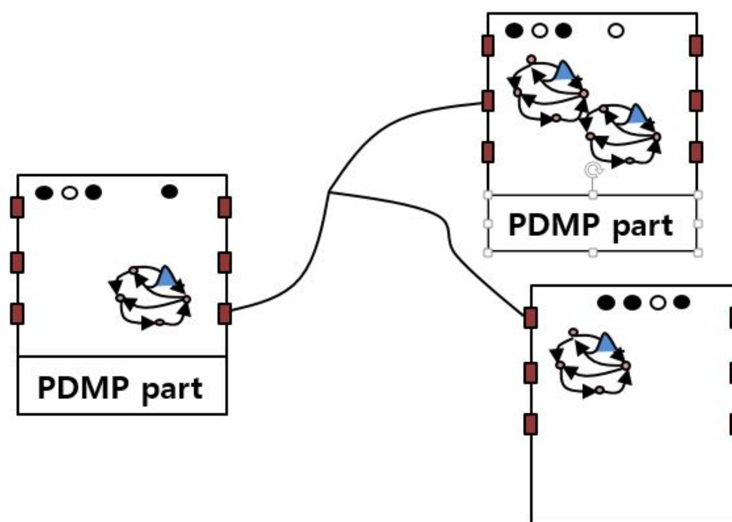


Figure 16

Specific System Modeling

3.1.3 Creation of a quantification model

This third step is based on the knowledge base and on the system model. It consists in the creation of a quantifiable model with new input data about simulation parameters, new definition of the desired indicators, etc.

4 Generic modeling

The generic modeling consists in producing generic models for every component which belongs to the targeted class of the system.
The components' models are basic elements of a knowledge base. They are used to model every constituent of a given class of systems, whether material or conceptual. Programmatically, a component is a class derived form the PyCATSHOO class ***CComponent***.

```
...
 class Heater(Pyc.CComponent):
     def __init__(self, name):
         Pyc.CComponent.__init__(self, name)
```

This excerpt creates a class called Heater. This class has a constructor which requires, as a unique parameter, a string called *name*.

The creation of an instance of an existing class is done within a system as follows:

```
...
    self.aHeater = Heater(externalHeaterName)
```

Here an instance of Heater is created. *externalHeaterName* is a string which gives the external name of the created instance.

A class is mainly made up of three kinds of information. We will describe this information in the next section using the PyCATSHOO Python API.

4.1 Variables

Variables characterize the intrinsic state of a component. Thus, for instance, a dam's upstream basin may be characterized by the water level, a spent fuel pool may be characterized by the level and the temperature of the water, etc.
The declaration of a variable is done inside a class as follows :

4.1.1 addVariable()

```
...
    self.pythonVariableName = self.addVariable(externalVariableName,
                                               variableType,
                                               initialValue,
                                               toReinitialize)
```

Where :

- *externalVariableName* is a string which gives the external name of the variable.
- *variableType* gives the variable type which may be **Pyc.TVarType.t_double**, **Pyc.TVarType.t_float**, **Pyc.TVarType.t_int**, **Pyc.TVarType.t_bool** or **Pyc.TVarType.t_complex**.
- *initialValue* gives the initial value of the variable. This value may be overridden by an external xml file which references the variable by its external name *externalVariableName*.
- *toReinitialize* is an **optional** boolean parameter. When True, the initial value will be affected to the variable immediately after every discrete event occurrence. Its default value is False

A variable is a rather complex object. Thus, specific "setters" and "getters" are required to modify and retrieve its value.

4.1.2 setXValue()

```
...
    self.variable.setDValue(aPythonDoubleValue)
```

```
...
    self.variable.setFValue(aPythonFloatValue)
```

```
...
    self.variable.setIValue(aPythonIntegerValue)
```

```
...
    self.variable.setBValue(aPythonBooleanValue)
```

These setters modify the variable value in a way that is compatible with its type.

4.1.3 value()

```
...
    self.variable.value()
```

This getter asks for the variable value whatever its type is.

4.2 References

References are simply receptacles for external information incoming from the component's outside. This may be for instance the values of incoming flow rate measured by a sensor.

4.2.1 addReference()

```
...
    self.outsideTemperature = self.addReference(referenceExternalName)
```

Where :

- *referenceExternalName* is a string which gives the external name of the variable.

A reference may only be seen as the image of an actual remote element that the object cannot access. There is therefore no direct "setters" for the references.
References getters are slightly different from those of variables. To illustrate this difference, let's imagine a tank which is fed by several valves. We do not necessarily know in advance the number of valves. That is why, in the model of the tank, we will only have one reference as a receptacle for all the incoming flow rates. This means that the flow rates reference will have a behavior close to that of a vector as shown in the following excerpts.

```
...
    self.flowRate = self.addReference("flowRate")
```

Declaration of a reference.

4.2.2 cnctCount(), value(), xValue())

The following excerpt gives an illustration of a reference use.

```
...

    sumOfFlowRates = 0
    for i in range(self.flowRate.cnctCount()):
       sumOfFlowRates = sumOfFlowRates + self.flowRate.value(i)
```

Here self.flowRate.cnctCount() gives the number of incoming values which is equal to the number of connected valves and self.flowRate.value(i) gives the flow rate of the ith value which is the incoming flow rate from the ith connected valve. In case of possible confusion it is recommended to use self.reference.xValue(index) where x can be d, f, i, b, c according to the expected type (double, float, integer, bool, complex). This will be fixed in a next release.

References provide the flowing convenient operators:

4.2.3 sumValue()

```
...
    self.reference.sumValue(defaultValue)
```

Gives the **sum** of all incoming values connected to *self.reference*. This may replace the loop given above.

4.2.4 productValue()

```
...
    self.reference.productValue(defaultValue)
```

Gives the **product** of all incoming values connected to *self.reference*.

4.2.5 Reference.andValue

```
...
    self.reference.andValue(defaultValue)
```

This call is only relevant for references to boolean variables. It applies the **and** operator to all the values connected to *self.reference* and return the result.

4.2.6 Reference.orValue

```
...
    self.reference.orValue(defaultValue)
```

This call is only relevant for references to boolean variables. It applies the **or** operator to all the values connected to *self.reference* and returns the result.

All these operators return *defaultValue* when self.reference is empty.

4.3 Message boxes

In PyCATSHOO we tend to respect the encapsulation principles thus we provide the means, namely message boxes, to do so.
A component can exhibit several message boxes. A message box draws up a list of a component variables whose values are readable from the outside of the component. It also provides slots for incoming values that feed the component references.
The creation of a message box can be done as follows:

4.3.1 addMessageBox()

```
...
    messageBox = self.addMessageBox(messageBoxName)
```

Creates a new message Box whose name is *messageBoxName*.

4.3.2 addMessageBoxImport(), addImport()

Add an incoming slot -that feeds *self.reference*- to an existing message box.

```
...
    self.addMessageBoxImport(messageBoxName, self.reference, slotName)
    #or
    messageBox.addImport(self.reference, slotName)
```

4.3.3 addMessageBoxExport(), addExport()

Add an outgoing slot -whose source is *self.variable*- to an existing message box.

```
...
    self.addMessageBoxExport(messageBoxName, self.variable, slotName)
    #or
    messageBox.addExport(self.variable, slotName)
```

Where :

- *messageBoxName* is a string that gives the external name of the message box
- *slotName* is a string that gives the external name of the slot
- *self.variable* is a component variable
- *self.reference* is a component reference

4.4 Automata, states and transitions

The Automata are the means used by PyCATSHOO to model the dynamic behavior of a component and to make it an autonomous actor.
An automaton is comprised of a set of states. It is declared within a class as well as its states as follows:

4.4.1 addAutomaton()

```
...
    self.anAutomaton = self.addAutomaton(automatonName)
```

Creates a new automaton named *automatonName*.

4.4.2 addState()

```
...
    self.stateX = self.addState(automatonName, stateXName, stateXIndex)
```

Adds to an existing automaton a state named *stateXName* with an Index equal to *stateXIndex*.

```
...
    self.stateY = self.anAutomaton(stateYName, stateYIndex)
```

Adds to an existing automaton a state named *stateYName* with an Index equal to *stateYIndex*.

4.4.3 setInitState()

```
...
    self.anAutomaton.setInitState(stateXName)
```

Designates *stateXName* as the initial state of its automaton.

```
...
    self.anAutomaton.setInitState(self.stateX)
```

Designates *self.stateX* as the initial state of its automaton.

Where :

- *automatonName* is a string which gives the external name of the automaton

- *stateXName* and *stateYName* are strings which designate the names of the states
  A name of a state must be unique not only in the scope of the automaton but also in the scope of the component
- *stateXIndex* and *stateYIndex* are integers which designate the indexes of the states
  An index of a state is an integer which is unique in the scope of the automaton

It is strongly recommended to designate the initial state of an automaton just after the creation of its states. The initial state may obviously be modified later either during the creation of a system or through an xml parameter file.
An automaton may be asked to return its current state or the index of the latter through the following calls:

4.4.4 currentState()

```
...
    self.anAutomaton.currentState()
    self.anAutomaton.currentIndex()
```

These two calls indicate the current state of the automaton *anAutomaton*. The first one returns a reference to the state. The second returns the current state index in the automaton.

4.4.5 addTransition()

Once all the automaton states are declared the creation of the transitions between these states can be done by the instructions given below.

```
...
    self.aTransition = self.stateX.addTransition(transitionName)
```

Creates a new transition starting form the previously created state *self.stateX* and named *transitionName*

4.4.6 setDistLaw()

```
...
    self.aTransition.setDistLaw(probabilityLawType, probabilityParameter)
```

This call is optional. When it is omitted the transition is processed immediately after its condition becomes True.
Otherwise, this call sets the probability law that determines when the transition can be fired

The parameter *probabilityLawType* designates the probability distribution. It can take one of the folowing values : - **Pyc.TLawType.expo** for expnential distribution. - **Pyc.TLaxType.inst** for discrete distribution. - **PycTLawType.defer** for determenistic transition with fixed delay.
Other kindes of probability distributions will be available in next versions of PyCATSHOO. Meanwhile, the user can create its own distributions.

4.4.7 addTarget()

```
...
    self.aTransition.addTarget(self.stateY, transitionType)
```

Adds *self.sateY* as a target to the transition

Where :

- *transitionName* is a string that gives the external name of the transition.
- *probabilityLawType* designates the type of probability law used to randomly sample the instant where the transition can be fired. This parameter may take one of the three following values:
  - **Pyc.TLawType.expo**: for exponential probability law. In this case *probabilityParameter* designates the transition rate.
  - **Pyc.TLawType.inst**: called instantaneous probability law. It means that when the transition condition is satisfied the transition is fired immediately. This transition can lead to two or more targets.
    When two targets are possible *probabilityParameter* gives the probability to reach the first target declared and the second one may be reached with a probability equal to *1 - probabilityParameter*
  - **Pyc.TLawType.defer**: for a deterministic transition fired after a duration equal to *probabilityParameter* starting from the instant when the condition of the transition becomes true
- *transitionType* designates the type of the transition.
  This parameter can take one of the three following values:
  - **Pyc.TTransType.fault**: When the transition is assimilated to a failure
  - **Pyc.TTransType.rep**: When the transition is assimilated to a repair
  - **Pyc.TTransType.trans**: for the other types of transitions

4.4.8 setCondition()

```
...
    self.aTransition.setCondition(...)
```

This call is optional. When it's omitted the transition can be fired without condition. Otherwise, this call sets a condition that must be satisfied before the probability law is asked to randomly sample the instant when the transition can be fired. This call has several variants as we can see below:

```
...
1     self.setCondition(value)
```

Here, *value* is a python boolean variable or literally True or False.

```
...
2     self.setCondition(self.variable)
```

Here, *self.variable* is a reference to a component state variable. The value of this variable may change over the simulation time and so does the transition condition.

```
...
3    self.setCondition(self.variable, takesTheOpposite)
```

This call leads to the same result as the previous one if the parameter *takesTheOpposite* is False. Otherwise it is equivalent to the following call:

```
...
        self.setCondition(self.not_variable)
```

where *self.not_variable* is equal to the negation of *self.variable*.

```
...

4    self.setCondition(self.methodReference)
```

Here, the transition condition is be the result of the call to the component method *self.methodReference*. The latter can be replaced by a Python ***lambda function*** which returns a boolean value.

```
...
5    self.setCondition(self.methodReference, takesTheOpposite)
```

Here the meaning of the parameter *takesTheOpposite* is the same as in the case number 3.

4.4.9 setInterruptible()

When the probability law is **Pyc.TLawType.expo** or **Pyc.TLawType.defer**, the instant when the transition must be fired is drawn at the instant when the transition condition becomes true and the countdown begins just after that. The default behavior is that this countdown never stops even though, in the meantime, the condition becomes false. To change this behavior and make the countdown stop when the condition becomes false the following call must be done:

```
...
    self.aTransition.setInterruptible(True)
```

If the condition becomes true again, a new countdown starts without taking into account the previously interrupted countdown.
To come back to default behavior the following call has to be done:

```
...
    self.aTransition.setInterruptible(False)
```

4.4.10 setModifiable()

A transition between two states in a component automaton can be fired several times during a simulation. This means that the time when this transition is fired is also drawn (calculated) several times. In the default behavior, all of these times are drawn using the initial value of *probabilityParameter*, the probability law parameter which is the transition rate for an exponential law.
There are two other alternatives to this behavior that can be set up with the two following calls:

```
...
    self.aTransition.setModifiable(Pyc.TModificationMode.discrete_modification)
```

```
...
    self.aTransition.setModifiable(Pyc.TModificationMode.continuous_modification)
```

The time when the transition must be fired is computed, in the case of the behavior induced by the first call, using the current value of *probabilityParameter* which is the value given to the probability law parameter at the most recent change. This behavior allows to take account of discrete evolution probability law parameters.
The second call allows to consider the continuous evolution over time of the probability law parameter *probabilityParameter*. This means that the integral form of the probability law is used to draw the instant when the transition must be fired. The following equation reminds the cumulative distribution function of the used probability law when it has an exponential form.

$$\mathrm{P}(T < t) = 1 - \mathrm{e}^{-\int_0^t probabilityParameter(u)\,\mathrm{d}u}$$

As *probabilityParameter* evolves continuously over time and it might even depend on the component state variables, this parameter must be driven by the PDMP (Piecewise Deterministic Markov Process) manager that will be introduced in the next sections.
To come back to the default behavior the following call must be done:

```
...
    self.aTransition.setModifiable(Pyc.TModificationMode.not_modifiable)
```

4.5 Events and sensitive methods

In PyCATSHOO, an event corresponds to the occurrence of a modification in one of the system components and marks some key instants in a simulation. For each one of these events the knowledge base developer can set a method, called sensitive method, that PyCATSHOO automatically calls when the event occurs.
There are five types of events in PyCATSHOO.

4.5.1 addStartMethod()

The beginning of the simulation corresponds to the instant 0 just after all states and variables are reset to their initial values. One may have, for instance, to finalize the initializations by giving to the induced variables the appropriate initial values. To do so we have to designate a method in charge of this task with the following call:

```
...
    self.addStartMethod(startMethodName, self.myStartMethod)
```

startMethodName and *self.myStartMethod* give respectively the internal method name and the reference of the method that has to be added to the component in order to accomplish the required tasks at the begining of a simulation.

4.5.2 addSensitiveMethod() for states

Every time a state is entered (activated) or left (deactivate), designated sensitive methods are automatically called by PyCATSHOO. The following call designates these methods :

```
...
    self.stateX.addSensitiveMethod(methodName, self.methodReference, when)
```

methodName and *self.methpdReferences" give the internal name of the method and its reference. The designated method has to be added to the component in order to accomplish required tasks at the entering and/or leaving* self.stateX.
when* is an **optional** parameter which can take one of the three values; -1, 1 or 0 (default value).

```
* -1 means that the method is called when the state is left (deactivated)
*  1 means that the method is called when the state is entered (activated)
*  0 means that the method is called when the state is entered (activated)
   and when it is left (deactivated)
```

4.5.3 addSensitiveMethod() for automata

Every time an automaton changes its current state the designated sensitive methods are automatically called by PyCATSHOO. The following call designates these methods:

```
...
    self.automatonX.addSensitiveMethod(methodName, self.myMethod, when)
```

methodName and *self.myMethod* give respectively the internal name of the method and its reference. The designed method has added to the component in order to accomplish the required tasks when the automaton changes its state.
when is an **optional** parameter which can take one of the three values; -1, 1 or 0 (default value).

```
* -1 means that the method is called when the automaton enters a state whose
     index is lower than that the one of the state that has been left.
*  1 means that the method is called when the automaton enters a state which
     index is higher than that of the left state
*  0 means that the method is called every time the automaton changes
     its current state
```

4.5.4 addSensitiveMethod() for references

Every time a reference receives a new value, the designated sensitive methods are automatically called by PyCATSHOO. The following call designates these methods:

```
...
    self.referenceX.addSensitiveMethod(methodName, self.myMethod)
```

Here, *methodName* and *self.myMethod* give respectively unlike the internal name and the reference of the method to be called when *self.referenceX* receives a new value. *self.myMthod* has to be added to the component.

4.5.5 addSensitiveMethod() for transitions

Every time a transition is fired the designated sensitive methods are automatically called by PyCATSHOO. The following calls designate these methods:

```
...
    self.tranistionX.addSensitiveMethod(methodName, self.myMethod)
```

Here, *methodName* and *self.myMethod* give respectively unlike the internal name and the reference of the method to be called when the transition is fired. *self.myMethod* has to be added to the component.

4.6 PDMP Manager

The PDMP Manager manages the Piecewise Deterministic Markov Process common to the whole system. It is therefore in charge of solving ordinary differential equations (ODE) which govern the evolution over time of physical phenomena. It is also in charge of the interaction between physical phenomena and the discrete behavior of the system. This is achieved by providing, at every instant of the deterministic phase, the means that ensure the coherence between the shape and the parameters of the PDMP Flow -i.e. its system of ODE- and the current state of the system. Although the PDMP Manager is a single entity shared by the whole system component, its definition is distributed over all the system's components. It is therefore distributed over the different classes of such components. In this section we will see how the ingredients of this definition are created in every component class.

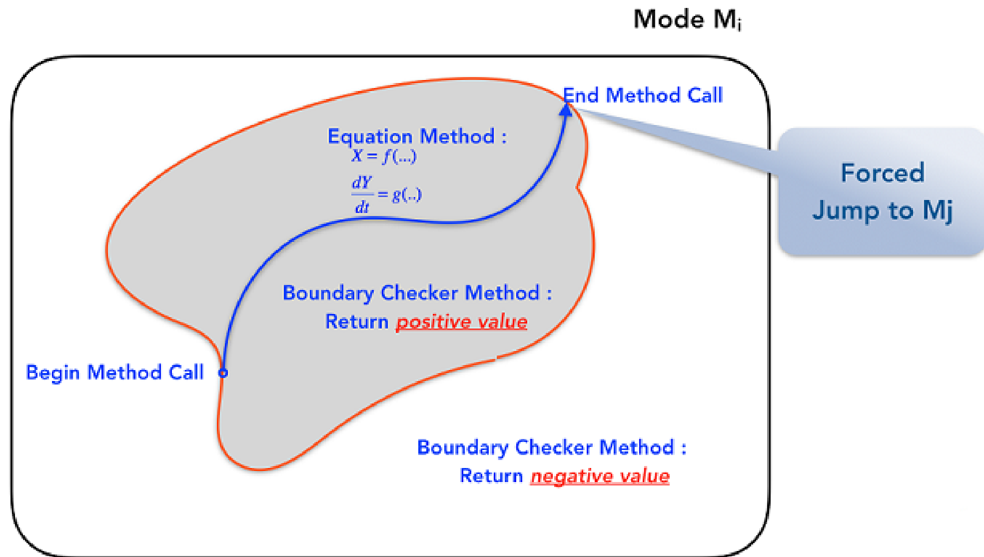


Figure 17

The PDMP ingredients in case of a trajectory finished by a forced jump

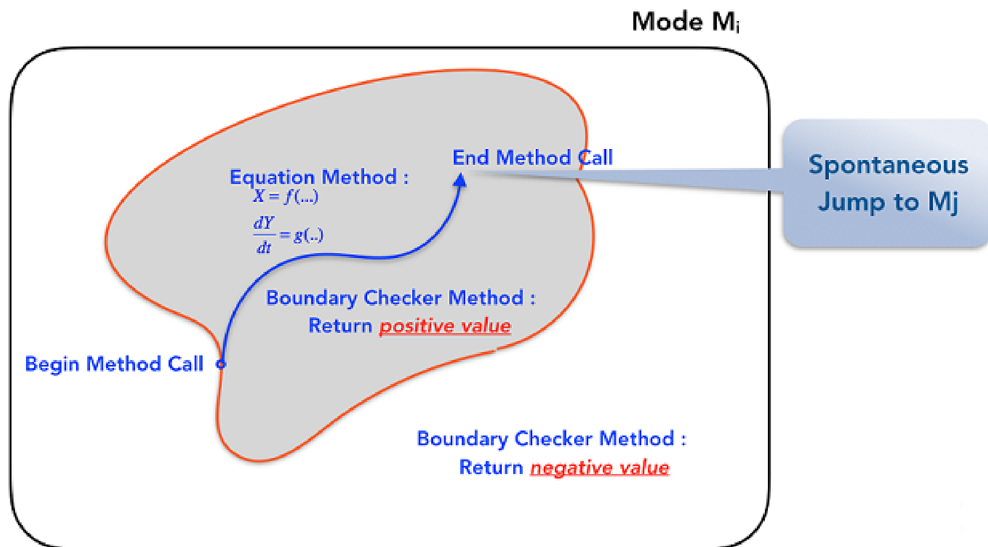


Figure 18

The PDMP ingredients in case of a trajectory finished by a spontaneous jump

4.6.1 addPDMPManager()

```
...
    pdmpManager = self.addPDMPManager(pdmpManagerName)
```

> *pdmpManagerName* is a string which gives the name of the PDMP Manager to create.
> Several different PDMP Managers can be created. But only one can be active at a given instant.
> Even if this call can be done inside several components with the same name, only one PDMP Manager will be created with that name.

A PDMP Manager manages the evolution over time of a subset of state variables of the system. This subset includes a subset of the component state variables. Hence, for every component we have to designate its managed state variables. This can be done in two ways according to the manner these variables are calculated:

4.6.2 addPDMPODEVariable(), addODEVariable()

```
...
    self.addPDMPODEVariable(pdmpManagerName, self.variableX)
    #or
    pdmpManager.addODEVariable(self.variableX)
```

> These calls inform the PDPM Manager which name is *pdmpManagerName* or which is referenced by the variable *pdmpManager*, that it is in charge of the variable *self.variableX* and that this variable has to be calculated thanks to a system of ODEs.

4.6.3 addPDMPExplicitVariable(), addExplicitVariable()

```
...
    self.addPDMPExplicitVariable(pdmpManagerName, self.variableY)
    #or
    pdmpManager.addExplicitVariable(self.variableY)
```

> These calls inform the PDPM Manager which name is *pdmpManagerName* or which is referenced by the variable *pdmpManager*, that it is in charge of the variable *self.variableY* and that this variable has to be calculated thanks to an explicit expression.

4.6.4 addPDMPEquationMethod(), addEquationMethod()

An ODE system consists in a set of first order differential equations. These equations as well as the explicit expressions must be declared inside a method which reference has to be given to the PDMP Manager with one of the following calls:

```
...
    self.addPDMPEquationMethod(pdmpManagerName,
                               equationMethodName,
```

```
                                    self.equationMethod,
                                    order)
#or
pdmpManager.addEquationMethod(equationMethodName,
                                  self,
                                  self.equationMethod,
                                  order)
```

These calls designate, for the PDMP manager which name is *pdmpManagerName* or which is referenced by the variable *pdmpManager*, the component method where the differential equations and explicit expressions are declared.

Where :

- *pdmpManagerName* is the name of the PDMP Manager.
- *equationMethodName* is a string which gives the name of that method.
- *self.equationMethod* is the reference of a component method designated as the PDMP Manager method. This reference is **optional**. If omitted, the designated class method name will be the one given by *equationMethodName*.
- *order* is an **optional** parameter which default value is 0.
  This parameter indicates the order in which this method is called. Indeed several methods may be used for a given PDMP Manager either inside a same component or in different components. In this rare case, the order of the calls to these methods may matter.

4.6.5 setDvdtODE()

The declaration of a first order differential equation is done, inside the equation method previously added to the PDMP Manager, as follows:

```
...
    self.variableX.setDvdtODE(aPythonExpression())
```

This call sets *aPythonExpression* as the derivative expression of *self.variableX*

Where :

- *aPythonExpression* is a Python expression involving any simple Python variables and/or any PyCATSHOO variables and references

4.6.6 setDValue

The declaration of an explicit equation is done, inside the equation method previously added to the PDMP Manager, as follows:

```
...
    self.variableY.setDValue(aPythonExpression)
```

This call sets *aPythonExpression* as the expression whose value is to be assigned to the variable *self.variableY*

Where :

- *aPythonExpression* is a Python expression involving any simple Python variables and/or any PyCATSHOO variables and references

Here we have an illustration of the Python implementation of a PDMP Manager equation method. This illustration is drawn from the heated tank test case. It gives the following expression of the failure as an explicit function of the temperature inside the tank.

$$\lambda = \lambda_0 \times \frac{b_1 \times e^{b_c \times (T-20)} + b_2 \times e^{b_d \times (20-T)}}{b_1 + b_2}$$

```
...
    def equationMethod(self):
      iFlow    = self.pi_inFlow.sumValue()
      oFlow    = self.pi_outFlow.sumValue()
      self.po_level.setDvdtODE((iFlow - oFlow) / self.po_area.value())
      lmbda = self.po_lambda0.dValue()*\
                     ( b1 * np.exp(+ bc * (self.pi_tankTemperature.value(0) - 20)) +
                       b2 * np.exp(- bd * (self.pi_tankTemperature.value(0) - 20))
                     ) / (b1 + b2)
      self.po_lambda.setDValue(lmbda)
```

4.6.7 addPDMPBoundaryCheckerMethod(), addBoundaryCheckerMethod()

The PDMP addresses a sequence of deterministic phases which may be interrupted by stochastic or deterministic discrete jumps. As mentioned above, a PDMP manager ensures the coherence between these deterministic phases and jumps. It is therefore in charge of scheduling the discrete jumps and in particular the deterministic ones. The latter must be executed when the boundary of the region where the current states are valid is reached. In order to accomplish this task, the PDMP manager needs to have means to detect the crossing of the boundary beyond which the current states and evolution become incompatible. This means relies mainly on the boundary checkers' methods defined as follows:

```
...
    self.addPDMPBoundaryCheckerMethod(pdmpManagerName,
                                      boundaryCheckerMethodName,
                                      self.boundaryCheckerMethod)
    #or
    pdmpManager.addBoundaryCheckerMethod(boundaryCheckerMethodName,
                                         self,
                                         self.boundaryCheckerMethod)
```

Where:

- *pdmpManagerName* is the name of the PDMP Manager
- *boundaryCheckerMethodName* is a string which gives the name of the method
- *self.boundaryCheckerMethod* is the reference of a component method designated as a boundary checker

The role of a boundary checker method consists in the accomplishment of a set of appropriate tests to determine if the current continuous state variables are still in a region coherent with the current discrete states. In this case the method must return a positive value otherwise, it must return a negative value.
In most cases the use of bounadry checker methods is not mandatory and can be replaced by the folowing methods.

4.6.8 addPDMPWatchedTransition()

Roughly speaking, the bounadriy checker methods help detecting crossing the boundaries of current states valid regions. But in general the condition of a boudary crossing is also the condition of a deterministic transition. It may be, for example, a transition between OPEN an CLOSE states of a valve which must close when the level in the vessel exeeds a particular threshold. In such a case, we dont need to implement any boudary checker method, we just have to ask PyCATSHOO to watch the transition by the following call:

```
...
    self.addPDMPWatchedTransition(pdmpManagerName, transition)
```

Where:

- *pdmpManagerName* is the name of the PDMP Manager
- *transition* is a reference to a determinitic transtion whose condition involves determenistic and continuous variables

5 System construction

Once a knowledge base is available i.e. when every class of component is modeled, it becomes quite easy to construct a system model which belongs to the class of systems targeted by such a knowledge base.
A system construction consists in creating its components by instantiation of the knowledge base classes. These components are then connected to each others via their message boxes according to the architecture of the system.
The components created must be gathered within a class derived from the PyCATSHOO class ***CSystem***. To create such a class we can proceed as follows:

```
...
    class MySystem(Pyc.CSystem):
        def __init__(self, systemName):
            Pyc.CSystem.__init__(self, systemName)
```

> This excerpt creates a Python class called *MySystem*. The constructor of this class requires a string which designates the external name of the system instance.

Within this class, in the constructor itself or in a dedicated method, the constituents of the system are created by instantiating the knowledge base classes according to the system composition:

```
...
    self.aHeater = Heater("MainHeater")
    self.room    = Room("Room")
```

> In this excerpt we create an instance of the ***Heater*** class. This instance will be known outside the system by its external name "MainHeater". An instance called "Room" of the class ***Room*** is also created.

Once the instances are created, we proceed to the connexion of their message boxes according to the architecture of the modeled system. Such connexions are created by one of the following calls:

```
...
    self.connect(self.aHeater, aHeaterMessageBoxName, self.room, aRoomMessageBoxName)
```

> In this call, we create a link between the message box called *aHeaterMessageBoxName* of the component *self.aHeater* and the message box called *aRoomMessageBoxName* of the component *self.room*.

```
...
    self.connect("MainHeater", aHeaterMessageBoxName, "Room", aRoomMessageBoxName)
```

> In this call, we create a link between the message box called *aHeaterMessageBoxName* of a component called "MainHeater" and the message box called *aRoomMessageBoxName* of a component called "Room".

6 Quantification

The quantification is done in 9 steps:

1. Setting the file of parameters and initial values
2. Call to the system construction method
3. Construction of prior simulation indicators
4. Setting the sequence filters
5. Setting the simulation parameters
6. Launching the simulation
7. Setting post-simulation indicators
8. Post-simulation sequences filtering and printing
9. Loading a file of results

6.1 Setting parameters' and initial values' files

6.1.1 File of parameters

A parameters file modifies the initial values of the variables and this modification takes effect just after the creation of the latter. This means that if in the parameter file we set the value of a variable which external name is "mu" to 10^{-5}, the result of the following sequence will be 10^{-5} even though if its defaults value to 0.1.

```
...
    self.po_mu = self.addVariable("mu", Pyc.TVarType.t_double, 0.1)
    print (self.po_mu.value())
```

This parameter file must be loaded before the creation of the system components.
The next excerpt gives the syntax of a parameter file:

```
...
<PY_PARAM>
    <VAR_P NAME="componentName.externalVariableName" INITV="alternativeInitialValue"/>
    <VAR_P NAME="#regularExpression" INITV="alternativeInitialValue"/>
</PY_PARAM>
```

The attribute **NAME** of the element **VAR_P** refers to a variable. The latter is identified by the *componentName* which is the name of the component to which it belongs and by *externalVariableName* which is the external name of the variable.
The value *alternativeInitialValue* is literally written. It will replace the one passed as default value to the variable creation method.
The regular expression "#regularExpression" may replace an explicit name of a variable in order to refer to several variables and to give them the same alternative initial value. The character **#** indicates that the following string is a regular expression.
As a reminder here we have some control characters used in regular expressions:

- . represents any character
- * means that the previous one may be repeated 0 or more times
- + means that the previous one may be repeated 1 or more times
- \ removes the control meaning from a control character so it will be used a it is

For instance "#Pump.*\.mu" matches any string which starts with "Pump" followed by any repetition of characters ".*" and then by a point "\." followed by the string "mu". This also means: any element which external name is "mu" of any component which external name begins by the string "Pump".
A Parameters' file may be load by the following instruction:

```
...
    system.loadParameters(fileName)
```

6.1.2 File of initial values

As well as the parameter file, the file of initial values is also an XML file which is loaded by the same instruction as above. But unlike the file of parameters, it must be loaded after the creation of all the components of the system. This means that the modification set by loading this become effective only just before the beginning of the simulation.
The file of initial values allows to:

- Modify the initial values of the variables
- Modify the initial states of the automata
- Set the simulation parameters
- Choose the variables, states and automata whose values, taken during the simulations, must be memorized
- Choose the transitions whose occurrences during the simulations are memorized
- Set the ODE solver parameters

This file can also be used to complete the composition and the architecture of the system. This functionality will not be developed in the current version of this document.
The next excerpt gives the file of initial values syntax:

```
...
    <PY_PARAM TRACE="0 or 1 or 2"
            NAME="nameOfParemetreSet"
            TMAX="1000"
            SEQ_NB="1000"
            RNG="yarn5"
            RGN_S="0"
            RNG_BS="50"
            RES_FILE="ResultFile.xml"
            RES_XML="0 or 1"
    >

            <TRACE_TR  NAME="explicitNameOrRegularExpression" LEVEL="0 or 1 or 2"/>
            <TR NAME="explicitNameOrRegularExpression" TRACE="0 or 1 or 2"/>

            <TRACE_VAR NAME="explicitNameOrRegularExpression" LEVEL="0 or 1 or 2"/>
            <VAR NAME="explicitNameOrRegularExpression" TRACE="0 or 1 or 2"/>

            <TRACE_ST  NAME="explicitNameOrRegularExpression" LEVEL="0 or 1"/>
            <TR NAME="explicitNameOrRegularExpression" TRACE="0 or 1"/>

            <TRACE_AUT NAME="explicitNameOrRegularExpression" LEVEL="0 or 1"/>
            <AUT NAME="explicitNameOrRegularExpression" TRACE="0 or 1"/>

            <VAR NAME="explicitNameOrRegularExpression" INITV="alernativeInitialValue"/>
            <ST  NAME="explicitNameOrRegularExpression" INITV="0 or 1"/>

            <MONIT_VAR NAME="explicitNameOrRegularExpression"/>
            <VAR NAME="explicitNameOrRegularExpression"  MONIT="0 or 1"/>

            <MONIT_ST NAME="explicitNameOrRegularExpression"/>
            <ST  NAME="explicitNameOrRegularExpression"  MONIT="0 or 1"/>

            <MONIT_AUT NAME="explicitNameOrRegularExpression"/>
            <AUT NAME="explicitNameOrRegularExpression"  MONIT="0 or 1"/>

            <MONIT_TR NAME="explicitNameOrRegularExpression"/>
            <TR  NAME="explicitNameOrRegularExpression"  MONIT="0 or 1"/>


            <ODE NAME="pdmpManager" DT="1" DTM="5" DTC="0.001" SCHEME="1"/>


        </PY_PARAM>
```

File header

The root element **PY_PARAM** of this XML file has the following attributes:

- ***TRACE***: This attribute can take one of the three following values:
  - 0 : ⇒ the number of sequences are not printed during the simulation
  - 1 : ⇒ the number of sequences are printed only if some elements are traced (see below)
  - 2 : ⇒ the number of sequences are printed during the simulation
- ***Name***: name of the parameter set given in this file
- ***TMAX***: duration of the system observation
- ***SEQ_NB***: number of simulations to be executed by PyCATSHOO
- ***RNG***: name of the random number generator to use. This attributes can take three different values:
  mt19937 Mersenne Twister 19937 generator
  KISS (Keep It Simple Stupid by G. Marsaglia)
  yarn5 (yet another random number generator) based on a multiple recursive generator with 5 feedback taps. This generator must be used when the simulations are distributed over several different cores.
- ***RGN_S***: gives the seed of Monte Carlo simulation. If set to 0 (default value), the seed is actually chosen randomly thanks to the computer internal clock.
- ***RNG_BS***: this attribute is only useful when ***RNG*** is **yarn5**. It gives the size of a generated block of numbers to skip between simulations when they are distributed over several cores.
- ***RES_FILE***: When given, this attribute asks PyCATSHOO to produce two results files in binary and XML format. Its value composes the names of these two files.
- ***RES_XML*** : this attribute may take 1 which means that the transitions are stored in the xml file rather than in the binary or 0 which means that the transitions are only stored in the binary file

Tracing instructions

In the elements **TRACE_XYZ** the attribute ***NAME*** gives a reference to one or several element to trace i.e. which values are displayed in the console during the simulation. This reference is given by *explicitNameOrRegularExpression* which can be "componentName.externalElementName" or "#regularExpression". In the latter, all the elements that have their own name and the name of their parent satisfying the regular expression will be displayed.
As for **TRACE_TR**, the attribute ***LEVEL*** can take three values :

- 0: ⇒ The transition is not traced.

- 1: ⇒ Only the instant when the transition is fired is displayed
- 2: ⇒ This also displays the instant when the waiting time before the transition is fired
  This trace consists in printing this waiting time, except that this waiting time might not be known when the rate of transition depends on the continuous state variables. In this case the "?" character is displayed next to the estimated waiting time.

As for **TRACE_VAR**, the attribute ***LEVEL*** can also take three values :

- 0: ⇒ the variables are not traced
- 1: ⇒ the values of the variable(s) are printed every time they change
- 2: ⇒ the values of the variable(s) are printed but only at the end of the deterministic phases

TRACE_ST and **TRACE_AUT** are about states (entering and leaving) and automata (changes of indexes)
Note that these two instructions are equivalent:

```
...
        <TRACE_XYZ  NAME="explicitNameOrRegulaExpression" LEVEL="0 or 1 or 2"/>
        <XYZ NAME="explicitNameOrRegulaExpression" TRACE="0 or 1 or 2"/>
```

Initialization instructions

- The following instruction selects one or several variables via *explicitNameOrRegularExpression* and gives them *alternativeInitialValue* which is an alternate value to the one given at the variable creation method.

```
...
<VAR NAME="explicitNameOrRegularExpression" INITV="alternativeInitialValue"/>
```

- The following instruction selects one or several states via *explicitNameOrRegurlaExpression* and set it (them) as the initial state(s) of its automaton (their automata) when the attribute **INIT** is equal to 1.

```
...
<ST NAME="explicitNameOrRegularExpression" INIT="0 or 1"/>
```

Monitoring instructions

The following instructions select one or several elements of type (Variable: VAR, State: ST, Automaton: AUT, Transition: TR) and, for further post processing analysis, ask PyCATSHOO to monitor i.e. to save their evolution over time during all the simulation. The elements are selected thanks to *explicitNameOrRegularExpression* which gives the explicit name of the elements or a pattern (regular expression).

```
...
<MONIT_[ VAR | ST | AUT | TR] NAME="explicitNameOrRegulaExpression"/>
```

As for the transitions, PyCATSHOO memorizes their name, nature and the instant when they had been fired.
Note that the two following instructions are equivalent:

```
...
<MONIT_[ VAR | ST | AUT | TR] NAME="explicitNameOrRegulaExpression"/>
<[ VAR | ST | AUT | TR] NAME="explicitNameOrRegulaExpression" MONIT="1"/>
```

PDMP Manager configuration

The configuration of the PDMP Manager can be done by the following instruction :

```
...
<ODE NAME="pdmpManager" DT="1" DTM="5" DTC="0.001" SCHEME="1"/> <!-- S1  -->
```

- ***NAME*** indicates the name of PDMP Manager to configure.
- ***DT*** gives the solver's time step.
- ***DTM*** gives the time step between two instants when the variables' values are memorized.
- ***DTC*** gives the time precision for the seeking algorithm of the boundary crossing of the state variables' regions.
- ***SCHEME*** gives the ***id*** of the ODE solver to use. PyCATSHOO has twelve solver schemes:

Id	ODE Solver scheme
1	euler
2	runge kutta 4
3	runge kutta cash karp 54
4	runge kutta dopri 5
5	runge kutta fehlberg 78
6	modified midpoint
7	controlled rk cash karp54
8	controlled rk dopri5
9	controlled rk fehlberg78
10	controlled bulirsch stoer
11	dense rk dopri 5
12	dense bulirsch stoer

6.2 Call to the system construction method

Here we proceed to a simple call the construction method which belongs to the class derived from ***CSystem*** dedicated to hold the system's composition and architecture.

6.3 Construction of prior simulation indicators

The construction of this kind of indicators must be done before the simulation is launched and after the creation of the system. At this moment we have a reference to this system. Let's denote *system* such a reference. The creation of these indicators can be done by one of the followings instructions:

```
...
        1. anIndicatorX = system.addIndicator("nameOfIndicator",
                                               "nameOfAComponent.nameOfElement",
                                               "natureOfElement")

        2. anIndicatorY = system.addIndicator("nameOfIndicator",
                                               "nameOfAComponent.nameOfElement",
                                               "natureOfElement",
                                               "operator",
                                               value)

        3. anIndicatorZ = system.addIndicator("nameOfIndicator",
                                               anIndicatorFunction)
```

Where :

- The indicator is designated by the string *nameOfIndicator* as its identifier. This identifier can help to retrieve the indicator in case of processing indicators reloaded from a result file produced by a previous simulation.
- The main element used to calculate the indicator is selected thanks to its external name *nameOfElement* and thanks to the external name *nameOfAComponent* of the component to which it belongs. This element can be a variable, a state or an automaton.
- The nature of the element is designated by the string *natureOfElement* which takes one of these three values: "VAR", "ST" or "AUT".
- We can give an operator *operator* and a value *value* which specify the operation to apply to the designated element to make up a boolean indicator. *operator* can take one of these values: "<", "<=", ">", ">=", "=" or "==". The two latter are equivalent.
- An indicator can also be equivalent to the result of a call to a pre-implemented function *anIndicatorFunction*.

In the instruction **1**, the function can be written as follows:

$$f(nameOfComponent.nameOfElement) = nameOfComponent.nameOfElement$$

In the instruction **2**, the function is an indicator function on *nameOfComponent.nameOfElement* that can be written as follows:

$$f(\text{nameOfComponent.nameOfElement}) = \mathbb{1}_{\{True\}}(\text{nameOfComponent.nameOfElement})$$

which is equal to:

$$\begin{cases} 1 & \text{if} \quad \text{nameOfComponent.nameOfElement } operator \text{ value} = \text{True} \\ 0 & \text{if} \quad \text{nameOfComponent.nameOfElement } operator \text{ value} = \text{False} \end{cases}$$

In the instruction **3**, the function does not necessarily depend on any particular element. It can be written as follows :

$$f() = anIndicatorFunction()$$

The indicators are computed progressively during the simulations at a set of instants that the user must specify by the following call:

```
...
    system.addInstants(0, system.tMax(), numberOfInstants)
```

This call specifies a vector of instants ranging from 0 to the observation time of the system and which length is equal to *numberOfInstants*. PyCATSHOO will compute the indicators at every instants in this vector.
After the creation of an indicator, its nature has to be defined. This is done by the following call:

```
...
    anIndicatorX.setRestitutions(indicatorNature)
```

indicatorNature can take an expression like: type_1 or type_2 or .... or type_n, where type_x specifies the nature of an indicator that PyCATSHOO will compute. type_x cane take one of the following values:

- **Pyc.TIndicatorType.mean_values**: Asks for the evolution over time of the indicator function mean.
  The mean evolution is then retrieved after the end of the simulation by the following call:

```
...
     vectorOfMeans = anIndicatorX.means()
```

- **Pyc.TIndicatorType.std_dev**: Asks for the evolution over time of the standard deviation.
  The standard deviation evolution is then retrieved after the end of the simulation by the following call:

```
...
     vectorOfStdDevs = anIndicatorX.stdDevs()
```

- **Pyc.TIndicatorType.all_values**: Asks for all the values taken by the indicator function for every simulation carried out at a given instant.
  These values are retrieved after the end of the simulation by the following call:

```
...
     vectorOfValues = anIndicatorX.values(indexOfAnInstant)
```

indexOfAnInstant gives an index in the vector of instants where the indicators have been calculated. This call retrieves the values taken by the indicator at the given instant at every simulation carried out by PyCASTHOO.

- **Pyc.TIndicatorType.distribution**: Asks for the distributions of the indicator.
  These distributions can be retrieved after the end of the simulations by the following calls:

```
...
     vectorOfDistributionValues = anIndicatorX.distributions(indexOfAnInstant)
```

indexOfAnInstant gives an index in the vector of instants where the indicators have been calculated.
This call retrieves the distribution values taken by the indicator at the given instant at every simulation carried out by PyCASTHOO.

- **Pyc.TIndicatorType.monitor_values** : This is not actually used to calculate specific indicators. It is always used in addition to others indicator types in order to ask PyCATSHOO to consider the indicators as monitored elements. All the values taken by the indicators at every instant and every simulation carried out will be memorized by PyCATSHOO.
- **Pyc.TIndicatorType.quantile_gt**: Asks for the evolution over time of the values which are greater than the quantile equal to *quantileValue* expressed as a percentage. This value must be set with the following instruction:

```
...
    anIndicatorX.setPctQuantileGtValue(quantileValue)
```

This indicator is retrieved by the following call:

```
...
     vectorOfQuantiles = anIndicatorX.quantilesGt()
```

- **Pyc.TIndicatorType.quantile_le**: Asks for the evolution over time of the values which are lesser than *quantileValue* expressed as a percentage. This value must be set with the following instruction:

```
...
    anIndicatorX.setPctQuantileLeValue(quantileValue)
```

This indicator is retrieved by the following call:

```
...
     vectorOfQuantiles = anIndicatorX.quantilesLe()
```

6.4 Setting sequence filters

During the simulations, PyCATSHOO keeps in memory all the monitored transitions and then builds a sequence for every simulation. But there are several situations where some of these sequences are not relevant for the analyses of the results. In this case it would be interesting to make PyCATSHOO forget the irrelevant sequences especially as this saves memory space. To do so, a sequence filtering can be asked before the beginning of the simulation by using the following calls:

```
...
    system.setSeqFilter(mySequenceFilterFunction)
    system.setKeepFilteredSeqForInd(aBoleanValue)
```

The first call sets a boolean function as a filter. So at the end of every simulation i.e. at the end of the construction of every sequence, PyCATSHOO calls the function *mySequenceFilterFunction* with the sequence as its unique parameter. If the function returns True, the sequence will be memorized. Otherwise it is forgotten by PyCATSHOO. The second call allows to choose if PyCATSHOO has to take account of the forgotten sequences when computing indicators (*aBooleanValue* = True) or not (*aBooleanValue* = False).
The following excerpt gives an example of *mySequenceFilterFunction* filter function:

```
...
    def mySequenceFilterFunction(sequence):

        system    = Pyc.CSystem.glSystem()

        anElement = system.getMonitoredElt("Tank.OverFlow", "ST")

        return sequence.value(anElement, system.tMax()) == 1
```

The first instruction in this function asks for the reference of the system.
The second instruction asks for the reference to an element of type State ("ST") which corresponds to the state called "Overflow" of the component called "Tank". The last instruction compares to 1 the value taken by this element at the final time of the simulation in the context of the sequence which reference is passed to the function. The result of this comparison is then returned by the function. This means that the function returns True if the state called Overflow of the component called "Tank" is reached at the end of the simulation. In this case the corresponding sequence is kept in memory otherwise it is dismissed.
An other convenient function to use in sequence filtering is given below:

```
...
    vectorOfInstants = Pyc.VectorDouble()
    vectorOfInstants.append(t1)
    vectorOfInstants.append(t2)
    ....

    sequence.realized(anElement,  predicate, vectorOfInstants)
```

This call applies to a sequence. It takes as first parameter an element as defined above. The second parameter is a predicate i.e. a boolean function which operates on the value taken by the element *anElement*. The third parameter is a vector of doubles which represents different instants.
The function *sequence.realized* returns a list of booleans, one for each instant in *vectorOfInstants*. The ith boolean is true if *predicate* returns True at least once at an instant lesser than the ith instant. Otherwise the boolean is False.

6.5 Setting simulation parameters

Before starting the simulation, several parameters must be set either using the initial value file (see 6.1.2) or by the calls explained below.

- Setting the Monte Carlo simulation seed:

```
...
    system.setRNGSeed(theSeed)
```

If *theSeed* is 0, the seed is actually randomly chosen thanks to the internal clock of the computer.

- Setting the random number generator:

```
...
    system.setRNG(theGeneratorName)
```

theGeneratorName can take one of the three values : "KISS", "yarn5" or "mt19937"

- Setting the number of sequences to simulate:

```
...
    system.setNbSeqToSim(nbSequencesToSimulate)
```

nbSequencesToSimulate gives the number of sequences to simulate.

- Setting the elements to monitor:
An element can refer to a variable, a state, an automaton or a transition. Each of these elements is identified by its external name and by the external name of the component to which it belongs. The python variable which holds the reference to an element can also be used as its identifier.
When an element, of type Automaton, State or Variable is monitored, PyCATSHOO keeps in memory all the values taken by the element during all the simulated sequences. These values are then stored in the result file if this storage is asked. These values can also be retrieved thanks to the PyCATSHOO class *CAnalyser* which will be presented below.
When a transition is monitored, it will be present in the simulated sequences with its occurrence time, otherwise it will be skipped.
The following instructions ask PyCATSHOO to monitor elements:

```
...
  system.monitorVariable(elementName, monitoringLevel)

  system.monitorState(elementName, monitoringLevel)

  system.monitorAutomaton(elementName, monitoringLevel)

  system.monitorTransition(elementName, monitoringLevel)
```

- *elementName* can be an explicit name which identifies a unique element or a regular expression which references a set of elements.

- *monitoringLevel* if 0 the element will not be monitored, if 1 the element will be monitored.

- Setting the name of the result file:

```
...
 system.setResultFileName(filename, storeSequenceInBinFile)
```

- *fileName* gives the name of the XML file where the results will be stored. PyCATSHOO will create an other file whose name is *fileName* suffixed by .bin. This file is a binary version of the XML file.

- *storeSequenceInBinFile* this is a boolean parameter. When True, the sequences will be stored in the binary file rather than in the XML file

6.6 Launching simulation

Once the system is build, its initialization is done and the simulation parameters are set, the simulation can be launched as follows :

```
...
    system.simulate()

    if system.MPIRank() > 0 :
        exit(0)
```

This sequence of instructions launches the simulation and at the end of the latter, asks MPI if the rank of the current process is greater than 0. If so the process exits.
The rank of a process can be greater than 0 if several parallel processes have been launched. In this case only the process with the rank 0 can continue the post processing of all the parallel simulations results. Indeed, at the end of simulation the partial results of the simulations carried out by the processes with non zero rank are automatically transfered to the process of rank 0. These partial results are also automatically gathered as if they were produced by only one process If no result file has been specified, the results can still be saved by the following statement.

```
...
    system.dumpResults(filename, storeSequenceInBinFile)
```

- *fileName* gives the name of the XML file where the results will be stored. PyCATSHOO will create an other file whose name is *fileName* suffixed by .bin. This file is a binary version of the XML file.
- *storeSequenceInBinFile* this is a boolean parameter. When True, the sequences are stored in the binary file rather than in the XML file

The post processing can then be done as explained in the next section.

6.7 Setting post-simulation indicators

At the end of the simulation, the post-processing consists either in retrieving the indicators set before launching the simulation or in constructing new indicators based on the monitored elements. In this section we will focus on the indicators based on the monitored elements which, as reminder, are not optimal in their use of memory.
The first thing to achieve is to create a system analyzer:

```
...
    analyzer = Pyc.CAnalyser(system)
```

The created analyzer holds all the values taken by the monitored elements during all the sequences. Thus, all the statistical calculations can be applied to these results. However, the values taken by the same monitored element during two different sequences are not given for the same instants in the two sequences. Thus prior to any statistical calculation we have to make a projection of these values over a common vector of instants. The following convenient function is dedicated to this aim:

```
...
    vectorOfInstants = stepsVector(system.tMax(), stepNumber):
```

This instruction builds a vector which length is *stepNumber* + 1 with instants from 0 to *system.tMax()*
Some common statistical calculations are already implemented in the *CAnalyzer* class. Their use is illustrated below :

```
...
1   pctQuantileGt      = 1.
2   pctQuantileLe      = 1.
3
4   vectorOfInstants = stepsVector(system.tMax(), 1000):
5
6   def indFct(x):
7       return x > 100
8
9   meanTemperature = analyzer.meanValues("room.temperature", vectorOfInstants)
10
11  quantileLe = analyzer.quantilesLe ("room.temperature", vectorOfInstants, pctQuantileLe)
12  quantileGt = analyzer.quantilesGt ("room.temperature", vectorOfInstants, pctQuantileGt)
13
14  cpdfValues1 = analyzer.realized  ("room.temperature", vectorOfInstants,  indFct)
15
16  cpdfValues2 = analyzer.realized  ("room.temperature", vectorOfInstants,  lambda x: x > 100)
17
18  cpdfValues3 = analyzer.realized  ("room.temperature", vectorOfInstants,  ">", 100)
```

The lines 1 and 2 give the values of the quantiles as percentages.
The line 4 builds a common vector of instants. It contains 1001 doubles going from 0 to *system.tMax()* by steps of *system.tMax()/1000* where *system.tMax()* is the specified observation time. All the following indicators are calculated at the instants given in this vector.
The line 9 computes the evolution of the mean of the element "temperature" of the component "Room".
The lines 11 and 12 compute the quantiles given by *quantileGt* and *quantileLe* percentages. These quantiles correspond to the evolution over time of the mean value of the Room temperature such that *pctQuantileGt* percent of temperature values at the same instants are greater than this value and *pctQuantileLe* percent of temperature values at the same instants are lesser than this value.

The lines 14, 16 and 18 are equivalent as they compute the cumulative probability distribution function of the following indicator function:

$$\mathbb{1}_{Room.temperature>100} = \begin{cases} 1 & \text{if} \quad \text{Room.temperature} \ > \ 100 \\ 0 & \text{if} \quad \text{Room.temperature} \ <= \ 100 \end{cases}$$

6.8 Post-simulation sequences' filtering and printing

When transitions are monitored, the sequences which pass the filters set before the simulation can be visualized in any html viewer. Such a visualization is possible since the sequences can be translated into html format by the following instruction :

```
...
    analyzer.printFilteredSeq(percentage, xmlFileName, "PySeq.xsl")
```

This instruction prompts the creation of an HTML presentation of the sequences which represent *percentage* percent in term of probability of the sequences produced by the simulations. The result of this transformation is on the one hand an XML file whose name is given by the parameter *xmlFileName* and on the other hand an HTML file which name is composed of the *xmlFileName* base name with the "html" extension.
At this stage it is still possible to proceed to another filtering of the sequences before the creation of their HTML presentation. This filtering can be done exactly in the same way as the filtering before the simulation previously presented.

6.9 Loading a results' file

To load a file of results, a system instance must be previously created. To do so this sequence of instructions can be used:

```
...
    system = Pyc.CSystem("theNameOfMySystem")
    system.loadResults("TheResultsFileNameOfAPreviousSimulation.xml")
```

The first instruction calls the constructor of an empty instance of the PyCATSHOO class *CSystem*. Therefore it is not required to recreate explicitly the original system. This creation will be automatically done by loading the file of results which holds all the informations about the corresponding system.
The second call loads an XML file where the results of a previous simulation have been stored.
After these instructions a new post-processing can be done in almost the same way as it is done after a call to *system.simulate()*. The only difference is related to the references to the indicators potentially created in the previous simulation. Here, theses references must be rebuild thanks to their names as follows:

```
...
    anIndicatorX = system.indicator("nameOfIndicator")
```

7 Implementation of a test case

The test case introduced below takes an example of a room with one or several heating devices aimed to maintain the room temperature between two thresholds: $maxTemperature = 20°C$ and $minTemperature = 15°C$. The room is assumed to be insufficiently thermally insulated. Thermal leaks are assumed to be proportional to the difference between the room temperature $temperature$ which initial value is $initialTemperature = 17°C$ and the outside temperature which is $outsideTemperature = 13°C$. The coefficient of proportionality is the leakage rate $leakageRate = 0.1W.°C^{-1}$.
The heating devices have their own thermostats and sensors. They start and stop the heaters according to the room temperature. They can be connected to each others so that they operate in a passive redundancy. This means that only one heater operates at a given instant and that a heater can be declared as slave of a master heater. The slave heater can start only when the master is out of order. It stops when the latter is repaired.
A heater capacity i.e. the nominal power it is able to supply is equal to $nominalPower = 5W$. The power it supplies at a given instant is equal to $power = nominalPower$ when it is operating and it is equal to $power = 0\mathbf{W}$ when it has been stopped.
We assume that the devices may fail at any instant with the failure rate $lambda = 0.01h^{-1}$ and can be repaired with a rate $mu = 0.1h^{-1}$.

Figure 19

Overview of the test case Heated room

The energy balance given below gives the ordinary differential equation which governs the evolution over time of the room temperature:

$$\frac{d(temperature)}{dt} = power - leakageRate \times (temperature - outsideTemperature)$$

The aim here is to give a generic model for the concepts and components of the system class described above in order to be able to address systems represented by the following figure 20:

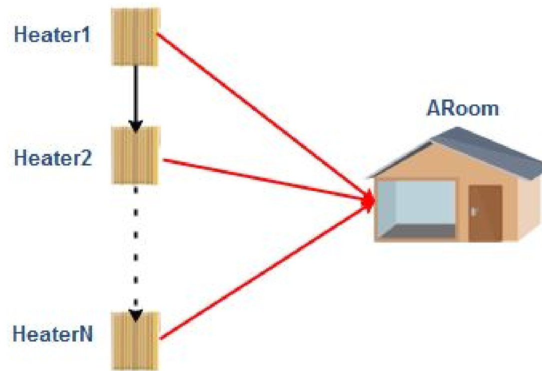


Figure 20

A generic heated room system

In figure 20, the black arrows stand for the passive redundancy. The origin of this arrow is connected to a master heater and its target is connected to a slave heater. The red arrows represent the thermal power supplied by the heaters to the room.
We will proceed progressively in the four following steps.

7.1 Step 1: Simple Heaters

In this step we will set aside the heated room and focus on the heaters with a simple operating mode. The heaters stop in case of a failure which may occur according to an exponential probability law. The heaters can be repaired equally according to an exponential probability law. They automatically start when repaired.
We must here create a class **Heater** where we will implement the following items:

1. The state variables
2. The behavior i.e. automata (states and transitions)

7.1.1 Heater state variables

In this step will retain for the heater the following state variables:

- **nominalPower**: the value of thermal power that the heater supplies when it is operating
- **power**: the value of thermal power supplied by the heater at a given instant
- **lambda**: its failure rate
- **mu**: its repair rate

7.1.2 The heater behavior

Defining the behavior of a component consists in defining its different states, the transitions between these states and the actions carried out by the component when they are entered or left.

Two automata define the heater's behavior. The first is the dysfunctional one. It has two states: *OK* and *KO*. The transitions between *OK* and *KO* states are governed by exponential probability laws.
The second automaton is a functional one. It has two states:
1. *ON*: if it is OK and (it has no master or (it has a master and there is a starting request))
2. *OFF*: if it is in not OK or (it has a master and there is no starting request)
The transitions between these two states are obviously deterministic and only their conditions *OFF2ONCondition* and *ON2OFFCondition* must be implemented in the Heater class.
When in *ON* state the thermal power supplied by the heater must be: $power = nominalPower$ and when on *OFF* state it must be $power = 0W$.
We have to ensure that this rule is satisfied at the beginning and at every instant of a simulation. To do so, we will implement, in the heater class, a sensitive method called *updateSuppliedPower* which will be in charge of this task. We will then ask PyCATSHOO to automatically call it at the beginning of a simulation and at every time the states of the functional automaton changes.

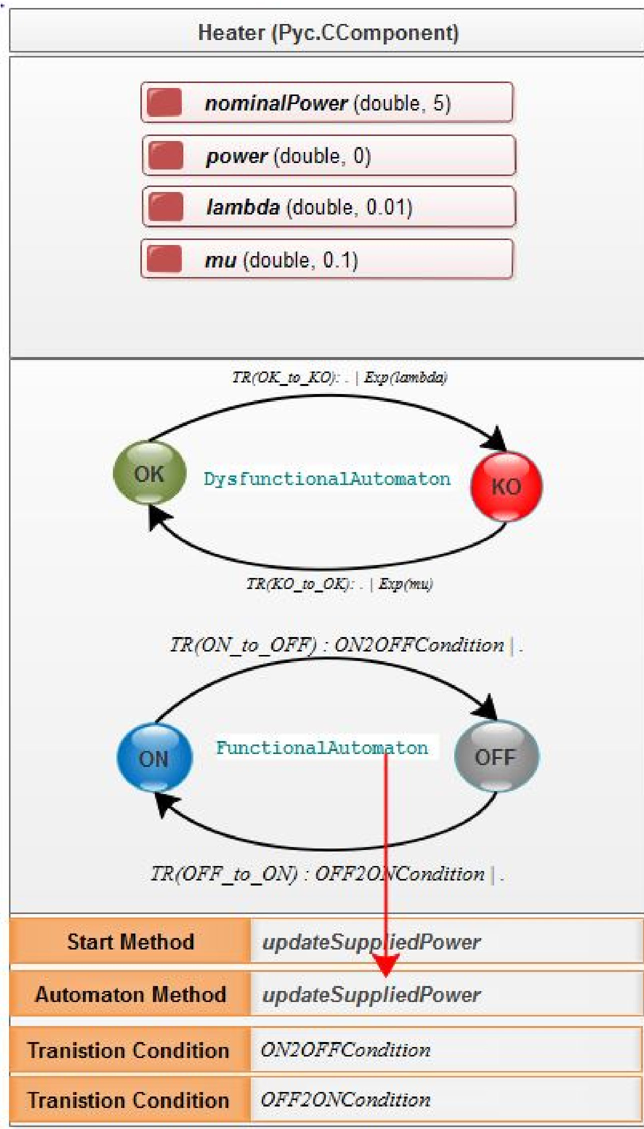


Figure 21

The Heater class for the step S1

```
In [2]: # coding: latin-1
        '''
        Created on 07.06.2017

        @author: Hassane CHRAIBI - hassane.chraibi@edf.fr

        '''

        from Util import *                                                          #S1


        #####################################################################################
        class Heater(Pyc.CComponent):                                               #S1

            def __init__(self, name):                                               #S1
                Pyc.CComponent.__init__(self, name)                                 #S1

                # The intrinsic state variables -------------------------------------------
                # The nominal power value supplied by the heater when on state ON
                # This value is held by a variable which type is double, with 5 as
                # initial value and known as "nominalPower"
                # Note : The initial values given here may be overridden by values
                # given by a parameter file loaded before
                # simulation
                self.po_nominalPower  = self.addVariable("nominalPower",
                                                        Pyc.TVarType.t_double, 5)   #S1

                # Holds the power supplied : 0 when OFF and po_nominalPower when ON
                self.po_power         = self.addVariable("power"       ,
                                                        Pyc.TVarType.t_double, 0)   #S1

                # The failure rate of the heater
                self.po_lambda = self.addVariable("lambda",
                                                 Pyc.TVarType.t_double, 0.01)       #S1

                # The repair rate of the heater
                self.po_mu     = self.addVariable("mu"    ,
                                                 Pyc.TVarType.t_double, 0.1)        #S1


                # The Automata declaration---------------------------------------------------
                # heater dysfunctional automaton
                # Creation of an automaton
                self.aDysfunctional = self.addAutomaton("DysfunctionalAutomaton")   #S1
                # Creates a state named OK and adds it to the automaton
                self.stateOK     = self.addState("DysfunctionalAutomaton", "OK", 1) #S1
                # Creates a state named KO and adds it to the automaton
                self.stateKO     = self.addState("DysfunctionalAutomaton", "KO", 0) #S1
```

```
# Sets OK state as the initial state
self.setInitState("OK")                                                     #S1

# Creation of the transition starting from the OK state
trans = self.stateOK.addTransition("OK_to_KO")                              #S1
# Setting of an exponential law as the probability law of the
# transition law to the failure state
# po_lambda is the parameter of this law
trans.setDistLaw(Pyc.TLawType.expo, self.po_lambda)                         #S1
# Definition of the target of the transition
trans.addTarget(self.stateKO, Pyc.TTransType.fault)                         #S1

# The definition of the repair transition
trans = self.stateKO.addTransition("KO_to_OK")                              #S1
# Setting of an exponential law as the probability law of
# the transition law to the fixed state
# po_mu is the parameter of this law
trans.setDistLaw(Pyc.TLawType.expo, self.po_mu)                             #S1
# Definition of the target of the transition
trans.addTarget(self.stateOK, Pyc.TTransType.rep)                           #S1



# heater functional automaton
# Creation of an automaton
self.aFunctional = self.addAutomaton("FunctionalAutomaton")                 #S1
# Creates a state named OK and adds it to the automaton
self.stateON  = self.addState("FunctionalAutomaton", "ON" , 1)              #S1
# Creates a state named KO and adds it to the automaton
self.stateOFF = self.addState("FunctionalAutomaton", "OFF", 0)              #S1
# Sets OK state as the initial state
self.setInitState("OFF")                                                    #S1

# Sets the method to be called when the functional automaton changes
# in order to update the value of the supplied heating power according to
# the states ON and OFF
self.aFunctional.addSensitiveMethod("updateSuppliedPower")                  #S1

# Creation of the transition starting from the OFF state
trans = self.stateOFF.addTransition("OFF_to_ON")                            #S1
# Sets condition of the transition
# The transition is fired if the FunctionalAutomation state OK is active
trans.setCondition(self.OFF2ONCondition)                                    #S1
# Definition the target of the transition
trans.addTarget(self.stateON, Pyc.TTransType.trans)                         #S1

# Creation of the transition starting from the ON state
```

```
        trans = self.stateON.addTransition("ON_to_OFF")                                  #S1
        # Sets condition of the transition
        # The transition is fired if the FunctionalAutomation state KO is active
        trans.setCondition(self.ON2OFFCondition)                                         #S1
        # Definition of the target of the transition
        trans.addTarget(self.stateOFF, Pyc.TTransType.trans)                             #S1



        # Makes updateSuppliedPower method to be called at the beginning of every
        # simulation in order to update the value of the supplied heating power
        # according to the initial states (ON or OFF)
        self.addStartMethod("UpdateSuppliedPower", self.updateSuppliedPower)           #S1


    # This method returns True if the heater hast to switch to ON
    def OFF2ONCondition(self):                                                           #S1
        # We switch to ON only if the heater is OK
        return self.stateOK.isActive()                                                   #S1


    # This method returns True if the heater has to switch to OFF
    def ON2OFFCondition(self):                                                            #S1
        # We switch to OFF when the heater is KO
        return self.stateKO.isActive()                                                    #S1


    # This method is called every time the functional automaton changes
    # and also at the beginning of every simulation
    def updateSuppliedPower(self):                                                        #S1
        if self.stateON.isActive():                                                       #S1
            # If ON, the heater delivers its nominal heating power
            self.po_power.setDValue(self.po_nominalPower.value())                         #S1
        else :                                                                            #S1
            # If OFF, the heater does not deliver any  heating power
            self.po_power.setDValue(0)                                                    #S1
```

7.1.3 Step 1 - testing

The test performed here consists in creating a simple system with one heater. The simulation of such a system will be devoted to the calculation of the expected thermal power provided by a heater.

```
In [3]: ####################################################################################
        class MySystem(Pyc.CSystem):                                                      #S1
            def __init__(self, name):                                                     #S1
                Pyc.CSystem.__init__(self, name)                                          #S1
```

```
                    # Instantiation of a Heater
                    self.aHeater = Heater("aHeater")                                      #S1
```

We also have to give the initial values and the configuration of the simulation. To do so, we will use an XML initial values file with the following items:

- Initial values for the **aHeater** component:
  - $nominalPower = 5W$
  - $lambda = 0.01h^{-1}$
  - $mu = 0.1h^{-1}$
  - OK and ON are the initial states
- Maximum duration of the system observation = 100 hours
- NUmber of simulations (sequences) = 1000000
- Pseudo random number generator = yarn5
- Seed for the Pseudo random number generator = 0
- level of trace = 0

We also ask, in this XML file, to monitor the heater power.

```
<PY_PARAM NAME="S1-Parameters" TMAX="100" SEQ_NB="1000000" RNG="yarn5"  RNG_S="0"><!-- S1 -->

    <TRACE_TR NAME="#.*" LEVEL="0"/>                                            <!-- S1 -->


    <VAR NAME="aHeater.nominalPower"  INITV="5"/>                               <!-- S1 -->

    <VAR NAME="aHeater.lambda"        INITV="0.01"/>                            <!-- S1 -->
    <VAR NAME="aHeater.mu"            INITV="0.1"/>                             <!-- S1 -->

    <ST NAME="aHeater.OK"             INIT="1"/>                                <!-- S1 -->
    <ST NAME="aHeater.ON"             INIT="1"/>                                <!-- S1 -->

    <MONIT_VAR NAME="aHeater.power"/>                                           <!-- S1 -->

</PY_PARAM>                                                                     <!-- S1 -->
```

Here is a commented Python main program which launches the simulation and displays the graph of the evolution over time of the thermal power supplied by the heater.

In [4]:
```
########################################################################################
if __name__ == '__main__':                                                          #S1

    try :
        # An instance of the system is constructed (Only one system can be
        # constructed in a session)
        system = MySystem("S1")                                                     #S1

        # The simulation parameters and initial values are loaded from an XML file
        system.loadParameters("HeatedRoom_S1.xml")                                  #S1

        # Launches the simulation
        beginTime = time.time()                                                     #S1
        system.simulate()                                                           #S1

        # In case of parallel simulation we end all the launched instances
        # except the main one which deals with post processing
        if system.MPIRank() > 0:                                                    #S1
            exit(0)                                                                 #S1

        endTime = time.time()                                                       #S1
        print ("Simulation time = %.2f s"% (endTime - beginTime))                   #S1


        # We create an anlyzer which holds the values of all the monitored elements
        analyser = Pyc.CAnalyser(system)                                            #S1

        # This vector will hold the instants where the values of the indicators
        # will be calculated
        vectorOfInstants = stepsVector(system.tMax(), 500)                          #S1

        # Calculates the mean values of the supplied power
        meanValues = analyser.meanValues("aHeater.power", vectorOfInstants)         #S1


        # This is a call to a convenient function which draws the curve of the
        # indicator evolution over time
        plot(vectorOfInstants, True, True,                                          #S1
             (
                 (("Simulation of %d Sequences") % system.nbSequences(),
                  "Time",
                  "Expected supplied power",
                  ((meanValues, 'r', "aHeater.power"),)),
             )
            )


    except Exception as e:                                                          #S1
```

```
            print(type(e),e)                                                          #S1
            Pyc.printMessages()                                                       #S1
```

```
Simulation time = 10.47 s
```

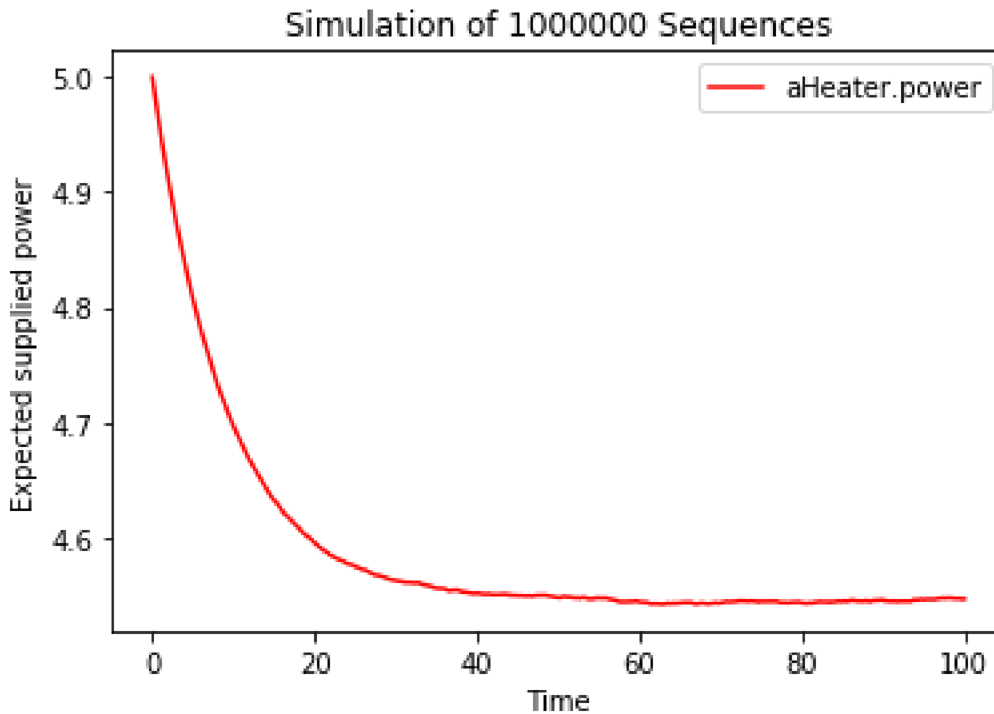


Figure 22

The supplied power expected from the Heater in the step S1

The asymptote of the curve produced above indicates that the expected supplied thermal power is about $4.54W$. This value is coherent with the values given by the theoritical expression :

$$\frac{\mu}{\lambda + \mu} \times nominalPower = 5 \times \frac{0.1}{0.1 + 0.01} = 4.5454W$$

7.2 Step 2: Heaters with master / slave operating

In this step, we aim to introduce a master / salve operating for the heaters which has to ensure the following properties:

1. Every heater can be a master or / and a slave for another heater.
2. When it is a master, a heater must transmit its KO state (True if **KO** is active and False otherwise) to the rest of the system through a message box that we will call **"MB-toSlave"** in order that the receiver considers it as a start or stop request.
3. When it is a slave, a heater must be ready to receive boolean values (True or False) through a message box we will call **"MB-toMaster"**. Such a heater must consider that a reception of at least one True value is equivalent to a start request. Otherwise the heater must stop.
4. If a heater has no master this means that it has a permanent request to start.

To be able to satisfy these requirements we have to:

a. Add a reference called ***startingRequest*** to **Heater** class.

b. Add two message boxes to a the **Heater** class : **"MB-toSlave"** and **"MB-toMaster"**

c. Export the **KO** state through the message box **"MB-toSlave"**
d. Import the values into ***startingRequest*** via **"MB-toMaster"**
e. Modify the *OFF2ONCondition* and *ON2OFFCondition* methods so that :

   A Heater passes from **OFF** state to **ON** state if the heater is **OK** and, it either has no master or there is at least one True value in ***startingRequest***. As a reminder, the latter condition is equivalent to an ***OR*** operator over all the values held by ***startingRequest***.
   A Heater passes from **ON** state to **OFF** state if the heater is **KO** or, if it has a master and there is no True value in ***startingRequest***.

7.2.1 Heater reference

We then have to add a reference with the name ***startingRequest***

7.2.2 The heater behavior

The only differences here are the conditions required to go from **ON** to **OFF** and from **OFF** to **ON**.

7.2.3 The heaters communication with the rest of the system

We then have to create two message boxes: **"MB-toSlave"** and **"MB-toMaster"**. Both will have a port that we will call **"request"**.
The port of **"MB-toSlave"** is an outgoing one. It is in charge of communicating the state **KO** which means that the transmitted value will be equal to **True** when the state **KO** is active and **False** otherwise.
The port of **"MB-toMaster"** is an incoming one. It is in charge of receiving booleans which semantic is a starting order when one of them is **True**.

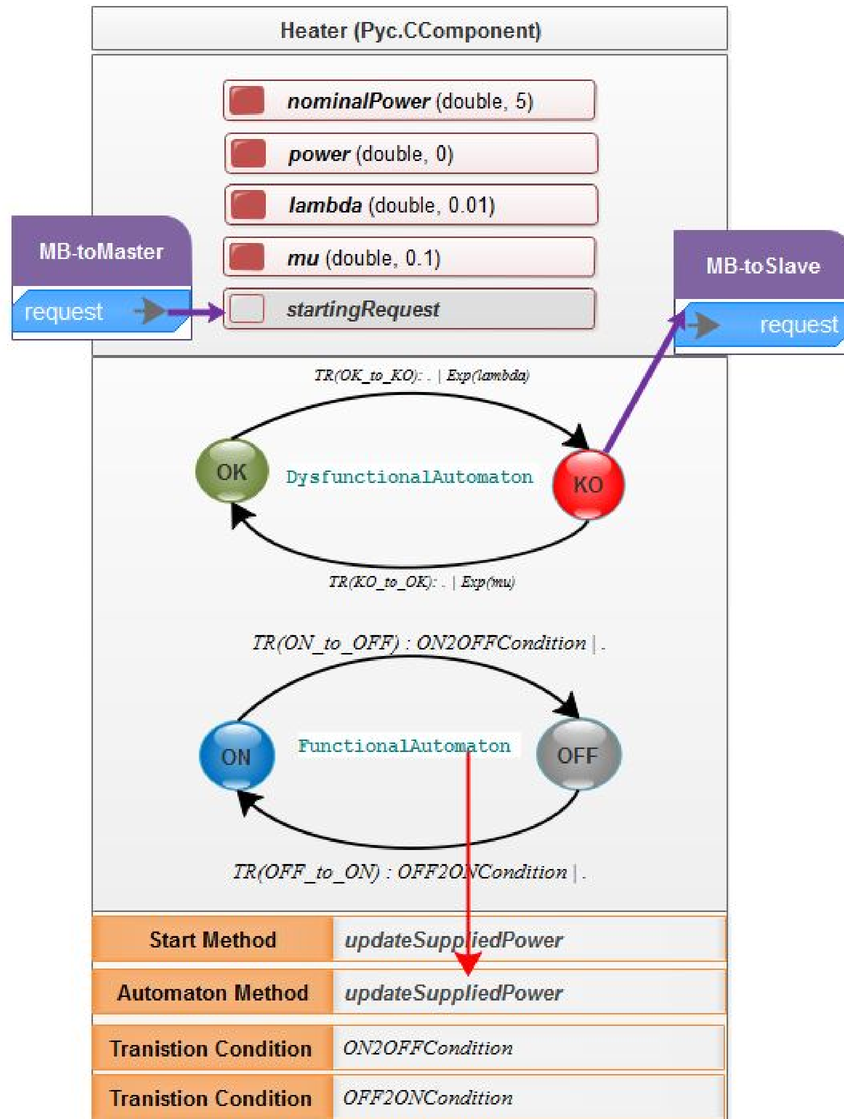


Figure 23

The Heater class for the step S2

```
In [1]: # coding: latin-1
        '''
        Created on 07.06.2017

        @author: Hassane CHRAIBI - hassane.chraibi@edf.fr

        '''

        from Util import *                                                                   #S1


        ##########################################################################################
        class Heater(Pyc.CComponent):                                                        #S1

            def __init__(self, name):                                                        #S1
                Pyc.CComponent.__init__(self, name)                                          #S1

                # The intrinsic state variables ---------------------------------------------
                # The nominal power value supplied by the heater when on state ON
                # This value is held by a variable which type is double, with 5 as
                # initial value and known as "nominalPower"
                # Note : The initial values given here may be overridden by values
                # given by a parameter file loaded before
                # simulation
                self.po_nominalPower  = self.addVariable("nominalPower",
                                                    Pyc.TVarType.t_double, 5)           #S1

                # Holds the power supplied : 0 when OFF and po_nominalPower when ON
                self.po_power         = self.addVariable("power"       ,
                                                    Pyc.TVarType.t_double, 0)           #S1

                # The failure rate of the heater
                self.po_lambda = self.addVariable("lambda",
                                             Pyc.TVarType.t_double, 0.01)               #S1

                # The repair rate of the heater
                self.po_mu     = self.addVariable("mu"    ,
                                             Pyc.TVarType.t_double, 0.1)                #S1

                # The references to external information ------------------------------------
                # This is a reference which holds the received requests that make the heater
                # starting when there is a True value
                self.pi_startingRequest = self.addReference("startingRequest")          #S2


                # The automata declaration--------------------------------------------------
                # heater dysfunctional automaton
                # Creation of an automaton
```

```python
self.aDysfunctional = self.addAutomaton("DysfunctionalAutomaton")           #S1
# Creates a state named OK and adds it to the automaton
self.stateOK     = self.addState("DysfunctionalAutomaton", "OK", 1)        #S1
# Creates a state named KO and adds it to the automaton
self.stateKO     = self.addState("DysfunctionalAutomaton", "KO", 0)        #S1

# Sets OK state as the initial state
self.setInitState("OK")                                                    #S1

# Creation of the transition starting from the OK state
trans = self.stateOK.addTransition("OK_to_KO")                             #S1
# Setting of an exponential law as the probability law of the
# transition law to the failure state
# po_lambda is the parameter of this law
trans.setDistLaw(Pyc.TLawType.expo, self.po_lambda)                        #S1
# Define the target of the transition
trans.addTarget(self.stateKO, Pyc.TTransType.fault)                        #S1

# The definition of the repair transition
trans = self.stateKO.addTransition("KO_to_OK")                             #S1
# Setting of an exponential law as the probability law of
# the transition law to the fixed state
# po_mu is the parameter of this law
trans.setDistLaw(Pyc.TLawType.expo, self.po_mu)                            #S1
# Define the target of the transition
trans.addTarget(self.stateOK, Pyc.TTransType.rep)                          #S1



# heater functional automaton
# Creation of an automaton
self.aFunctional = self.addAutomaton("FunctionalAutomaton")                #S1
# Creates a state named OK and adds it to the automaton
self.stateON  = self.addState("FunctionalAutomaton", "ON" , 1)             #S1
# Creates a state named KO and adds it to the automaton
self.stateOFF = self.addState("FunctionalAutomaton", "OFF", 0)             #S1
# Sets OK state as the initial state
self.setInitState("OFF")                                                   #S1

# Sets the method to be called when the functional automaton changes
# in order to update the value of the supplied heating power according to
# the states ON and OFF
self.aFunctional.addSensitiveMethod("updateSuppliedPower")                 #S1

# Creation of the transition starting from the OFF state
trans = self.stateOFF.addTransition("OFF_to_ON")                           #S1
# Sets the condition of the transition as a boolean function
trans.setCondition(self.OFF2ONCondition)                                   #S1
```

```
        # Define the target of the transition
        trans.addTarget(self.stateON, Pyc.TTransType.trans)                              #S1

        # Creation of the transition starting from the ON state
        trans = self.stateON.addTransition("ON_to_OFF")                                  #S1
        # Sets the condition of the transition as a boolean function
        trans.setCondition(self.ON2OFFCondition)                                         #S1
        # Define the target of the transition
        trans.addTarget(self.stateOFF, Pyc.TTransType.trans)                             #S1



        # Message boxes -----------------------------------------------------------------
        # This message box will link to the master heaters
        messageBox = self.addMessageBox("MB-toMaster")                                   #S2
        # Transmits to self.pi_startingRequest the state KO of the master heater
        # in order to provoke starting when True
        messageBox.addImport(self.pi_startingRequest, "request")                         #S2

        # This message box will link to the  slave heater
        messageBox = self.addMessageBox("MB-toSlave")                                    #S2
        # Transmits state KO to the slave heater in order to make it start
        # when state is KO
        messageBox.addExport(self.stateKO, "request")                                    #S2


        # Makes updateSuppliedPower method to be called at the beginning of every
        # simulation in order to update the value of the supplied heating power
        # according the initial states (ON or OFF)
        self.addStartMethod("updateSuppliedPower", self.updateSuppliedPower)            #S1


    # This method returns True if the heater hast to switch to ON
    def OFF2ONCondition(self):                                                           #S1
        # We don't switch to ON if the heater is not OK
        if not self.stateOK.isActive():                                                  #S2
            return False                                                                 #S2

        # We don't switch to ON if there is a Master and there is no request
        if not self.pi_startingRequest.orValue(True) :                                   #S2
            return False                                                                 #S2

        return True                                                                      #S2


    # This method returns True if the heater has to switch to OFF
    def ON2OFFCondition(self):                                                           #S1
        # We switch to OFF if the heater is not OK
```

```
            if not self.stateOK.isActive():                                         #S2
                return True                                                         #S2

            # We switch to OFF if there is a Master and there is no request
            if not self.pi_startingRequest.orValue(True) :                          #S2
                return True                                                         #S2

            return False                                                            #S2

        # This method is called every time the functional automaton changes
        # and also at the beginning of every simulation
        def updateSuppliedPower(self):                                              #S1
            if self.stateON.isActive():                                             #S1
                # If ON, the heater delivers its nominal heating power
                self.po_power.setDValue(self.po_nominalPower.value())               #S1
            else :                                                                  #S1
                # If OFF, the heater does not deliver any  heating power
                self.po_power.setDValue(0)                                          #S1
```

7.2.4 Step 2 - testing

The test performed here consists in creating two heaters. The first one will play the role of a master and the second will be considered as a slave. The simulation of such a system will be devoted to the calculation of the expected thermal power provided by the two heaters. And we will verify that :

$$aMasterHeater.power = 5 - aSlaveHeater.power$$

```
In [2]: ###########################################################################################
        class MySystem(Pyc.CSystem):                                                #S1
            def __init__(self, name):                                               #S1
                Pyc.CSystem.__init__(self, name)                                    #S1

                # Instantiation of a Master and Slave Heater
                self.aMasterHeater = Heater("aMasterHeater")                        #S2
                self.aSlaveHeater  = Heater("aSlaveHeater")                         #S2

                # connecting heaters slave and master
                self.connect("aMasterHeater", "MB-toSlave", "aSlaveHeater", "MB-toMaster") #S2
```

We also have to give the initial values and the configuration of the simulation. To do so we will use an XML parameters' file with the following items:
We will give the same initial values to the two heaters except that for the slave, the **OFF** will be the initial state of the functional automaton while the **ON** will still be the initial state for the master.

```
<PY_PARAM NAME="S2-Parameters" TMAX="100" SEQ_NB="100000" RNG="yarn5"  RNG_S="0"><!-- S2 -->
```

```
    <TRACE_TR NAME="#.*" LEVEL="0"/>                                        <!-- S1 -->


    <VAR NAME="aMasterHeater.nominalPower"  INITV="5"/>                     <!-- S2 -->
    <VAR NAME="aSlaveHeater.nominalPower"   INITV="5"/>                     <!-- S2-->

    <VAR NAME="aMasterHeater.lambda"        INITV="0.01"/>                  <!-- S2 -->
    <VAR NAME="aMasterHeater.mu"            INITV="0.1"/>                   <!-- S2 -->

    <VAR NAME="aSlaveHeater.lambda"         INITV="0.01"/>                  <!-- S2 -->
    <VAR NAME="aSlaveHeater.mu"             INITV="0.1"/>                   <!-- S2 -->

    <ST NAME="aMasterHeater.OK"             INIT="1"/>                      <!-- S2 -->
    <ST NAME="aMasterHeater.ON"             INIT="1"/>                      <!-- S2 -->

    <ST NAME="aSlaveHeater.OK"              INIT="1"/>                      <!-- S2 -->
    <ST NAME="aSlaveHeater.OFF"             INIT="1"/>                      <!-- S2 -->

    <MONIT_VAR NAME="aMasterHeater.power"/>                                 <!-- S2 -->
    <MONIT_VAR NAME="aSlaveHeater.power"/>                                  <!-- S2 -->

</PY_PARAM>                                                                 <!-- S1 -->
```

Hereafter a commented Python main program which launches the simulation and draws the evolution over time curve of the thermal power supplied by the heater.

```
In [3]: ####################################################################################
        if __name__ == '__main__':                                                  #S1

            try :
                # An instance of the system is constructed (Only one system can be
                # constructed in a session)
                system = MySystem("S2")                                             #S1

                # The simulation parameters and initial values are loaded from an XML file
                system.loadParameters("HeatedRoom_S2.xml")                          #S1

                # Lanches the simulation
                beginTime = time.time()                                             #S1
                system.simulate()                                                   #S1

                # In case of parallel simulation we end all the launched instances
                # except the main one which deals with post processing
                if system.MPIRank() > 0:                                            #S1
                    exit(0)                                                         #S1

                endTime = time.time()                                               #S1
                print ("Simulation time = %.2f s"% (endTime - beginTime))           #S1
```

```
        # We create an anlyzer which holds the values of all the monitored elements
        analyser = Pyc.CAnalyser(system)                                                      #S1

        # This vector will hold the instants where the values of the indicators
        # will be calculated
        vectorOfInstants = stepsVector(system.tMax(), 500)                                    #S1

        # Calculate the mean values of the supplied power
        mMeanValues = analyser.meanValues("aMasterHeater.power", vectorOfInstants) #S2
        sMeanValues = analyser.meanValues("aSlaveHeater.power", vectorOfInstants)  #S2

        # This is a call to a convenient function which draws the curves of the
        # indicators' evolution over time
        plot(vectorOfInstants, True, True,                                                     #S2
             (
                 (("Simulation of %d Sequences") % system.nbSequences(),
                  "Time",
                  "Expected supplied power",
                  ((mMeanValues, 'b', "aMasterHeater.power"),)),
                 (("Simulation of %d Sequences") % system.nbSequences(),
                  "Time",
                  "Expected supplied power",
                  ((sMeanValues, 'r', "aSlaveHeater.power"),)),
             )
            )


    except Exception as e:                                                                     #S1
        print(type(e),e)                                                                       #S1
        Pyc.printMessages()                                                                    #S1
```

```
Simulation time = 2.44 s
```

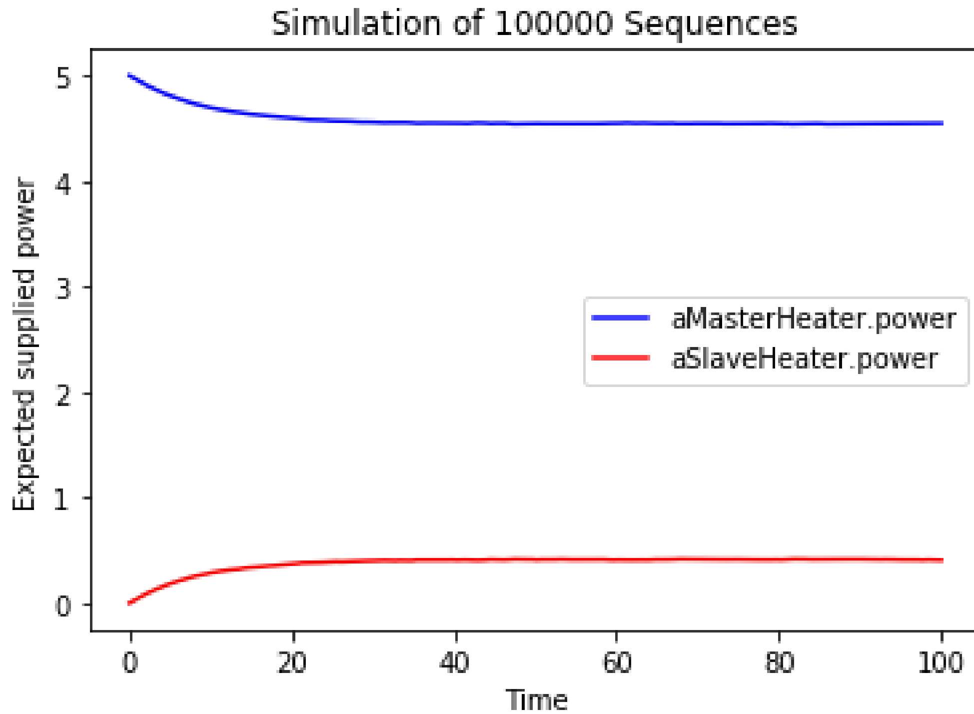


Figure 24

The Heaters expected supplied power for the step S2

The asymptotes of the two curves produced above indicate that the expected supplied thermal power of the two heaters satisfy the following constraints :

$$aMasterHeater.power = 5 - aSlaveHeater.power$$

7.3 Step 3: Introduction of the heated room

In this step we will focus on the room to be heated by the two heaters introduced above. As for the heaters a Room class will be created with the following informations.

7.3.1 Room state variables

The room state variables are :

- **leakageRate**: the rate of the thermal power leaks through the room walls
- **temperature**: the room temperature which evolves over time

- **initialTemperature**: a constant which gives the room temperature at the beginning of the simulations
- **outsideTemperature**: a constant which gives the outside temperature

7.3.2 Room references

The power supplied by the heater is received by the room class into a reference called: ***power***.

7.3.3 The room behavior

There will be no automata in the room class. The room behavior is limited to a deterministic physical phenomenon which governs the temperature evolution over time. Beyond this we have only to ensure that at the beginning of every simulation the state variable **temperature** takes the value of the variable **initialTemperature**. A method in charge of this task has to be added to the **Room** class. We will call it **"start"**.

7.3.4 The room communication with the rest of the system

A room has to communicate with the heaters which supply it with the thermal power. Then, there will be a message box in the **Room** class that we will call **MB-toHeaters**. This message box receives the values of the thermal power provided by all the connected heaters and stores these values in the reference ***power*** of the **Room** class. As mentioned above, the heaters hold a servo control of their operation according to the temperature of the room. This temperature should be collected thanks to an internal temperature sensor. A simplified model of that consists here in the **MB-toHeaters**, a room message box, which sends the values of the temperature to all connected heaters. We then have to create an outgoing channel in this message box and connect it to the room state variable **temperature**.

7.3.5 The room PDMP Manager part

As said before, the physical phenomena is represented by the following ordinary differential equation :

$$\frac{d(temperature)}{dt} = \sum_i power_i - leakageRate \times (temperature - outsideTemperature)$$

This means that there will be a PDMP Manager in the system and that the **Room** part of this PDMP manages the state variable **temperature**. A method that we will call ***pdmpMethod*** has to be defined in the room class. It has to implement the ODE which governs the temperature evolution over time.

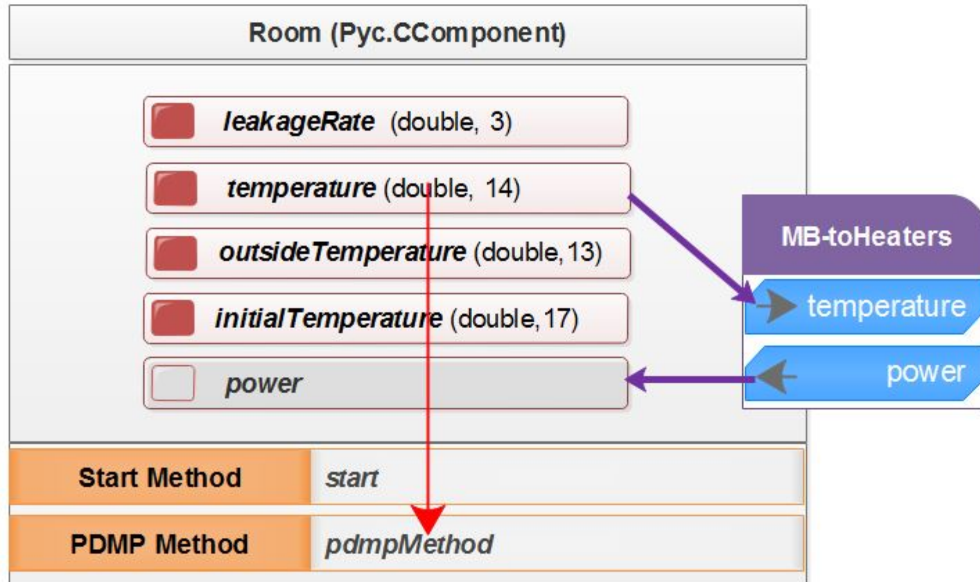


Figure 25

The Room class for the step S3

```
In [1]: # coding: latin-1
        '''
        Created on 07.06.2017

        @author: Hassane CHRAIBI - hassane.chraibi@edf.fr

        '''

        from Util import *                                                                    #S1

        #####################################################################################
        class Room(Pyc.CComponent):
            def __init__(self, name):
                Pyc.CComponent.__init__(self, name)

                # The intrinsic state variables ---------------------------------------------------
                # The heat leakage rate in the room
                self.po_leakageRate = self.addVariable("leakageRate",
                                                       Pyc.TVarType.t_double, 3.0)          #S3

                # The temperature of the room
                self.po_temperature = self.addVariable("temperature",
                                                       Pyc.TVarType.t_double, 17)           #S3
                # The temperature of the outside
                self.po_outsideTemperature = self.addVariable("outsideTemperature",
                                                              Pyc.TVarType.t_double, 13)    #S3
                # The initial temperature of the room
                self.po_initialTemperature = self.addVariable("initialTemperature",
                                                              Pyc.TVarType.t_double, 17)    #S3

                # The references to external information -------------------------------------
                # This value will hold the value(s) of power supplied by the heater(s)
                self.pi_power = self.addReference("power")                                  #S3

                # Message boxes ---------------------------------------------------------------
                # This message box will link to the heater(s)
                messageBox = self.addMessageBox("MB-toHeaters")                             #S3
                # From the heaters, the values of the power they supply is received
                messageBox.addImport(self.pi_power, "power")                                #S3
                # To the heater, the value of the current temperature will be sent
                messageBox.addExport(self.po_temperature, "temperature")                    #S3

                # The PDMP Manager ------------------------------------------------------------
                pdmpManager = self.addPDMPManager("PDMP-Manager")                           #S3
                pdmpManager.addEquationMethod("pdmpMethod", self)                           #S3
                pdmpManager.addODEVariable   (self.po_temperature)                          #S3

                # Makes start method to be called at the beginning of every simulation
```

```
        self.addStartMethod("start", self.start)                                  #S3

    # Start method makes the currentTemperature to be equal to the
    # initialTemperature at the beginning of the simulation
    def start(self):                                                              #S3
        self.po_temperature.setDValue(self.po_initialTemperature.dValue())        #S3

    # The pdmpMethode gives the derivative of the room temperature
    # The derivative expression is deduced from the energy
    # balance in the room
    def pdmpMethod(self):                                                         #S3
        self.po_temperature.setDvdtODE \
            (                                                                     #S3
              self.pi_power.sumValue(0) -                                         #S3
              self.po_leakageRate.value() *                                       #S3
              (self.po_temperature.value() - self.po_outsideTemperature.value())  #S3
            )                                                                     #S3
```

Now that the room is modeled, we have to update the Heater class in order to be able to:

- send the value of the supplied thermal power.
- simulate the existence of an internal thermostat with the maximum value that the room temperature must not exceed and the minimum value that the room temperature must not exceed.
- receive the temperature of the room so that the heater can compare it to the programmed minimum and maximum values.

Thus, the adaptation of the heater class consists in :

- adding two state variables ***minTemperature*** and ***maxTemperature*** which hold the programmed values of the internal thermostat.
- adding a reference ***roomTemperature*** which holds the received value of the room temperature.
- adding a message box **""MB-toRoom"** dedicated to send the value of the thermal power supplied and to receive the room temperature.
- updating the *OFF2ONCondition* method such that the condition of starting will be that the temperature of the room is lesser than ***minTemperature***.
- updating the *ON2OFFCondition* method such that the condition of stopping will be that the temperature of the room is greater than ***maxTemperature***.
- adding ***ONBoundaryChecker***, a boundary checker associated with **ON** state which ensures that $temperature > maxTemperature$ is a forbidden region when the heater is **ON**.
- adding ***OFFboundaryChecker***, a boundary checker associated with **OFF** state which ensures that $temperature < minTemperature$ is a forbidden region when the heater is **OFF** and **KO**.

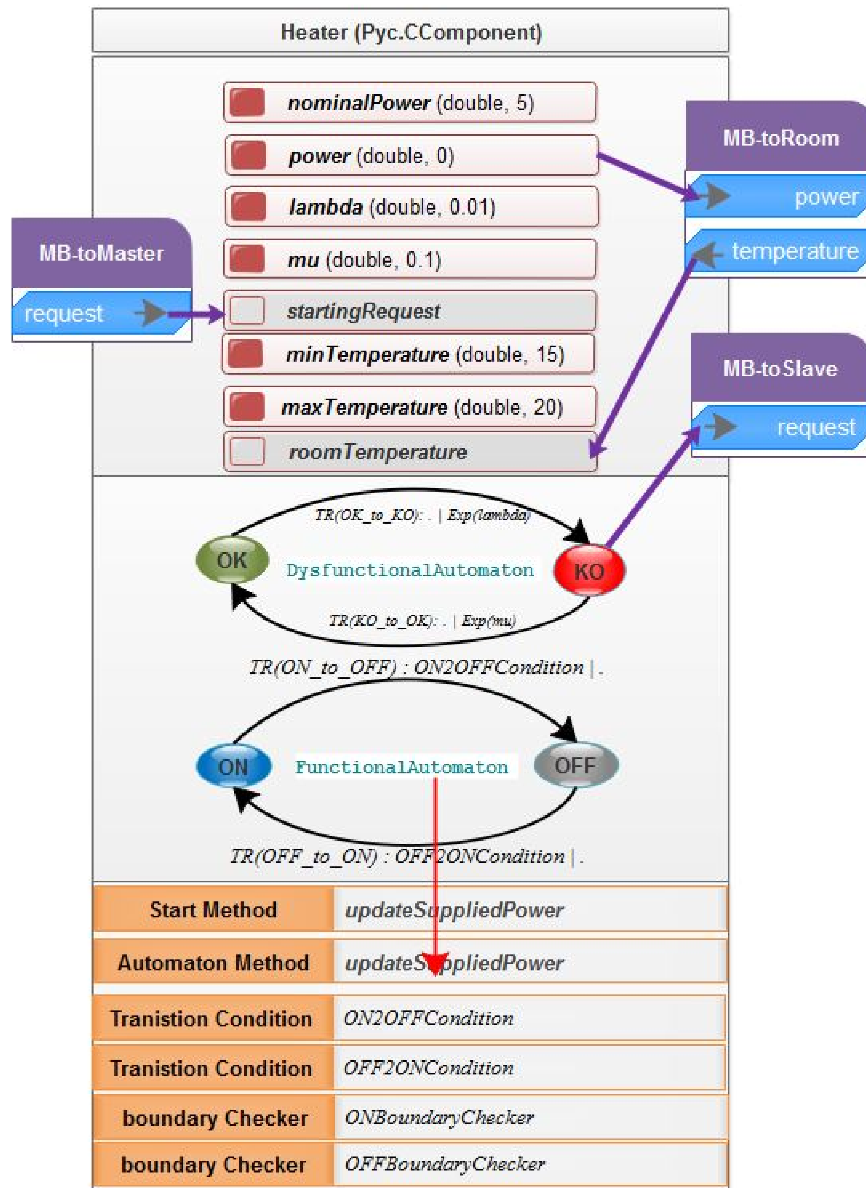


Figure 26

The Heater class for the step S3

In [2]:
```
# coding: latin-1
'''
Created on 07.06.2017

@author: Hassane CHRAIBI - hassane.chraibi@edf.fr

'''

from Util import *                                                          #S1


##################################################################################
class Heater(Pyc.CComponent):                                               #S1

    def __init__(self, name):                                               #S1
        Pyc.CComponent.__init__(self, name)                                 #S1

        # The intrinsic state variables ---------------------------------------------
        # The nominal power value supplied by the heater when on state ON
        # This value is held by a variable which type is double, with 5 as
        # initial value and known as "nominalPower"
        # Note : The initial values given here may be overridden by values
        # given by a parameter file loaded before
        # simulation
        self.po_nominalPower  = self.addVariable("nominalPower",
                                        Pyc.TVarType.t_double, 5)           #S1

        # Holds the power supplied : 0 when OFF and po_nominalPower when ON
        self.po_power         = self.addVariable("power"       ,
                                        Pyc.TVarType.t_double, 0)           #S1

        # The failure rate of the heater
        self.po_lambda = self.addVariable("lambda",
                                Pyc.TVarType.t_double, 0.01)                #S1

        # The repair rate of the heater
        self.po_mu     = self.addVariable("mu"    ,
                                Pyc.TVarType.t_double, 0.1)                 #S1

        # When the temperature of the heated room goes below minTemperature
        # the heater must start if requested
        self.po_minTemperature = self.addVariable("minTemperature",
                                        Pyc.TVarType.t_double, 15)          #S3

        # When the temperature of the heated room exceeds maxTemperature
        # the heater must stop
        self.po_maxTemperature = self.addVariable("maxTemperature",
                                        Pyc.TVarType.t_double, 20)          #S3
```

```
# The references to external information ------------------------------------
# This is a reference which holds the received requests that make the heater
# starting when there is a True value
self.pi_startingRequest = self.addReference("startingRequest")           #S2
# This reference will be  bound to the current temperature of the room
self.pi_roomTemperature = self.addReference("roomTemperature")           #S3


# The Automata declaration---------------------------------------------------
# heater dysfunctional automaton
# Creation of an automaton
self.aDysfunctional = self.addAutomaton("DysfunctionalAutomaton")        #S1
# Creates a state named OK and adds it to the automaton
self.stateOK     = self.addState("DysfunctionalAutomaton", "OK", 1)      #S1
# Creates a state named KO and adds it to the automaton
self.stateKO     = self.addState("DysfunctionalAutomaton", "KO", 0)      #S1

# Sets OK state as the initial state
self.setInitState("OK")                                                  #S1

# Creation of the transition starting from the OK state
trans = self.stateOK.addTransition("OK_to_KO")                           #S1
# Setting of an exponential law as the probability law of the
# transition law to the failure state
# po_lambda is the parameter of this law
trans.setDistLaw(Pyc.TLawType.expo, self.po_lambda)                      #S1
# The definition of the target of the transition
trans.addTarget(self.stateKO, Pyc.TTransType.fault)                      #S1

# The definition of the repair transition
trans = self.stateKO.addTransition("KO_to_OK")                           #S1
# Setting of an exponential law as the probability law of
# the transition law to the fixed state
# po_mu is the parameter of this law
trans.setDistLaw(Pyc.TLawType.expo, self.po_mu)                          #S1
# The definition of the target of the transition
trans.addTarget(self.stateOK, Pyc.TTransType.rep)                        #S1



# heater functional automaton
# Creation of an automaton
self.aFunctional = self.addAutomaton("FunctionalAutomaton")              #S1
# Creates a state named OK and adds it to the automaton
self.stateON  = self.addState("FunctionalAutomaton", "ON" , 1)           #S1
# Creates a state named KO and adds it to the automaton
self.stateOFF = self.addState("FunctionalAutomaton", "OFF", 0)           #S1
```

```
# Sets OK state as the initial state
self.setInitState("OFF")                                                    #S1

# Sets the method to be called when the functional automaton changes
# in order to update the value of the supplied heating power according to
# the states ON and OFF
self.aFunctional.addSensitiveMethod("UpdateSuppliedPower",
                                    self.updateSuppliedPower)               #S1

# Creation of the transition starting from the OFF state
transOff2On = self.stateOFF.addTransition("OFF_to_ON")                      #S1
# Sets the condition of the transition as a boolean function
transOff2On.setCondition(self.OFF2ONCondition)                              #S1
# The definition of the target of the transition
transOff2On.addTarget(self.stateON, Pyc.TTransType.trans)                   #S1

# Creation of the transition starting from the ON state
transOn2Off = self.stateON.addTransition("ON_to_OFF")                       #S1
# Sets the condition of the transition as a boolean function
transOn2Off.setCondition(self.ON2OFFCondition)                              #S1
# The definition of the target of the transition
transOn2Off.addTarget(self.stateOFF, Pyc.TTransType.trans)                  #S1



# Message boxes --------------------------------------------------------------
# This message box will link to the master heaters
messageBox = self.addMessageBox("MB-toMaster")                              #S2
# Transmits to self.pi_startingRequest the state KO of the master heater
# in order to provoke starting when True
messageBox.addImport(self.pi_startingRequest, "request")                    #S2

# This message box will link to the  slave heater
messageBox = self.addMessageBox("MB-toSlave")                               #S2
# Transmits state KO to the slave heater in order to make it starting
# when state is KO
messageBox.addExport(self.stateKO, "request")                               #S2

# This message box will link to the  room
messageBox = self.addMessageBox("MB-toRoom")                                #S3
# Transmits the value of the power supplied
messageBox.addExport(self.po_power, "power")                                #S3
# Receives the value of the room current temperature
messageBox.addImport(self.pi_roomTemperature, "temperature")                #S3



# The PDMP Manager -----------------------------------------------------------
```

```
        pdmpManager = self.addPDMPManager("PDMP-Manager")                        #S3
        # Designate transition to watch in order to check if the system
        # is still in region consistent with the current states
        self.addPDMPWatchedTransition("PDMP-Manager", transOff2On)               #S3
        self.addPDMPWatchedTransition("PDMP-Manager", transOn2Off)               #S3



        # Makes updateSuppliedPower method to be called at the beginning of every
        # simulation in order to update the value of the supplied heating power
        # according the initial states (ON or OFF)
        self.addStartMethod("UpdateSuppliedPower", self.updateSuppliedPower)    #S1


    # This method returns True if the heater has to switch to ON
    def OFF2ONCondition(self):                                                   #S1
        # We don't switch to ON if the heater is not OK
        if not self.stateOK.isActive() :                                         #S1
            return False                                                         #S1

        # We don't switch to ON if there is a Master and there is no request
        if not self.pi_startingRequest.orValue(True) :                           #S2
            return False                                                         #S2

        # We switch to ON if the temperature is too low
        return (self.pi_roomTemperature.value(0) <                               #S3
                    self.po_minTemperature.value())                              #S3


    # This method returns True if the heater must switch to OFF
    def ON2OFFCondition(self):                                                   #S1
        # We switch to OFF if the heater is not OK
        if not self.stateOK.isActive() :                                         #S1
            return True                                                          #S1

        # We switch to OFF if there is a Master and there is no request
        if not self.pi_startingRequest.orValue(True) :                           #S2
            return True                                                          #S2

        # We switch to OFF if the temperature is too high
        return (self.pi_roomTemperature.value(0) >                               #S3
                self.po_maxTemperature.value())                                  #S3


    # This method returns -1 (i.e. is in forbidden region) if:
    #    the heater is OK and ON and
    #      (the temperature is higher than the maximum threshold or
    #       the heater has no starting request)
```

```
# Otherwise it returns 1
def ONBoundaryChecker(self):                                                    #S3
    if self.stateKO.isActive():                                                 #S3
        return 1. # No boundary crossed                                         #S3

    if not self.stateON.isActive():                                             #S3
        return 1. # No boundary crossed                                         #S3

    if (self.pi_roomTemperature.value(0) > self.po_maxTemperature.value() or
        not self.pi_startingRequest.orValue(True)):                             #S3
        return -1. # A boundary has been crossed to the forbidden region        #S3
    else :                                                                      #S3
        return 1. # is still in valid region                                    #S3


# This method returns -1 (i.e. is in forbidden region) if:
#     the heater is OK and OFF and
#       (the temperature is lower than the minimum threshold and
#        the heater has a starting request)
# Otherwise it returns 1
def OFFBoundaryChecker(self):                                                   #S3
    if self.stateKO.isActive():                                                 #S3
        return 1. # No boundary crossed                                         #S3

    if not self.stateOFF.isActive():                                            #S3
        return 1. # No boundary crossed                                         #S3

    if (self.pi_roomTemperature.value(0) < self.po_minTemperature.value() and
        self.pi_startingRequest.orValue(True)):                                 #S3
        return -1. # A boundary has been crossed to the forbidden region        #S3
    else:                                                                       #S3
        return 1. # is still in valid region                                    #S3



# This method is called every time the functional automaton changes
# and also at the beginning of every simulation
def updateSuppliedPower(self):                                                  #S1
    if self.stateON.isActive():                                                 #S1
        # If ON, the heater delivers its nominal heating power
        self.po_power.setDValue(self.po_nominalPower.value())                   #S1
    else :                                                                      #S1
        # If OFF, the heater does not deliver any  heating power
        self.po_power.setDValue(0)                                              #S1
```

7.3.6 Step 3 - testing

The test performed here consists in adding a room to S2 system and in connecting this room to the two heaters. The indicators to calculate are the evolution of the mean room temperature over time and also the two temperature quantiles :

1. The temperature lower than the temperature of only 1% of the simulated sequences
2. The temperature higher than the temperature of only 1% of the simulated sequences

```
In [3]: #####################################################################################
        class MySystem(Pyc.CSystem):                                                    #S1
            def __init__(self, name):                                                   #S1
                Pyc.CSystem.__init__(self, name)                                        #S1

                # Instantiation of a Master and Slave Heater
                self.aMasterHeater = Heater("aMasterHeater")                            #S2
                self.aSlaveHeater  = Heater("aSlaveHeater")                             #S2
                self.room          = Room("Room")                                       #S3

                # connecting heaters slave and master
                self.connect("aMasterHeater", "MB-toSlave", "aSlaveHeater", "MB-toMaster") #S2
                # connecting heaters and the room :
                self.connect("aMasterHeater", "MB-toRoom", "Room", "MB-toHeaters")      #S3
                self.connect("aSlaveHeater" , "MB-toRoom", "Room", "MB-toHeaters")      #S3
```

We also have to update the initial values and the configuration of the simulation. To do so we will use a third XML file with the following items:

```
<PY_PARAM NAME="S3-Parameters" TMAX="100" SEQ_NB="10000" RNG="yarn5"  RNG_S="0"><!-- S3 -->

    <TRACE_TR NAME="#.*" LEVEL="0"/>                                            <!-- S1 -->


    <VAR NAME="aMasterHeater.nominalPower"  INITV="5"/>                         <!-- S2 -->
    <VAR NAME="aSlaveHeater.nominalPower"   INITV="5"/>                         <!-- S2-->

    <VAR NAME="aMasterHeater.lambda"        INITV="0.01"/>                      <!-- S2 -->
    <VAR NAME="aMasterHeater.mu"            INITV="0.1"/>                       <!-- S2 -->

    <VAR NAME="aSlaveHeater.lambda"         INITV="0.01"/>                      <!-- S2 -->
    <VAR NAME="aSlaveHeater.mu"             INITV="0.1"/>                       <!-- S2 -->

    <ST NAME="aMasterHeater.OK"             INIT="1"/>                          <!-- S2 -->
    <ST NAME="aMasterHeater.ON"             INIT="1"/>                          <!-- S2 -->

    <VAR NAME="#.*\.maxTemperature"         INITV="20."/>                       <!-- S3 -->
    <VAR NAME="#.*\.minTemperature"         INITV="15."/>                       <!-- S3 -->

    <ST NAME="aSlaveHeater.OK"              INIT="1"/>                          <!-- S2 -->
```

```
    <ST NAME="aSlaveHeater.OFF"            INIT="1"/>                          <!-- S2 -->



    <VAR NAME="Room.leakageRate"           INITV="0.1" />                      <!-- S3 -->
    <VAR NAME="Room.initialTemperature"    INITV="17."/>                       <!-- S3 -->
    <VAR NAME="Room.outsideTemperature"    INITV="13."/>                       <!-- S3 -->

    <MONIT_VAR NAME="aMasterHeater.power"/>                                    <!-- S2 -->
    <MONIT_VAR NAME="aSlaveHeater.power"/>                                     <!-- S2 -->
    <MONIT_VAR NAME="Room.temperature"/>                                       <!-- S3 -->

    <ODE NAME="PDMP-Manager" DT="1" DTM="1" DTC="0.001" SCHEME="1"/>           <!-- S3 -->

</PY_PARAM>                                                                    <!-- S1 -->
```

Below, we give a commented Python main program which launches the simulation and draws the curves of the evolution over time of the mean of the room temperature and the two quantiles:

1. Temperatures lesser than temperatures of 1% of simulated sequences.
2. Temperatures greater the temperatures of 1% of simulated sequences.

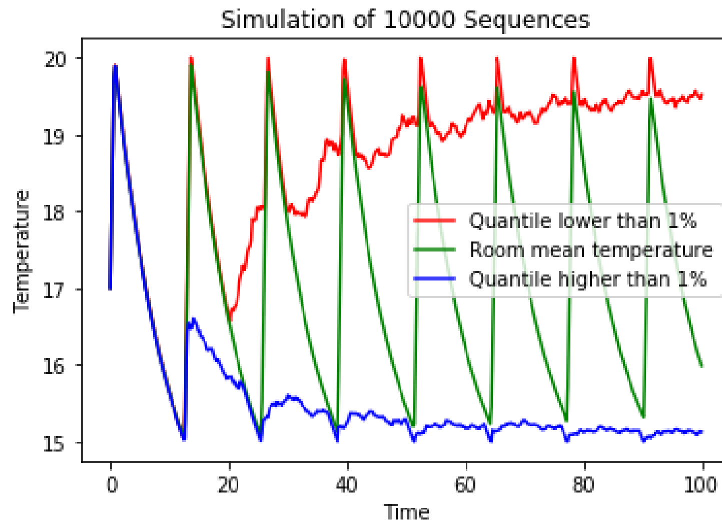
This program also draws the curve of the cumulative probability distribution of the indicator function : Temperature < 14.°C and calculates the mean time spent by the room with a temperature under 14.°C.

```
In [4]: ######################################################################################
        if __name__ == '__main__':                                                       #S1

            try :
                # An instance of the system is constructed (Only one system can be
                # constructed in a session)
                system = MySystem("S3")                                                  #S3

                # The simulation et initial values are load from an XML file
                system.loadParameters("HeatedRoom_S3.xml")                               #S3

                # Lanches the simulation
                beginTime = time.time()                                                  #S1
                system.simulate()                                                        #S1

                # In case of parallel simulation we end all the launched instances
                # except the maine one which deals with post processing
                if system.MPIRank() > 0:                                                 #S1
                    exit(0)                                                              #S1

                endTime = time.time()                                                    #S1
                print ("Simulation time = %.2f s"% (endTime - beginTime))                #S1
```

```
# We create an anlyzer which holds the values of all the monitored elements
analyser = Pyc.CAnalyser(system)                                              #S1

# This vector will holds the instants where the values of the indicators
# will be calculated
vectorOfInstants = stepsVector(system.tMax(), 500)                            #S1

# Calculates the evolution over time of the mean of the room temperature :
temperatureMeans = analyser.meanValues("Room.temperature",
                                       vectorOfInstants)                      #S3

# Calculates two temperature curves :
# 1 : temperature such that only 1% of the the simulated sequences
# give a higher temperature (red curve)
# 2 : temperature such that only 1% of the the simulated sequences
# give a lower temperature (blue curve)
quantilesGt01 =analyser.quantilesGt("Room.temperature",                      #S3
                                    vectorOfInstants, 1.)                    #S3

quantilesLe01 =analyser.quantilesLe("Room.temperature",                      #S3
                                    vectorOfInstants, 1.)                    #S3

#This is call to a convenient function which draws the curves of the
# indicators' evolution over time
plot(vectorOfInstants, True, True,                                            #S3
     (                                                                        #S3
         (("Simulation of %d Sequences") % system.nbSequences(),              #S3
          "Time",                                                             #S3
          "Temperature",                                                      #S3
          ((quantilesLe01, 'r', "Quantile lower than 1%"),)),                 #S3

         (("Simulation of %d Sequences") % system.nbSequences(),              #S3
          "Time",                                                             #S3
          "Temperature",                                                      #S3
          ((temperatureMeans, 'g', "Room mean temperature"),)),               #S3

         (("Simulation of %d Sequences") % system.nbSequences(),              #S3
          "Time",                                                             #S3
          "Temperature",                                                      #S3
          ((quantilesGt01, 'b', "Quantile higher than 1%"),)),                #S3
     )                                                                        #S3
    )                                                                         #S3

# This vector will hold the cumulative probability distribution
# of the Indicator Temperature < 14
hotTemperatureOccurrences = analyser.realized("Room.temperature",             #S3
                  vectorOfInstants,                                           #S3
```

```
                    lambda t: t < 14)                                               #S3
    # This is call to a convenient function which draws the curve of the
    # cumulative probability distribution calculated above
    plot(vectorOfInstants, True, True,                                              #S3
      (                                                                             #S3
        (("Simulation of %d Sequences") % system.nbSequences(),                     #S3
          "Time",                                                                   #S3
          "Probability",                                                            #S3
          ((hotTemperatureOccurrences, 'r', "CPDF Ind : Temperature < 14."),)       #S3
        ),                                                                          #S3
      )                                                                             #S3
       )                                                                            #S3

    # Claculates the mean time spent under 14.°C between 0 an tMax
    meanResidenceTime = analyser.residenceTime("Room.temperature",                  #S3
                                               0, system.tMax(),                    #S3
                                               lambda t: t < 14)                    #S3

    print("-----------------------------------------")                              #S3
    print("Mean Time spent under 14.°C : %.2f hours"%meanResidenceTime)             #S3
    print("-----------------------------------------")                              #S3



except Exception as e:                                                              #S1
    print(type(e),e)                                                                #S1
    Pyc.printMessages()                                                             #S1
```

```
Simulation time = 16.13 s
-----------------------------------------
Mean Time spent under 14.°C : 0.07 hours
-----------------------------------------
```



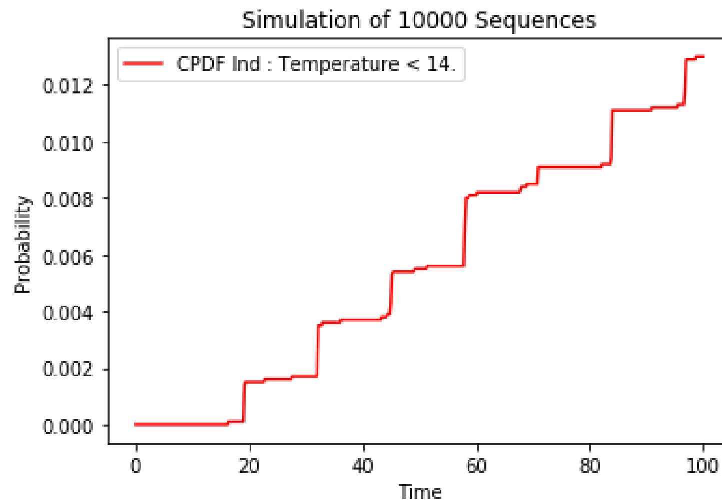


Figure 27

The first graph gives the mean and the quantile curves of the room temperature
The second graph gives the cumulative probability distribution of the undesired event

7.4 Step 4: Drawing sequences

In this step we will reuse the system S3 to which we will add the instructions that produce the simulated sequences. The first instructions designate which transitions must be shown in the produced sequences. Such transitions are those which are monitored. We choose to show only the failure events of the heaters ***OK_to_KO*** and repairs events ***KO_to_OK***. So in the XML file we have to add the following instructions :

```
<MONIT_TR  NAME="#.*\.OK_to_KO"/>                                              <!-- S4 -->
<MONIT_TR  NAME="#.*\.KO_to_OK"/>                                              <!-- S4 -->
```

We will then visualize among the produced sequences only those where the temperature drops under 14°C. To do so we will define a function which operates as a filter since it returns True if the room temperature drops under 14°C before the end of the sequence simulation and return False otherwise. This function has to be set as a filter on the analyzer used to retrieve the simulated sequences.

```
In [4]: ####################################################################################
        if __name__ == '__main__':                                                     #S1

            try :
                # An instance of the system is constructed (Only one system can be
                # constructed in a session)
                system = MySystem("S4")                                                #S3

                # The simulation et initial values are load from an XML file
                system.loadParameters("HeatedRoom_S4.xml")                             #S3

                # Lanches the simulation
                beginTime = time.time()                                                #S1
                system.simulate()                                                      #S1

                # In case of parallel simulation we end all the launched instances
                # except the maine one which deals with post processing
                if system.MPIRank() > 0:                                               #S1
                    exit(0)                                                            #S1

                endTime = time.time()                                                  #S1
                print ("Simulation time = %.2f s"% (endTime - beginTime))              #S1


                # We create an anlyzer which holds the values of all the monitored elements
                analyser = Pyc.CAnalyser(system)                                       #S1

                # This vector will holds the instants where the values of the indicators
                # will be calculated
                vectorOfInstants = stepsVector(system.tMax(), 500)                     #S1

                # Calculates the evolution over time of the mean of the room temperature :
```

```
temperatureMeans = analyser.meanValues("Room.temperature",
                                       vectorOfInstants)                    #S3

# Calculates two temperature curves :
# 1 : temperature such that only 1% of the the simulated sequences
# give a higher temperature (red curve)
# 2 : temperature such that only 1% of the the simulated sequences
# give a lower temperature (blue curve)
quantilesGt01 =analyser.quantilesGt("Room.temperature",                     #S3
                                    vectorOfInstants, 1.)                   #S3

quantilesLe01 =analyser.quantilesLe("Room.temperature",                     #S3
                                    vectorOfInstants, 1.)                   #S3

#This is call to a convenient function which draws the curves of the
# indicators' evolution over time
plot(vectorOfInstants, True, True,                                          #S3
     (                                                                      #S3
         (("Simulation of %d Sequences") % system.nbSequences(),            #S3
          "Time",                                                           #S3
          "Temperature",                                                    #S3
          ((quantilesLe01, 'r', "Quantile lower than 1%"),)),               #S3

         (("Simulation of %d Sequences") % system.nbSequences(),            #S3
          "Time",                                                           #S3
          "Temperature",                                                    #S3
          ((temperatureMeans, 'g', "Room mean temperature"),)),             #S3

         (("Simulation of %d Sequences") % system.nbSequences(),            #S3
          "Time",                                                           #S3
          "Temperature",                                                    #S3
          ((quantilesGt01, 'b', "Quantile higher than 1%"),)),              #S3
     )                                                                      #S3
    )                                                                       #S3

# This vector will hold the cumulative probability distribution
# of the Indicator Temperature < 14
hotTemperatureOccurrences = analyser.realized("Room.temperature",           #S3
                  vectorOfInstants,                                         #S3
                  lambda t: t < 14)                                         #S3
# This is call to a convenient function which draws the curve of the
# cumulative probability distribution calculated above
plot(vectorOfInstants, True, True,                                          #S3
  (                                                                         #S3
    (("Simulation of %d Sequences") % system.nbSequences(),                 #S3
      "Time",                                                               #S3
      "Probability",                                                        #S3
      ((hotTemperatureOccurrences, 'r', "CPDF Ind : Temperature < 14."),)   #S3
```

```
        ),                                                              #S3
      )                                                                 #S3
       )                                                                #S3


    # Claculates the mean time spent under 14.°C between 0 an tMax
    meanResidenceTime = analyser.residenceTime("Room.temperature",      #S3
                                               0, system.tMax(),        #S3
                                               lambda t: t < 14)        #S3

    print("-----------------------------------------")                  #S3
    print("Mean Time spent under 14.°C : %.2f hours"%meanResidenceTime)  #S3
    print("-----------------------------------------")                  #S3


    # This function is a filter which returns true for a sequence
    # where the room temperature drops under 14°C at least once
    # before system.tMax()
    def mySeqFilter(sequence):                                          #S4
        value =  sequence.realized(system.room.po_temperature,          #S4
                                   lambda x : x < 14,                   #S4
                                   vectorOfInstants)                    #S4
        return value[-1]                                                #S4

    # Sets the filter defined above
    analyser.setSeqFilter(mySeqFilter)                                  #S4
    # This call produces an XML and an HTML files of all the simulated
    # sequences where the room temperature drops under 14°C at least once
    # before system.tMax()
    analyser.printFilteredSeq(100,                                      #S4
                              "Filtred-Seq-HeatedRoom_S4.xml",          #S4
                              "PySeq.xsl")                              #S4

except Exception as e:                                                  #S1
    print(type(e),e)                                                    #S1
    Pyc.printMessages()                                                 #S1
```

Number of simulated sequences : 10000

Number of filtred sequences : 153

Sequence probability	Sequence ending reason	Description of sequence branches					
		n°	Mean time of firing	Transition description			
				name	final state	type	law
0.0013	Normal	1	30.6442	aMasterHeater.OK_to_KO	KO	fault	exp
		2	37.3094	aSlaveHeater.OK_to_KO	KO	fault	exp
		3	53.6684	aSlaveHeater.KO_to_OK	OK	repair	exp
		4	64.0062	aMasterHeater.KO_to_OK	OK	repair	exp
0.001	Normal	1	52.1683	aSlaveHeater.OK_to_KO	KO	fault	exp
		2	61.706	aMasterHeater.OK_to_KO	KO	fault	exp
		3	74.6036	aMasterHeater.KO_to_OK	OK	repair	exp
		4	85.4304	aSlaveHeater.KO_to_OK	OK	repair	exp
0.0007	Normal	1	32.2418	aSlaveHeater.OK_to_KO	KO	fault	exp
		2	40.5998	aMasterHeater.OK_to_KO	KO	fault	exp
		3	63.7547	aSlaveHeater.KO_to_OK	OK	repair	exp
		4	72.748	aMasterHeater.KO_to_OK	OK	repair	exp
0.0005	Normal	1	9.79719	aMasterHeater.OK_to_KO	KO	fault	exp
		2	16.2194	aMasterHeater.KO_to_OK	OK	repair	exp
		3	43.171	aMasterHeater.OK_to_KO	KO	fault	exp
		4	48.624	aSlaveHeater.OK_to_KO	KO	fault	exp
		5	66.1603	aMasterHeater.KO_to_OK	OK	repair	exp
		6	73.88	aSlaveHeater.KO_to_OK	OK	repair	exp
0.0004	Normal	1	16.4108	aSlaveHeater.OK_to_KO	KO	fault	exp
		2	23.4685	aMasterHeater.OK_to_KO	KO	fault	exp

Figure 28

An excerpt of the filtered sequences produced by the simulation for the step S4