Dynamic reliability modeling and assessment with PyCATSHOO: Application to a test case

Hassane CHRAIBI,
EDF R&D/MRI, Clamart, France
hassane.chraibi@edf.fr

Abstract: PyCATSHOO is a new tool using, in an original way, the stochastic hybrid automata as a representation language of the Piecewise Deterministic Markov Processes.
PDMP are a well-known framework suited for the modeling and assessment of a large range of system classes when, in such systems, the interaction between the stochastic discrete events and the deterministic evolution of the continuous state variables has to be taken into account. However, it is still facing several difficulties, especially in the case of large systems where the interaction, between discrete and continuous states, makes the system modeling very arduous.
In the PyCATSHOO approach, a system behavior is modeled thanks to two kinds of automata. There are, on the one hand, the stochastic hybrid automata which stand for the PDMPs and, on the other hand, the pure discrete stochastic automata which model the discrete system behavior. These two kinds of automata inhabit object structures enriched by paradigms borrowed from computer science such as Object-Oriented, Multi-Agent Systems and functional programming paradigms. Such paradigms are behind the compositional modeling potentialities which make PyCATSHOO able to address the difficulties led by continuous and discrete state interactions.
This paper is aimed at illustrating some PyCATSHOO capabilities by reporting its use in the reliability assessment of a system derived from a well-known heated tank test-case. The difference between the former and the variant presented here stands in the valves that fill and empty the tank. In the original test case these devices open and close instantaneously while, in the variant we have assessed in this paper, the transitions between open and closed states are not instantaneous.
This test case exhibits an interaction between the stochastic discrete events and the deterministic evolution of the continuous state variables governed by a two dimensional and non-linear differential equation. Moreover, the failure rate of one of the devices involved in the studied system depends on the continuous state variables of the system.
Therefore, through this test case, we show in this paper how PyCATSHOO helps dealing with the two main characteristics of a stochastic hybrid system.

Keywords: Dynamic reliability, stochastic hybrid systems, probabilistic safety assessment, piecewise deterministic Markov processes, PyCATSHOO.

1 INTRODUCTION

Dynamical reliability assessments refer to those reliability assessments based on a hybrid model of the assessed system. A hybrid model is characterized by the ability to account for the interactions between the evolutions of two kinds of the system state variables: discrete variables which are subject to random changes and continuous variables which follow deterministic paths between two successive discrete events.

There is no need to recall the expected benefit from dynamical reliability assessments for a large class of system. Nuclear power plants are among such systems. Indeed, in the level 2 PSA [1] [2], the importance of the interaction between the process and the thermal hydraulic of an accident is now widely admitted [3] while the Event Tree and Fault Tree (ET/FT) approaches mostly used in the Nuclear domain have reached their limits and do not sufficiently take into account the interaction between the evolutions of physical phenomena and the discrete states of the system.

Dynamical reliability assessments face two kinds of difficulties. The first one concerns the quantification tools as, due to the complexity of the hybrid models, only simulation is still reasonably usable for real industrial systems. The second difficulty concerns the formulation of such systems models which account for the two kinds of state variables and for their complex interactions.

This paper mainly concerns how PyCATSHOO (PythoniC Object Oriented Hybrid Stochastic AuTomata), a recently developed tool, deals with the second difficulty. So, in this paper, we will not describe the embedded PyCATSHOO Monte Carlo Simulation engine. Indeed, even if this engine is ready for running in multi-core machines and computer clusters, up to now, we are still working on its improvement by experimenting some acceleration methods such as sequential Monte Carlo methods. We will rather describe the compositional modeling approach proposed by PyCATSHOO which borrows several ideas from computer science approaches and which is based on an existing theoretical framework that is "Piecewise Deterministic Markov Processes" (PDMP or PDP) that PyCATSHOO models as stochastic hybrid automata.

We will begin this paper by briefly introducing these notions, and we will show how they are used in PyCATSHOO in order to help with system compositional modeling. Then, we will describe the test case we used to illustrate PyCATSHOO capabilities and the kind of results that this new tool produces.

2 PyCATSHOO BASICS

2.1 PDMP principles

Thanks to its mathematical properties, the PDMP framework is suited for PSA of a large class of systems in which hybrid modeling is required.

PDMP have been introduced by M. H. A. Davis [4] and their mathematical formulation has been recalled by several authors such as in [5], [6] and [7]. So we will just mention some of their main principles.

In the PDMP framework, two kinds of transitions may occur. The first one corresponds to spontaneous transitions which move a system component from a discrete state to another one. These transitions are triggered by events such as component failures or repairs and come up at instants governed by probabilistic laws. The second kind corresponds to the forced transitions. These transitions occur when a limit is reached by continuous state variables. Such a limit may, for instance, be an ambient temperature value which causes a switching of a heater thermostat.

According to the mathematical properties of PDMP, the trajectory of a system may be constructed iteratively. At the beginning, the system evolves deterministically until the occurrence of the first transition which belongs to one of the two kinds of transitions mentioned above. This transition may lead to a discontinuity of the state variables and / or to a mode modification i.e. to the modification of the differential equation which governs the evolution of the continuous state variables over time. The new mode and the new value of the state variables vector may be set according to the system logic. This new position of the system may also be governed by a probability distribution. Once in a new position, the system goes on a new deterministic path until a new transition occurs and so on.

2.2 Stochastic Hybrid Automata as a behavior modeling formalism

At first glance, the PDMP path construction principles may give the illusion of a convenient usability of the PDMP. But in reality their use presents a lot of difficulties. Indeed, On the one hand, only Monte Carlo Simulation can be reasonably considered as a quantification method in case of large systems. And, on the other hand, the modeling task tends to become more and more arduous when the size and complexity of the studied systems increase. This is due to the fact that few graphical formalisms are able to represent PDMP in a convenient way and to produce a suitable model for quantitative and qualitative evaluation [8]. Dynamically Colored Petri Nets DCPN is one of these formalisms that were proven to be equivalent to PDMP [9]. The automata formalism is another one. It has been used as a representation language of PDMP in several works and has given the PDMP the compositional modeling capability they lack [10]. PyCATSHOO adopts the latter formalism as a means description of the behavior of the system components.

As a reminder of the stochastic hybrid automata principles, here we have the list of its ingredients as reported by several authors such as in [11], [12] & [13].

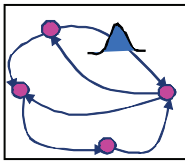
A stochastic hybrid automaton is a 6-tuple: $SHA = (M, X, Inv, T, f, (m_0, x_0))$. Where:

- M is a set of discrete states, locations or modes.
- $X \subseteq \mathbb{R}^n$ is a continuous state space.
- $Inv : M \to P(X)$ is a function that gives, for each mode $m \in M$, $Inv(m) \subseteq X$, a subset of the continuous state space which represents the invariant condition of the mode m. A mode invariant condition is a condition that has to be satisfied (true) as long as the associated mode is active. An active mode has to be left (deactivated) at the instant its invariant condition becomes false.
- T is the set of transitions. Each $\tau \in T$ is 6-tuple $(m, m', \lambda_\tau, R_\tau, guard_\tau, Inv_\tau)$ where:
  - m and m' are respectively source and destination of the transition.
  - λ_τ is a jump rate function that gives probability of discrete state transition.
  - R_τ may be a deterministic function or a transition measure that resets probabilistically the values of the continuous state variables when the jump from m to m' occurs.
  - $guard_\tau$ is a subset of state space. It represents conditions under which the transition is enabled (may occur). The guards are only relevant for spontaneous transitions.
  - Inv_τ is a subset of the continuous state space. It represents conditions under which the transition is forced (must occur). Inv_τ satisfies: $\bigcup_{\tau \in T, source(\tau)=m} Inv_\tau = \neg Inv(m)$ the complement of $Inv(m)$.

    The notion of invariant is only relevant for forced transitions.
- $f : M \to f_m$ is a vector field which characterizes the continuous dynamics for every mode.
- $m_0 \in M$ is an initial discrete state (initial mode) and x_0, an initial continuous state.

2.3 Brief description of PyCATSHOO modeling's main principles

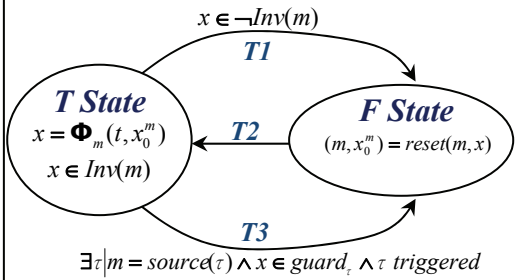
As mentioned above, in the PyCATSHOO approach, automata help to model a component behavior. So the first step when modeling a system with PyCATSHOO consists in defining the states of each of the system components and in giving the conditions of the transitions between these states.



This graph stands for an automaton. The nodes represent the states or modes. The oriented rows represent the transitions between them. One of these transitions has got a probability distribution which governs the instant of its occurrence. Such a transition is a stochastic one. The other transitions are deterministic.
This figure does not represent the condition of the transitions occurrence.

Figure 1 : An example of automaton which models the behavior of a system component

The transitions may be deterministic or stochastic and may be coupled with continuous physical phenomena. In this case, a hybrid stochastic automaton has to be created. PyCATSHOO provides required tools for creating such an automaton which is called the PDPM Controller. This automaton helps to manage the interaction between the discrete behaviors and the continuous ones.

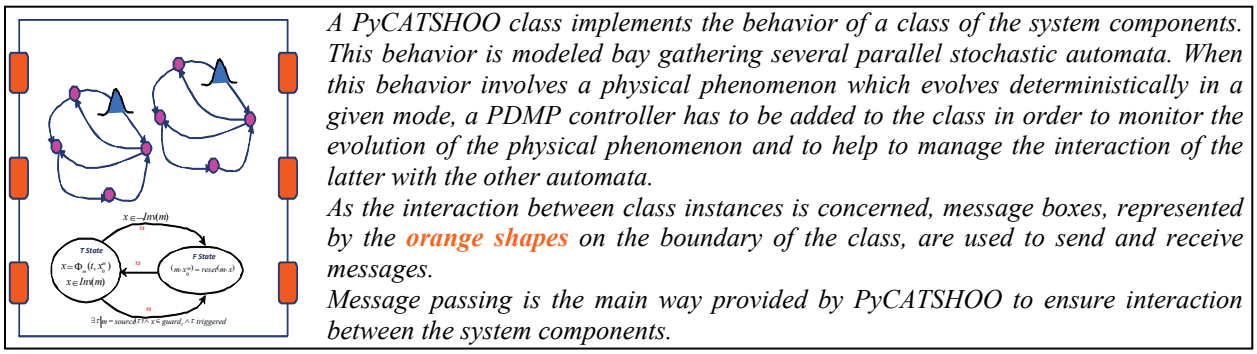


This graph represents the automaton associated with a PDMP Controller.
T State *represents a tangible state where the system evolves deterministically according to its differential equations in a given mode m.*
T1 *is a transition which occurs when the state variables of the system leave the specific validity domain of the current mode.*
T3 *is a transition which occurs when a spontaneous transitions occurs (failure of a component, etc.). Such a transition must have the current mode as source and its guard condition must be true.*
T1 *and* ***T3*** *lead to* ***F State****, a fugitive state, where the state variables and the mode are reset to new values. Then an instantaneous jump to* ***T State*** *makes the system start a new cycle.*

Figure 2 : The PDMP controller principles

All these automata are encapsulated in a class hierarchy of objects. Hence, these automata give proactive capabilities to such objects which represent the system components.

Objects are autonomous and their behaviors are only determined by their own goal as the multi agent system components do [14]. PyCATSHOO does not use each of the component automata to construct a global one. The dynamic of the system is still distributed over their components which interact by passing messages. So, every class has got a set of message boxes dedicated to sending and receiving messages. Furthermore, every class agrees a contract which makes its objects post, in precise circumstances, some messages in their own message boxes. On the other hand, received messages may change the state variables of an object which may lead to enabling or disabling some of its automata transitions and then to change the object behaviors.



A PyCATSHOO class implements the behavior of a class of the system components. This behavior is modeled bay gathering several parallel stochastic automata. When this behavior involves a physical phenomenon which evolves deterministically in a given mode, a PDMP controller has to be added to the class in order to monitor the evolution of the physical phenomenon and to help to manage the interaction of the latter with the other automata.
As the interaction between class instances is concerned, message boxes, represented by the ***orange shapes*** *on the boundary of the class, are used to send and receive messages.*
Message passing is the main way provided by PyCATSHOO to ensure interaction between the system components.

Figure 3 : A PyCATSHOO class which stands for a class of the system components

At the end of the first step, a generic modeling is done. The second step consists in describing the analyzed system which means:

- Creating its components by instantiating the classes created in the first step,
- Giving its architecture by creating the appropriate links between the objects message boxes.

The final modeling step consists in providing inputs and specifying the expected outputs.

In the next section, we will unfold these steps while processing a test case.

3 A VARIANT OF THE HEATED TANK TEST CASE

As mentioned above, the test case we have used to illustrate the PyCATSHOO approach and capabilities is derived from the one used by several authors such as in [5], [6] and others.

As illustrated by Figure 4, the system consists of a tank subject to a thermal source whose power decreases over time. To maintain the temperature of the system under a critical value, the tank is fed by a cooler fluid provided through two valves, V_1 and V_2 which also ensure that the level in the tank does not go below a low threshold. The tank is fitted with a third valve V_3 devoted to ensure that the level of the fluid in the tank never exceeds a high threshold. Under a certain level, the position of this valve does not allow fluid draining even when it is open.

Our objective here is to evaluate the reliability of the mission which consists in keeping the temperature of the system under 100°C and in preventing overflows.

The behavior of the valves in this variant of the heated tank test case differs from the original one. Indeed, the operating (opening and closing) of the three valves are not instantaneous. Furthermore, the three valves can only fail on demand with a probability γ or during operating (opening and closing) with different rates (λ_{vi}). Solely the valve V_3 is assumed to be fallible even when it is waiting and fully open. λ_w The rate of such a failure mode depends on the fluid temperature and their consequence is an undesired closing. The valves data are given in Table 1.

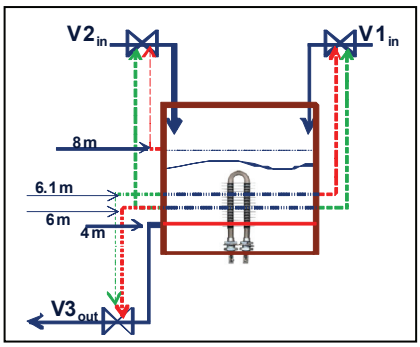


Figure 4 : The heated tank

Table 1 : Valves data

	Probability of failure on demand	**Failure Rate when operating**	**Operating time**	**Opens when**	**closes when**	**Position**	**Initial state**	**Max flow rate**
V_1	$\gamma = 0.01$	$\lambda_{v1}= 2.2831\ 10^{-3}$	$ot_{v1}=50$	level ≤ 6	level ≥ 6.1		Ok & Closed	1
V_2	$\gamma = 0.01$	$\lambda_{v2}= 2.8571\ 10^{-3}$	$ot_{v2}=50$	level ≤ 6	level ≥ 8		Ok & Open	1.01
V_3	$\gamma = 0.01$	$\lambda_{v3}= 1.562510^{-3}$	$ot_{v3}=50$	level ≥ 6.1	level ≤ 6	4	Ok & Open	1

While operating a valve, its flow is assumed to vary linearly over time so, the flow is: $F_{vi} = \alpha_{vi}.t + \beta_{vi}$ where α_{vi} and β_{vi} depend on the state of the valve. The failure rate of undesired closing failure of the V_3 valve is given according to θ, the temperature of the fluid in the tank:

$$\lambda_w = (0{,}8.e^{0{,}05756.(\theta-20)} + 0{,}2.e^{-0.2301(\theta-20)}).10^{-4} \qquad (1)$$

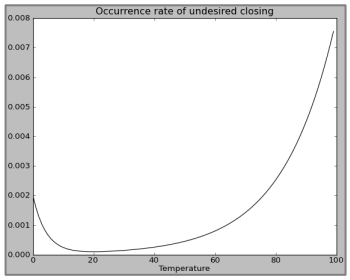


Figure 5 : λ_w: Occurrence rate of undesired closing of V_3 valve over temperature

The thermal production of the tank heater is assumed to decrease linearly over time: $\Delta H = 80 - 8.10^{-2}.t$

Table 2 : Tank data

Initial level	**Initial Temperature**	**Cooler temperature**	**Critical temperature**
7	31	θ_c=15	100

The two dimensional differential equation that governs the tank level and temperature is:

$$\frac{d}{dt}\begin{bmatrix} level \\ \theta \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{3} F_{vi} \\ \dfrac{(F_{v1}+F_{v2}).(\theta_c - \theta) + (80 - 8.10^{-2}.t)}{level} \end{bmatrix} \qquad (2)$$

3.1 PyCATSHOO modeling

PyCATSHOO is based on a library which provides the tools required by the modeling and assessment of a hybrid system. This library is developed in the Python language and uses several open source scientific computing tools such as Numpy Scipy and Simpy [15]. Up to now, PyCATSHOO does not have any graphical user interface. Its textual interface is however quite convenient. In order to illustrate the use of PyCATSHOO we will give the main Python instructions which help to define the key elements used by the PyCATSHOO approach.

3.1.1 Step 1: The generic modeling

This step consists in developing a set of classes called "PyCATSHOO Knowledge Base" (PyKB). For our test case the PyKB includes two classes: ***Valve*** and ***Tank***.

***Valve* class:**
The creation of this class includes the creation of its automaton as shown in Figure 6.

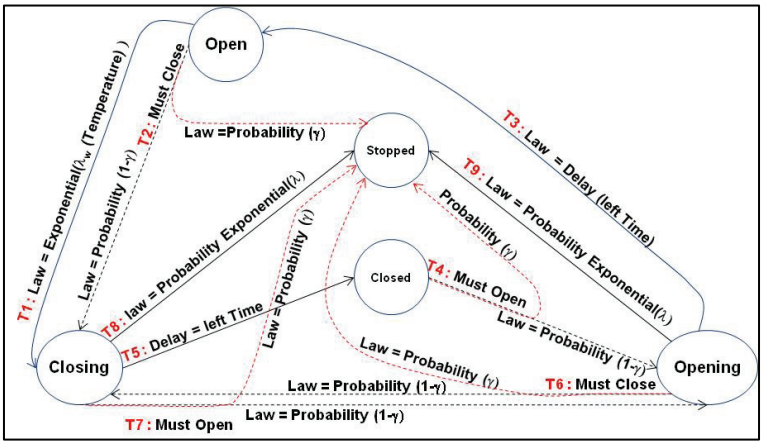


Figure 6 : The automaton that defines the behavior of the class ***Valve***

The automaton is created in the declarative way by firstly declaring the states with instructions such as *self.addState("Open")*. The automaton creation is then finalized by creating the transitions. Here we have some examples of the transition creation instructions:

- *self.addTransition("T9", "Opening","Stopped" , law = HCExpoPLaw(rate=lmbda))*
  This instruction introduces a transition which represents a failure of the valve while it is opening. The time of the occurrence of such a failure is a random variable which follows an exponential distribution introduced by the argument: *law = HCExpoPLaw(rate=lmbda))* with a rate equal to *lmbda*.
  PyCATSHOO provides several kinds of distribution and its library has an abstract class that one can derive to create its own distributions. This is particularly useful when the probability distribution parameters evolve according to the state variables of the system.
- *self.addVariableTransition("T1", "fireAgain", "Open" ,"Closing" ,*
  law = MyLaw((lambda : self.variable), rate=lmbda),
  condition = (lambda : self.started))
  This transition corresponds to the failure the V_3 valve which leads to the undesired jump to the closing state when the valve is required to stay in the open state. This transition may occur at a time which follows a random variable governed by a probability distribution with parameters that depend on the state variables of the tank. The noteworthy parameter of this instruction is:
  law = MyLaw((lambda : self.variable), rate=lmbda) which represents a specific distribution of the occurrence time of the transition. The expression of the rate of this distribution is given in a method that belongs to *MyLaw* class.

```
def rateFunction (self, v, t):
    #v[0] is the tank level and v[1] is its temperature, self.rate=10^-4
    return (0.8*np.exp(0.05756*(v[1]-20)) + 0.2*np.exp(-0.2301*(v[1]-20)))*self.rate
```

Once the class automata are created, we have to add message boxes which will give the ability to the instances of such a class to interact with the other system objects. The ***Valve*** class has got two message boxes declared as:

- *self.addMessageBox(HCMessageBox("Sensor",HCMessageBox.IN , self.getTankLevel))*
  This message box is used by the ***Valve*** class to receive the value of the tank level. The third parameter used by this message box declaration gives the method to be invoked when a new value is received. This method simply changes the value of an attribute which may modify the result of the evaluation of the valve automata transitions conditions.
- *self.addMessageBox(HCMessageBox("Tank",HCMessageBox.OUT, self.sendValveStateToTank))*
  This message box is used by the ***Valve*** class to send information about its state to the tank every time this is useful. This occurs in case of a failure or when the valve state changes.

***Tank* class:**
This class models the behavior of the tank component. We will not go into the details of the message boxes and the automata of this class. Instead, we will explain the creation of its main constituent that is the PDMP Controller. This creation is achieved thanks to the following declaration:

```
self.continuousVariable = HCPDMPController("tankState", ["currentLevel", "currentTemperature"], self,
                                [self.initialLevel, initialTemperature], [0.01, 0.1],
                                 maximalTimeStep, self.minimalTimeStep, collectingTimeStep,
                                 duration, self.getLevelValue, fDerivative,
                                 odeSolverMethod="lsoda", odeExtraParameters={"rtol":0.1})
```

We will limit the description of this declaration to the description of the underlined parameters:

- *["currentLevel", "currentTemperature"]*: This parameter means that the PDMP Controller will have to manage a two dimensional state variable vector.
- *[self.initialLevel, initialTemperature]*: This parameter gives the initial value of the state variable vector.
- *duration*: This parameter gives the system observation duration,
- *self.getLevelValue*: This is a method of ***Tank*** class that is invoked every time the level reaches a limit of the boundary of the current system mode. Indeed the PDMP Controller is able to detect and stop at the instant where a boundary is reached. In return, the tank has to inform the PDMP Controller when a spontaneous transition occurs. The PDMP Controller provides a method for this aim. Hence, it also stops in case of spontaneous transition occurrences and changes the evolving continuous state variables field.
- *fDerivative*: This is a function that computes the derivative of the state variables vector. In our test case, this function is written as follow:

```
def fDerivative(y, t, alpha, beta, coolerTemperature):
# alpha, beta, coolerTemperature : the parameter of the differential equation in the current mode
# dev is the derivative of the state variable vector - y is the state variable vector at the t time
    dev    = np.zeros(2)
    dev[0] = alpha[0]*t + beta [0]
    dev[1] = ((alpha[1]*(coolerTemperature - y[1]) + alpha[3])*t +
              beta [1]*(coolerTemperature - y[1]) + + beta[3])/(y[0])
    return dev
```

3.1.2 Step 2: System modeling

The system modeling consists in creating objects which will stands for the system components and in creating the paths of the messages exchanging by linking message boxes according to the architecture of the system given in Figure 7. This is done thanks to the following calls to PyCATSHOO library:

```
V1 = Valve("V1", self.lambdaV1, self.gamma, self.pvOperationTime, self.pv1MaxLevel,
                 self.pv1MinLevel, self.pv1FlowRate, Valve.FILL , Valve.CLOSED  )
V2 = Valve("V2", self.lambdaV2, self.gamma, self.pvOperationTime, self.pv2MaxLevel,
                 self.pv2MinLevel, self.pv2FlowRate, Valve.FILL , Valve.OPEN  )
V3 = Valve("V3", self.lambdaV3, self.gamma, self.pvOperationTime, self.pv3MaxLevel,
                 self.pv3MinLevel , self.pv3FlowRate, Valve.EMPTY, Valve.OPEN,
                 threshold=self.tankMinLevel)
```

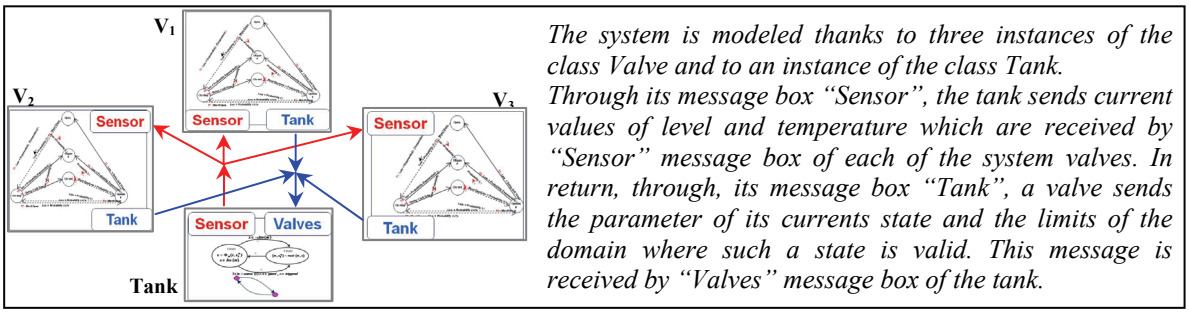
```
tank = Tank("Tank", self.tankMaxLevel, self.tankMinLevel, self.tankInitialLevel,
                    self.tankInitialTemperature, self.tankMaxTemperature,
                    self.coolerTemperature, self.tankPower, self.minimalTimeStep,
                    self.maximalTimeStep, self.collectingTimeStep, self.duration)

HCLink("v123_t", [(0, tank, "Valves"), (1, V1, "Tank"  ),  (2, V2  , "Tank"  ), (3, V3, "Tank"  )])
HCLink("t_v123", [(0, tank, "Sensor"), (1, V1, "Sensor"),  (2, V2  , "Sensor"), (3, V3, "Sensor")])
```

3.1.3 Step 3: Inputs and outputs definition

The final step consists in assigning values to the parameters passed in the previous calls and in selecting the system variables and/or the system states of interest. This is done by creating a monitor of such items. Here are examples of the PyCATSHOO instructions which help to make such selections:

```
1. temperatureMonitor = tank.monitorVariable("currentTemperature")
2. overflowMonitor    = tank.monitorState   ("Overflow")
```

The first instruction asks PyCATSHOO to memorize the evolution of the variable called "`currentTemperature`", which stands for the temperature of the fluid in the tank. The second one asks the memorization of the instants where the tank enters the `"Overflow"` state. This state must have been defined in the ***Tank*** class.



The system is modeled thanks to three instances of the class Valve and to an instance of the class Tank.
Through its message box "Sensor", the tank sends current values of level and temperature which are received by "Sensor" message box of each of the system valves. In return, through, its message box "Tank", a valve sends the parameter of its currents state and the limits of the domain where such a state is valid. This message is received by "Valves" message box of the tank.

Figure 7 : The PyCATSHOO model of the heated tank system

3.2 The quantification

PyCATSHOO has got an embedded Simulation Monte Carlo engine which can be used either on mono core or multi-core computers or on a cluster computer. Once a simulation is terminated, the state and variables monitors previously created can be used to produce the outputs of interest by asking PyCATSHOO to do some appropriate statistical computations such as:

```
1.  occVTempHot    = temperatureMonitor.occurrences(lambda t : t > 100.)
2.  mean           = temperatureMonitor.meanData   ()
```

The first instruction produces a curve which gives the evolution over time of the probability that the temperature exceeds 100 °C. The second one computes the mean temperature curve.

Below, we give for different input values, the evolution over time of the occurrence probability of the undesired events defined by level > 10 and temperature > 100 °C and, beforehand, we give in Figure 8 the evolution of temperature and level for one run simulation and without any failure.

In Figure 9, we assume that the V_3 valve does not fail when it is waiting in an open state. Hence, as the failures only occur when operating, one can notice the succession of ramps which are clearly visible in the left curve and which coincide with those of the left curve of Figure 8.

In Figure 10 and in the Figure 11 we assume that the V_3 valve may fail and jump to an undesired closing state when it is open and waiting. The rate λ_w of such failure is assumed to be constant and equal to λ_{v3} in the Figure 10 and, in the Figure 11, it is assumed to depend on the temperature according to the equation (1).

We can notice that in the cases of Figure 10 and Figure 11, the curves are smoother than the case of Figure 9. Indeed, in those cases failure may occur at almost any time. So the probability of occurrence of the undesired event is also more regularly distributed over time.

We can also notice the increasing rapidity of the probability of the overflow and of exceeding 100°C between 0 and 200 hours in Figure 11 with regard to Figure 10. This is due to the high values of the occurrence rate of the undesired closing of the V_3 valve when temperature is high as shown in Figure 5.

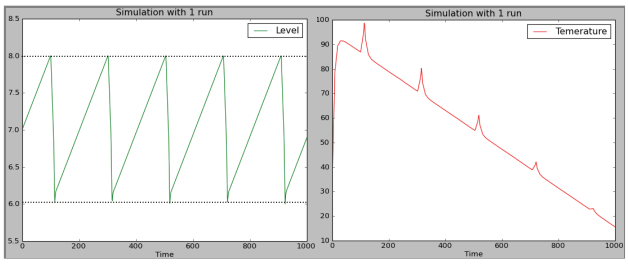


Figure 8 : Level and temperature evolution in one run and without any failure

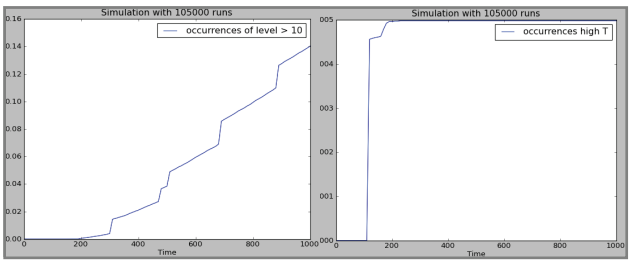


Figure 9 : Undesired events occurrences when failures only occur when operating

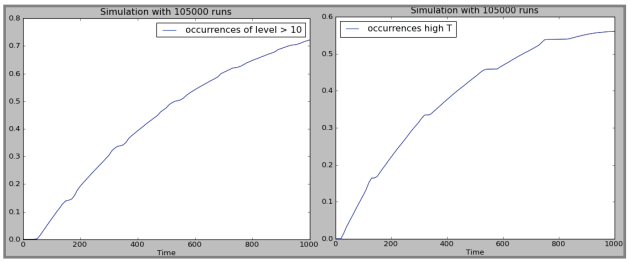


Figure 10 : Undesired events occurrences when failures also occur while waiting with a constant rate

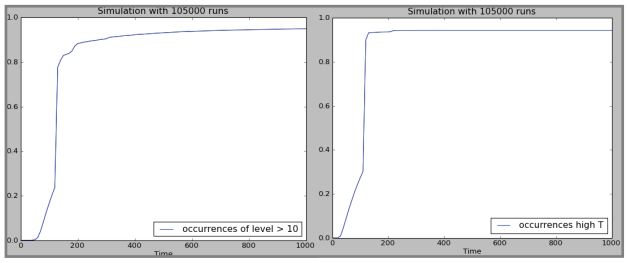


Figure 11 : Undesired events occurrences when failures also occur while waiting with a variable rate

4 CONCLUSION

In this paper we have described a new approach of using proven tools that are the PDMP framework and the stochastic hybrid automaton as a modeling formalism. This approach does not resort to building a global system automaton. Instead, as in a multi-agent system approach, it keeps the automata that model the behavior of a system component, encapsulated in an object which autonomously evolves while interacting with the other system components through messages passing. This approach, deals with the complexity of a system by focusing on the modeling of the behavior of its components and, thanks to the PDMP Controller,

it overcomes the additional difficulty introduced by hybrid systems that is the interaction between the random discrete events and deterministic continuous physical phenomena.

This paper illustrates the use of PyCATSHOO by assessing the dependability of a variant of the heated tank system and shows how easy it is even if the graphical user interface is still lacking. This variant differs from the original one. In particular, it does not assume that the state changes of the system devices are instantaneous. On the other hand it assumes that most of the failures only occur during the state changing of these devices.

Several other test cases have proven the validity of PyCATSHOO and, currently, it is used by a PhD student to model a realistic size system from the hydropower generation domain. However, PyCATSHOO is still under development. Lots of improvements are still undergoing such as introducing a Monte Carlo acceleration method and implementing an efficient way to identify the most probable faulty sequences that lead to the assessed system undesirable events.

References

[1] CNRA Working Group on Inspection Practices. Level 2 PASA methodology and severe accident management. Organisation for economic co-operation and development. OCDE/GD(97)198. www.oecd-nea.org/nsd/docs/1997/csni-r1997-11.pdf.

[2] V. Kopustinskas, J. Augutis, S. Rimkevicius. Dynamic reliability and risk assessment of the accident localization system of the Ignalina NPP RBMK-1500 reactor. Reliability Engineering and System Safety 87(2005) 77–87.

[3] M. Marseguerra, E. Zio, J. Devooghtb, P.E. Labeau. A concept paper on dynamic reliability via Monte Carlo simulation. Mathematics and Computers in Simulation 47 (1998) 371-382.

[4] M. H. A. Davis. Piecewise-deterministic Markov processes: a general class of non-diffusion. Journal of the Royal Statistical Society. vol. 46, no. 3, pp. 353–388, 1984.

[5] William LAIR. Modélisation dynamique de systèmes complexes pour le calcul de grandeurs fiabilistes et l'optimisation de la maintenance. http://tel.archives-ouvertes.fr/docs/00/64/46/94/PDF/These_WilliamLair_2011_publique.pdf.

[6] H Zhang K Gonzalez F Dufour and Y Dutuit. Piecewise Deterministic Markov Processes and Dynamic Reliability. Proceedings of the Institution of Mechanical Engineers Part O Journal of Risk and Reliability 222, 04 (2008) 545-551.

[7] L. G. Pola, M. L. Bujorianu, J. Lygeros, M. D. Di Benedetto. Stochastic Hybrid Models: An Overview with applications to Air Traffic Management. ADHS, Analysis and Design of Hybrid System, Saint-Malo, 2003.

[8] S.N.Strubbe, A.A. Julius, A.J. van der Schaft. CommunicatingPiecewise Deterministic Markov Processes. in Proc. IFAC Conf. Analysis and Design of Hybrid Systems 2003, pp 349-354, St. Malo, France.

[9] M.H.C. Everdij and H.A.P. Blom. Piecewise Deterministic Markov Processes represented by Dynamically Coloured Petri Nets Revised edition (2003). National Aerospace Laboratory NLR. NLR-TP-2000-428.

[10] G.A. Pérez Castañeda. Evaluation par simulation de la sûreté de fonctionnement de systèmes en contexte dynamique hybride. Rapport de Thèse de l'Institut National Polytechnique de Loraine. 30 mars 209.

[11] Christos G. Cassandras, Stéphane Lafortune. Introduction to Discrete Event Systems – Second Edition. 2008 Springer Science + Business Media, LLC.

[12] A. Agung Julius. Approximate Abstraction of Stochastic Hybrid Automata. Hybrid Systems: Computation and Control. LNCS 3927, pp. 318–332, 2006.

[13] T. A. Henzinger. The theory of hybrid automata. Proc. 11th Annual IEEE Symp. Logic in Computer Science, Los Alamitos, CA: IEEE Computer Society Press, 1996, pp. 278-292.

[14] Michael Wooldridge.Introduction to MultiAgent Systems. John Wiley & Sons, Ltd. 2001.

[15] http://www.python.org/, http://numpy.scipy.org/, http://www.scipy.org/, http://simpy.sourceforge.net/.