Integrated dynamic probabilistic safety assessments with PyCATSHOO: a new coupling approach

Hassane CHRAIBI[a], Jean-Christophe HOUDEBINE[b],
Claudia PICOCO[c], Valentin RYCHKOV[d]
[a] EDF, Palaiseau, France, hassane.chraibi@edf.fr
[b] ARISTE, Antony, France, jean-christophe.houdebine@wanadoo.fr
[c] EDF, Palaiseau, France, claudia.picoco@edf.fr
[d] EDF, Palaiseau, France, valentin.rychkov@edf.fr

Abstract: Integrated dynamic probabilistic safety assessment (IDPSA) approaches provide a valuable complement to the PSA classic methods that no longer needs to be justified. These hybrid approaches gather in the same model stochastic discrete events behavior and deterministic and time-dependent one which account for physical phenomena. However, these approaches are still facing several challenges such as modeling complexity, computational costs, data availability, post-processing difficulties etc. EDF contributes to meeting these challenges by developing the PyCATSHOO tool, which, among others, addresses the modeling complexity and calculation costs. The improvement effort of this tool is still ongoing and focuses on another challenge, namely the coupling methods between models that deal with discrete stochastic aspects and physical codes. In most experiments conducted to date, this coupling has been carried out thanks to ad hoc solutions and required a significant effort. However, a solution exists which could benefit IDPSA models. This solution is the FMI (Functional Mockup Interface) standard widely used in 0D/1D physical simulations. We have recently integrated this standard into PyCATSHOO. This article reports on this integration and gives an illustration based on the well-known Heated Tank system.

1. INTRODUCTION

The need to integrate physical phenomena in probabilistic safety assessment models is nowadays an obvious fact in a wide range of cases and several methods were implemented to ensure such an integration. As shown in a survey form [1], these methods are mainly based on extensions of fault trees and event trees' formalisms. Most of them are variants of dynamic event trees and take the time into account as a discrete magnitude.

As for continuous-time methods, this survey, which was conducted in 2013, has mentioned two ones. The first one is the "continuous cell-to-cell-mapping" (CCCMT) detailed in [2]. This technique only accounts for instantaneous changes when, for instance, a set point is reached. The second one was developed by [3]. It introduced the notion of stimulus, which is prior to an action to be executed after a deterministic or a stochastic delay.

Since then, EDF has developed a method that contributes to lifting the current limitations of probabilistic safety approaches which integrate deterministic physical models. We implemented this method in the PyCATSHOO tool [4]. Besides being a modeling and probabilistic assessment tool for discrete systems, PyCATSHOO offers advanced functionalities for designing and simulating 0D/1D models. It also offers tools to facilitate the interaction between stochastic discrete behaviors and deterministic and continuous behaviors.

These models, entirely based on PyCATSHOO, can be very effective because of the close integration of these two types of behavior. However, it is necessary to allow the reusability of external physical models when they are already available.

As shown in [5], the use by PyCATSHOO of external physical models is already possible.
It benefits from all the features offered by PyCATSHOO, including the parallel simulation capability.

However, it requires a dedicated manual coupling which must comply with the working schema of the PyCATSHOO tool. That is why we have added to the functionalities offered by PyCATSHOO, the co-simulation capabilities that meet the FMI (Functional Mock-up Interface) standard.

In this paper, we will briefly review the theoretical foundations of the PyCATSHOO tool and their implementation. We will then review the main issues risen by coupling with physical models in a probabilistic evaluation.

In section 4, we will present the FMI standard and, in particular, its co-simulation variant.

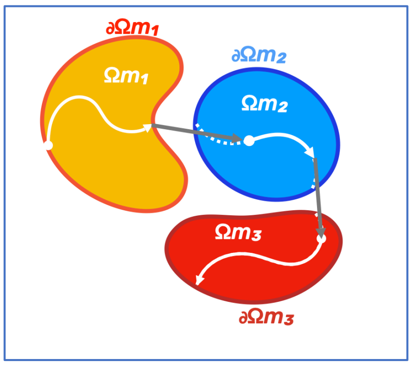
Before describing the implementation in PyCATSHOO of the co-simulation variant of the FMI standard, we will recall in section 5 the Heated Tank case study, which we have addressed in this paper. The Heated Tank system is now well known and has been used by many authors such as [6] & [7]. We have also simulated this system with PyCATSHOO through a model that integrates its discrete stochastic behavior and the physics equations that govern the deterministic evolution of the liquid level and the temperature in the reservoir.

In section 6, we will explain the approach implemented in PyCATSHOO, via the FMI standard to interact with a physical code in a probabilistic evaluation. We will explain in section 7 how the analyst uses an external model that encapsulates the physics equations that govern the level and temperature in the reservoir without explicitly calling the FMI standard functionalities that we have integrated into PyCATSHOO.

2. Introduction to PyCATSHOO

PyCATSHOO is based on the mathematical framework of Piecewise Deterministic Markovian Processes (PDMP) introduced by [8].

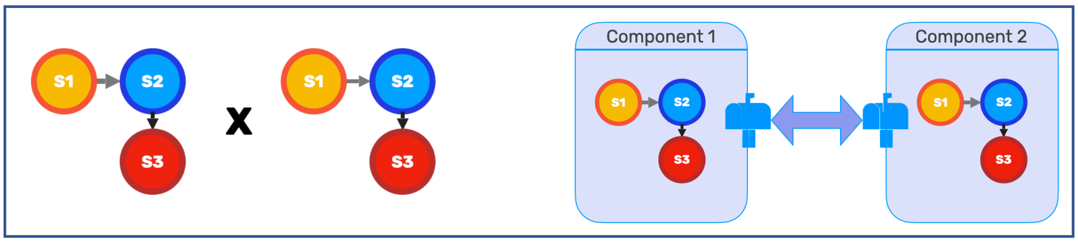
Figure 1: Spontaneous and forced jumps in a PDMP



The mathematical framework of PDMP is based on the decomposition of the modeled process into a set of operating modes of the studied system. It binds to each of these modes, a space Ωm_i, where continuous state variables evolve deterministically according to physical equations. These equations can be ordinary differentials, algebraic differentials, or explicit equations. The evolution of state variables can be interrupted for two reasons. The first is stochastic and could correspond to a failure in operation with a fixed or variable occurrence rate. This interruption causes a spontaneous jump into another randomly selected operating mode. The second type of interruption is due to the reaching, by the state variables, of $\partial\Omega m_i$, the boundary of the space in which they are supposed to evolve. This second type of interruption causes a forced jump into another randomly selected operating mode. The forced jump could correspond to reaching a threshold that causes the activation of an actuator.
For simple systems, one could assimilate the operating modes to states. Figure 1 could thus stand for a stochastic automaton because the transitions between its discrete states are also stochastic. This automaton also has a hybrid character. Indeed, as long as a discrete state is active, the continuous variables evolve deterministically.

However, this transposition cannot be applied as it is, to a complex system compound by numerous interacting components. Indeed, a system operating mode is, in this case, a combination of the states of its elementary components. This means that the system dynamic has to be modeled by a huge global automaton. Such an approach may lead to a combinatorial explosion. Besides, the construction of the global automaton is difficult to realize and hard to simulate by the current computing means. To overcome these difficulties, PyCATSHOO introduces the notion of hybrid distributed stochastic automata (ASHD) within autonomous and communicating objects, as illustrated in figure 2. The formal description of this approach is given in [9].

Figure 2: Distribution as an alternative to the composition of elementary object states



To summarize, the PyCATSHOO modeling of a system begins with the modeling of its components. The granularity of these components depends on the study purpose and the nature of available data. According to the PyCATSHOO approach, a component is a communicating autonomous actor. As illustrated in figure 3, an object is modeled by the following elements:

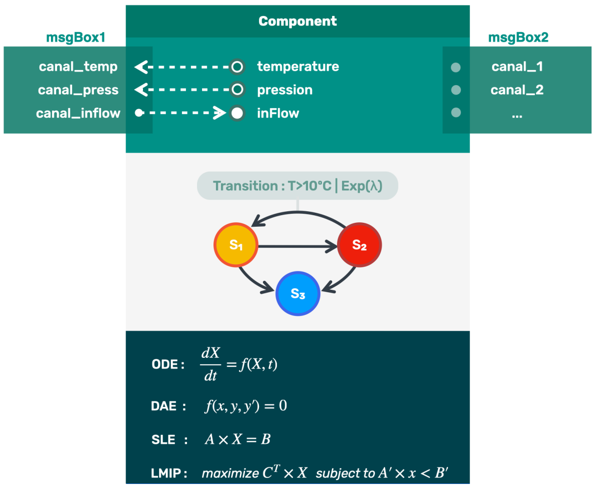
- Continuous or discrete state variables. Some of these variables can be readable by the rest of the system outside the component. According to PyCATSHOO vocabulary, these variables are called "intrinsic" variables.
- Variables that contain information coming from the rest of the system outside the object. These variables are called "references" in the PyCATSHOO vocabulary.
- Message boxes. They hold outgoing channels that expose the component's intrinsic variables to the outside world. They also hold incoming channels that feed the reference variables with information coming from the world outside the component.
- Hybrid stochastic automata which comprise states and transitions. The latter can be instantaneous deterministic, delayed deterministic, stochastic, or instantaneous with discrete probability distributions. Transitions can also be conditional on boolean expressions involving intrinsic and reference variables.
- Ordinary differential equations (ODE), algebraic differential equations (ADE) or systems of linear equations (SLE). These equations govern the evolution of continuous state variables. PyCATSHOO gathers these equations into a global system and ensures the simultaneous solving of the equations declared in all the system components.
- Linear constraints on some of the object variables and an objective function to be maximized by a Linear Mixed Integer Programming method (LMIP). PyCATSHOO gathers these constraints in a global optimization problem.

The above information is to be given in a declarative way. Ordinary differential equations can be declared by giving methods that calculate the derivatives of the variables. Their solving is synchronized with the rest of the system dynamic and PyCATSHOO handles it transparently.

Similarly, we declare transitions by specifying their starting and target states, their conditions as methods with a boolean return, and by giving their probability law.

Unlike approaches only dedicated to 0D/1D modeling and simulation, PyCATSHOO does not require the model developer to perform any explicit probabilistic computations, such as drawing jumping time by reversing cumulative probability function, etc.

Figure 3: A PyCATSHOO object



A procedural complement (callback methods) can be added to this information and associated with discrete events. This allows, for instance, to propagate the consequences of an automaton state change.

In computer terms, the components models are classes written in either Python or C++ language. In PyCATSHOO vocabulary, we call these classes a knowledge base (KB). A knowledge base can stand for a particular category of systems (Electrical networks, telecommunications, thermohydraulic, etc.). It can also implement a generic formalism, such as reliability diagrams.

A knowledge base can then be used to model a particular system. We do this modeling by creating objects as instances of these KB classes. The links between the message boxes of these objects stand for the system architecture.

3. Coupling with physical models in probabilistic assessments

In the panorama of dynamic approaches that we have cited above, we mainly find approaches that extend the formalism of event trees as in [10]. The most widely used of these approaches introduce variants that differ in the way the branching points are determined and in the calculating method of the undesired event probability. For example, we can notice the following variants:

- The temporal branching points correspond to intervals chosen by the analyst. Their probabilities are deduced from fragility curves using values of continuous state variables calculated by the physical model.
- The physical branching points correspond to thresholds (e.g., setpoints that should lead to a technical or human system action). Their occurrence time results from solving a crossing problem by the physical model.
- The branching occurs at instants randomly drawn according to failure or repair rates.
- A systematic exploration of the branching points helps discover the sequences that lead to the undesired event. Summing the probabilities of these sequences gives the probability of the undesired event as it is done in discrete dynamic event tree. In this variant, the combinatorial explosion is prevented by a minimum truncation threshold on the probability of a sequence.
- Alternatively, the probability of the undesired event can be calculated directly with Monte Carlo simulation as it is done in Monte Carlo dynamic event tree.

PyCATSHOO can also produce drivers that implement these features. Moreover, it offers the possibility of on-the-fly generation of branching points that can be due to deterministic behaviors from the physical model or stochastic behaviors driven by probability laws that can depend on the state variables of the physical model.

All these approaches have to take care of the following aspects of coupling with numerical models of physics:

- The first one is about saving the system state at a particular time step. Such a saving may increase the efficiency of the calculations. Indeed, a branching point can be visited by the solver several times in the same conditions. Here, it is worth not redo a calculation already performed.
- The second issue is related to the first one. It concerns the detection of threshold crossing by the system state variables. Indeed, this crossing detection requires a dichotomy and backtracking before resuming the simulation with a smaller time step than the previous one.

These two points are among the most important when coupling a probabilistic evaluation with a physical model of the studied system physics. This coupling is likely to require significant work. Besides, this work is often very dependent on the physical model. In case of a change of the latter, the coupling must be, at least partially, redone.

4. FMI Standard and co-simulation

In some drivers (e.g., RAVEN [11]), a proprietary Application Programming Interface (API) is provided to facilitate the coupling with a physical code and address the two issues described above. The coupling is then realized by implementing the API methods within a dedicated interface. It's the interface that allows communication between the driver and the physical code. Each coupling with a different physics code may demand the implementation of a new interface depending on the characteristics of the code itself.

We aimed to provide PyCATSHOO with a similar API that we based on the FMI standard to ensure that any physical code using FMI can be almost automatically "coupled" with a PyCATSHOO driver.

Before illustrating the use of FMI by PyCATSHOO, we give here its key principles extracted from the specification document [12] of the version 2.02.

The FMI (Functional Mock-up Interface) standard was developed by the European ITEA2 MODELISAR project from 2008 to 2011[*]. It has been evolving since 2011 within the framework of the Modelica Association's permanent project. The FMI standard is presented as a specification and an application programming interface (API) that an executable called FMU (Functional Mock-up Unit) must implement. The latter will play the role of all or part of the physical model to be used in a dynamic PSA. A physical model can thus be natively developed as an FMU, i.e., according to FMI specifications. It can also be wrapped by a software that implements these specifications. Here, the wrapper is to link the functionalities of the physical model and the FMI API. In both cases, the internal structure of the FMU data and its algorithms are not exposed and can evolve without questioning their usability. Indeed, an FMU is aimed to be used by a master, a simulation environment, which can drive several FMUs thanks to the API of the FMI standard.

According to the FMI standard, the FMUs can be of two types:

- The FMUs for model exchange that do not have solvers. They delegate to the simulation driver the complete control of the simulation and the solving of the model equations. They have two drawbacks:
    a. These FMUs are not autonomous since they do not hold a solver; therefore, we cannot validate them independently without a driver.
    b. The pilot is complex to build since it is in charge of solving all the equations.

- FMUs for co-simulation that have their own solvers and their own integration steps. This is the type of FMU that we will describe and that we have implemented in PyCATSHOO.

[*]https://itea4.org/publication/download/910-modelisar-seizing-the-high-ground.pdf

5. Assessment of the Heated Tank system with PyCATSHOO

5.1. System description

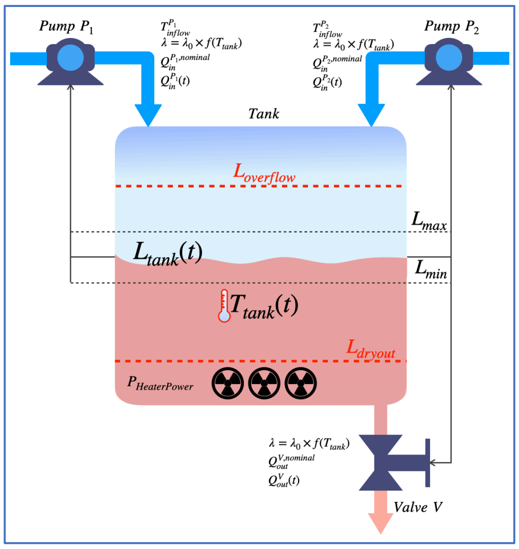
The Heated Tank system comprises the following components:

- A Tank that holds a liquid.
- A Heater immersed in the liquid held by the tank. It emits a constant thermal power.
- Two Pumps that operate in on/off mode. When opened, they bring a cold liquid to the tank with a constant nominal flow rate to cool the tank and avoid its drying out.
- A Valve to prevent the tank from overflowing.

The pumps are regulated according to the liquid level $L_{tank}(t)$ in the tank. They open when $L_{tank}(t) < L_{min}$ and close when $L_{tank}(t) > L_{max}$.

The valve is also regulated according to the liquid level in the tank. It opens when $L_{tank}(t) > L_{max}$ and closes when $L_{tank}(t) < L_{min}$.

Figure 4: Schematic diagram of the Heated Tank



The pumps and the valve are subject to two failure modes: "stuck open" and "stuck closed". The occurrence rate of these failures depends on the temperature $T_{tank}(t)$ in the tank according to the following equation:

$$\lambda = \lambda_0[b_1 e^{b_c(T_{tank}-20)} + b_2 e^{-b_d(T_{tank}-20)}] \quad (1)$$

As in the original version of this case study, we did not consider any repairs†.

The objective of this case study is to evaluate the evolution over time of the probabilities that the level in the Tank goes outside the interval $[L_{dryout}, L_{overflow}]$ and that the temperature of the Tank goes beyond the $T_{burning}$ limit.

† In the framework of a benchmark about stochastic model simulation [13], we introduced repairs in another version of the Heated Tank system to lower probabilities of the undesired events.

5.2. Integrated modeling with PyCATSHOO

As stated in section 2, the first modeling step in the PyCATSHOO modeling approach is to produce a behavioral model for each of the Tank, Pump, Valve and Heater components. We implemented these models in 4 different classes. The source code of these classes is given in Python language on pycatshoo.org. Here, we will just describe the main PyCATSHOO modeling concepts and the concepts that will have to be changed when using an FMU. So, we will only give some excerpts of the Pump and Tank classes.

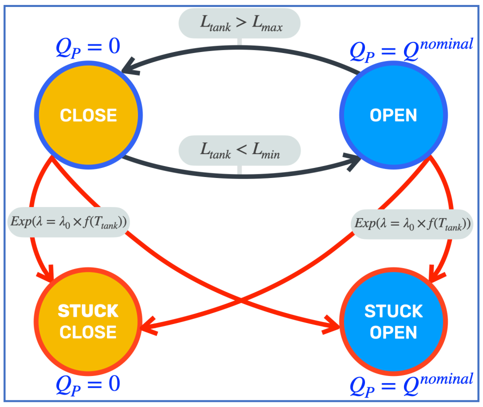
5.2.1 Model's excerpts the Pump class

We declare the state variables and parameters in the following way:

```
self.p_nominalFlow = self.addVariable("nominalFlow", Pyc.TVarType.t_double, 1.5)
self.v_flow        = self.addVariable("flow"       , Pyc.TV"rType.t_double, 1.5)
self.p_lambda0     = self.addVariable("lambda0"    , Pyc.TVarType.t_double, 0.001)
self.v_lambda      = self.addVariable("lambda"     , Pyc.TVarType.t_double, 0.001)
```

Figure 5 pictures an automaton that stands for the dynamic behavior of the Pump.

Figure 5: The automaton of the class Pump



The python source that stands for this automaton is:

```
self.a_automaton       = self.addAutomaton("FuncAutomation")
self.s_openState       = self.addState("FuncAutomation", "OPEN"      , 0)
self.s_closeState      = self.addState("FuncAutomation", "CLOSE"     , 1)
self.s_stuckOpenState  = self.addState("FuncAutomation", "STUCKOPEN" , 2)
self.s_stuckCloseState = self.addState("FuncAutomation", "STUCKCLOSE", 3)
self.a_automaton.setInitState(self.s_openState)
self.a_automaton.addSensitiveMethod("updatFlow", self.updateFlow)
```

This excerpt also gives the default initial state and the name of the callback method that changes the value of the pump output flow at every jump of the automaton.
The following excerpt shows how to create transitions:

```
trans_o2c  = self.s_openState.addTransition("OPEN_to_CLOSE")
trans_o2c.setCondition("closeCondition", self.closeCondition)
trans_o2c.addTarget(self.s_closeState, Pyc.TTransType.trans)
……
trans_o2so = self.s_openState.addTransition("OPEN_to_STUCKOPEN")
trans_o2so.setDistLaw(Pyc.IDistLaw.newLaw(self, Pyc.TLawType.expo, self.v_lambda))
trans_o2so.addTarget(self.s_stuckOpenState, Pyc.TTransType.fault)
trans_o2so.setModifiable(Pyc.TModificationMode.continuous_modification)
```

The first transition created in this excerpt is deterministic. PyCATSHOO fires this transition when the boolean method ***closeCondition*** returns *True*. The second transition is stochastic. PyCATSHOO fires it at a time randomly drawn according to an exponential law with a rate given by λ. Here, λ is a continuous variable that evolves over time. So, we have to ask PyCATSHOO to manage this variable by a Piecewise-Deterministic-Markovian-Process (PDMP).

```
self.addPDMPManager           ("pdmpManager")
self.addPDMPExplicitVariable  ("pdmpManager" , self.v_lambda)
self.addPDMPEquationMethod    ("pdmpManager" , "odeMethod",self.odeMethod)
self.addPDMPWatchedTransition ("pdmpManager" , trans_o2c)
```

In this excerpt, we ask the PDMP to manage the continuous variable lambda that evolves according to the explicit equation (1) implemented in the method ***odeMethod***. We also ask the PDMP to watch the transition ***trans_o2c*** (open to close). Indeed, this transition stands for the crossing of a threshold. When PyCATSHOO detects this crossing, it stops the equations' integration and performs the appropriate PDMP mode changes.

Unlike tools or languages only dedicated to 0D/1D modelling and simulation such as Modelica, these declarations are sufficient for PyCATSHOO to handle the integrations and draw the transition instants in a coordinated way with the rest of the modelled behavior.

5.2.2 Model excerpts from the Tank class

The two main continuous state variables of the Tank class are the temperature and the liquid level.

```
self.v_temperature = self.addVariable("temperature", Pyc.TVarType.t_double, 20.)
self.v_level       = self.addVariable("level"      , Pyc.TVarType.t_double,  7.)
```

The PDMP must manage these variables, but this time through a system of ordinary differential equations.

```
self.addPDMPEquationMethod       ("pdmpManager","odeMethod", self.odeMethod)
self.addPDMPODEVariable          ("pdmpManager", self.v_level)
self.addPDMPODEVariable          ("pdmpManager", self.v_temperature)
```

The method ***odeMethod*** implements the derivative expressions of the two variables:

```
def odeMethod(self):
    iFlow    = self.r_inFlow.sumValue()
    oFlow    = self.r_outFlow.sumValue()

    self.v_level.setDvdtODE((iFlow - oFlow) / self.p_area.dValue())

    sumiFiT = 0
    for i in range(self.r_inFlow.nbCnx()):
        sumiFiT = sumiFiT + self.r_inFlow.dValue(i) * \
                            (self.r_inFlowtemperature.dValue(i) - self.v_temperature.value())

    self.v_temperature.setDvdtODE(
        (sumiFiT + self.r_power.dValue(0)) / (self.v_level.dValue() * self.p_area.dValue()))
```

PyCATSHOO will then use one of its solvers to monitor the evolution over time of all the system variables in coordination with the deterministic and stochastic discrete events.

6. Dialog principles between PyCATSHOO and an FMU

Currently, PyCATSHOO only implements the co-simulation protocol of the FMI standard. So, we can use PyCATSHOO to develop FMUs according to this protocol and generate, mostly automatically, an FMU descriptor. We will not describe these functionalities in this paper. We will focus on the situation where PyCATSHOO is a driver that uses external FMUs in co-simulation mode.

In a simplistic co-simulation approach, the use of such an FMU can consist in performing a simulation by stopping at pre-defined times to monitor the values of some variables of the physical model. We used the approach on the original integrated Heated Tank model we described in section 5. We developed an external driver which merely retrieves, from the FMU, the values of the variables of interest at specific time steps. This approach can be helpful when the modelling tools used do not allow for efficient operation in a Monte Carlo simulation framework due, for instance, to slow computations or inability to parallelize simulations by distributing them across cluster nodes.

In a more complex approach, the driver accounts for the events that occur in the FMUs. An FMU is then supposed to stop its simulation when an event occurs before the end of the time step it has been asked to simulate. This approach, described in [14], requires that the driver respond to the events presented by the FMU. This approach has several constraints:

- The events modeled in the FMU must match those expected by the driver.
- The breakpoint of the FMU must have the properties expected by the driver in terms of accuracy and crossing conditions.
- With parallel management of several FMUs, it is necessary that these FMUs manage backward steps of their simulation so that the driver can synchronize them at the first stopping point of an FMU.

The approach we have adopted is different. It ensures that the FMU is not designed specifically for the driver. The detection of the events that the driver must handle is entirely done by the driver itself. The FMU can thus be a simple simulation model whose future states cannot be predicted. In this context, the driver detects a threshold crossing by going beyond the threshold and searching for the time of crossing, starting from the previous time step. Thus, the only constraint that this approach imposes on the FMU is that it must be able to move back to previous time steps whose state must have been saved.

The implementation in PyCATSHOO of functionalities that make it able to use co-simulation FMUs has been done to minimize the effort to be provided by the driver designer. The FMU is loaded by PyCATSHOO thanks to a simple ***addFMU(pathFMU)*** command that performs all the loading operations and controls and places the FMU in an initialization situation. The loaded FMUs are then handled by PyCATSHOO similarly to other standards PyCATSHOO components. They can have a list of variables identical to the standard PyCATSHOO variables.

The driver designer can then use all the PyCATSHOO tools to change the initial values of variables, add callback methods that update discrete variables, request indicator calculations, request monitoring, etc.

The simulation of a complete sequence by a PyCATSHOO model that uses FMUs remains transparent to the driver designer.

When the driver does not have to manage the threshold of the FMU variables, this simulation performs the following tasks:

- Starting a sequence: the FMUs are taken out of the initialization mode,
- For each PyCATSHOO time interval before the next transition time calculated as in f:
    c. Integration of the differential equations until the desired time or a threshold crossing on the variables the driver is in charge of.
    d. Advancement of the FMUs to the current time.
    e. Application of active transitions.
    f. Management of callback methods.
    g. Control of sequence termination conditions.
    h. Calculation of the next transition time.
    i. If the conditions for stopping the sequence are not met, resume in a.
- End of the sequence: the FMUs are reset and put back into the initialization mode, where the variables take their initial values.

When the driver has to manage threshold crossings involving FMU variables, the latter must have been added to the driver's current solver. The driver is to detect the realization of the crossing condition and of searching for the crossing point. This search is done by dichotomy, by asking the FMU in charge of the variable concerned by the crossing to resume the simulation, with a smaller step, from the previous time step. As expected by PyCATSHOO, the integration is then stopped immediately after the crossing and after the synchronization of all the FMUs. This synchronization equally covers the driver solver if there is one. Then, PyCATSHOO resumes its tasks by controlling the sequence termination conditions and calculating the next transition time.

7. Heated Tank: First use of the FMI standard with PyCATSHOO

7.1 Implementation details

In the following illustration, the solving of the differential equations governing the evolution of the tank state variables are handled by an external FMU. This FMU will then have the behavior of the Tank class and the driver will be in charge of the rest of the modelling. The driver will hold the control of discrete deterministic and stochastic events and will host the Pump, Valve, and Heater classes. As for the class Tank, the driver will host a proxy of its actual implementation in the FMU. So, we will describe the changes we made in the Tank class to transform it into such a proxy.

- ***Changes in variable declarations:***
  The temperature and level variables in the Tank proxy will be bound to those of the FMU which must be loaded first:

```
self.fmu           = self.system().addFMU("fmufolder/Tank.fmu", "MyFMU")
self.v_temperature = self.fmu.variable("FMUTank.Temperature")
self.v_level       = self.fmu.variable("FMUTank.Level")
```

  The input FMU variables must also be bound to the proxy variables:

```
self.v_power        = self.fmu.variable("FMUTank.Power")
```

- ***Changes in the PDMP of the driver***
  We inform the local PDMP of driver that there is an FMU in charge of all or part of the continuous variables.

```
pdmp.addFMU( self.system().FMU("MyFMU"))
```

  We bind to the PDMP of the driver a method that PyCATSHOO automatically calls at the beginning of every deterministic piece of the PDMP timeline. The role of this method is to propagate the values of the discrete variables to the FMUs.

```
self.addPDMPBeginMethod("pdmpManager", "beginMethod",  self.beginMethod)
```

  Hereafter an example of the implementation of this method:

```
 def beginMethod(self):
     ……
     self.v_power.setDValue(self.r_power.sumValue())
     ……
```

7.1 Outcomes

As expected, this implementation gave the same results as the monolithic PyCATSHOO model, i.e., without FMU. However, the computation times were not identical. Using FMUs noticeably slowed down the simulations. The reason is the state saving tasks that were not required in the monolithic version of the Heated Tank model.

Figure 6: Same results with and without FMU

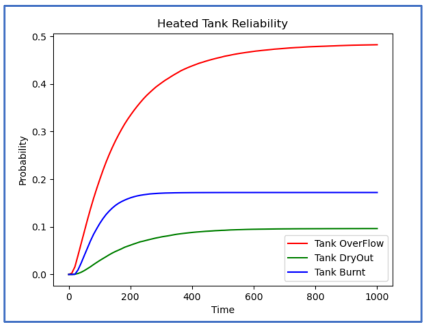


Table 1 gives the computation times in seconds for different numbers of simulated sequences in monolithic modeling and in modeling with FMU. We performed these simulations with a compiled version of the model written in C++. We dispatched them on ten execution threads on a laptop with a six-core Intel(R) Core (TM) i7-8750H processor and 32 GB of memory.

Table 1: Computation times for different numbers of simulated sequences

	10^3 sequences	**10^4 sequences**	**10^5 sequences**	**10^6 sequences**
Modeling with FMU	1.7 s	7.7 s	69 s	727 s
Monolithic modeling	1.2 s	2.20 s	16.4 s	180 s

Note that, in both cases, the computation time evolves linearly according to the number of simulated sequences.

8. CONCLUSION

This study is an illustration of the using capabilities of the FMI standard by PyCATSHOO. It has highlighted a transparent way for the driver designer to use the FMI standard API.

The approach we adopted for implementing the FMI standard in PyCATSHOO, helps keeping the driver in charge of managing the events that may correspond to the branching times, whatever the nature of these events. This leads to the most important advantage of this approach, namely that implementing an FMU wrapper does not require deep modifications to the physical models. The only requirements are to ensure the model's ability to save its state at a computation instant and to resume computation at a previous step.

The increase in computation time observed in the illustration outcomes when using FMUs, is not systematic. For the Heated Tank study, the tasks of state saving, which is not required in the monolithic original model, explain it. But, when it happens, this drawback is compensated by more modularity and by the ability to replace physical models with minimal effort.

However, in the study presented in this paper, this increase remains acceptable, especially since it evolves perfectly linearly with the number of simulated sequences.

The system that we have studied in this article is fairly representative of the functionalities that the FMI standard should offer to the task of coupling physical model in probabilistic assessments. But our perspectives are to carry out industrial studies based on physical models wrapped in FMUs.

So far, our works about FMI only covered the co-simulation protocol, which offers a relevant solution to the model coupling tasks. But we still have to evaluate the interest of the second way of implementing the FMI standard, namely the model exchange protocol.

References

[1] T. Aldemir. *"A survey of dynamic methodologies for probabilistic safety assessment of nuclear power plants",* Annals of Nuclear Energy, 52, pp.113-124, (2013).
[2] B. Tombuyses and T. Aldemir. "*Continuous Cell-To-Cell Mapping"*. Journal of Sound and Vibration, 202(3), pp395-415, (1997).
[3] P. E. Labeau and J. M. Izquierdo. "*Modeling PSA Problems—I: The Stimulus-Driven Theory of Probabilistic Dynamics"*, NUCLEAR SCIENCE AND ENGINEERING, 150, pp 115–139, (2005).

[4] H. Chraibi, J.C. Houdebine, A. Sibler.*"PyCATSHOO: Toward a new platform dedicated to dynamic reliability assessments of hybrid systems",* PSAM13 (2016).
[5] N. Brînzei, C. Duval, H. Chraibi, M. Hassanaly. *"Modeling the Consequences of Feared Event by Stochastic Hybrid Automata",* Proceedings of the 30th European Safety and Reliability Conference and the 15th Probabilistic Safety Assessment and Management Conference, pp 4875-4882, (2020).
[6] Zhang H., De Saporta B., Dufour F., Deleuze G.*, "Dynamic Reliability by Using Simulink and Stateflow"*, Chemical Engineering Transactions, 33, 529-534, (2013).
[7] M. Bouissou, X. de Bossoreille, "*From Modelica models to dependability analysis"*, IFAC PapersOnLine, Volume 48, Issue 7, Pages 37-43, (2015).
[8] M. H. A. Davis. *"Piecewise-Deterministic Markov Processes: A general class of non-diffusion stochastic models"*, J. R. Stat. Soc. B 46:353-388, (1984).
[9] L. Desgeorges, P.Y. Piriou, T. Lemattre, H. Chraibi. "Formalism and semantics of PyCATSHOO: A simulator of distributed stochastic hybrid automata". RESS, Volume 208, (April 2021), 107384.
[10] C. Picoco, T. Aldemir, V. Rychkov, A. Alfonsi, D. Mandelli, C. Rabiti. *"Coupling of RAVEN and MAAP5 for the Dynamic Event Tree analysis of Nuclear Power Plants".* Proceedings of the 27th European Safety and Reliability Conference, June 18-22, PORTOROZ SLOVENIA (2017).
[11] C. Rabiti, A. Alfonsi, J. Cogliati, D. Mandelli, R. Kinoshita, S. Sen, C. Wang, P. W. Talbot, D. P. Maljovec, M. G. Abdo, "*RAVEN User Manual"*, INL/EXT-15-34123 (2022)
[12] Modelica Association*. "Functional Mock-up Interface for Model Exchange and Co-Simulation"*, Document version 2.02 (2020). https://github.com/modelica/fmi-standard/releases/download/v2.0.3/FMI-Specification-2.0.3.pdf.
[13] A. Abate, H.A.P Blom, M. Bouissou, N. Cauchi, H. Chraibi, J. Delicaris, S. Haesaert, A. Hartmanns, H. Ma and More Authors. *"ARCH-COMP21 Category Report: Stochastic Models".* In G. Frehse, & M. Althoff (Eds.), 8th International Workshop on Applied Verication of Continuous and Hybrid Systems, EPiC Series in Computing, 80, pp. 55-89, (2021).
[14] F. Cremona, M. Lohstroh, D. Broman, M. Di Natale, E. A. Lee, and S. Tripakis*. "Step revision in hybrid Co-simulation with FMI"*, conference paper (2016).