Calcul de fiabilité dynamique par simulation de Monte-Carlo : gains apportés par la cosimulation

Jean-Philippe Tavella, Bouissou Marc, Philippe Carer

Institut pour la Maîtrise des Risques
Sûreté de Fonctionnement - Management - Cindyniques

Calcul de fiabilité dynamique par simulation de Monte-Carlo : gains apportés par la cosimulation

TAVELLA Jean-Philippe
Département SYSTEME
EDF Lab Paris Saclay
7 Boulevard Gaspard Monge
91120 Palaiseau
jean-philippe.tavella@edf.fr

BOUISSOU Marc
Département PERICLES
EDF Lab Paris Saclay
7 Boulevard Gaspard Monge
91120 Palaiseau
marc.bouissou@edf.fr

CARER Philippe
Département SYSTEME
EDF Lab Paris Saclay
7 Boulevard Gaspard Monge
91120 Palaiseau
philippe.carer@edf.fr

***Résumé* —** Cet article présente l'intérêt de la cosimulation pour les études de fiabilité de système. On rappelle tout d'abord les difficultés rencontrées dans certains cas d'étude de fiabilité comme le « heated tank » qui nécessite de combiner modélisation fonctionnelle et simulation de Monte Carlo pour générer les pannes (fiabilité hybride). A partir d'un modèle sous Modelica du « heated tank » un fichier FMU est généré. La cosimulation avec les outils DNG permet dans un temps acceptable de faire de nombreuses simulations de fiabilité pour converger vers des résultats statiquement significatifs. On montre un exemple de 100.000 instances sur le cas du « heated tank » pour un temps de calcul de 175 secondes.

Mots-clefs — fiabilité hybride, Modelica, Heated tank , simulation de Monte carlo, Co-simulation, FMU, DNG

***Abstract*—** This paper is about the use of cosimulation computation method to solve difficult system reliability analysis. The used case of the "heated tank" is presented. It is necessary for this example of reliability analysis to use functional model with Modelica and Monte Carlo simulation to generate the failure of the component of the system. Based on the Modelica model of the "heated tank" a FMU file is created. The DNG tool uses the FMU file to do cosimulation. An example with "heated tank" shows the result obtained through a 100.000 instances in 175 seconds.

Keywords — Hybrid Reliability, Modelica, Heated tank, Monte Carlo simulation, Co-simulation, FMU, DNG

I. INTRODUCTION

Classiquement dans les études de fiabilité et disponibilité des systèmes, il est d'usage de considérer chaque système comme architecturé en composants, chaque composant possédant en général deux états $enMarche$ ou $enPanne$.
On affecte traditionnellement à ces composants un taux de panne λ et un taux de réparation μ. On prend aussi en compte les pertes simultanées de composants en redondance λ_{DCC} de DCC (Défaillance de Causes Communes). Et ensuite on calcule la probabilité que le système reste en fonctionnement sur une durée de mission donnée (fiabilité), ou fonctionne à un instant donné (disponibilité).
Il existe depuis longtemps des outils comme KB3-Figseq [8] pour effectuer ces calculs de fiabilité et disponibilité système basés sur la méthode des graphes de Markov. Avec cependant des limitations dans certains cas, comme par exemple lorsque :

- Les taux de défaillance des composants ne sont plus constants, mais paramétrés par des grandeurs physiques qui sont susceptibles de varier au cours de la mission du système ;
- Les évènements redoutés ne sont plus la panne totale du système, mais le fait qu'une grandeur physique ne reste pas en-dessous ou au-dessus d'une valeur admissible (comme la température d'une installation ou la valeur du courant traversant un conducteur par exemple) ou qu'elle sorte de certaines plages de fonctionnement (comme la tension dans les réseaux de distribution HTA/BT) ;
- Les causes physiques à l'origine des DCC de composants redondants doivent être prises en compte.

Ces différents cas nécessitent de réaliser des études dites de « fiabilité dynamique » qui couplent la modélisation physique d'un système réel avec une génération de pannes/réparations de ses composants afin de calculer les grandeurs fiabilistes par la méthode de simulation de Monte Carlo.
Dans ce contexte, il se trouve que les modèles de la physique qui, par nature dépendent du temps, sont faciles à construire avec un langage orienté équations comme Modelica [1]. Des outils logiciels comme Dymola (commercialisé par Dassault Systèmes) ou OpenModelica (disponible en open-source auprès de l'OSMC) permettent d'éditer des modèles Modelica et d'en simuler le comportement dynamique sans grande difficulté.
Cependant, les systèmes réels sont toujours plus complexes que la physique sous-jacente de leur fonctionnement nominal, qui reste par nature déterministe. Ces systèmes ne peuvent ni se représenter uniquement par une modélisation déterministe de variables d'état physiques, ni être réduits à une simple représentation stochastique discrète de la réalité.

Pour estimer les probabilités d'événements indésirables pouvant conduire à l'échec de la mission des systèmes réels, il faut en passer par des modèles hybrides, c'est-à-dire mêlant des variables physiques (souvent continues) avec des variables aléatoires (souvent discrètes).
Modelica apporte une solution pour la construction de modèles hybrides stochastiques, comme cela a déjà été montré dans des études antérieures (voir [2] et [3]). Cependant, ces modèles peuvent se heurter à une difficulté de temps de calcul, en particulier avec la **méthode de Monte-Carlo qui se base** sur un nombre important de simulations, chacune étant une réalisation aléatoire des trajectoires des différentes variables du modèle.

II. OBJECTIF DE L'ARTICLE

L'objectif de cet article est de démontrer l'intérêt de la technique de cosimulation pour l'étude des systèmes hybrides en fiabilité dynamique. Il est organisé comme suit :

- La section III rappelle l'intérêt du langage Modelica pour la résolution des systèmes d'équations algébro-différentielles et la génération de nombres aléatoires. Elle précise également ce qu'est la cosimulation, en particulier dans le contexte du standard de couplage FMI [4]. Enfin, elle donne un rapide état de l'art des modèles et outils qui permettent de faire des calculs de fiabilité dynamique.
- La section IV expose le cas d'étude choisi, ainsi que les outils retenus pour sa mise en œuvre.
- La section V présente les résultats et les performances obtenus avec la cosimulation.
- La section VI propose quelques pistes d'amélioration pour l'avenir.

III. ETAT DE L'ART

A. *Apport de Modelica dans les calculs de fiabilité dynamique*

Pour mémoire, dans l'étude probabiliste des durées de vie, les objets mathématiques d'intérêt sont les suivants :

- Une fonction de répartition F de la variable T, définie sur R^+ par : $F(t) = P(T \leq t)$.
  $F(t)$ est la probabilité que le composant tombe en panne avant la date t. C'est une fonction croissante, égale à 0 quand $t = 0$ et qui vaut 1 au bout d'une durée infinie.
- De manière complémentaire, la fonction de survie $\bar{F}$ est définie par $\bar{F}(t) = P(T > t)$. $\bar{F}(t)$ peut aussi s'écrire : $\bar{F}(t) = 1 - F(t)$.
  Elle est la probabilité que le composant fonctionne encore à la date t. C'est une fonction décroissante, égale à 1 quand $t = 0$ et qui vaut 0 au bout d'une durée infinie.
- Remarque : dans le domaine biomédical ou pharmaceutique $\bar{F}(t)$ est appelée habituellement fonction de survie $S(t)$ soit $\bar{F}(t) = S(t)$ et dans le domaine industriel elle est appelée fiabilité $R(t)$ (reliability en anglais) soit $\bar{F}(t) = R(t)$.
- Le taux de hasard $\lambda(t)$ lié à la fonction de répartition $F(t)$ est définie par :
  $\lambda(t) = \frac{F'(t)}{\bar{F}(t)}$, ce qui revient à l'équation différentielle :
  $\frac{dF(t)}{dt} = (1 - F(t)) * \lambda(t)$.

Quand le taux de hasard est indépendant du temps, on peut poser : $\lambda(t) = Cte = \lambda$. Et l'équation différentielle présente une solution analytique simple : la loi exponentielle. Il est alors facile de générer les dates des défaillances, à partir d'un générateur de nombres pseudo aléatoires équirépartis sur l'intervalle [0,1].
Dans le cas général où le taux de hasard dépend du temps, une solution proposée dans la publication [2] est de remonter à la résolution de l'équation différentielle avec un taux de hasard $\lambda(t)$ variable pour générer les temps d'occurrence de défaillance.
Et le langage Modelica se prête bien à une intégration dans le temps, par exemple en adoptant l'écriture suivante :
$der(F) = (1 - F) * hazardRate;$
Dans cette équation, la variable $hazardRate$ représente le taux de hasard $\lambda(t)$.

B. *Apport de la cosimulation pour les études basées sur la méthode de Monte-Carlo*

On appelle cosimulation le fait de faire fonctionner simultanément deux ou plusieurs simulateurs, qui échangent des données via une interface normalisée, à une cadence déterminée par le pas de cosimulation.
Le principal intérêt de la cosimulation réside dans le facteur d'accélération des performances potentiellement offert par la parallélisation des calculs. Ce gain est maximal à condition de répartir les calculs dans des composants faiblement couplés entre eux (pour minimiser les attentes et les temps d'échange des données) et si possible équilibrés en termes de charge CPU.
Si la cosimulation utilise un standard de couplage tel que le FMI [4], les composants qui forment la cosimulation sont appelés des FMU (Functional Mock-up Units) et les échanges entre les FMU d'un système sont normalisés.
Dans le cas d'une simulation de Monte Carlo, c'est le même système réel qui fait l'objet d'un grand nombre de tirages (chacun comportant sa part d'aléas) de sorte que les différentes réplications de ce système sont totalement indépendantes, ce qui simplifie la situation en termes de couplage.
L'outil open-source DNG [5] développé par EDF R&D permet de construire graphiquement un schéma de calcul basé sur des FMU connectées entre elles par des canaux de communication portant chacun une liste de variables à échanger. Les FMU gérées par DNG doivent être conformes à la dernière version officielle du standard de cosimulation (FMI for CoSimulation 2.0, voir [4]).

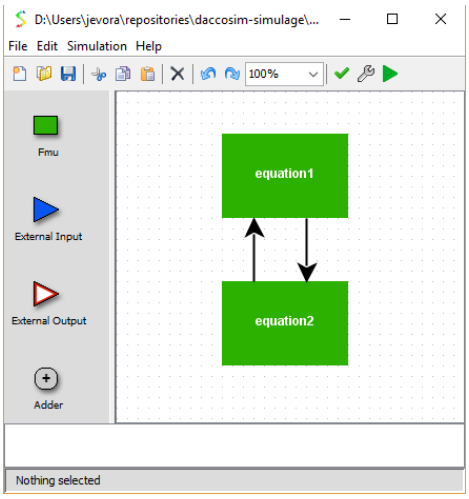

Fig.1 : Editeur graphique de l'outil DNG

Le noyau de calcul de DNG peut être sollicité directement en mode batch grâce à un fichier de script d'extension $.dng$ décrivant textuellement un schéma de calcul et ses paramètres d'exécution. La figure 2 est un exemple de script $.dng$. Ce script instancie 50.000 fois la même FMU physique, définit les variables d'intérêt de chaque instance logique et précise quelques paramètres de la cosimulation (notamment un temps de simulation de 1.000 unités de temps) :

```
FMU heatedTank1 "fmu/heatedTank.fmu"
Variable heatedTank1 i Integer discrete
Variable heatedTank1 tank.failure_code Integer discrete
Variable heatedTank1 tank.end_time Real discrete
                              ...
FMU heatedTank50000 "fmu/heatedTank.fmu"
Variable heatedTank50000 i Integer discrete
Variable heatedTank50000 tank.failure_code Integer discrete
Variable heatedTank50000 tank.end_time Real discrete
InitialValue heatedTank50000.i 50000 beforeInit

Log heatedTank1.tank.failure_code heatedTank1.tank.end_time
                              ...
Log heatedTank50000.tank.failure_code heatedTank50000.tank.end_time
Export , . 0.0
IterativePropagationInitializer 20 1.0E-5
ConstantStepper 1000.0
Simulation 0.0 1000.0
```

Fig.2 : Exemple de script de cosimulation pour DNG

Toutes les réplications d'une cosimulation de Monte Carlo sont théoriquement calculables en parallèle, avec un degré de parallélisme à régler en fonction de la plateforme informatique utilisée pour effectuer les calculs. Cette base hardware peut aller d'une simple machine multicœurs, à un cluster de calcul composé de plusieurs machines physiques en réseau.
En compressant le fichier de script $.dng$ avec un répertoire contenant toutes les FMU physiques (une seule dans notre étude), on obtient un fichier d'extension $.dngx$ directement exécutable par DNG.
Il est également possible de définir des campagnes de tests composées de plusieurs cosimulations lancées en série. La figure 3 est un exemple de script $.dcs$. Ce script lance en série deux fois la même cosimulation décrite par un fichier $.dngx$ généré pour 50.000 instances logiques du système, de sorte qu'au total ce sont 100.000 trajectoires qui sont simulées :

```
input:<path>\heatedTank50000.dngx
input:<path>\heatedTank50000.dngx
```

Fig.3 : Exemple de campagne de tests pour DNG

De manière générale, une cosimulation impose un pas de temps identique à toutes les FMU mais pas nécessairement un pas fixe. Qu'il soit fixe ou variable, chaque pas de calcul de DNG est un macro-pas au sein duquel c'est le solveur numérique embarqué dans chaque FMU qui gère ses propres micro-pas, selon ses capacités. Par exemple avec le solveur $Dassl$ ou $Cvode$ de Dymola dans la FMU, on a bien des micro-pas variables chainés entre eux pour chaque macro-pas demandé par le master de la cosimulation.
Il n'y a donc qu'un seul temps système dans le master de cosimulation DNG et toutes les FMU sont synchronisées selon la même horloge. Cette synchronisation des macro-pas n'affecte en rien la gestion interne des micro-pas qui est assurée par le solveur embarqué dans chacune des FMU.

C. Génération de nombres aléatoires en Modelica

Dans les simulations de Monte Carlo, la variable T est déterminée en tirant un nombre aléatoire $r \in [0,1]$ et en résolvant l'équation : $F(T) = r$.
Il est donc primordial de disposer d'un générateur de nombres aléatoires en Modelica, ce qui est très facile en combinant des fonctions disponibles en base dans la bibliothèque standard MSL.
Par exemple dans la séquence suivante :

$seed = Modelica.Math.Random.Utilities.initializeImpureRandom($
$Modelica.Math.Random.Utilities.automaticGlobalSeed());$
$r = Modelica.Math.Random.Utilities.impureRandom(id = seed);$

La fonction $impureRandom()$ permet de générer un nombre aléatoire r dans la plage $[0,1]$ en utilisant l'algorithme $xorshift1024*$ initialisé par une graine $seed$ elle-même calculée à partir des fonctions standards suivantes :

- $automaticGlobalSeed()$ qui retourne une graine globale calculée à partir de l'identifiant du processus informatique du code et de l'heure locale (exprimée sous la forme du nombre entier de millisecondes écoulées depuis le début de l'heure courante) ;
- $initializeImpureRandom()$ qui génère un état initial pour l'algorithme $xorshift1024*$ à partir de la graine globale calculée précédemment.

Une petite difficulté apparaît à ce stade en considérant qu'en mode cosimulation, tous les tirages d'une simulation de Monte Carlo sont exécutés comme des réplications parallèles à partir du même processus informatique et exactement à la même milliseconde.
Une manière simple de différentier les aléas de ces réplications consiste à définir un paramètre entier i ayant une valeur différente pour chaque instance du système, ce qui permettra d'obtenir des graines globales distinctes.

Moyennant cette astuce, les deux lignes de code Modelica précédentes deviennent :

```
seed = Modelica.Math.Random.Utilities.
                    initializeImpureRandom(
  Modelica.Math.Random.Utilities.automaticGlobalSeed()
                    + i);
r = Modelica.Math.Random.Utilities.
                    impureRandom(id = seed);
```

Il est alors facile de résoudre l'équation : $F(T) = r$ en Modelica grâce à une équation *when* pouvant ressembler à celle-ci :

```
when F > pre(r) then
  componentIsWorking =
            not(pre(componentIsWorking));
end when;
```

On note que cette équation permet de passer d'un état $enMarche$ ($componentIsWorking == true$) à un état $enPanne$ ($componentIsWorking == false$) et vice-versa. La publication [6] explique comment ce mécanisme peut être encapsulé dans un « bloc de défaillance », facile à ajouter aux composants d'un modèle déterministe de système pour le transformer en modèle stochastique.

C. Autres approches en fiabilité dynamique

Un modèle hybride stochastique adapté à une simulation de Monte Carlo peut être approché par un modèle discret ; pour ce faire, il existe un grand nombre d'outils gratuits et commerciaux. Les références [11] et [12] évaluent la « popularité » de 42 de ces outils. On peut ajouter à cette liste les outils développés à EDF sur la base du langage Figaro : KB3 pour la modélisation et YAMS pour le calcul par simulation de Monte Carlo.

Si on veut être plus rigoureux et ne s'intéresser qu'aux formalismes et outils offrant une modélisation vraiment hybride, le choix se réduit considérablement. Il y a un certain nombre d'outils d'origine universitaire, dont le formalisme d'entrée est très contraint, probablement trop contraint pour permettre la modélisation de systèmes industriels réels. Dans cette catégorie, on trouve les outils de traitement des réseaux de Petri dits continus, hybrides, ou fluides (cf. [9] pour une synthèse), et des outils ayant chacun son propre langage de modélisation [15].

Pour des applications industrielles, il faut des outils offrant une grande liberté de modélisation. De manière très similaire à ce que nous faisons avec les outils Modelica, un modèle Matlab/Simulink peut être complété par un comportement aléatoire dû à des défaillances : le cas test traité ci-après a été modélisé de cette manière dans la référence [13].

Enfin, il y a quelques outils vraiment faits pour la simulation de systèmes stochastiques hybrides : sans prétendre à l'exhaustivité, nous pouvons citer Vensim (http://vensim.com/), Anylogic (www.anylogic.fr) et Pycatshoo (www.Pycatshoo.org). Vensim, contrairement aux deux derniers, ne propose pas de modélisation orientée objet, ce qui limite la complexité des modèles qu'on peut construire avec cet outil, pour des raisons purement pratiques [10].

Anylogic est construit à partir du langage Java et est souvent présenté comme un outil de modélisation par agents, alors que Pycatshoo est accessible en Python ou C++ et revendique plutôt une approche orientée objets. La différence entre agents et objets n'est pas évidente, en revanche le langage sous-jacent est important car l'utilisateur qui construit les bibliothèques est amené à écrire ses modèles dans ce langage. Pour une catégorie de systèmes donnée, une fois la bibliothèque mise au point, on peut rapidement construire des modèles de systèmes par assemblage d'objets instances des classes décrites en bibliothèque. Anylogic est surtout utilisé pour modéliser des processus industriels dans le but d'optimiser leur fonctionnement ; Pycatshoo permet également de faire de l'optimisation mais il comporte des fonctions plus spécialisées dans le calcul de fiabilité des systèmes. En particulier, il sert de terrain d'expérimentation pour des méthodes innovantes d'accélération de la simulation de Monte Carlo, bien utiles pour les systèmes très fiables [14]. En conclusion de ce tour d'horizon des solutions pour faire de la fiabilité dynamique, on peut se demander pourquoi mettre en avant la solution décrite dans le présent article alors qu'il existe des outils spécialisés. Cette solution peut être intéressante lorsqu'on possède déjà un modèle de système en Modelica parce qu'il a dû être développé pour la conception et/ou le dimensionnement d'un système. Redévelopper un modèle équivalent en partant de zéro pour un outil tel que Anylogic ou Pycatshoo pourrait être très coûteux en temps d'ingénieur par rapport au simple ajout de blocs de défaillance, plus des modifications minimes sur un modèle Modelica existant, selon la méthode décrite dans [6]. La même remarque s'applique aux modèles en Matlab/Simulink.

IV. CAS D'ETUDE

A. Le modèle "heated Tank"

Le cas d'étude choisi ($heated\ Tank$) est décrit en détail dans l'article [6]. Ce cas a aussi été choisi parce qu'il fait partie des benchmarks sur lesquels sont testés différents outils dans le cadre d'une compétition internationale amicale qui a lieu chaque année [7]. Le cas heated Tank est décliné en plusieurs variantes, et celle que nous traitons ici est le cas sans réparation mais avec des taux de défaillance de composants dépendants de la température. L'absence de réparation conduit à des probabilités suffisamment grandes pour que l'on n'ait pas besoin d'un trop grand nombre de simulations pour obtenir des résultats précis.

Pour mémoire, le système se modélise par plusieurs composants comme illustré par la figure suivante :

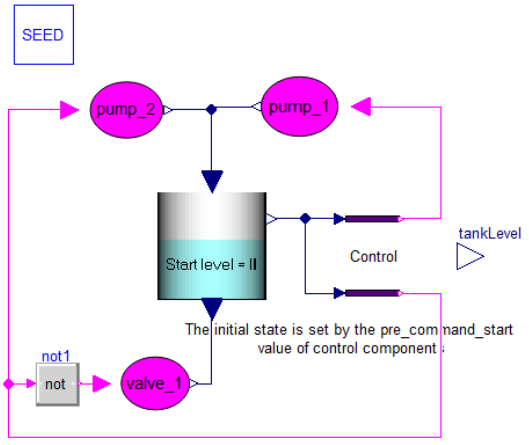

Fig.4 : Aperçu graphique du système $heatedTank$ sous Dymola

Les deux pompes et la vanne (en violet dans la figure) sont les seuls composants à pouvoir tomber en panne et ils disposent chacun de leur propre taux de panne. Un fluide est chauffé dans le réservoir et deux contrôleurs pilotent les pompes et la vanne en fonction du niveau de liquide dans ce réservoir qui doit rester entre une valeur basse (min_level) et une valeur haute (max_level).
Le modèle Modelica nous a été fourni par les auteurs de la publication [6]. Développé à l'origine avec OpenModelica, il s'est avéré directement portable sous Dymola. Simplement, nous l'avons très légèrement modifié pour ne pas utiliser la même graine globale pour chaque réplication. En revanche, au sein d'une même réplication, ce sont les tirages successifs d'un unique générateur de nombres pseudo aléatoires qui sont utilisés pour les composants pompes et vanne, comme dans le modèle d'origine.

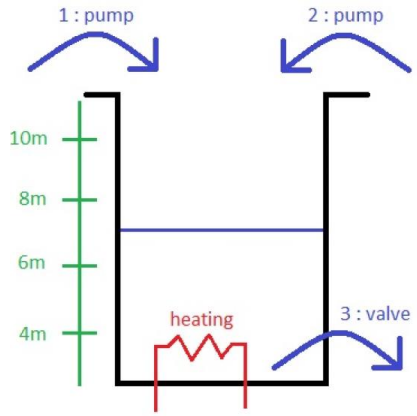

Fig.5 : Schéma de principe du cas $heatedTank$

Le modèle Modelica de ce cas d'étude est conçu pour se terminer par une instruction Modelica $terminate()$ dès que le système atteint l'un des trois événements redoutés : $overFlowLevel$ (le réservoir risque de déborder), $dryOutLevel$ (le réservoir est vide) et $boiling$ (la température du fluide est trop élevée). En effet, l'objectif étant de réaliser un calcul de la probabilité d'atteindre l'un des états indésirables avant un certain temps, il est inutile de compliquer le modèle en y ajoutant les équations de la physique qui permettraient de prolonger les simulations dans ces états indésirables.
Dans la publication [6], les auteurs ont développé un script Modelica (fichier d'extension $.mos$) pour sérialiser les réplications de Monte Carlo et conserver la valeur des variables d'intérêt en cas d'arrêt « prématuré », c'est-à-dire avant la fin de la durée de mission.
A la différence d'un modèle Modelica, le langage de script Modelica n'est pas portable et ne peut pas être utilisé en l'état sous Dymola. Dans le cadre de notre étude, nous n'en n'avons de toute façon pas besoin puisque l'objectif est simplement de tester et éprouver la technique de la cosimulation pour paralléliser les réplications de Monte Carlo en partant du principe que, si une FMU logique s'arrête prématurément à cause d'un $terminate()$, elle n'empêche pas les autres instances de la cosimulation de se poursuivre, ce qui permet à la fin des calculs de disposer de résultats exploitables pour l'ensemble des tirs, quel que soit le nombre de ceux qui se sont bien ou mal terminés.
Comme dans la publication [6], la durée de mission est fixée à $1000\ h$.

B. *Les outils utilisés*

Avec le standard FMI [4], la réentrance d'une FMU se traduit, dans le fichier $.xml$ décrivant la FMU, par la présence d'un drapeau booléen appelé $canBeInstantiatedOnlyOncePerProcess$ avec la valeur $false$.
Sur le papier, les deux outils OpenModelica et Dymola exportent des FMU avec ce drapeau à $false$.
Pourtant, après essais, aucune FMU exportée d'OpenModelica ne fonctionne en réentrance, y compris pour seulement deux instances logiques. Les FMU non réentrantes obligent DNG à charger en mémoire toutes les instances devant participer en parallèle à la cosimulation. La cosimulation reste possible, mais les performances obtenues sont alors de peu d'intérêt.
Avec Dymola les FMU semblent bien réentrantes, ce qui permet théoriquement de créer autant d'instances logiques que nécessaire avec un code binaire physiquement chargé une seule fois en mémoire. Toutefois, cette capacité n'est disponible sous Dymola qu'avec le solveur numérique $Cvode$ embarqué dans la FMU.
En pratique, nous avons tout de même rencontré sous Dymola une limitation aléatoire sur le nombre de FMU logiques exploitables pour une même FMU physique, l'aléa semblant dépendre à la fois du modèle compilé, du nombre (important) d'instances et du degré de parallélisation de ces instances, lui-même dépendant du système d'exploitation de la machine. Le problème a été corrigé par l'éditeur Dassault Systèmes et la correction a été faite pour la version $2022x\ Refresh$ mise à disposition dès le mois de novembre 2021 à EDF R&D qui possède le statut particulier de site client pilote de Dymola.
En parallèle, nous avons profité de ce temps disponible pour améliorer les performances de DNG sur la lecture des fichiers de script de grande taille comme ceux devant être générés dans notre étude (à titre indicatif, le script $.dng$ qui gère 50.000 instances logiques de la même FMU physique du système, comprend plus de 300.000 lignes de texte). Cette version DNG s'appelle 2022 2.7.0 et elle est disponible en téléchargement depuis le site de l'outil [5].

Pour faire tourner notre cas d'étude, nous avons utilisé un PC équipé d'un processeur Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz et de 32,0 Go de RAM installée, sous le système d'exploitation Windows 10 Professionnel à 64 bits. Cette machine dispose de 6 cœurs physiques (12 cœurs logiques), offrant ainsi une certaine capacité de parallélisation.

C. *Pré et post-traitement de la cosimulation avec DNG*

Nous avons utilisé la version 2.7.16 du langage Python et l'interface Spyder 3.3.3 pour développer notre code de pré et post traitement de la cosimulation. De manière plus précise, ce script Python enchaîne automatiquement les étapes suivantes :

- Génération de la FMU physique à partir d'un script $.mos$ lançant Dymola en mode silencieux ;

- Génération des scripts de la cosimulation (le fichier $.dng$ pour décrire un groupe d'instances logiques devant être traitées en parallèle et le fichier $.dcs$ pour structurer la campagne de tests en plusieurs groupes d'instances logiques à sérialiser) ;
- Génération du fichier principal de la cosimulation (le fichier $.dngx$, comprenant le script $.dng$ et la FMU physique du système) ;
- Lancement de la cosimulation en mode batch silencieux ;
- A partir des résultats de la cosimulation (un ou plusieurs fichiers $.csv$ selon le paramétrage de la campagne), construction d'un unique fichier $.csv$ de résultats permettant le tracé des probabilités d'occurrence des événements redoutés ;
- Suppression de tous les fichiers intermédiaires qui, en outre, sont de taille relativement modeste, même pour de nombreuses réplications ;
- Appel du composant open-source OMPlot de la suite OpenModelica pour visualiser les résultats ;
- Comptage de quelques temps intermédiaires et du temps total pris par le script Python.

V. RESULTATS ET PERFORMANCES

A. Maximiser la parallélisation

En première approche, nous avons choisi de tester des campagnes ne lançant qu'une seule cosimulation, donc en maximisant la parallélisation des instances logiques de la FMU système sur notre machine de tests.
Afin de tester la montée en charge de la cosimulation, nous avons progressivement augmenté le nombre d'instances, de 10 à 100.000, tout en étant bien conscients que lorsque le nombre de trajectoires voulues est trop faible, les résultats ne sont pas significatifs.
Le tableau de la figure 6 détaille les performances relevées. Les temps de calcul de la partie cosimulation pure sont assez variables compte-tenu des aléas dans l'équilibrage de charge de Windows et du caractère non déterministe des pannes générées dans les différentes instances. Les temps sont exprimés en secondes et arrondis à la valeur entière la plus proche :

Nb Instances	Export FMU	Création Script DNG	Exécution Campagne	Traitements Résultats	Temps Total
10	7	ϵ	29	ϵ	35
20	7	ϵ	29	ϵ	35
50	7	ϵ	28	ϵ	35
100	7	ϵ	28	ϵ	35
200	7	ϵ	28	ϵ	35
500	7	ϵ	29	ϵ	36
1000	7	ϵ	29	ϵ	36
2000	7	ϵ	30	ϵ	38
5000	7	ϵ	34	ϵ	41
10000	7	ϵ	41	1	49
20000	7	ϵ	60	2	69
50000	8	ϵ	253	10	271
100.000	7	ϵ	~	~	~

Nota : ϵ signifie que le temps est négligeable (<<1 s) et ~ signifie que la valeur n'est pas disponible.

Fig.6 : Performances relevées en cosimulation pure

Pour 100.000 instances, la cosimulation échoue, ce qui explique la dernière colonne du tableau de résultats. La raison est un manque de mémoire (Invalid memory access), car même si une seule FMU est chargée en mémoire, il faut tout de même disposer d'un tableau interne de variables et pointeurs pour chacune des instances logiques. Ce problème a été résolu en ajoutant un peu de sérialisation, ce qui de toute façon s'avère nécessaire puisque les performances de calcul se dégradent assez rapidement au-delà de 10.000 instances.

Nous donnons ci-après un exemple du fichier de trace produit par l'exécution du script Python complet du cas à 50.000 instances en parallèle :

```
Script heatedTank.py started at 2021-12-31 09:58:30

Step 1: Running Dymola in silent mode with the script
heatedTank.mos for creating the FMU heatedTank.fmu
Split time is approx. 7.8 seconds
Step 2: Creating the .dng script heatedTank50000.dng
Step 3: Creating heatedTank50000.dngx with FMU
heatedTank.fmu and dng script heatedTank50000.dng
Step 4: Creating the .dcs script heatedTank50000.dcs with
1 dngx files
Split time is approx. 8.1 seconds
Step 5: Running heatedTank50000.dcs from a batch in
silent mode
Split time is approx. 261.2 seconds
Step 6: Removing temporary files and printing results:
- Nb of successful runs is 12691 on 50000
- Nb of failures for dryOutLevel is 4659
- Nb of failures for overFlowLevel is 24504
- Nb of failures for boiling is 8146
Step 7: Creating the file heatedTank50000.csv ready to be
viewed with OMPlot

Global time to execute is approx. 270.6 seconds
```

Fig.7 : Exemple de trace d'une exécution en cosimulation pure (50.000 instances)

B. Mixer parallélisation et sérialisation

Cette seconde approche vise à diminuer le nombre d'instances traitées simultanément par la cosimulation en introduisant une dose de sérialisation dans la campagne et en nous limitant au cas de 100.000 instances, qui représente le nombre de simulations de référence choisi dans la publication [6].

La montée en charge s'effectue cette fois-ci en sérialisant un nombre croissant de paquets d'instances, chaque paquet représentant une cosimulation, avec donc une parallélisation des instances qu'elle contient.
Le tableau de la figure 8 détaille les performances relevées, les temps étant toujours exprimés en secondes et arrondis à la valeur entière la plus proche.

Nb paquets	2	5	10	20	50	1^E2	1^E3	1^E4
Nb instances par paquet	5^E4	2^E4	1^E4	5^E3	2^E3	1^E3	100	10
Export FMU	7	7	7	7	8	7	7	7
Création scripts DNG	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ
Exécution campagne	447	250	171	158	144	129	197	2^E3
Traitement résultats	39	38	38	38	39	39	38	59
Temps total	493	295	216	203	190	175	235	2^E3
Exécution /Temps total	91%	85%	79%	78%	76%	74%	84%	99%

Fig.8 : Performances relevées en mixant sérialisation et parallélisation pour 100.000 simulations

L'analyse des résultats amène les commentaires suivants :

- Autour de 7 s, le temps nécessaire à la génération de la FMU est remarquablement court sous Dymola $2022x\ Refresh$ (prototype de la future version Dymola 2023). A titre de comparaison, le même modèle Modelica exporté en une FMU depuis la dernière version d'OpenModelica (version 1.18.1 de décembre 2021) requiert environ 14 s, soit deux fois plus de temps que Dymola[1].
- Le temps de création des fichiers nécessaires à la campagne (fichiers d'extension $.dng$, $.dngx$ et $.dcs$) est toujours quasi-négligeable (bien inférieur à 1 s), quel que soit le nombre et la combinaison des instances.
- Le temps de post-traitement des fichiers $.csv$ résultats de toutes les cosimulations des campagnes est très stable, sauf pour la dernière combinaison essayée pour laquelle il y a 10.000 très petits fichiers à traiter. Dans tous les cas, ce temps ne dépend que de notre code Python qui pourrait sûrement être optimisé.
- De manière logique, la dernière ligne du tableau de la figure 8 indique que dans tous les cas de figure, le temps nécessaire pour l'exécution de la campagne fait l'essentiel du temps total, au-delà de 74%.
- Sur la machine utilisée, la meilleure performance est obtenue pour une campagne sérialisant 100 cosimulations de 1.000 instances chacune.
- Pour 100.000 occurrences, le volume des données à traiter (fichiers $.csv$) est indépendant du degré de sérialisation (autour de 8,5 Mo). DNG produit en outre des traces internes réparties en un nombre de fichiers $.log$ dépendant de la campagne mais dont le volume total reste également relativement constant, autour de 12 à 13 Mo.

Le cas extrême (sérialisation maximale et parallélisation minimale) serait représenté par une série de 100.000 cosimulations d'une seule instance. Ce cas n'a pas été essayé car son temps d'exécution devrait être très important, bien au-delà des attentes de cette étude. De manière plus générale, lorsque la sérialisation est importante, on multiplie le temps nécessaire à la décompression de chacun des fichiers $.dngx$, puis du fichier de la FMU qu'il contient (extension $.fmu$) et le temps de lecture du script $.dng$. De plus, DNG recharge aussi à chaque fois la FMU physique en mémoire.

Pour 100.000 occurrences, la courbe des performances (temps d'exécution de la campagne et temps total) en fonction du nombre de paquets sérialisés est la suivante :

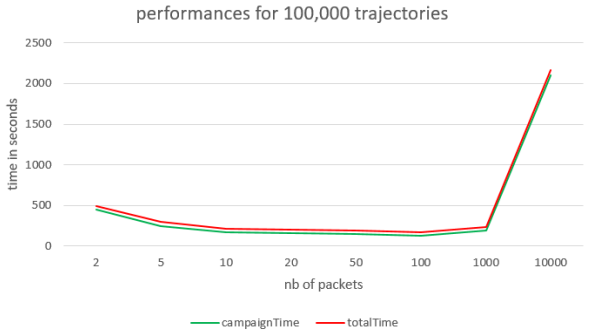

Fig.9 : performances obtenues en fonction de la sérialisation choisie pour 100.000 occurrences

Le minimum de cette courbe se situe entre 100 et 1.000 paquets, c'est-à-dire en parallélisant entre 1.000 et 100 occurrences. On voit donc bien que l'accélération permise par DNG est supérieure au nombre de cœurs physiques (6) ou logiques (12) de la machine utilisée pour les essais.

C. Analyse des résultats

On rappelle que les résultats finaux sont les probabilités d'occurrence de trois événements redoutés : $overFlowLevel$ (le réservoir risque de déborder), $dryOutLevel$ (le réservoir est vide) et $boiling$ (la température du fluide est trop élevée). Plus le nombre de trajectoires est important et plus les résultats obtenus dessinent des courbes bien lissées. En revanche, la répartition entre sérialisation et parallélisation n'a pas d'influence sur la qualité graphique de la visualisation des résultats.
Comme dans la publication [6], chaque courbe de probabilité tend vers une asymptote : $p = 0{,}5$ pour l'évènement $overFlowLevel$, $p = 0{,}1$ pour l'évènement $dryOutLevel$ et $p = 0{,}16$ pour l'évènement $boiling$.
A titre indicatif, la figure suivante reproduit le résultat obtenu pour 100.000 tirs répartis en 100 paquets de 1.000 trajectoires.

[1] Toutefois, pour compléter la comparaison, notons que la taille du fichier de la FMU du système exportée d'OpenModelica fait seulement 429 Ko contre 998 Ko pour la FMU Dymola.

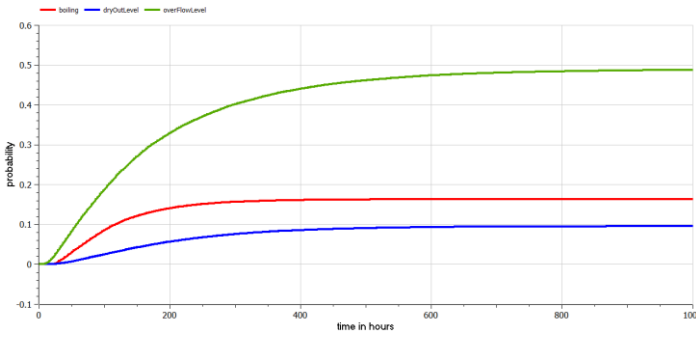

Fig.10 : Visualisation des résultats sous OMPlot (100.000 occurrences réparties en 100 paquets)

VI. CONCLUSIONS ET PERSPECTIVES

Le cas d'étude choisi ici est très simple mais il est représentatif des problèmes rencontrés en fiabilité dynamique. Il permet d'apporter une preuve d'intérêt de la technique de cosimulation pour l'étude des systèmes hybrides.

En raison de l'absence totale de couplage induit par l'indépendance des différentes instances de FMU logiques décrivant les tirs de la méthode de Monte Carlo, les résultats ne sont pas biaisés par la cosimulation. L'équilibrage de charge est parfait puisque les FMU logiques sont strictement identiques entre-elles en termes de poids de calcul. Et surtout, les calculs sont très sensiblement accélérés.

Cette étude constitue un nouveau retour d'expérience de l'utilisation de notre outil DNG dans un contexte métier différent. En effet, dans ce cas particulier où la campagne de tests répète plusieurs fois le même schéma de calcul, il doit être possible d'éviter le rechargement et la décompression des fichiers $.dngx$ et $.fmu$ pour chacun des paquets sérialisés. Il conviendrait également de contrôler de manière plus optimale les allocations mémoire dans DNG puisque le code du master est un $.jar$ (fichier Java compressé) et qu'il est possible de positionner des options Java telles que $-Xms < size >$ ou $-Xmx < size >$.

En termes d'évolution fonctionnelle pour gérer de plus gros modèles, il faudra certainement étendre et tester la cosimulation DNG selon une distribution multi-machines.

Sur un plan plus théorique enfin, il serait intéressant de prédire ce que serait la configuration de cosimulation la plus efficace (compromis sérialisation/parallélisation) en fonction du modèle sous test (une métrique minimale étant le nombre et la complexité de ses équations) et de l'architecture de la plateforme de calcul (nombre de nœuds de calcul, nombre de cœurs de calcul de chaque nœud).

REFERENCES

[1] Association Modelica : https://modelica.org

[2] Marc Bouissou, Hilding Elmqvist, Martin Otter, Albert Benveniste "Efficient Monte Carlo simulation of stochastic hybrid systems", Proceedings of the 10th International Modelica Conference March 10-12, 2014, Lund, Sweden.

[3] Philippe Carer, Xavier De Bossoreille, Martin Otter, « Modelica et simulation de Monte Carlo pour l'étude de fiabilité d'un Data Center », 20ième congrès de maîtrise des risques et de sûreté de fonctionnement - Saint-Malo, 11-13 octobre 2016.

[4] Standard FMI : https://fmi-standard.org

[5] Plateforme DNG : https://bitbucket.org/simulage/daccosim/wiki

[6] Marc Bouissou, Lena Buffoni, « Generic method to transform a Modelica simulation model into a dynamic reliability model », 22ième Congrès de Maîtrise des Risques et Sûreté de Fonctionnement λμ22, en distanciel, 12-15 octobre 2020.

[7] Abate, A., Blom, H. A. P., Bouissou, M., Cauchi, N., Chraibi, H., Delicaris, J., Haesaert, S., Hartmanns, A.,
Ma, H., & More Authors (2021). ARCH-COMP21 Category Report: Stochastic Models. In G. Frehse, & M.
Althoff (Eds.), 8th International Workshop on Applied Verification of Continuous and Hybrid Systems (pp. 55-
89). (EPiC Series in Computing; Vol. 80).

[8] Marc Bouissou & Yannick Lefebvre, « A Path-Based Algorithm to Evaluate Asymptotic Unavailability
for Large Markov Models ». In: Proceedings of RAMS'2002, 2002. doi:10.1109/RAMS.2002.981616.

[9] M. Dotoli, M.P. Fanti, A. Giua, C. Seatzu, First-order hybrid Petri nets. An application to distributed manufacturing systems, Nonlinear Analysis: Hybrid Systems, Volume 2, Issue 2, 2008, doi:10.1016/j.nahs.2006.05.005.

[10] M. Bouissou, I. Chubarova, H. Chraibi, Critical comparison of two user friendly tools to study Piecewise Deterministic Markov Processes (PDMP): season 2, Proceedings of ESREL 2013, Amsterdam.

[11] Dias, L. M., Vieira, A. A., Pereira, G. A., & Oliveira, J. A. (2016, December). Discrete simulation software ranking—A top list of the worldwide most popular and used tools. In 2016 Winter Simulation Conference (WSC) (pp. 1060-1071). IEEE.
[12] Vieira, A. A. C., Dias, L. S., Santos, M. Y., Pereira, G., & Oliveira, J. A. (2019). A ranking of the most known freeware and open source discrete-event simulation tools. Proceedings of the European Modeling and Simulation Symposium, 2019.

[13] Huilong Zhang, Benoîte de Saporta, Francois Dufour, Gilles Deleuze, "Dynamic Reliability by Using Simulink and Stateflow". Chemical engineering transactions Vol 33, 2013. doi:10.3303/CET1333089

[14] G. Chennetier, H. Chraibi, A. Dutfoy & J. Garnier, "Importance Sampling and Sensitivity Analysis for Reliability Assessment of Hybrid Dynamic Systems Represented by Piecewise Deterministic Markov Processes",
31st ESREL conference, 19-23 Sept. 2021, Angers, France.