Méthodologie et algorithmes pour la distribution large échelle de co-simulations de systèmes complexes : application aux réseaux électriques intelligents (Smart Grids)

Cherifa Dad

CentraleSupélec

N° d'ordre : 2018-02-TH

CENTRALESUPELEC

École doctorale IAEM

« Informatique, Automatique, Électronique-Électrotechnique et Mathématiques »

Laboratoire UMI GT-CNRS 2958

THÈSE DE DOCTORAT

Domaine STIC

Spécialité : Informatique

Soutenue le 26 juin 2018

Par :

Cherifa DAD

Méthodologie et algorithmes pour la distribution large échelle de co-simulations de systèmes complexes : Application aux réseaux électriques intelligents (*Smart Grids*)

Composition du jury :

Directeur de thèse	M. Stéphane VIALLE	Professeur à CentraleSupélec
Président du jury	M. Michaël KRAJECKI	Professeur à l'Université de Reims Champagne-Ardenne
Rapporteurs	M. Hervé GUYENNET	Professeur à l'Université de Franche-Comté
	M. Michaël KRAJECKI	Professeur à l'Université de Reims Champagne-Ardenne
Examinateurs	M. Mongi BEN GAID	Chef de projet à IFP Energies nouvelles
	M. Mathieu CAUJOLLE	Chef de projet à EDF Lab Saclay
	M^{me} Sara TUCCI-PIERGIOVANNI	Ingénieur chercheur au CEA LIST
Invité	M. Jean-Philippe TAVELLA	Expert Simulation à EDF Lab Saclay

«La connaissance s'acquiert par l'expérience, tout le reste n'est que de l'information.»
par *Albert Einstein*

DÉDICACE

Je dédie mon travail à ma famille.

REMERCIEMENTS

Tout d'abord, je souhaiterais remercier les rapporteurs, pour le temps qu'ils ont accordé à la lecture de cette thèse et à l'élaboration de leurs rapports très détaillés. Je remercie Monsieur Michaël KRAJECKI et Monsieur Hervé GUYENNET d'avoir accepté cette charge et de l'intérêt qu'ils ont porté à mes travaux.

Je souhaiterai remercier encore une fois Michaël KRAJECKI le président du jury de ma thèse et le jury de thèse, composé de, Monsieur Mongi BEN GAID, Monsieur Mathieu CAUJOLLE, et madame Sara TUCCI-PIERGIOVANNI. Merci pour vos retours sur mes travaux, cela m'est très important.

Je remercie CentraleSupélec de m'avoir accueillie dans sa structure pour permettre de faire mes preuves en tant que chercheur, et je remercie également EDF, notre collaborateur industriel.

Je tiens à exprimer mes sincères remerciements à mon directeur de thèse, Monsieur Stéphane VIALLE, pour sa confiance, son aide et ces critiques objectives. Il était toujours disponible pour m'apporter conseils avisés. Merci pour tout.

Mes vifs remerciements vont à Monsieur Jean-Philippe TAVELLA qui m'avait apporté un soutien sans faille dans la conduite de mes travaux de recherche, et qui a souvent été à mon écoute.

Mathieu, merci de m'avoir accompagné dans la deuxième partie de la thèse, tes compétences techniques et ta précision m'inspireront toujours.

Mes remerciements vont également à mes parents adorés, à mes soeurs et à mes frères car sans leurs constants soutiens et leurs encouragements, je n'aurai pas pu mener ce travail. Merci pour vos conseils, vos attentions, votre patience ainsi que votre confiance.

Un énorme merci et clin d'œil à tous les doctorants avec qui j'ai partagé ces 3 ans : Muhammad, Hassan, Sara, Ninel, Hicham, Marwene, Chanez, Timothé, Guillaume, shane, Amel, Rana, Wahab, Rawdha, Alban, Chi-Hak, Pierre-Louis. Un merci à tous les acteurs de CentraleSupélec. Merci à Yann, Dorothée, Fabienne, Gilles, Sébastien, Eric, Patrick, Claudine.

Résumé

L'apparition des réseaux électriques intelligents, ou « *Smart Grids* », engendre de profonds changements dans le métier de la distribution d'électricité. En effet, ces réseaux voient apparaître de nouveaux usages (véhicules électriques, climatisation) et de nouveaux producteurs décentralisés (photovoltaïque, éolien), ce qui rend plus difficile le besoin d'équilibre entre l'offre et la demande en électricité et qui impose d'introduire une forme d'intelligence répartie entre leurs différents composants. Au vu de la complexité et de l'ampleur de la mise en œuvre des *Smart Grids*, il convient tout d'abord de procéder à des simulations afin de valider leur fonctionnement. Pour ce faire, CentraleSupélec et EDF R&D (au sein de l'institut RISEGrid) ont développé DACCOSIM, une plate-forme de co-simulation s'appuyant sur la norme FMI [1](*Functional Mock-up Interface*), permettant de concevoir et de mettre au point des réseaux électriques intelligents et de grandes tailles. Les composants clés de cette plate-forme sont représentés sous forme de boîtes grises appelées FMU (*Functional Mock-up Unit*). En outre, les simulateurs des systèmes physiques des *Smart Grids* peuvent faire des retours arrière en cas de problème dans leurs calculs, contrairement aux simulateurs événementiels (unités de contrôle) qui, bien souvent, ne peuvent qu'avancer dans le temps. Pour faire collaborer ces différents simulateurs, nous avons conçu une solution hybride prenant en considération les contraintes de tous les composants, et permettant d'identifier précisément les types d'événements auxquels le système est confronté. Cette étude a débouché sur une proposition d'évolution de la norme FMI.

Par ailleurs, il est difficile de simuler rapidement et efficacement un *Smart Grid*, surtout lorsque le problème est à l'échelle nationale ou même régionale. Pour pallier ce manque, nous nous sommes focalisés sur la partie la plus gourmande en calcul, à savoir la co-simulation des dispositifs physiques. Ainsi, nous avons proposé des méthodologies, approches et algorithmes permettant de répartir efficacement et rapidement ces différentes FMU sur des architectures distribuées. L'implantation de ces algorithmes a déjà permis de co-simuler des cas métiers de grande taille sur un cluster de PC multi-cœurs. L'intégration de ces méthodes dans DACCOSIM permettra aux ingénieurs d'EDF de concevoir des « réseaux électriques intelligents de très grande taille » plus résistants aux pannes.

Mots clés : co-simulation, placement des tâches, exécution distribuée et parallèle, large échelle, pas de temps variable, événement de contrôle-commande.

1. http ://fmi-standard.org

Abstract

The emergence of Smart Grids is causing profound changes in the electricity distribution business. Indeed, these networks are seeing new uses (electric vehicles, air conditioning) and new decentralized producers (photovoltaic, wind), which make it more difficult to ensure a balance between electricity supply and demand, and imposes to introduce a form of distributed intelligence between their different components. Considering its complexity and the extent of its implementation, it is necessary to co-simulate it in order to validate its performances. In the RISEGrid institute, CentraleSupélec and EDF R&D have developed a co-simulation platform based on the FMI [2] (Functional Mock-up Interface) standard called DACCOSIM, permitting to design and develop Smart Grids. The key components of this platform are represented as gray boxes called FMUs (Functional Mock-up Unit). In addition, simulators of the physical systems of Smart Grids can make backtracking when an inaccuracy is suspected in FMU computations, unlike discrete simulators (control units) that often can only advance in time. In order these different simulators collaborate, we designed a hybrid solution that takes into account the constraints of all the components, and precisely identifies the types of the events that system is facing. This study has led to a FMI standard change proposal.

Moreover, it is difficult to rapidly design an efficient Smart Grid simulation, especially when the problem has a national or even a regional scale. To fill this gap, we have focused on the most computationally intensive part, which is the simulation of physical devices. We have therefore proposed methodologies, approaches and algorithms to quickly and efficiently distribute these different FMUs on distributed architectures. The implementation of these algorithms has already allowed simulating large-scale business cases on a multi-core PC cluster. The integration of these methods into DACCOSIM will enable EDF engineers to design « large scale Smart Grids » which will be more resistant to breakdowns.

Keywords : Co-simulation, task placement, distributed and parallel execution, large-scale, variable time step, control-command event.

2. http ://fmi-standard.org

TABLE DES MATIÈRES

INTRODUCTION

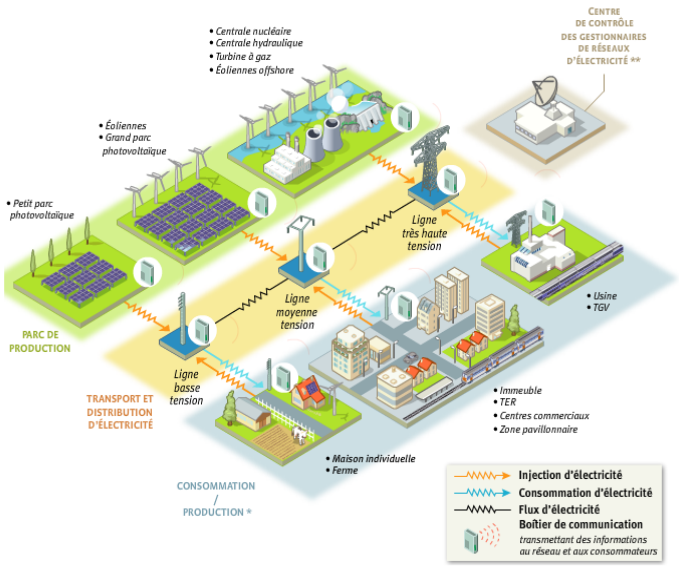
1 Contexte scientifique et industriel

Le thème de cette thèse s'est inscrit dans le cadre du projet industriel POMME (Plateformes Orientées Métiers de Multi-simulations Électriques), mené par EDF [3] R&D dans le cadre de l'institut *RISEGrid*[4]. Cet institut fédère les efforts de recherche de CentraleSupélec et d'EDF R&D autour de la thématique des *Smart Grids*. Ces recherches abordent de nombreux domaines présents dans les *Smart Grids*, comme le routage de l'électricité, les systèmes d'informations, le contrôle-commande, la co-simulation des *Smart Grids*, la gestion optimisée des réseaux électriques, les réseaux de télécommunication associés, etc. Les travaux de recherche de cette thèse se sont concentrés sur la co-simulation distribuée des *Smart Grids*.

Le *Smart Grid*, ou le réseau électrique intelligent, est le domaine applicatif de nos travaux. Il représente un réseau de distribution d'électricité moderne qui introduit de nouvelles technologies de l'information en parallèle du système électrique, afin d'assurer à tout moment un équilibre entre l'offre et la demande, et d'optimiser la production, la distribution et la consommation d'électricité [1]. Il s'agit en fait de faire évoluer un réseau de distribution classique vers un système auto-adaptatif, en faisant circuler de l'information entre les consommateurs et les fournisseurs. La mise au point d'un tel système complexe nécessite de pouvoir le simuler, de manière réaliste et à grande échelle. De manière générale, un simulateur de *Smart Grid* comporte des simulateurs d'un réseau électrique, d'un réseau télécom et d'un système d'information qui s'échangent des données de manière permanente. Le but initial du projet POMME était donc de développer une plate-forme de co-simulation dédiée aux *Smart Grids* afin d'étudier les systèmes

3. le leader de la production et de la distribution d'électricité en France.
4. *RiseGrid* a été fondé par CentraleSupélec et EDF R&D en 2012.

électriques intelligents. Il s'agissait (1) d'assembler dans un même environnement des composants logiciels existants, hétérogènes, réalisés avec des technologies différentes et par des experts différents, et (2) de développer une plate-forme capable d'exécuter cet assemblage de simulateurs sur un cluster de PC multi-coeurs, en utilisant la norme FMI (*Functional Mock up Interface*) choisie par EDF. Pour cela, les deux acteurs du projet (CentraleSupélec et EDF) ont développé une plate-forme de co-simulation s'appuyant sur la norme FMI, appelée DACCOSIM (*Distributed Architecture for Controlled CO-SIMulation*) et opérationnelle sur architecture distribuée.



Source : Commission de Régulation de l'Enérgie (http://www.smartgrids-cre.fr)

FIGURE 1 – Exemple de système complexe (*Smart Grid*)

2 Problématique

Comme on l'a présenté auparavant, notre sujet de thèse s'inscrit dans un projet de co-simulation distribuée à large échelle. Il concerne plus précisément : (1) l'échange des données et la coordination des différents composants de la co-simulation de la plate-forme DACCOSIM, et (2) l'obtention de performances et la capacité de passage à l'échelle de DACCOSIM sur des clusters de PC multi-coeurs.

La problématique industrielle de nos travaux de recherche se résume en fait par les deux questions suivantes : Comment la co-simulation de système électrique intelligent s'appuyant sur DACCOSIM gère-t-elle la synchronisation et l'échange des données ? Peut-on co-simuler efficacement et rapidement un gros système électrique avec DACCOSIM sur un ensemble de machines multi-coeurs ?

2.1 Concevoir un modèle hybride

Les *Smart Grids* représentent des systèmes complexes contenant de nombreux composants variés qui interagissent fortement entre eux : centrales électriques, réseaux électriques, réseaux de télécommunication et de contrôle, sites de consommation, etc. Pour co-simuler un *Smart Grid*, il faut donc être capable de simuler des composants physiques évoluant en temps continu (comme le réseau de distribution d'électricité et les unités de contrôle-commande), des composants évoluant en temps discrets ou par événements (comme les réseaux télécom et les systèmes d'informations), et leurs interactions. Les différentes parties des *Smart Grids* ont ainsi des notions différentes du temps, c'est pour cela que leur interopérabilité reste difficile puisque chaque partie s'appuie sur son propre modèle de temps (discret ou continu).

Un des apports attendus de cette thèse est donc la proposition d'un modèle hybride qui assure une progression cohérente du temps et une synchronisation des données de l'ensemble des composants d'une co-simulation utilisant la plate-forme DACCOSIM.

2.2 Concevoir une distribution effectuée sur clusters de co-simulation

La deuxième problématique de cette thèse est celle de la performance et du passage à l'échelle des co-simulations de *Smart Grids*. Nous nous sommes intéressés à la partie la plus gourmande en calcul dans la simulation d'un *Smart Grid*, qui est la simulation des systèmes physiques. Néanmoins, ces co-simulations font face à des tailles de problèmes très importantes (*Smart Grids* de tailles régionales ou nationales) ayant des composants hétérogènes avec des temps de traitements très longs. La co-simulation de ces systèmes ne peut pas s'exécuter dans la mémoire d'un seul noeud de calcul mais plutôt dans les mémoires d'un ensemble de machines. L'utilisation d'une telle architecture distribuée permet un passage à l'échelle pour traiter de gros problèmes en accélérant les calculs. Contrairement aux applications classiques de calcul intensif, les composants traités dans cette thèse sont des boîtes grises, hétérogènes, qui échangent beaucoup de petites données de manière fréquente. Pour remédier à ce problème, on doit s'appuyer sur des méthodes qui répartissent efficacement le graphe de FMU d'une co-simulation DACCOSIM sur les différents PC d'un cluster. On recherche notamment à équilibrer la charge entre les nœuds de calcul si le problème contient des calculs importants, et/ou à minimiser les communications inter-nœuds si le problème implique beaucoup de messages échangés entre ses FMU.

Les apports attendus de cette deuxième partie sont donc d'aboutir à des co-simulations à la fois précises et performantes, tout en les validant sur des *use-cases* métiers fournis

par EDF.

3 Plan de mémoire

Ce mémoire de thèse est structuré autour de six principaux chapitres qui se résument comme suit :

— Dans le chapitre 1, nous introduisons la technique de co-simulation pour simuler les systèmes complexes, plus particulièrement les *Smart Grids*, et nous présentons quelques solutions de co-simulation proposées dans la littérature.

— Dans le chapitre 2, nous présentons un état de l'art sur le placement des tâches et décrivons quelques aspects architecturaux des machines que nous avons utilisées. Ensuite, nous positionnons notre problématique par rapport à l'existant.

— Dans le chapitre 3, nous introduisons l'outil de co-simulation DACCOSIM basé sur la norme FMI, et détaillons son architecture logicielle ainsi que son fonctionnement.

— Dans la première partie du chapitre 4, nous proposons un modèle de temps hybride dédié à des systèmes qui utilisent à la fois la plate-forme DACCOSIM pour simuler des composants physiques en suivant la norme FMI, et une plate-forme événementielle pour simuler des composants de contrôle-commande et le réseau informatique. Dans la deuxième partie de ce chapitre, nous proposons une approche plus originale entièrement basée sur la norme FMI et sur DACCOSIM, mais qui demande d'améliorer à la fois la norme FMI et l'API JavaFMI.

— Dans le chapitre 5, nous décrivons, à l'aide d'algorithmes et de diagrammes, trois approches et plusieurs heuristiques qui permettent de placer des FMU hétérogènes sur un cluster de PC multi-cœurs. L'approche choisie dépendra ensuite du problème visé.

— Dans le chapitre 6, nous évaluons les approches proposées dans le chapitre 5 et interprétons les résultats d'expérimentations obtenus sur des cas d'utilisations métiers fournis par EDF R&D.

Dans la fin de ce mémoire, nous ouvrons des perspectives pour les futures recherches et concluons notre travail de thèse.

Première partie

État de l'art : Co-simulation distribuée sur Cluster de PC multi-cœurs

CHAPITRE 1

CO-SIMULATION APPLIQUÉE AUX *SMART GRIDS*

Sommaire

1 Introduction

L'essor des systèmes cyber-physiques (SCP) offre aujourd'hui de nouvelles perspectives dans le domaine des réseaux de distribution d'énergie. Ce type de systèmes est difficile à concevoir, et à développer, il couvre plus de domaines et comprend de nombreux simulateurs hétérogènes très interactifs entre eux et qu'il faut rassembler au sein d'un seul environnement d'études. À titre d'exemple, on peut citer les réseaux électriques intelligents (*Smart Grids*). Les simulateurs qui les composent ont été développés par des experts différents, avec des langages et des outils variés. Il serait donc particulièrement ardu et coûteux de réintroduire et redévelopper tous ces outils hétérogènes pour les regrouper et les intégrer dans un seul environnement de modélisation, alors qu'il est plus rationnel de réutiliser des simulateurs déjà existants pour traiter des problèmes à des petites et à des grandes échelles. Pour toutes ces raisons, il semble pertinent de se pencher sur une nouvelle technique dite co-simulation qui permet de réutiliser et combiner ces mêmes outils hétérogènes. Cette technique n'est toujours pas providentielle et de nombreux défis doivent encore être relevés : (1) interconnecter les modèles événementiels (en particulier les événements de type contrôle-commande) avec des modèles continus (physique des bâtiments, des centrales de production, du réseau électrique...), tout en prenant en compte leurs propres critères (taille, retour en arrière en cas de calculs imprécis), (2) simuler des gros systèmes qui passeront à l'échelle, et (3) s'assurer de la qualité des résultats.

Ce chapitre est consacré, dans un premier temps, à définir la nouvelle technique de simulation appelée co-simulation permettant de simuler des composants variés. Nous décrirons ensuite les deux principaux standards de la co-simulation, à savoir la norme FMI (*Functional Mock-up Interface*) et la norme HLA (*High Level Architecture*). Comme l'un des points les plus importants de cette technique réside dans le couplage hybride entre des simulateurs axés sur le temps et des simulateurs gérant des événements, nous présenterons un état de l'art des modèles de temps hybrides pour les systèmes complexes, plus particulièrement pour les *Smart Grids*. Enfin, nous conclurons ce chapitre par un tableau récapitulatif qui compare les solutions de co-simulation présentes dans la littérature.

2 Terminologies et notions de base

Dans cette première partie de l'état de l'art, il nous semble indispensable de définir certaines notions de base qui seront utilisées dans notre travail.

— Un **système** regroupe un ensemble d'éléments qui interagissent entre eux selon certaines règles. Il représente un objet d'étude associé à des objectifs bien définis

par les utilisateurs (le fonctionnement attendu, la dynamique du système...).

— **Simuler** un composant signifie l'exécution d'un modèle visant à reproduire le comportement d'un système réel.

— Le **modèle** représente la spécification et la représentation abstraite d'un système réel ou virtuel. La définition que donne *Walliser* de celui-ci est la suivante :

Definition 2.1 *La notion de modèle recouvre toute représentation d'un système réel, qu'elle soit mentale ou physique, exprimée sous forme verbale, graphique ou mathématique [76].*

On trouve généralement dans la littérature deux types de modèles, à savoir les modèles à temps continu dont les entrées et les sorties sont fonctions de variables continues, et les modèles à temps discret dit événementiels dont les entrées et les sorties sont discrètes à base de variables discrètes.

— Chaque **modèle de simulation** (S) doit contenir des entrées (u_i) et des sorties (v_i), comme l'illustre la figure 1 (a). Les sorties de ce simulateur sont liées directement aux entrées des autres simulateurs, en appliquant la fonction f. Grâce à ces entrées/sorties, nous pouvons coupler les modèles de simulation entre eux. La figure 1(b) illustre une co-simulation de trois simulateurs différents $(S), (S')et(S'')$, qui communiquent et échangent leurs données pendant une simulation.

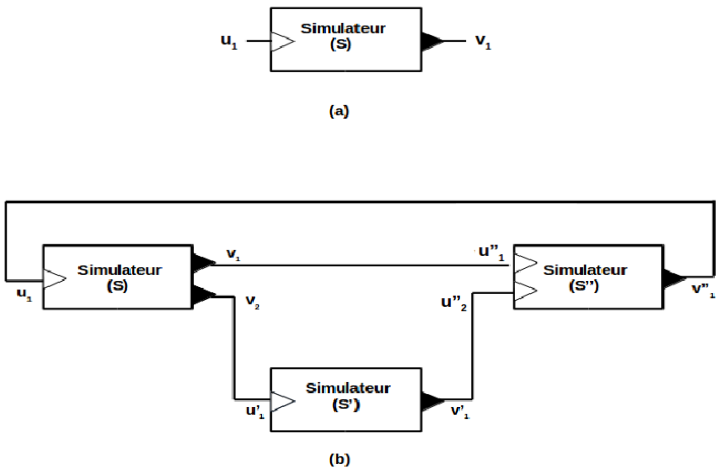


FIGURE 1 – Système contenant trois simulateurs physiques

— **Le temps simulé** est différent du temps d'exécution de la co-simulation ; il représente le temps mesuré par une horloge logique globale du système qui avance plus ou moins rapidement en fonction de l'exécution de la co-simulation.

— **Un pas de temps** ou un pas de calcul-communication (h) représente un intervalle de temps entre deux portions de calculs successifs. Ce pas de temps peut être

divisé en des instants de temps réduits dit pas de temps internes ; ceux-ci sont utilisés par le solveur numérique interne du simulateur. À l'issue de chaque pas de temps, une propagation des valeurs des sorties obtenues sera effectuée vers les entrées connectées, cette dernière étape représente la phase de communication où les différents composants échangent des données entre eux.

— **La durée de co-simulation** représente la succession des pas de temps $(\sum_{i=1}^{n}(h_i))$ tel que h_i est un pas de temps et n est le nombre de pas de temps (nombre d'itération), comme l'indique la figure 2.

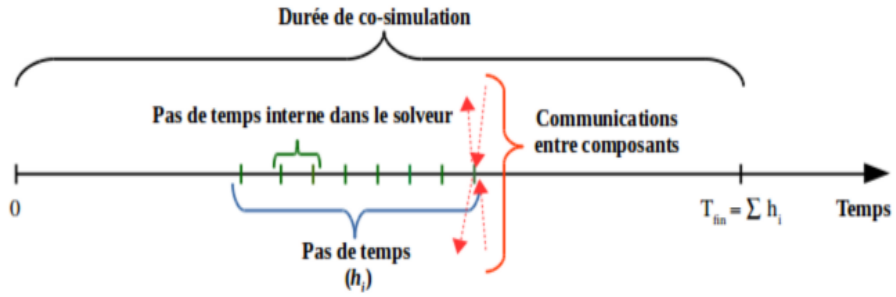


FIGURE 2 – Simulation d'un composant physique

— **Un événement** est une circonstance ou une action apparue dans la simulation, il représente un changement d'état à un moment précis, et la **date d'occurrence** de cet événement en temps simulé est présentée par une estampille temporelle dite *timestamp* [18].

— On appelle **composant** physique tout simulateur qui imite le comportement réel d'un dispositif physique.

— D'après [58] dans le chapitre 9, un **solveur** est représenté comme une réalisation de l'algorithme d'intégration qui permet d'évaluer le signal continu x au point t tel que $t \in \mathbb{R}$. Cet algorithme déduit avec précision la forme du signal en sortie. On trouve généralement des solveurs à pas de temps constant [65] ou à pas de temps variable [26].

— **Contrainte de causalité**

Un système distribué est un système informatique qui exécute un ensemble de sous-programmes du même système sur un ensemble de ressources distribuées. Il apparaît à l'utilisateur comme une seule entité cohérente. Notons que chaque ressource matérielle est munie de sa propre mémoire et de sa propre unité de calcul. Dans un système distribué, un simulateur qui respecte la contrainte de causalité assure que l'exécution de ses événements sera effectuée dans l'ordre croissant de leurs estampilles. Autrement dit, la contrainte de causalité dans un système distribué représente une cohérence entre les événements. Cette propriété garantit que la

sortie y (le phénomène futur) de ce système ne doit pas précéder son entrée x (le passé) : le futur de ce système n'influe pas son passé (x → y) [64, 29].

Definition 2.2 *La contrainte de causalité est garantie quand la simulation parallèle d'un modèle M produit les mêmes résultats que l'exécution séquentielle de ce même modèle M [29].*

— **Gestion du temps dans les systèmes complexes**

D'après *R. Fujimoto* [29], la gestion de temps dans les systèmes distribués s'appuie sur deux types d'algorithmes, dits optimiste et pessimiste qui assurent la cohérence des échanges de données :

1. **Synchronisation optimiste** : c'est une stratégie qui ne respecte pas la contrainte de causalité, mais qui détecte les erreurs temporelles de la simulation en faisant un retour dans le passé (grâce à un mécanisme de *Rollback* [44]). De manière générale, ce mode optimiste a été développé dans le but d'améliorer la précision des résultats de la simulation. Néanmoins, cette stratégie de synchronisation peut s'avérer trop lente à cause des retours en arrière du simulateur trop fréquents, et nécessite d'avoir beaucoup de mémoire. Les simulateurs qui adoptent cette stratégie doivent être en mesure d'intégrer dans leurs algorithmes des mécanismes pour faire des *Rollback*.

2. **Synchronisation Conservative (ou pessimiste)** : Contrairement à l'approche optimiste, la causalité temporelle dans ce type de synchronisation doit être assurée à tout moment, c'est-à-dire que les simulateurs doivent s'attendre les uns les autres avant d'être exécutés. Ainsi, à aucun moment, un simulateur ne doit revenir dans son passé pour se corriger en cas de problème au niveau des calculs [29]. Généralement, on trouve ce type d'approche dans les simulateurs événementiels où on ne peut jamais revenir en arrière pour reproduire les événements. Ce mode de fonctionnement s'avère trop pessimiste dans de nombreux cas (comme dans la simulation des dispositifs physiques par exemple).

   Il existe plusieurs algorithmes implémentés qui s'appuient sur cette approche. On trouve la méthode conservative la plus utilisée dans les systèmes événementiels dits CMB [1][17], qui gère le temps global du système. Selon cette méthode, deux processus logiques communiquent entre eux par le biais d'un canal, et les messages estampillés triés décrivent les événements du système. Chaque processus accède à l'ensemble des événements du canal pour proposer

1. *Chandy Misra Bryant*

une valeur de temps future. Cet algorithme choisit la valeur minimale des estampilles d'événements se trouvant dans chaque canal. Cette valeur minimale de tous les messages représente la valeur minimale de l'ensemble des canaux. Elle garantit qu'aucun événement estampillé inférieur à cette valeur minimale ne survienne par la suite.

Dans notre travail de thèse, nous utilisons une plate-forme s'appuyant sur une méthode de simulation hybride. Cette plate-forme respecte la contrainte de causalité et en cas de besoin, elle peut effectuer des *rollbacks*. Cette dernière sera présentée dans le chapitre 3.

3 Simulation des systèmes complexes : réseaux électriques intelligents

Un *Smart Grid*, ou réseau électrique intelligent, dispose d'unités de contrôle (centralisées ou distribuées) qui interagissent en temps réel. Ces interactions ont lieu d'une part entre ces composants et le réseau électrique pour prélever des informations (pilotage à distance du réseau et reconfiguration de sa topologie), et d'autre part avec certains clients pour offrir une flexibilité au réseau de distribution (résolution ou limitation de contraintes locales de tension ou de transit), en utilisant un système de communication performant. Les *Smart Grids* comportent à la fois des composants physiques (dispositifs du réseau électrique) évoluant en temps continu, et des composants numériques évoluant en temps discret (réseau de télécommunication et unités de contrôle), hautement connectés avec le monde physique environnant. Au vu de la complexité d'un tel système, la première exigence est de le maîtriser puis de l'exploiter à l'optimum afin de valider son fonctionnement. Néanmoins, pour pouvoir le simuler sans simplifier ni les modèles ni leurs interactions, il est nécessaire d'adopter un nouveau type de simulation dit « co-simulation » ou bien « multi-simulation » (co-simulation purement logicielle donc sans HIL, voir la section suivante).

3.1 Processus de validation pour les systèmes complexes

Dans la littérature, on trouve globalement trois possibilités de validation pour les systèmes complexes [27]. De la simulation de ces systèmes (MIL) à la simulation en temps réel (SIL et HIL) [27], on peut les résumer comme suit :

— ***Model-In-the-Loop*** (MIL) est une technique utilisée pour modéliser le comportement d'un système. Elle est adoptée pour valider par simulation les différents algorithmes de contrôle et les corriger avant la fabrication des prototypes.

— ***Software-In-the-Loop*** (SIL) est un type de processus qui comporte deux parties : la première existe dans l'outil de simulation natif et la deuxième partie est un code exécutable. Autrement dit, ce type s'assure que le code fonctionne correctement sans aucun matériel ; c'est une méthode efficace pour évaluer et critiquer un logiciel avant son utilisation dans le monde réel.

— ***Hardware-In-the-Loop*** (HIL) est une plate-forme qui exécute à la fois le modèle et le matériel. Elle est utilisée pour valider les performances du matériel en contrôlant les entrées de la simulation en temps réel [2], c'est-à-dire qu'elle combine le *hardware* et le *software* dans le but d'avoir une analyse complète du système. Avec cette approche, on pourrait détecter les anomalies tout au long de son développement avant même l'installation du matériel sur le terrain [45]. Cela évitera à la fois la perte de temps et le coût inhérent à sa réparation ou à sa modification. C'est une technique efficace dans les systèmes de contrôle-commande, comme dans les applications des véhicules électriques hybrides [57].

3.2 C'est quoi une co-simulation ?

La co-simulation est une technique de simulation des systèmes complexes, permettant de combiner plusieurs modèles hétérogènes réalisés avec des outils différents et par des experts issus de domaines variés. Elle est considérée comme un environnement unique, regroupant des simulateurs existants afin d'éviter de redévelopper à nouveau tous les composants de ce système.

La multi-simulation est similaire à la co-simulation, mais sans configuration (HIL) (*Hardware-In-the-Loop*). Dans le présent manuscrit, nous utilisons le terme « co-simulation » qui englobe celui de « multi-simulation ». À noter que seul le terme de co-simulation est connu dans le monde Anglo-Saxon. Les différents outils de la co-simulation échangent des données et interagissent fortement avec les autres composants du système par un envoi de messages. Notons également que nous travaillons, dans cette thèse, avec la co-simulation de *uses cases* (cas d'utilisation) des réseaux électriques intelligents métiers fournis par EDF R&D.

D'un point de vue informatique, une co-simulation se présente comme l'exécution d'un graphe de tâches hétérogènes fortement couplées. Une co-simulation comprend trois phases principales, décrites dans l'article [8] et présentées ci-dessous :

1. La phase d'initialisation : cette étape primordiale permet d'élaborer le graphe de tâches (préparer les communications des composants), et d'initialiser les entrées et les sorties du système global.

2. Désigne une simulation où le temps de la simulation progresse exactement comme le temps physique.

2. La phase de co-simulation proprement dite qui permet de lancer l'exécution de tous les composants de notre système.
3. La phase de clôture représente la fin de la co-simulation globale et le rapatriement des résultats.

De manière générale, la co-simulation permet de valider le bon fonctionnement des systèmes complexes hybrides.

3.3 Contraintes imposées par les *Smart Grids*

Étant donné que cette thèse est liée directement à EDF, nous prenons en compte les contraintes imposées par l'environnement industriel des *Smart Grids* (contraintes des systèmes événementiels et des systèmes physiques). Il s'agit donc de combiner des logiciels simulant des composants événementiels de type événement et contrôle-commande avec des logiciels simulant les composants électriques (physiques). Ces modèles de simulation sont soumis aux contraintes citées dans les sous-sections suivantes.

3.3.1 Co-simulation de systèmes physiques

La simulation des dispositifs physiques s'appuie sur des solveurs qui discrétisent le temps continu en intervalles de temps dits pas de temps. La figure 2 précédente illustre un modèle de temps d'un composant physique avec un pas de temps (temps de calcul et communication) constant noté (t_{cc} ou h_i) à l'itération i. Les **solveurs** numériques de ces dispositifs physiques utilisent des pas de temps internes notés t_{ci} pour résoudre les équations de la physique (ODE [3], DAE [4]) [56], c'est-à-dire des composants qui intègrent et mélangent des équations différentielles du temps avec des équations algébriques. À l'issue d'un pas de temps (h_i), ces composants physiques communiquent et échangent des données. Ainsi, les solveurs des composants physiques peuvent utiliser des pas de temps constants, ou des pas de temps auto-adaptatifs (variables). Notons qu'un pas de temps ne doit être ni trop petit pour ne pas ralentir la co-simulation, ni trop grand pour éviter de causer des erreurs dans les calculs [29].

Dans le cas où les simulateurs sont synchrones, tous les composants du modèle utilisent et partagent le même pas de temps (h_i). Lorsque celui-ci s'achève, les composants envoient leurs sorties (résultats) vers les entrées d'autres composants. Cependant, l'exécution de cette méthode peut entraîner beaucoup de communications, surtout si le pas de temps est petit. En revanche, si le système comporte des simulateurs asynchrones, la co-simulation peut théoriquement choisir des pas de temps variables. Mais dans ce cas, il est très

3. Ordinary Differential Equation
4. Differential Algebraic Equations

difficile d'implémenter une solution qui synchronise les données échangées au sein du même système, surtout lorsque les composants utilisent des pas de temps variables avec la possibilité de retourner en arrière (déclencher le mécanisme de *rollback*).

Un système contient plusieurs composants physiques qui s'influencent mutuellement, mais selon deux types de couplages, à savoir le couplage fort et le couplage faible. Ils sont résumés comme suit :

— **Couplage fort et fréquent**
Dans ce type de couplage, les composants physiques du système fonctionnent avec un pas de temps unique (h_i) pour tous les composants du système, en s'échangeant fréquemment leurs résultats intermédiaires après chaque pas de temps. On peut utiliser des pas de temps constants ou des pas de temps auto-adaptatifs (petit).

— **Couplage faible**
Dans ce type de couplage, les composants physiques du même système sont d'avantage autonomes, fonctionnent de manière relativement indépendante, et avec une contrainte de temps relâchée. Le couplage faible se prête bien à des solutions de type SMA [5] et on trouve des solutions qui utilisent des pas de temps différenciés [6].

Dans les travaux de cette thèse, nous utilisons nécessairement un couplage fort et fréquent à cause des constantes de temps des composants physiques sous-jacentes.

Dans les systèmes complexes, les composants physiques génèrent souvent des événements de type ***state events*** vers les composants discrets. Par exemple, si la température dépasse un certain seuil, le composant physique déclenche une alarme qui représente un événement interne envoyé au contrôleur. Il est impossible de définir la véritable date d'apparition d'un événement interne (comme une alarme), car celui-ci peut apparaître au milieu d'un pas de temps (h_i). Le modèle d'un simulateur de système physique ne peut donc pas connaître ces événements à l'avance. À l'inverse, les composants de contrôle génèrent souvent des événements de type *time events* facilement prédictibles.

Mécanisme de *Rollback*

Dans le cas d'un pas de temps auto-adaptatif ou variable, un simulateur de sous-système physique peut revenir en arrière et adopter un pas de temps plus petit s'il détecte une perte de précision dans ses calculs (le système à l'instant $(T + h_i)$ revient à un état antérieur à l'instant (T)). Avec la co-simulation, tous les composants doivent évoluer en parallèle entre deux instants de temps t_{cc1} et t_{cc2}, et chaque composant vérifie la qualité de ses calculs (Fig.3 étape 1) avant de passer au pas de temps suivant ou de

5. Systèmes Multi-Agents.
6. Les pas de temps ne sont pas les mêmes pour tous les composants.

faire un *rollback* (Fig.3 étape 2). Par conséquent, si un de ces composants effectue un *rollback* et adopte un pas de temps plus petit, il obligera tous les autres composants à le suivre et à revenir en arrière (*rollback*) pour ré-exécuter leur modèle avec le nouveau pas de temps ($h' < h$) (Fig.3 étape 3). Ce mécanisme nécessite la sauvegarde de l'état du simulateur à chaque début de pas de temps (*checkpoint « ch_i »*), comme l'illustre la figure 3.

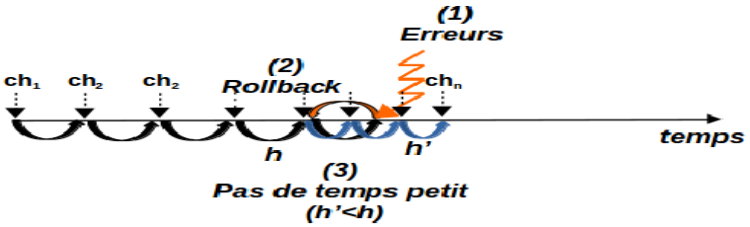


FIGURE 3 – Fonctionnement d'un composant physique (continu)

3.3.2 Co-simulation de systèmes événementiels

Les systèmes événementiels se fondent sur des événements dont le traitement est assuré par un simulateur événementiel dit « discret ». Ce type de système peut émettre des événements de contrôle à destination des composants physiques à tout moment. Généralement, les événements envoyés par des simulateurs discrets sont représentés sous forme d'impulsions idéales (type *Dirac*), comme le montre la figure 4, et se traduisent par des changements d'états.

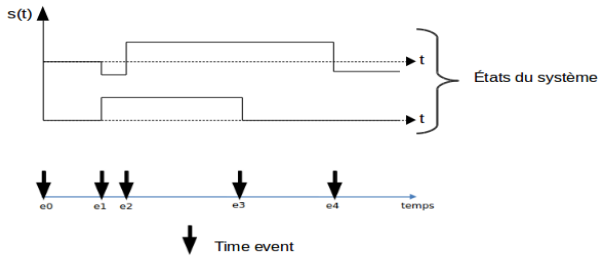


FIGURE 4 – La représentation du temps dans des simulateurs événementiels

Principe du *Look ahead*

Dans un système événementiel, chaque composant déclare un *look ahead* (L), qui est un espace temporel où ce composant n'émettra aucun message. Ce *look ahead* est primordial pour traiter les événements des composants de façon simultanée. Par conséquent, si ce paramètre est grand, il améliore les performances de la co-simulation globale, en permettant d'exécuter plusieurs composants en parallèle. Toutefois, la réactivité réelle de

ce système devient lente car les événements urgents ne doivent être pris en compte qu'à la fin de ce *look ahead* : le système réagit avec un décalage et ne peut émettre un nouveau *time event* que (L) secondes après avoir reçu un *state event* de la partie continue. Si au contraire le *look ahead* est de taille trop restreinte, il est impossible de favoriser le parallélisme car l'exécution devient séquentielle et lente, alors que la réactivité du système devient plus rapide et le traitement des événements se fait au fur et à mesure [29]. La figure 5 illustre un composant événementiel qui ne peut envoyer des messages qu'à partir de leur temps réel (T) plus leur *look ahead* ($T + L$) [77]. Toutefois, pour modéliser les systèmes réels de manière réaliste, il est nécessaire de maintenir un L faible. Notons que le composant « *fedarate* » sur la figure 5 représente le simulateur événementiel.

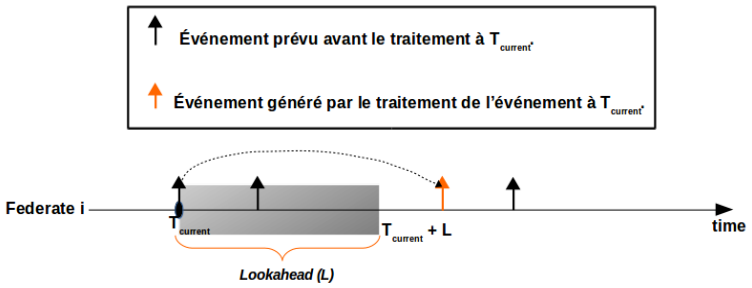


FIGURE 5 – Principe du *Lookahead*

4 Standards de co-simulation

Dans cette section, nous introduisons deux standards permettant de réaliser une co-simulation distribuée : *Functional Mock-up Interface* et *High Level Architecture*. En s'appuyant sur ces normes, il est possible d'effectuer une co-simulation, soit en développant un « master global » qui gère les interactions des composants et orchestre la co-simulation (cas du FMI), soit en utilisant un *middleware* de synchronisation déjà existant dans la norme pour synchroniser les communications (cas du HLA).

4.1 *Functional Mock-up Interface*

La norme FMI [7] [9] ou *Functional Mock-up Interface* est un outil de co-simulation qui a été défini dans le projet européen MODELISAR [8] de ITEA2 [9] ensuite repris en maintenance évolutive par le projet de l'association de MODELICA. Cet outil est une spécification *open source* qui permet de combiner différents modèles de simulation en les encapsulant dans des composants pour en faire des FMU (*Functional Mock-up Unit*),

7. https ://www.fmi-standard.org/
8. https ://itea3.org/project/modelisar.html.
9. Information Technology for European Advancement

comme l'illustre la figure 6. Cette norme simplifie les coopérations entre les différents composants, en s'appuyant sur le principe ($maître - esclave$) [8] présenté dans la section 5. Notons que, dans le contexte du projet, la fabrication des FMU se fait généralement avec le langage Modelica [10], un langage de modélisation des systèmes complexes.

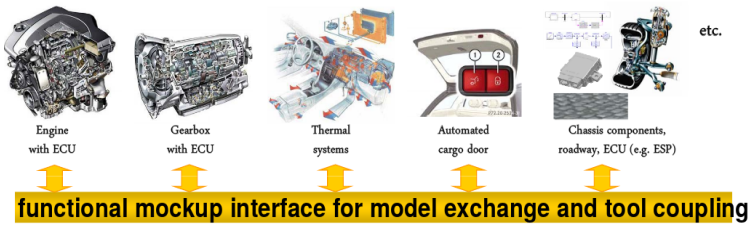


FIGURE 6 – *Functional Mockup Interface (FMI)*

Composant *Functional Mock-up Unit*

La norme FMI encapsule les composants d'un modèle dans des boites grises [11] appelées FMU. Une FMU est un dossier zippé qui contient les éléments suivants :

1. Un fichier XML nommé *modelDescription.xml* qui décrit toutes les variables de la FMU : des variables externes (entrées/sorties) ou des variables internes accessibles par le biais de l'interface définie par le standard FMI.
2. Un ensemble de codes binaires dépendants des systèmes d'exploitation (Linux ou Windows), offrant les fonctions de calcul du FMU ainsi que les primitives de son interface.
3. Des informations spécifiques comme la documentation et des icônes.

Modes de fonctionnement des FMU

Le principal atout de cette norme est de supporter deux modes de composants : le mode ME (*Model Exchange*) et le mode CS (*Co-simulation*)

1. **FMI pour le mode *Model Exchange*** : ce mode illustré par la figure (7-a) contient le modèle décrit de manière standardisée et c'est à l'utilisateur de développer son propre solveur. Ce mode est dit FMI-ME.
2. **FMI pour le mode *Co-Simulation*** : ce mode illustré par la figure (7-b) encapsule le modèle et un solveur numérique dans la même boite (FMU). Il préserve la confidentialité du modèle en protégeant sa propriété intellectuelle PI [12] (il ne communique pas les sources du modèle). Ce mode est dit FMI-CS.

10. https ://www.modelica.org/.
11. On connaît quelques informations qui se trouvent dans un fichier XML fourni par le fabricant des FMU ou par EDF.
12. Propriété Intellectuelle

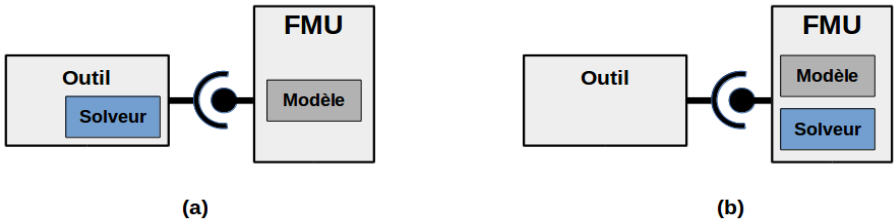


FIGURE 7 – Mode de co-simulation : (a) mode *Model Exchange* (b) mode *Co-Simulation*

Avec la norme FMI, on doit définir et implanter un « master global » qui sert à interfacer les FMU, établir des connexions et échanger des données entre les différents modèles interagissant pour composer le méta modèle de la co-simulation [8].
Quand on utilise la norme FMI, on est obligé d'intégrer *wrappers* qui encapsulent l'API du standard afin de manipuler ces FMU. Par exemple, en Java, on trouve JFmi, JavaFMI, et en C/C++, on trouve fmi++, etc. Toutes ces API n'offrent toutefois pas les mêmes fonctionnalités : certaines ne sont compatibles qu'avec certaines versions du standard ; d'autres n'implémentent pas toutes ses primitives. Dans notre cas, nous utilisons le package JavaFMI [13] en version Windows et en version Linux car il implémente la norme FMI-CS sans restrictions.

EDF a choisi la norme FMI pour le mode *Co-Simulation* car il intégre le modèle et son solveur numérique dans la même FMU, et permet aussi une meilleure protection de la PI. Dans cette thèse, nous utilisons un logiciel de co-simulation qui : (1) supporte le mode FMI-CS, (2) exploite les solveurs métiers de façon optimiste (retourner en arrière en cas de problèmes de précision dans les calculs), et (3) offre des possibilités de parallélisme et de distribution que n'autorise pas un solveur générique.

Limites de la norme FMI

L'émergence de la norme FMI permet à un utilisateur de combiner plusieurs modèles de domaines différents en se libérant des contraintes d'interfaçage et de synchronisation. Malgré les avantages très significatifs apportés à la simulation des systèmes complexes, la partie FMI-CS de la norme présente encore quelques limitations majeures qui sont indiquées ci-après :

— Gestion de couplage entre les composants physiques et les composants événementiels.

— Gestion précise des événements qui apparaissent au milieu d'un pas de temps.

De plus, on peut lister certains problèmes d'implantation :

13. http ://bitbucket.org/siani/javafmi/wiki/Home

— Il reste à la charge du développeur de définir un « master global » qui traduit le modèle structuré du système pour gérer toutes les communications échangées entre les composants du système, et orchestrer la co-simulation globale.

— L'utilisateur ne peut pas accéder à toutes les données des FMU parce qu'il voit la FMU comme une boite grise, et n'a accès qu'aux variables exposées par les FMU.

— Équilibrage des calculs (l'utilisateur doit découper ses modèles pour éviter les trop grosses FMU).

4.2 *High Level Architecture*

Cette section décrit brièvement la norme HLA (*High Level Architecture*[23]), qui a été developpée par DMSO [14] pour le département Américain de la défense DoD [15]. Elle représente un outil industriel de co-simulation distribuée qui se compose globalement d'un ensemble de composants à base de modèles événementiels.
Le principal objectif de HLA est de développer une plate-forme distribuée facilitant l'interopérabilité et la réutilisabilité de simulateurs hétérogènes dans des applications. Pour réaliser une co-simulation type HLA, nous devons avoir trois éléments importants qui se résument comme suit :

— **Federate** : Dans la terminologie de la norme HLA, chaque modèle de simulation ou un système réel individuel est encapsulé dans un composant dit *federate* [16]. Ces participants (*federates*) interagissent entre eux par l'intermédiaire du bus de co-simulation (RTI) ; un *federate* peut être de nature très différente des autres *federates* se trouvant dans le même système.

— ***Run-Time Infrastructure (RTI)*** : Les *federates* communiquent entre eux par des envois d'événements, à travers un bus de co-simulation dit *Run-Time Infrastructure*. Autrement dit, les requêtes émises par ces *federates* seront routées vers leurs destinations grâce à une interface de communication RTI, qui représente un point central. Ce bus de communication assure le contrôle de l'ensemble des *federates*, et garantit le bon fonctionnement de la co-simulation globale. Le RTI de la norme HLA permet à la fois de faire communiquer et de synchroniser les différents *federates* du système.

— La **fédération** regroupe et fait communiquer un ensemble de *federates* par un bus de communication RTI.

La figure 8 ci-dessous illustre l'architecture globale d'une co-simulation de type HLA qui

14. *Defense Modeling and Simulation Office*
15. *Department of Defence*
16. Un objet qui évolue dans le temps au cours de la simulation.

contient trois simulateurs sous forme de *federates* interconnectés entre eux par le bus d'événements *RTI*.

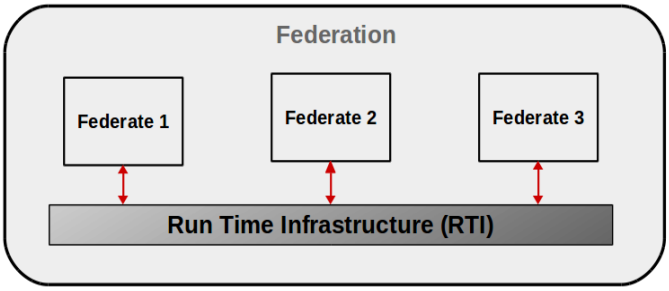


FIGURE 8 – Exemple d'une fédération HLA contenant trois simulateurs

Spécification de la norme HLA

Afin que l'ensemble des *federates* puissent s'interfacer avec le bus RTI, l'architecture HLA définit trois principaux composants permettant de garantir la correspondance sémantique de la co-simulation ; ces derniers sont présentés ci-après (d'après *Okan Topçu* et al [73] dans le chapitre 3) :

— Un ensemble de **règles** décrit les spécifications d'une fédération HLA, et les conditions pour ses participants (*federates*).

— Un modèle objet dit ***Object Model Template*** **(OMT)** spécifie l'ensemble des objets communs utilisés par une fédération afin de garantir l'interopérabilité de toutes les données du système. Notons que les fédérations doivent disposer d'un modèle d'objet (FOM) [17] de la fédération HLA pour définir les données du modèle qui sont conformes au modèle objet (OMT).

— ***Interface Specification*** **(IFSpec)** représente une API. Elle fournit les différents services pour interfacer le bus d'événements (RTI) avec l'ensemble des *federates*.

Gestion du temps

La gestion du temps est ancrée dans le bus d'événements RTI qui permet à l'ensemble des *federates* de progresser dans le temps. Ce bus RTI définit une horloge logique globale et il s'appuie soit sur une méthode conservative, soit sur une méthode optimiste qui viole la contrainte de causalité (voir la section 2 de ce chapitre). De manière générale, dans une fédération, l'avancement du temps se réalisera selon les mécanismes ci-dessous :

17. *Federation Model Template*

- Chaque *federate* a sa propre horloge locale interne appelée temps logique du *federate*, et les différents *federates* sont enchaînés entre eux par un temps coordonné [28].

- Généralement, la progression du temps d'un *federate* est définie par l'estampille du **message** reçu.
- Si le bus RTI utilise une méthode optimiste, la simulation progresse plus vite mais un ou plusieurs *federates* peuvent recevoir des messages dans le passé, cela provoquera des *rollbacks* pour que les *federates* reviennent à leur état cohérent. Ce type de méthode est peu utilisé dans la communauté HLA, car sa mise en œuvre devient rapidement complexe.
- On trouve, le plus souvent dans la littérature, des RTI qui s'appuient sur des méthodes conservatives qui n'acceptent pas de revenir en arrière et d'effectuer un *rollback*.

Limites de la norme HLA

Malgré sa capacité à traiter des systèmes complexes, plusieurs limitations pénalisent l'utilisation de cette norme.

1. La qualité du bus RTI : les spécificités se trouvant dans ce bus à événements sont souvent liées à son prix (les RTI libres sont les plus simples).
2. L'incompatibilité de cette norme avec d'autres simulateurs : On doit utiliser des simulateurs qui sont compatibles avec la norme HLA, or ce n'est pas tous les simulateurs existants qui peuvent fonctionner avec cette norme.

Il existe dans la littérature un formalisme qui est très utilisé dans les systèmes complexes, appelé le formalisme DEVS. DEVS ou *Discrete EVent System specification* a été introduit par *B.P. Zeigler* [79] pour modéliser un système quelconque. Il représente un formalisme modulaire qui permet de construire des modèles élémentaires ou composés. DEVS est un formalisme à événements qui considère que les informations échangées sont sous forme d'événements. Afin de reproduire le comportement exact du système, ce formalisme permet de décrire son modèle structurel. Il permet en fait d'interconnecter différents modèles atomiques entre eux. Le modèle DEVS est un 7-uplet de la forme suivante :

$$M =< X, Y, S, ta, \delta_{ext}, \delta_{int}, \lambda > \tag{1.1}$$

- X, Y représentent l'ensemble des événements entrants et sortants.
- S est l'ensemble des états du système.

- ta (*time advance*) désigne la durée de vie d'un état s_i.
- δ_{ext} représente la transition d'un état externe, c'est-à-dire une transition qui décrit la modification de l'état du système par un événement à son entrée.
- δ_{int} est la transition d'état interne qui révèle le changement de l'état du système pendant la durée de vie de l'état ta.
- λ désigne la fonction de sortie (les sorties du système).

La figure 9 illustre un schéma du DEVS. Il existe des extensions de ce formalisme, et

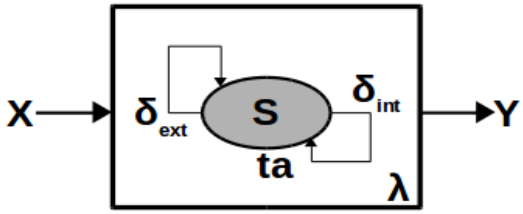


FIGURE 9 – Schéma du principe du formalisme DEVS

on trouve : DESS (*Differential Equation System Specification*) [79] pour les systèmes dynamiques à temps continu, DEVS&DESS (*Discrete EVent and Differential Equation Specified Systems*) [80] pour les modèles hybrides, G-DEVS (*Generalized Discrete EVent System Specification*) [78] pour résoudre les équations différentielles, etc.

5 Types de synchronisation pour la co-simulation

Cette section synthétise les trois principales techniques de synchronisation connues et adoptées par les différentes plates-formes de co-simulation présentées dans la section 6 [49].

1. **Approche maître-esclave** :
   Les composants du système sont des esclaves. Un programme dit « maître » contrôle et orchestre l'ensemble des communications de toute la co-simulation. Dans la figure 10-a, le programme qui supervise tous ces échanges se trouve dans le simulateur de réseau de télécom, comme dans l'article [51]. Notons que les nombres sur cette figure reflète l'ordre d'exécution des parties de la co-simulation globale.

2. **Approche de *Time-stepped*** :
   Les simulateurs individuels du système global exécutent leurs simulations de manière indépendante, mais ils arrêtent leurs simulations à des instants prédéterminés, i.e. des points de synchronisation. Ces points de rendez-vous représentent les instants auxquels tous les simulateurs peuvent échanger des données entre eux, comme l'indique la Fig. 10-b. Ce mécanisme peut engendrer des erreurs cumulées dans le

système. Ce phénomène d'accumulation intervient en particulier lorsque le pas de temps est trop important par rapport à la dynamique du système. Ce mécanisme est aussi une source d'imprécision si l'application nécessite des interactions imprévues et urgentes. Pour cette raison, la plupart des plate-formes ont renforcé cette méthode par l'ajout d'autres mécanismes (*middlware*), comme dans le cas de la solution présentée dans l'article [50].

3. **Approche *Global event-driven*** :
   L'idée de ce mécanisme est de créer une liste globale qui sauvegarde à la fois les pas de temps venant des composants physiques, et les événements produits par des composants événementiels. Cette approche permet de traiter l'ensemble de ces données (requêtes) de manière séquentielle. Elle ne peut traiter qu'une seule requête à la fois, comme l'illustre la figure 10-c. Les autres requêtes placées en file d'attente doivent attendre la fin de l'exécution pour être traitées par ordre chronologique. Ce mécanisme évite les erreurs engendrées par la méthode *Time-stepped*, mais reste relativement limité par sa nature séquentielle et l'impossibilité de passer à l'échelle, comme c'est notamment le cas de la plate-forme [52].

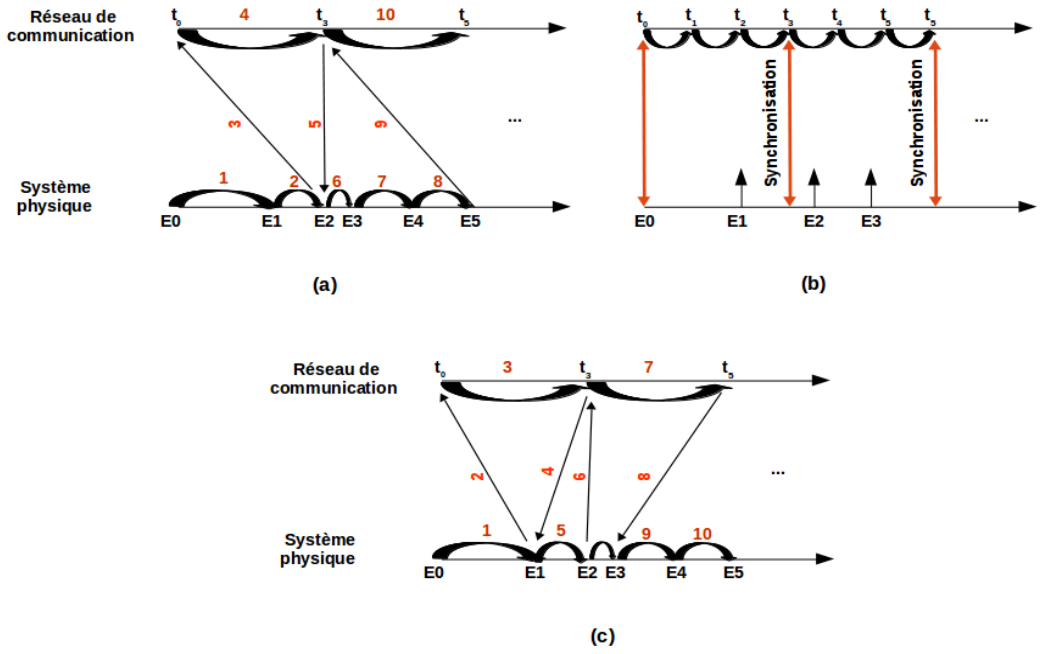


FIGURE 10 – Types de synchronisation : (a) Approche de Maître-esclave (b) Méthode de *Time-stepped* (c) Méthode *Global event-driven*

6 Solutions pour la question du temps dans une co-simulation

Les difficultés majeures d'une co-simulation sont l'interopérabilité et l'interfaçage des composants hétérogènes entre eux, afin de les présenter comme un seul environnement de co-simulation. Néanmoins, cette intégration des composants hétérogènes s'appuie sur des modèles hybrides qui font progresser l'ensemble de la co-simulation. Dans cette section, nous nous sommes intéressés à la gestion du temps des premières solutions présentées dans la littérature. Pour cela, nous avons classifié ces différentes solutions en quatre catégories : celles fondées sur la norme HLA, celles fondées sur la norme FMI, celles fondées sur les deux normes (HLA et FMI), ou encore les solutions de type *Ad Hoc* fondés sur leurs propres modèles de temps (voir la figure 11). Notons que les performances de ces solutions sont rarement évoquées et décrites dans la littérature. C'est pourquoi nous avons développé une solution *open source* de co-simulation orientée performances, reposant sur le standard FMI (DACCOSIM).

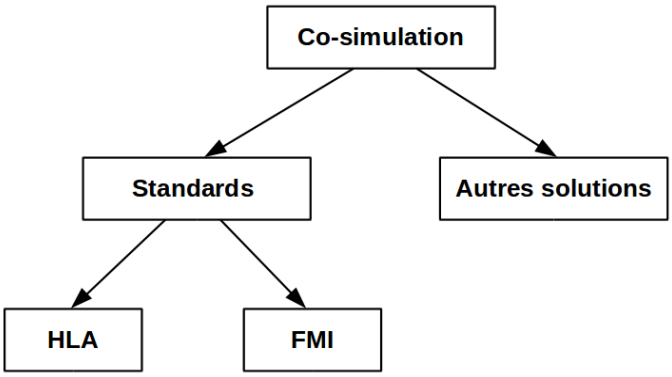


FIGURE 11 – Solutions de co-simulation

6.1 Solutions centrées HLA

Beaucoup de solutions exposées dans la littérature adoptent la norme HLA car elle facilite la portabilité et la distribution des composants (*federates*), et offre ainsi le modèle de temps pour interconnecter les différents dispositifs du système complexe (implémentation du bus RTI).

Bridge HLA (2003)

Il existe dans la littérature une solution multi-HLA qui permet de connecter plusieurs fédérations entre elles, chaque fédération utilise le même principe de gestion de temps présenté dans HLA (présenté dans la section 4.2). Les auteurs de l'article [25] définissent leur système comme une large fédération ou une inter-fédération qui comporte plusieurs

sous-fédérations. Chacune contient à la fois un ensemble de *federates* et son propre bus RTI. Dès lors, la solution pour connecter ces différentes fédérations est d'utiliser un pont HLA entre chaque couple de fédérations ; cette dernière est alors vue par les autres composants comme une simple fédération. Cette solution est particulièrement adaptée pour réduire le trafic du réseau et traiter des systèmes à large échelle. L'architecture d'une inter-fédération simple, basée sur un seul pont, est décrite dans la figure 12. Elle se compose de deux fédérations (1 et 2) liées entre elles grâce à un pont HLA qui permet d'observer le comportement de chaque fédération afin de le reproduire auprès d'autres fédérations [13].

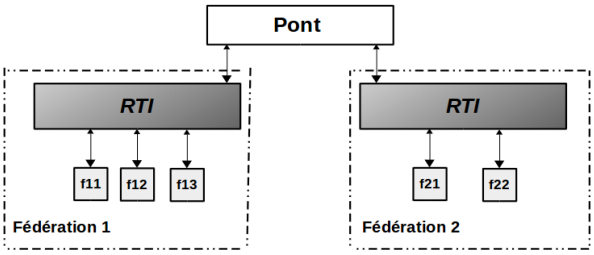


FIGURE 12 – Exemple : Deux fédérations reliées par un seul pont

EPOCHS (2003-2006)

Nous trouvons dans la littérature la première plate-forme de co-simulation qui traite à la fois des événements et des pas de temps. Développée en 2006, elle se nomme EPOCHS pour *Electric POwer and Communication Synchronizing Simulator*. EPOCHS [41, 42] est un environnement qui agrège un simulateur des systèmes électromagnétiques (PSCAD/EMTDC), un simulateur des systèmes électromécaniques(PSLF) et un simulateur de réseau de télecommunication (NS-2). Ces trois simulateurs hétérogènes sont supervisés par un composant de contrôle dit *Agent Head Quarter (AgentHQ)*, comme l'illustre la figure 13. *AgentHQ* présente un environnement unifié aux agents et agit comme un proxy lorsque des agents interagissent avec d'autres composants EPOCHS. Grâce à ce dernier, les agents peuvent obtenir et définir les valeurs du système d'alimentation, et peuvent envoyer et recevoir des messages les uns aux autres. Autrement dit, il correspond en fait à un « Master » qui coordonne et fait communiquer les différents agents du système. Toutefois, ces différents simulateurs sont connectés entre eux par le bus RTI de la norme HLA, qui a pour but de synchroniser et router toutes les données de chaque composant vers d'autres composants du système en utilisant l'approche *Time-stepped*. Ces différents simulateurs échangent des données hétérogènes dans les points de synchronisation (voir la figure 14). Autrement dit, elle utilise l'approche *Time-stepped* permettant d'exécuter ses fédérates jusqu'à ce qu'ils atteignent le point de synchronisation. Comme indiqué dans

l'article [41], le modèle de temps global de cette plate-forme est dominé par la méthode fixe où on ne peut pas interrompre un pas de temps pour traiter les événements émis entre temps par des simulateurs événementiels. Compte tenu de cela, il faut attendre jusqu'à la fin du pas de temps pour prendre en compte et router ces événements en décalant leur traitement au prochain pas de temps ; ce décalage et les erreurs qu'il entraîne constituent une limite de cette solution.

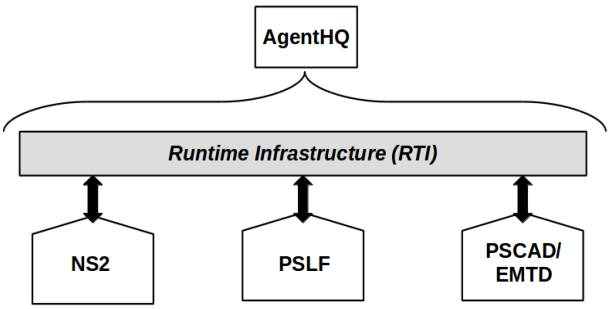


FIGURE 13 – La plate-forme EPOCHS [41]

Au point de synchronisation, ces différents composants envoient leurs données de sorties aux entrées des autres composants, et les événements produits pendant l'exécution ne vont être pris en compte qu'à la fin de ce point de synchronisation. La figure 14 illustre la synchronisation réalisée par la plate-forme EPOCHS ; la détection tardive de ces événements engendre des erreurs dans le système global. Cette plate-forme ne traite pas les événements apparus durant un pas de temps ce qui n'est pas compatible avec la réalité.

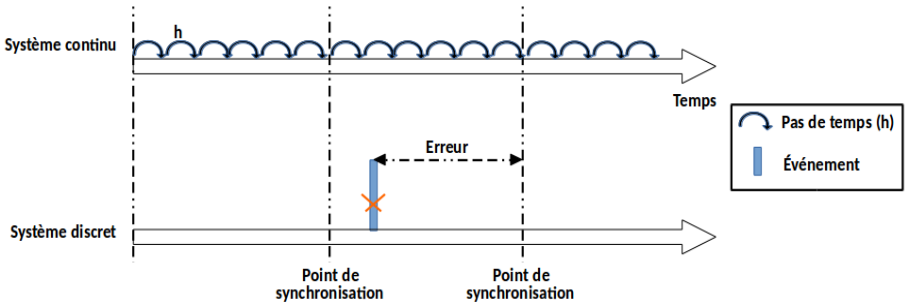


FIGURE 14 – Synchronisation des données dans EPOCHS [41]

Cependant, les composants de cette solution peuvent être exécutés sur une architecture distribuée pour accélérer la co-simulation globale du système (supporter une co-simulation large échelle) si le RTI utilisé est bien distribué. EPOCHS a justement été défini pour évaluer les performances des systèmes multi-agents (SMA).

INSPIRE (2012-2013)

Les auteurs de l'article [32] ont proposé une solution industrielle, dédiée à des *Smart Grids* conforme à une architecture réelle. Cette solution permet de co-simuler des composants électriques (simulé en *PowerFactory*) avec un réseau de communication (simulé en OPNET [18]) en se basant sur les protocoles qui respectent la norme IEC 61850. Cette norme s'installe de plus en plus dans les réseaux de transport et de distribution d'électricité en intégrant les exigences de communications entre les éléments du système, pour le contrôle-commande et les systèmes de protection [60]. Comme toutes les solutions centrées HLA, les auteurs de cette proposition adoptent le modèle de temps de la norme HLA, présenté par le bus RTI pour synchroniser l'ensemble des *federates*, comme la plate-forme EPOCHS. Ils ont représenté l'architecture hybride du simulateur en trois modules principaux : (1) le noyau de simulation (2) la couche réseau, et (3) la couche de gestion. Chaque module a été conçu pour fournir des fonctions individuelles pour la fédération globale. Notons que la co-simulation est pilotée par le système électrique : l'exécution des deux simulateurs se fait de manière séquentielle et alternée.

Les mêmes auteurs que l'article [32, 31] ont amélioré leur solution et ont proposé la plate-forme INSPIRE [37], qui est l'abréviation d'*Integrated Co-Simulation of Power and ICT Systems for Real-Time Evaluation*. Le principal objectif est d'assurer un ordre d'exécution des événements (TSO [19]) de toute la co-simulation afin d'obtenir des résultats de simulation reproductibles. Pour assurer un *TSO* pour une co-simulation distribuée, les auteurs de cet article adoptent une méthode conservatrice qui utilise les services de gestion de temps HLA. Elle permet de synchroniser leur composants avec la fédération et d'éviter les blocages. Dans le cas de la simulation du système physique, l'intervalle des pas de temps peut être ajusté en fonction de la précision et de la vitesse souhaitées de la simulation. Cette plate-forme oblige tous ses composants (*federates*) à proposer un *look ahead* afin d'éviter à la fois (1) l'interblocage de l'ensemble des simulateurs (on ne peut plus émettre de nouvel événement) et (2) l'apparition d'erreurs dans le système. Étant donné que les événements d'Opnet ne peuvent pas être traités tant qu'on a pas encore fini l'exécution, les auteurs de ce papier ont ajouté un autre paramètre dit MTAR [20] pour configurer la gestion du temps dans ce simulateur événementiel OPNET, comme l'indique la figure 15. Ce paramètre offre la possibilité de limiter cette période de synchronisation en mettant les événements apparus dans une file d'attente, elle définit en fait la durée maximale pour que la simulation puisse avancer dans le temps. Cette solution utilise des méthodes à pas de temps constants.

18. *OPtimized Network Engineering Tools*
19. TimeStamp Order
20. *Maximum Time Advance Request*

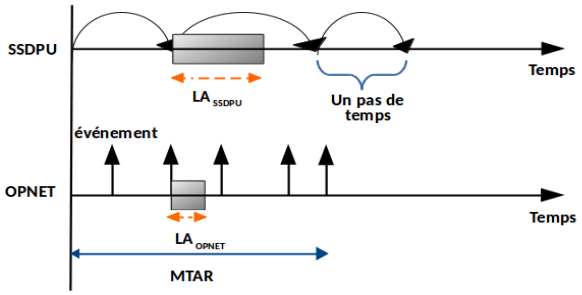


FIGURE 15 – La synchronisation en utisant la plate-forme *INSPIRE* [37]

6.2 Solutions centrées FMI

Le premier article introduisant la co-simulation en utilisant la norme FMI a été publié en 2011. Dans cet article [8], les composants sont représentés par des FMU qui communiquent entre elles grâce à un master global dit « maître », codé par les développeurs. Ses auteurs décrivent leur modèle par un graphe de FMU où les nœuds sont les composants (des FMU), et les arêtes représentent des données échangées entre ses derniers. Toutefois, pour lancer ces FMU, les auteurs ont rajouté des priorités à leurs composants. Ils ont noté que les FMU qui ont la même priorité s'exécutent en parallèle, et ils traitent des modèles de type *« data flows »*. Cette solution s'appuie sur une approche « *maître-esclave* » présentée dans la section précédente. Le « master » (ou maître) de cette solution s'appuie sur deux algorithmes, à savoir l'algorithme de *Gauss-Seidel* et l'algorithme de *Jacobi*[21].

Le logiciel de co-simulation *DACCOSIM* que nous avons utilisé dans cette thèse et qui sera abordé dans le chapitre 3, adopte la même approche que l'article [8]. La seule différence est que la solution décrite dans cet article concerne les *data flows* et ne fournit pas de mesures de performances ni sur une seule machine ni sur une architecture distribuée. À l'inverse, avec DACCOSIM, nous traitons des graphes de FMU indépendants, dont les FMU sont fortement connectées et réparties sur un cluster de PC multi-cœurs.

6.3 Solutions hybrides HLA/FMI

Il existe d'autres solutions décrites dans [6, 5, 2] combinant les deux normes présentées précédemment : la norme FMI pour coupler les composants physiques, et la norme HLA qui contrôle et orchestre la progression du temps de toute la co-simulation, mais en se fondant sur le mécanisme de gestion de bus RTI de la norme HLA. Les auteurs de la plate-forme C2WT [2] ont assemblé plusieurs simulateurs, et ont incorporé des FMU dans leur plate-forme en les encapsulant dans des féderates. C2WT est conçu comme une plate-forme de co-simulation à usage général permettant l'intégration de différents

21. Des méthodes itératives appliquées pour résoudre les systèmes linéaires.

simulateurs. Les recherches utilisant C2WT sont de nature militaire, en particulier dans le domaine des véhicules aériens sans pilote. Contrairement à notre équipe de projet DACCOSIM, ils traitent des FMU avec des pas de temps variables, mais qui s'exécutent sur une seule machine.

6.4 Solutions *Ad Hoc*

GECO (2010-2014)

À l'opposée des solutions présentées auparavant, il existe une solution nommée GECO [52] pour *Global Event-Driven Co-Simulation Framework*, qui utilise deux simulateurs : un simulateur physique qui est le PSLF et un simulateur de composants événementiels NS2. Cette plate-forme traite aussi la co-simulation dans les *Smart Grids*. Contrairement à EPOCHS, les auteurs de cette solution implémente leur propre modèle de temps. Ils utilisent une seule liste qui sauvegarde à la fois les pas de temps et les événements de manière séquentielle sans introduire de décalage, comme l'illustre la figure 16. Cette solution permet de résoudre les erreurs apparues au milieu d'un pas de temps. Mais elle ne peut pas passer à l'échelle et reste lente car elle est séquentielle et s'exécute sur une seule machine.

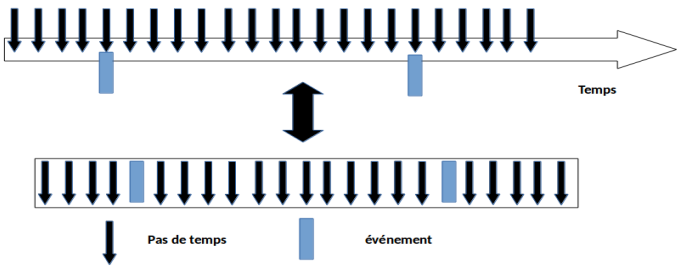


FIGURE 16 – La synchronisation en utilisant *GECO* [52]

VPNET

VPNET [50, 52] est un *framework* de co-simulation qui exploite deux simulateurs hétérogènes, à savoir le *Virtual Test Bed* (VTB) piloté par des instants de temps pour simuler le système multi-physiques, et OPNET piloté par des événements pour simuler le réseau de communication. En fait, l'outil VTB permet de simuler les systèmes d'énergie et les systèmes d'électronique de puissance. Les auteurs de l'article [50] ont intégré un mécanisme de synchronisation temporelle entre les simulateurs OPNET et VTB, i.e. un coordinateur de co-simulation mis entre VTB et OPNET basé sur des agents qui permettent de synchroniser des données échangées entre ces derniers. Ce mécanisme est similaire à la plate-forme « EPOCHS », et du coup, il accumule le même type d'erreurs

que celle-ci.

PowerNet (2007-2011)

Les auteurs de cet article [51] décrivent un autre environnement de co-simulation de *Smart Grids* appelé *PowerNet*. Ils réalisent un couplage *ad hoc* du simulateur réseau NS2 avec un simulateur de réseau électrique réalisé en Modelica. Les deux simulateurs s'exécutent dans des processus séparés, et ils communiquent entre eux via un pipe UNIX. Le simulateur NS2 pilote toute la co-simulation, cet environnement ne prend pas en compte les erreurs déclenchées par le simulateur physique (il ne traite pas les événements de type *state events*). La méthode de synchronisation adoptée par cette solution est à peu près la même que celle de la solution VPNET.

MECSYCO (2014-2015)

MECSYCO [16, 66] pour *Multi-agent Environment for Complex SYstems COsimulation* a été développée dans le cadre du projet MS4SG (*Multi-Simulation for Smart Grids*) entre le LORIA-INRIA et EDF R&D. Cette plate-forme est un environnement de multi-simulation événementiel permettant de regrouper différents modèles fortement connectés ; cette plate-forme est orientée multi-agents et basée sur le formalisme DEVS (voir la section 4.2 dans ce chapitre). Dès lors, cet environnement est représenté sous forme d'un ensemble d'agents autonomes avec de fortes interactions. Chaque agent de cette plate-forme gère un simulateur qui peut être différent de celui des autres agents, i.e. il repose sur le concept A&A [22]. En fait, les interactions entre les agents du système doivent être modélisées par des artefacts [66] (agents et artefacts). Ici, chaque agent est relié à un seul simulateur par un artefact de type « modèle », et il est relié aux autres simulateurs par un artefact de type « couplage ». Cet artefact de couplage permet de corréler les sorties d'un simulateur (modèle (M1)) vers les entrées d'un autre simulateur (modèle (M2)), comme l'illustre la figure 17.

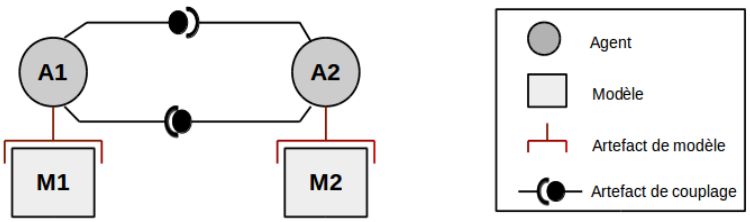


FIGURE 17 – Exemple de deux simulateurs hétérogènes interconnectés avec MECSYCO [16]

La gestion du temps dans MECSYCO se base sur une méthode conservative CMB (voir

22. Agents & Artefacts.

la section 2) respectant la contrainte de causalité, ce qui veut dire que cette plate-forme ne supporte pas le mécanisme de *rollback* en cas d'événements ou une dynamique trop importante du système (contrairement à la plate-forme DACCOSIM). MECSYCO utilise un *wrapper* similaire à un bus à événements qui se base sur des artefacts. Le point fort de la plate-forme MECSYCO est sa capacité à s'exécuter sur une architecture distribuée et hétérogène sous différents systèmes d'exploitation. Cependant, comme MECSYCO fait progresser le temps de la co-simulation avec une méthode conservative comme de nombreux bus d'événements, cette stratégie pénalise l'intégration des simulateurs physiques continus à pas de temps, qui parfois nécessitent le retour en arrière en cas d'écarts numériques trop importants dans les calculs des dispositifs physiques (suite à un événement ou une dynamique trop variable). Toutefois, afin d'avoir une bonne gestion d'événements, les auteurs de l'article [15] ont récemment implanté et intégré le mécanisme de *rollback* au niveau du modèle artefact.

7 Positionnement de notre solution

La plate-forme DACCOSIM est une solution centrée sur le standard FMI. Elle s'appuie sur un algorithme de « master global » pour faire avancer le temps de la co-simulation. Cette solution permet de coupler des composants synchrones hétérogènes (FMU) qui partagent le même pas de temps. DACCOSIM peut utiliser une méthode à pas variable et adopte une solution optimiste qui permet aux simulateurs de dispositifs physiques d'avancer dans le temps et de refaire leurs calculs en cas de perte de précision. Dans une co-simulation distribuée DACCOSIM, chaque FMU exécute un pas de temps en s'appuyant sur sa propre horloge interne, qui n'est pas nécessairement la même que celles des autres composants. Il s'appuie ensuite sur une horloge virtuelle globale gérée par le master de la co-simulation afin de synchroniser les communications entre FMU.

La synchronisation de DACCOSIM sera détaillée dans le chapitre 3 de ce manuscrit. Nous soulignons ici que nous n'avons trouvé aucun autre exemple de co-simulation distribuée à large échelle avec des composants en temps continu dans la littérature. DACCOSIM semble être le seul outil capable de réaliser des co-simulations distribuées et performantes, même à large échelle.

Récapitulatif

La synchronisation de la co-simulation peut se faire de trois façons différentes : avec la méthode maître-esclave, ou la méthode *time-stepped*, ou encore la méthode *Global event-driven*. Le Tableau 4 récapitule les différentes plate-formes trouvées dans la littérature [52, 49].

Plate-forme	Composants	Synchronisation	Distribué
EPOCH [41]	PSCAD, PSLF, NS2	*Time-stepped*	Oui
GECO [52]	PSLF et NS2	*Global event-driven*	Non
Inspire [37]	OPNET, PowerFactory	*Time-stepped*	Oui
C2WT [2]	*DIgSILENT* et *OPNET*	*Time-stepped*	Oui
MECSYCO[16]	composants événementiels	*Time-stepped*	Oui
PowerNet [51]	Modelica et NS2	maître-esclave	Limité
VPNET [50]	VTB et Opnet	*Time-stepped*	Limité
DACCOSIM [30]	FMU (continu et événementiel)	maître-esclave (avec maître hiérarchique et distribué)	Oui

TABLE 1.1 – Comparaison entre les solutions de co-simulation

8 Conclusion

Ce chapitre a été consacré à la co-simulation dans les réseaux électriques intelligents, plus particulièrement à la gestion du temps des simulateurs de systèmes complexes. La co-simulation permet principalement de combiner et d'interfacer de nombreux composants hétérogènes entre eux. Des approches *Ad Hoc* ou utilisant des standards de co-simulation ont été identifiées. Dans le contexte particulier imposé par EDF (standard FMI), un « master » de co-simulation spécifique qui gère toute la communication et l'orchestration du système, est nécessaire.

La présente thèse étant également orientée performance, l'étape suivante est donc d'analyser le comportement des composants de ce type de système sur une architecture distribuée. Mais avant cela, nous allons aborder un nouveau domaine qui est le placement des tâches sur une telle architecture informatique.

CHAPITRE 2

PLACEMENT DES TÂCHES DANS UNE ARCHITECTURE DISTRIBUÉE

Sommaire

1 Introduction

Le placement ou le *mapping*[1] des tâches sur une architecture distribuée prend de plus en plus d'importance en raison des exigences croissantes des applications industrielles dédiées aux grands systèmes complexes, comme dans le cas des co-simulations de *Smart Grids*.

Depuis quelques années, le placement des tâches se fait sur un cluster de PC multi-coeurs. Passant de processeurs mono-cœur à des processeurs multi-cœurs puis à des PC en réseau, ces améliorations ont permis de traiter ce type d'applications, en accélérant leur exécution. L'affectation de tâches sur les nœuds d'un *cluster* est identifiée comme un problème NP-complet[2] où la solution optimale est quasi-impossible à trouver en un temps raisonnable. Il est donc très difficile de savoir comment placer ces tâches[3] sur une architecture distribuée, voulant respecter simultanément plusieurs critères (ressources limitées, taille importante, tâches hétérogènes, volume élevé de communications...). Il est nécessaire pour l'utilisateur d'identifier des solutions efficaces[4] de bonne qualité, sans nécessairement garantir l'optimalité mais avec un temps d'exécution réduit, c'est-à-dire qu'il s'agit de trouver des solutions rapides offrant un temps de répartition nettement inférieur aux temps d'exécution des applications industrielles massives. Dans cette optique, l'un des objectifs de cette thèse est d'obtenir de meilleures performances sur un cluster de PC multi-cœurs en proposant des approches efficaces, décrivant comment placer au mieux les composants hétérogènes de nos applications afin d'avoir un temps de co-simulation raisonnable.

Dans le présent chapitre, nous abordons l'état de l'art du placement des tâches ainsi que ses méthodes de résolution. Nous listons ensuite les contraintes impactant potentiellement les performances des programmes parallèles. Pour conclure, nous valorisons le positionnement de notre problème par rapport aux problèmes exposés dans la littérature.

2 Placement des tâches

Ces dernières années, l'utilisation des systèmes distribués a connu une émergence importante, surtout pour exécuter de grosses applications. La distribution de telles applications sur une architecture distribuée est souvent adoptée car elle accélère la simulation totale du problème en répartissant chaque entité (tâche) du système sur un noeud physique du *cluster*. Autrement dit, chaque entité (tâche) exécute une partie du programme

1. Appellation du placement des tâches sur des ressources informatiques.
2. On trouve dans la littérature NP-hard, NP-difficile ...
3. Des jobs ou des applications qui désignent des unités de calcul.
4. des solutions proches de l'optimum.

de manière quasi-autonome et souvent parallélisable avec l'exécution d'autres parties. Mais, le problème majeur, dans ce type d'applications, est de savoir comment placer efficacement ces morceaux du programme sur un ensemble de PC, en utilisant soit des processeurs mono-coeur, soit des processeurs multi-coeurs (tout dépend de l'architecture des machines), afin de simuler tout le problème en un temps raisonnable.
De manière générale, on peut caractériser n'importe quel problème par un ensemble de tâches homogènes ou hétérogènes $T = \{t_1, t_2, ..., t_k\}$, et un ensemble de ressources informatiques $R = \{r_1, r_2, ..., r_n\}$ tel que $(n \leqslant k)$. En fait, le placement de ces tâches est l'affectation et l'assignation des tâches t_i sur des ressources informatiques r_j, en respectant quelques contraintes imposées (tâches parallèles, tâches indépendantes/dépendantes, contrainte de dépendance, etc.), comme l'illustre la figure 1. Notons que l'ensemble de ces tâches sont vues par un utilisateur externe comme un seul grand problème.

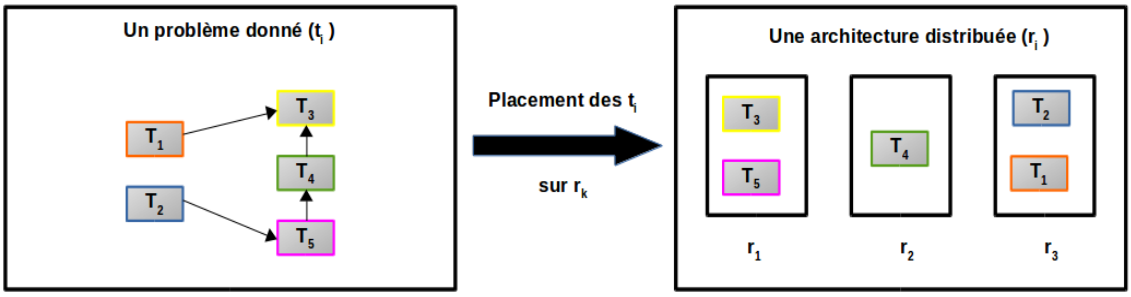


FIGURE 1 – Exemple d'un placement de tâches sur trois machines

2.1 Graphe des tâches

Les problèmes de placement sont généralement représentés sous forme d'un graphe de tâches $G = (V, E)$. L'ensemble V désigne les nœuds de G, soit les tâches à traiter, et l'ensemble E représente des liens (arc ou arêtes) de G qui font référence soit aux communications, soit aux interactions. Dans la littérature, on distingue deux types de graphe : les graphes de tâches dépendantes (TDG) [5] en cas de graphes de type *workflow*, et les graphes de tâches indépendantes (TIG) [6] en cas de composants autonomes indépendants qui s'exécutent en parallèle [24, 53].

— **Graphe de tâches dépendantes :** Il s'agit d'un graphe de tâches acyclique dirigé et ordonné, dans lequel les nœuds représentent des tâches de calculs, et les arcs entre ces tâches représentent des dépendances (ou des communications). De manière générale, la plupart des tâches dépendantes sont modélisées par des DAG [24] pour *Directed Acyclic Graphs*, qui respectent la contrainte de dépendance.

5. Task Dependency Graph
6. Task Interaction Graph

À titre d'exemple, on peut y rencontrer des problèmes de *workflow*.

Satisfaire une **contrainte de dépendance** signifie qu'il faudrait définir l'ordre global d'exécution des tâches de ce graphe : une tâche ne peut débuter son exécution que si les autres tâches qui lui sont connectées sont achevées, comme l'illustre la figure (3-a).

La figure 2 représente un exemple contenant un ensemble de composants qui suivent un certain ordre dans leur exécution. Autrement dit, c'est un flux de tâches qui évite aux composants du système de recevoir des données en désordre ; ce type de modèle assure la contrainte de précédente. Dans cette figure, il est impossible d'exécuter la tâche 4 sans avoir les résultats des deux tâches précédentes (2 et 3).

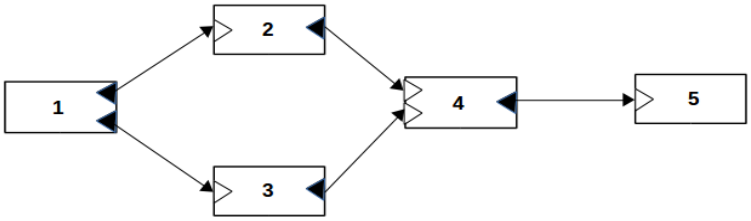


FIGURE 2 – Exemple d'une dépendance dans un graphe

Note : Dans notre cas, nous n'utilisons pas un *workflow* car les tâches de nos modèles s'exécutent en parallèle, et du coup nos modèles n'ont pas de contrainte de précédence.

— **Graphe de tâches indépendantes :** Ce type de graphe est représenté sous forme d'un graphe de tâches non orienté ; les nœuds de ce graphe sont des tâches de calcul et les arêtes désignent des interactions entre ces tâches (sans relation de dépendance), autrement dit, les tâches connectées par des liens peuvent s'exécuter simultanément (voir la figure 3-b).

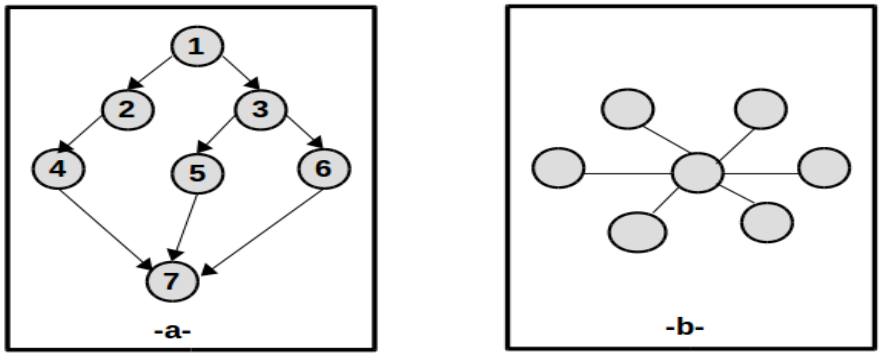


FIGURE 3 – (a) Graphe de tâches dépendantes. (b) Graphe de tâches indépendantes

La différence entre ces deux représentations est la suivante : les liens du graphe TDG

représentent des dépendances de contrôle et les liens du graphe TIG désignent les dépendances des données. Les tâches échangent des données ou utilisent des données partagées (par exemple les données en exclusion mutuelle).

En outre, on trouve dans la littérature des graphes qui ne contiennent aucun cycle dits graphes acycliques, et des graphes qui permettent à ses tâches de s'exécuter une infinité de fois dits graphes cycliques ou périodiques.

2.2 Types de placement des tâches

Dans cette section, on peut classer les méthodes de placement de tâches, ou *mapping*, en deux grandes familles, à savoir le placement dynamique (variable quand on avance dans le temps), et le placement statique (fixe pour toute la simulation).

— ***Mapping* statique** :
Le placement statique « *Off-line* » définit un placement fixe des tâches au moment de la compilation du programme. Cette méthode exige une connaissance préalable des temps d'exécution des différentes tâches avant l'exécution de l'algorithme de placement. Par conséquent, le système qui applique ce type de méthodes doit être stable car cette dernière n'est pas applicable à un système dont le poids de ses calculs varie dans le temps.

— ***Mapping* dynamique** :
La méthode dynamique « *On-line* » permet de placer des tâches qui varient avec le temps, et parfois d'attribuer de nouvelles tâches au moment de l'exécution. Contrairement au placement statique, dans les scénarios dynamiques, le temps pris pour placer chaque tâche sur une machine physique est pertinent, car il influence le temps d'exécution global de l'application.

Dans notre travail actuel, nous nous limitons à une méthode statique de placement car le nombre de tâches est fixe durant toute la co-simulation, et les temps de calcul de nos FMU sont relativement stables dans le temps.

2.3 Méthodes de résolution de placement des tâches

Le placement des diverses tâches (indépendantes ou dépendantes) sur un ensemble de machines multi-cœurs se situe dans la catégorie des problèmes complexes d'optimisation combinatoire. Il s'agit d'une famille de solutions permettant de trouver dans un ensemble des solutions réalisables (ou alternatives) une solution efficace. La littérature décrit des méthodes de résolution de ce type de problème ; cependant, elles doivent être sélectionnées avec précaution, dans la mesure où elles impactent directement les performances du système global.

Dans cette partie, nous avons synthétisé les grandes familles de solutions les plus utilisées dans les problèmes d'optimisation, plus particulièrement dans les problèmes combinatoires, à savoir les méthodes exactes et les méthodes approchées [19]. Les méthodes exactes résolvent ces problèmes à partir de modèles d'exécution et d'approches mathématiques, et les méthodes approchées englobent les heuristiques et les métaheuristiques. La figure 4 illustre les différentes méthodes qui seront abordées dans la suite de ce manuscrit.

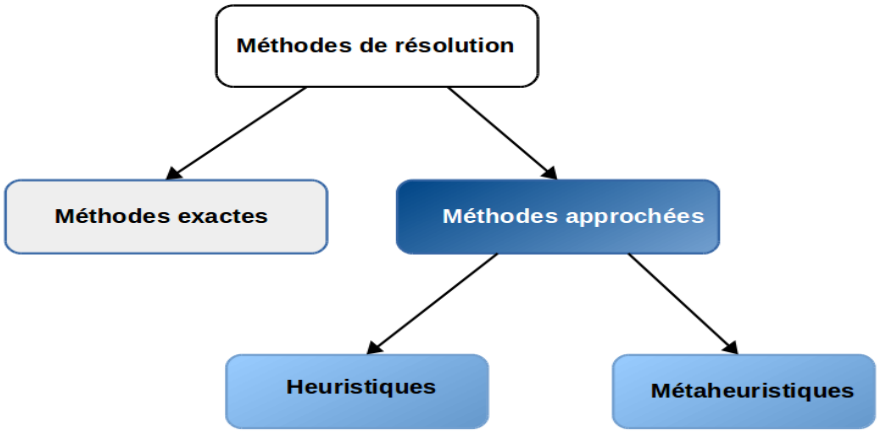


FIGURE 4 – Classification des méthodes combinatoires [11]

2.3.1 Méthodes exactes

Les méthodes exactes sont des méthodes exhaustives qui se décrivent mathématiquement. Elles représentent des méthodes complètes qui garantissent la solution la plus optimale [7], comme les algorithmes B&B (*Branch and Bound*) [48] qui s'appuient sur des méthodes de recherche arborescentes. Ces méthodes exactes sont applicables à un certain nombre de problèmes, elles permettent d'obtenir des solutions qui garantissent l'optimalité, mais uniquement dans la mesure où la taille du problème est raisonnable. Si le problème à une taille importante, le coût de la méthode croît exponentiellement, ce qui représente une limite importante de ce type de méthodes. Pour remédier à cette limitation, des chercheurs proposent d'autres méthodes permettant de repousser cette limite vers une solution efficace la plus proche possible de l'optimum : les méthodes approchées.

2.3.2 Méthodes approchées

Les méthodes approchées sont des méthodes approximatives, qui s'appliquent à des problèmes simples ou complexes dans le but de trouver une solution plus ou moins

7. Une solution dont son temps de simulation est minimal.

optimale en un temps raisonnable. Ces méthodes sont simples à mettre en œuvre ; par conséquent, elles ne garantissent généralement pas la solution optimale au problème.

Par des heuristiques

Les heuristiques sont des méthodes de résolution pour un problème bien défini, en se basant sur des expériences acquises, afin de trouver la solution proche de l'optimum et qui n'est pas nécessairement une solution la plus optimale. On recourt à ces méthodes lorsque la solution est quasi-impossible à trouver ce qui est le cas des problèmes de grandes tailles.
L'objectif de ces méthodes est donc de trouver rapidement une solution efficace sans garantir l'optimalité. En réalité, elles permettent de trouver un compromis entre la rapidité pour trouver une solution adéquate et la qualité de la solution trouvée (efficace et plus au moins optimale) [12, 54].
Les heuristiques appliquées dans les graphes de tâches abordés dans la section 2.1 ont comme but de minimiser le temps global de complétion, on dit aussi le *makespan*[8]. On peut citer ici quelques heuristiques très utilisées dans la littérature concernant les problèmes d'ordonnancement et de placement des tâches dans des cluster de PC. Mais avant d'appliquer les heuristiques présentées ci-dessous, il est nécessaire d'exécuter les différentes tâches sur une architecture distribuée afin d'obtenir une cartographie de leurs temps d'exécution sur chacune des machines du cluster.

Dans cette sous section, nous établissons une revue des principales heuristiques proposées dans la littérature.

- ***Opportunistic Load Balancing*** (OLB) : Cette heuristique permet d'affecter, de façon arbitraire, chaque tâche à une ressource disponible, sans tenir compte de son temps d'exécution sur la machine physique. On obtient généralement un *makespan* plus grand que l'optimal, ce qui représente l'inconvénient majeur de cette heuristique.
- ***Minimum Execution Time*** (MET) : Cette heuristique permet de prendre une tâche de manière aléatoire afin de la déposer sur la ressource qui donne le meilleur temps d'exécution mais indépendamment de la disponibilité des machines (même si la ressource en question est occupée). Cette heuristique pourrait créer un déséquilibre entre les charges des différents nœuds du cluster. À titre d'exemple, si on a 10 tâches en tout et 8 de ces dernières donnent des meilleurs résultats sur une machine (A), cet algorithme placera donc ces 8 tâches sur cette même machine (A) et deux tâches restantes sur une autre machine (B) [54].

8. Le temps d'exécution global de l'application.

- ***Minimum Completion Time*** (MCT) : Cette heuristique combine les avantages des deux heuristiques OLB et MET. Elle permet d'attribuer chaque tâche, de façon arbitraire, à une ressource qui donne le temps d'exécution le plus petit. Mais la limitation de cette heuristique est que certaines tâches (non placées) ne peuvent pas être affectées aux machines qui ont un temps d'exécution minimal : les nœuds de calcul qui donnent un temps d'achèvement petit sont déjà occupés par d'autres tâches [54] .
- ***Min-Min*** : Après avoir exécuté l'ensemble de tâches sur l'ensemble de ressources, cette heuristique consiste à placer la tâche t_i la plus petite [9] sur la ressource correspondante r_k. Ensuite, cette tâche placée sera retirée de la liste des tâches non placées, et ce processus sera répété tant que la liste des tâches non placées ne sera pas vide. Cette heuristique n'équilibre pas les charges entre les machines car elle commence par placer les petites tâches.
- **Max-Min** : Cette heuristique est similaire à l'heuristique Min-Min, mais elle commence par placer les grosses [10] tâches sur les ressources informatiques, pour ne pas retarder la fin de toute l'exécution du système (meilleur équilibrage de charge). En revanche, si le nombre de grosses tâches du système est plus grand que le nombre de petites tâches, cela peut augmenter le temps global de simulation du système suite à ce déséquilibre.
- **Duplex** : Cette heuristique combine les deux heuristique présentées auparavant Min-Min et Max-Min afin de choisir la meilleur des deux solutions.
- ***Heterogeneous Earliest-Finish-Time*** **(HEFT) :** L'algorithme HEFT [74] est fondé sur deux phases : la première phase classe et priorise les tâches, et la seconde sélectionne le processeur correspondant. En d'autres termes, la tâche dont la valeur de rang [11] *(highest upward rank)* est la plus élevée est placée sur le processeur correspondant, qui réduit au minimum son délai d'exécution. Cette heuristique à base de liste est la plus utilisée dans les problèmes d'ordonnancement et de placement des tâches, mais elle est uniquement applicable aux problèmes des tâches dépendantes type DAG.
- **Clustering** [12] : Dans les systèmes distribués, la stratégie de regroupement est une solution efficace pour réduire le coût de communication dans les graphes de calcul, surtout dans les graphes DAG. En fait, elle permet de regrouper les tâches qui communiquent beaucoup dans un seul groupe afin de les attribuer à un seul nœud

9. Tâche petite où le temps d'exécution est minimum.
10. Une tâche gourmande en calcul, on dit aussi une longue tâche.
11. la tâche ayant le plus haute distance à partir du nœud courant jusqu'au nœud de sortie.
12. regroupement.

de calcul. *P. SADAYAPPAN and al* [62] ont proposé une approche permettant d'équilibrer la charge entre les processeurs tout en supprimant les communications coûteuses en inter-processeurs, ce qui revient à maximiser les communications en intra-processeur. Elle repose sur deux phases importantes : la première phase est responsable de partitionner l'ensemble du graphe des tâches en plusieurs sous-ensembles, et a lieu de façon récursive. La seconde phase permet d'affecter les n sous-ensembles obtenus dans la première phase à autant de machines (n) ; chaque sous-ensemble est ainsi associé à une seule machine de calcul.

Par des méta-heuristiques

Ces méthodes dites « méta-heuristiques » consistent également à déterminer une ou plusieurs solutions efficaces, plus en moins optimales. Il s'agit d'heuristiques génériques adaptées à plusieurs problèmes, de méthodes évolutionnaires qui s'appuient sur des stratégies d'évolution [38]. Ces méthodes exploitent un espace de recherche pour trouver une bonne solution proche de l'optimum car les méta-heuristiques sont des méthodes non-déterministes qui n'assurent pas l'optimalité de la solution ([3] chapitre 1). Généralement, ce type de méthode permet de traiter les problèmes d'ordonnancement de production ; les exemples les plus connus sont les méthodes de trajectoire (en s'appuyant sur des méthodes de recherche locale), et des méthodes basées sur une population (algorithmes génétiques).

Les méthodes de trajectoire, dites méthodes de solution unique (recherche locale), permettent de faire évoluer une solution unique en construisant une solution pas à pas : autrement dit, elles permettent d'élaborer une trajectoire dans l'espace de recherche afin de converger vers des solutions optimales. Parmi les méthodes basées sur une population, on connaît notamment la méthode génétique (notée AG), introduite par *John Holland* dans son livre [40]. Celle-ci est fondée sur le principe de la génétique en passant de génération en génération pour trouver une solution proche de l'optimal. En Biologie comme en Informatique, ces méthodes nécessitent d'avoir une sémantique et des objets de base : un gène correspond à une propriété, un individu représente un ensemble de gènes, et une population est un ensemble d'individus.

En comparaison des heuristiques, qui visent à accélérer la recherche d'une solution, les méta-heuristiques obtiennent des solutions (presque) optimales, mais dans des délais parfois très longs. Dans le cadre de la présente thèse, nous nous sommes intéressés aux performances, en cherchant à trouver une heuristique garantissant une solution efficace (assez bonne) rapidement, plutôt qu'une méta-heuristique cherchant une solution plus optimale en un temps potentiellement indéterminé.

3 Performances sur machines parallèles

Quand on veut paralléliser une grosse application, il faut diviser l'application traitée en « n » morceaux afin de les placer sur une architecture distribuée, en utilisant un cluster informatique. Cette architecture distribuée ou *cluster* informatique est un ensemble de machines homogènes ou hétérogènes, offrant un espace de mémoire important, composé de nœuds physiques reliés par un réseau de communication.
Le principal objectif de ce type d'architecture est de fournir aux utilisateurs une grosse puissance de calcul potentielle qui dépend grandement des performances de chaque nœud et du réseau d'interconnexion. De plus, l'évolution architecturale des processeurs des noeuds de ce *cluster* (des processeurs mono-cœur vers les processeurs multi-coeurs) a permis une amélioration des performances. Ces processeurs peuvent communiquer entre eux soit par le biais d'une mémoire partagée en cas de programmes parallèles (architecture multi-processeurs), soit par envoi/réception de messages en cas de programmes distribués (architecture distribuée).

Definition 3.1 *Grappe de machines (en anglais cluster). Regroupement de machines indépendantes interconnectées par un même équipement réseau, localisées dans un même site et disposant d'au moins une machine spécifique de gestion appelée machine frontale*[13] *[43].*

Dans la partie qui suit, nous mettons l'accent sur deux aspects abordés précédemment, qui peuvent lourdement affecter les performances : l'architecture des machines et le réseau d'interconnexion.

3.1 Impact de l'architecture sur les performances

Ces dernières décennies, les processeurs multi-coeurs ont émergé sur le marché pour accélérer l'exécution des programmes parallèles. L'ensemble des coeurs qui se trouvent sur le même nœud physique partagent un cache qui est une mémoire partagée accessible par plusieurs programmes parallèles de façon simultanée. Cette mémoire partagée est considérée soit comme un seul bloc accessible par tous les cœurs du processeur, soit comme un ensemble de sous-mémoires. Toutefois, ce partage peut avoir des conséquences importantes sur les performances, selon les temps d'accès à la mémoire. Dans le cas d'une architecture NUMA[14], permettant de stocker des données dans des endroits différents en mémoire, les temps d'accès seront non uniformes. A l'inverse, les temps d'accès seront uniformes avec des architecture de type UMA[15], où le partage d'une même mémoire garantit un temps d'accès identique pour tous les cœurs du nœud [55].

13. Une machine frontale permet l'accès au gestionnaire de ressources, elle représente une porte d'entrée au cluster.
14. Non Uniform Memory Architecture.
15. Uniform memory Architecture.

De plus, les processeurs multi-cœurs peuvent engendrer une oscillation des performances due aux nombreuses interactions entres les programmes (congestion).

3.2 Importance du réseau d'interconnexion

Lors de l'exécution d'un ensemble de programmes parallèles qui communiquent fréquemment entre eux sur un cluster de PC, il faut disposer d'un support physique pour transférer les données échangées entre ces nœuds. Il s'agit d'un réseau de communication permettant d'interconnecter de nombreux nœuds, souvent considérés comme un ensemble de routeurs, de commutateurs branchés sur des serveurs et des périphériques de stockage.

Ce réseau d'interconnexion est constitué d'un ensemble de mécanismes pour classer les éléments physiques du réseau et des stratégies pour faire circuler les données au sein d'un câble. Il s'appuie généralement sur deux modes de transfert des données : le mode « *store-and-forward* » où le réseau attend l'intégralité des données avant de l'envoyer auprès de sa destination, et le mode « *cut-through* » qui démarre l'envoi des données dès la réception du premier paquet. Dans notre cas, nous avons utilisé le mode « *store-and-forward* ».

En outre, nous avons utilisé deux types de réseaux d'interconnexion à savoir : le réseau Ethernet standard et le réseau *InfiniBand*.

3.2.1 Réseaux locaux type Ethernet

Les réseaux locaux comme *Ethernet* sont les plus déployés au monde, ils permettent d'interconnecter et de faire circuler des données entre différents noeuds physiques. Les implantations les plus utilisées des réseaux locaux sont des réseaux de communication de 1 Gbit/s, et de 10 Gbit/s. Le matériel nécessaire pour concevoir les réseaux type Ethernet est bon marché comparativement aux réseaux type *Infiniband*, et ils sont moins coûteux à mettre en place. Mais l'inconvénient majeur est la congestion du réseau : plus les communications sont nombreuses et plus le temps est long car le débit reste relativement faible.

3.2.2 Réseau *Infiniband*

On trouve un nouveau type de réseaux d'interconnexion qui est *InfiniBand*, un réseau performant, qui prend en compte les communications concurrentes et la congestion du réseau de communication. Contrairement aux réseaux standards, les réseaux *Infiniband* se caractérisent par une faible latence, une bande passante élevée et une faible consommation d'énergie [34]. Ces différentes caractéristiques permettent de réduire les temps

de communication entre les différents noeuds du *cluster*, c'est pourquoi un tel réseau est utilisé généralement en HPC [16].

3.3 Modélisation des exécutions des programmes parallèles

L'estimation du temps global de la simulation dépend d'un ensemble de facteurs comme le nombre de coeurs, la fréquence du processeur, son cache, le type d'architecture du processeur, etc. Il est cependant très difficile, parfois quasi-impossible, de mesurer tous ces paramètres.

L'exécution d'un programme parallèle sur une grappe de calcul se caractérise habituellement par la répétition de deux phases importantes, à savoir, la phase de calcul et la phase de communication. D'une manière générale, quand on exécute une application parallèle sur une architecture distribuée, elle se présentera comme étant un seul grand programme qui lance un ensemble de sous-programmes sur un ensemble de machines, et fait communiquer tous ces programmes entre eux pour faire progresser l'exécution globale. Dans ce qui suit, nous avons listé les principales méthodes connues dans la littérature en vue de modéliser le temps de calcul et le temps de communication.

3.3.1 Modélisation du temps de calcul

Modéliser des temps de calcul oblige l'utilisateur soit à faire des expérimentations spécifiques types *Micro-benchmarking* de ses tâches, soit à appliquer des modèles d'estimations déjà existants afin de prédire le temps de calcul de chacune de ses tâches. Généralement, ces expérimentations s'appuient sur une analyse statique ; une méthode formelle qui permet de découper le programme en blocs sans l'avoir exécuté, comme *Slicing approaches* [59]. On trouve dans la littérature l'idée de *Saavedra et Smith* [61] s'appuyant sur une approche qui permet d'évaluer le programme en termes d'opérations élémentaires, dans le but d'estimer le temps de calcul. Cette solution n'a pas été conçue pour les programmes parallèles. Mais, l'auteur de l'article [10] a adopté le même principe sur l'ensemble des programme parallèles avec MPI (*Message-Passing Interface*). Malheureusement, dans notre cas, on ne peut pas appliquer ces méthodes d'estimations car nos tâches sont des boites grises (FMU) qui utilisent des solveurs permettant d'exécuter leurs calculs (leur propre code source), et du coup, on ne pourra pas accéder au code source de ces dernières.

Généralement, quand on veut estimer le temps de calcul d'une tâche, on part de l'équa-

16. High-Performance Computing.

tion suivante la plus utilisée dans la littérature :

$$T_{calcul} = NI * CPI * Tc \tag{2.1}$$

Où NI est le nombre d'instructions à exécuter par la machine, CPI est le nombre moyen de cycle d'horloge par instruction et Tc représente le temps de cycle du processeur. Pour calculer le NI du programme, il faut avoir des informations concernant le code source, ou bien adopter une analyse statique pour estimer ce paramètre.

Comme indiqué précédemment, il est très difficile, dans notre cas, d'obtenir ces informations car nos tâches étudiées sont des FMU, ce qui signifie qu'on ne peut pas déterminer le nombre d'instructions de nos tâches puisqu'on on ne peut pas accéder à leur code source. Pour remédier à ce problème, nous avons demandé à notre partenaire industriel EDF R&D de nous fournir quelques métadonnées sur leurs FMU métiers, et les développeurs des modèles des FMU nous ont communiqué le nombre d'équations de chaque tâche. Mais cela ne nous a pas suffit car chaque équation est intégrée par un ensemble d'instructions dont on ne connaît pas le format. Finalement, nous avons simplifié le problème en exprimant le temps de calcul d'une FMU en fonction de son nombre d'équations par une régression linéaire [17] sur nos mesures expérimentales. Autrement dit, nous avons modélisé le temps de calcul T_{Calcul} par une fonction de $nbr_{Equation}$ afin d'estimer le temps de calcul des FMU (nombre d'équations fourni par les développeurs de FMU).

3.3.2 Modélisation du temps de communication

Dans cette section, nous abordons deux modélisations des temps de communication. On commence par le modèle d'*Hockney* [39] qui représente une simple équation linéaire pour modéliser le temps de communication d'un message point-à-point. Pour cela, il est supposé que le temps d'envoi d'un message de taille m entre deux machines physiques est le suivant :

$$T_{Comm} = l + \frac{m}{Bw} \tag{2.2}$$

Où l est la latence et Bw la bande passante du réseau.
Ce modèle est indépendant de l'architecture et il ne peut pas modéliser la congestion du réseau. C'est pour cela que *Clement et Steed* [20] ont ajouté un nouveau paramètre au modèle d'*Hockney*, qui est un facteur de congestion λ des réseaux partagés, comme dans le cas d'Ethernet non commuté :

$$T_{Comm} = l + \frac{m \times \lambda}{Bw} \tag{2.3}$$

17. La régression linéaire est une méthode qui modélise la relation entre deux variables quantitatives continues.

Cependant, *Clement et Steed* supposent que tous les processus communiquent simultanément, ce qui n'est vrai que pour quelques modèles de communication collectifs. C'est pour cela que nous nous sommes limités dans cette thèse à utiliser le modèle simple générique d'*Hockney* .

3.4 Techniques de modélisation des performances

Pour augmenter l'espace de recherche, nous nous sommes orientés vers des méthodes analytiques alimentées par les hypothèses, afin de valider les méthodes expérimentales. À partir de là, il serait intéressant de modéliser le comportement exact de nos modèles, afin d'estimer les temps de calcul et de communication de chaque problème. Cette modalisation permet de donner une certaine souplesse pour déterminer les performances de ces problèmes. Pour se faire, nous listons les techniques de base que nous pourrions utiliser dans la modélisation des performances de nos applications.

3.4.1 Loi d'*Amdahl*

Nous trouvons dans la littérature le tout premier modèle qui détermine le gain d'accélération et de performance d'un programme donné, ce modèle a été conçu par *G.M. Amdahl* en 1967 [4]. Il s'intéresse à l'accélération d'un problème donnée, en prenant en compte une fraction séquentielle (f_s) du code que l'on ne peut pas paralléliser et une fraction parallélisable ($f_p = 1 - f_s$) du code. Le modèle d'*Amdahl* permet d'estimer l'accélération d'un programme parallèle sur plusieurs machines, en considérant que seule la fraction parallélisable peut être réduite, et que la taille du problème global reste constante. Le temps d'exécution parallèle sur « n » noeud de calcul est donné par l'équation 2.4 sachant que T(1) est le temps d'exécution séquentiel sur une seule ressource informatique :

$$T(n) = f_s \times T(1) + f_p \times \frac{T(1)}{n} \tag{2.4}$$

L'accélération calculée avec ce modèle est la suivante :

$$Speedup(n) = \frac{1}{f_s + \frac{f_p}{n}} \tag{2.5}$$

Où le n représente le nombre de ressources utilisées pour paralléliser l'application. Par exemple, si on a une fraction séquentielle estimée à 5% ($f_s = 0.05$) et un nombre de ressources égal à 8, le gain d'accélération théorique est égal à 5.92. Dans notre cas, nous avons comparé nos mesures de performances expérimentales à cette loi hyperbolique pour évaluer la qualité de nos parallélisations (voir le chapitre 6).

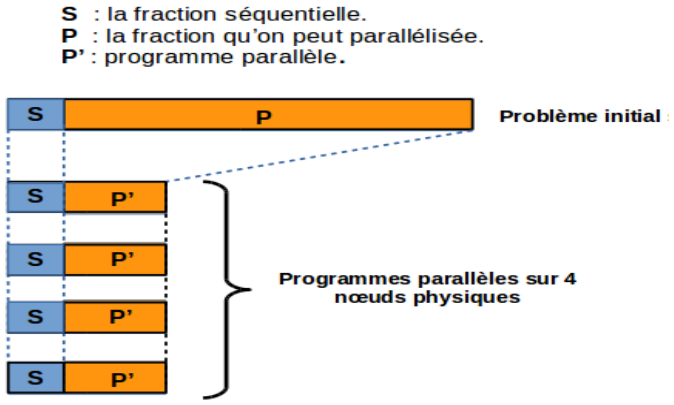


FIGURE 5 – Principe de parallélisation selon la loi d'*Amdahl*

3.4.2 Loi de *Gustafson*

La lois de *Gustafson* a été définie par *John Gustafson* en 1988 [35]. Contrairement au modèle de *G.M. Amdahl*, *Gustafson* s'est intéressé au parallélisme des grands problèmes, en considérant que l'on augmente la taille du problème quand on augmente le nombre de ressources utilisées. *John Gustafson* définit aussi une fraction séquentielle (f_s), et une fraction « parallélisée » ($f_p = 1 - f_s$), mais différentes de celles d'*Amdahl*. Il en déduit le temps d'exécution séquentiel comme suit :

$$T(1, q_i) = f_s \times T(n_i, q_i) + n_i \times f_p \times T(n_i, q_i) \tag{2.6}$$

tel que q_i est la taille du problème à exécuter, et n_i est le nombre de ressources.
Au final, *Gustafson* aboutit à une accélération illustrée dans l'équation 2.7, en considérant que la taille du problème (q) augmente quand le nombre de ressources (n) augmente :

$$Speedup(n) = f_s + f_p \times n \tag{2.7}$$

Nous avons mené des campagnes d'expérimentations sur des problèmes de tailles croissantes pour étudier plusieurs parallélisations de co-simulations vérifiant bien la loi de *Gustafson* (voir chapitre 6).

La loi de *Gustafson* fait l'hypothèse qu'on conserve le même temps de simulation lorsqu'on traite des problèmes plus gros, c'est-à-dire qu'on espère obtenir $T(n1, q1) = T(n2, q2) = T(n3, q3) = C^{ste}$, comme l'illustre la figure 6. Cette démarche vise à traiter des problèmes plus importants en exploitant efficacement les noeuds d'un cluster.

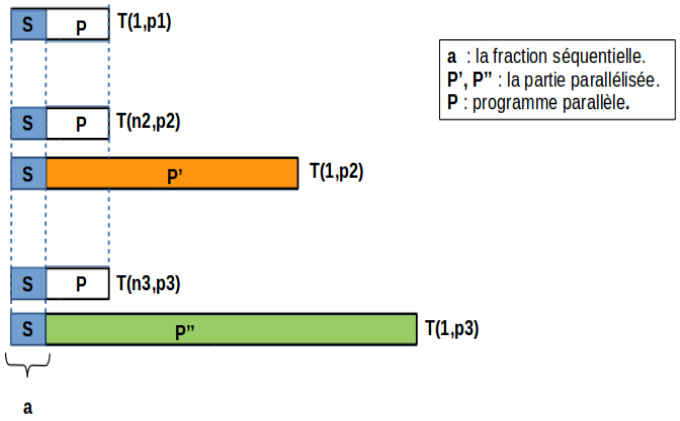


FIGURE 6 – Principe de parallélisation selon la loi de *Gustafson* [67]

4 Conclusion et positionnement de notre problème

Dans cette thèse, nous nous intéressons aux problèmes de placement des tâches de taille de plus en plus importante, de type boites grises sur une architecture de machines multi-cœurs. Vu que les problèmes traités dans notre cas sont des graphes de tâches hétérogènes, contenant des solveurs qui s'exécutent tous en parallèle pour un même pas de temps, on peut positionner ces derniers comme des modèles dégénérés qui représentent à la fois un graphe TIG [18] pour un pas de temps (avec lequel toutes ses FMU s'exécutent en parallèle de manière indépendante et autonome), et un graphe TDG [19] entre les pas de temps. Pour que les tâches commencent à exécuter un autre pas de temps, il faut disposer des résultats du pas précédent (contrainte de dépendance).

Dans notre cas, nous avons appliqué une méthode de *mapping* statique, car les charges de calculs de nos tâches sont quasiment fixes durant toute la co-simulation (vérifier par des expérimentations) ; les calculs sont effectués par des solveurs numériques qui les exécutent à chaque pas de temps. Notre contribution consiste à proposer des approches permettant de placer des tâches hétérogènes sur un cluster de PC multi-cœurs (voir la figure 7), c'est-à-dire des approches s'appuyant sur des heuristiques rapides à base de listes, tout en garantissant un temps d'exécution parallèle raisonnable et réduit. Elles seront abordées en détails dans le chapitre 5.

18. Task Interaction Graph.
19. Task Dependency Graph.

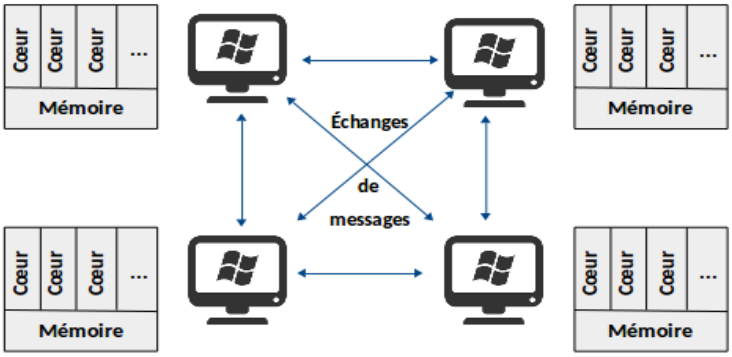


FIGURE 7 – Notre type de grille de calcul

CHAPITRE 3

PLATE-FORME DE CO-SIMULATION DISTRIBUÉE (DACCOSIM)

Sommaire

1 Introduction

En parallèle, à l'arrivée des *Smart Grids*, plusieurs outils de co-simulation sont apparus pour simuler le comportement des systèmes cyber-physiques et analyser leurs performances avant de les mettre en exploitation. Ces outils de co-simulation permettent d'analyser le comportement apparent de ces systèmes pour mettre au point les fonctions de contrôle qu'ils embarquent. Le développement de tels outils permettra de contrôler de manière plus fiable et plus sûre les réseaux électriques de demain, et de répondre ainsi aux exigences du marché en diminuant les coûts et en facilitant le raccordement de sources d'énergie renouvelable. L'objectif de ces outils est d'approcher de manière précise le comportement global d'un *Smart Grid* par la simulation. C'est également avec cet objectif ambitieux que CentraleSupélec et EDF R&D ont développé une plate-forme de co-simulation basée sur la norme FMI, permettant de concevoir et mettre au point des Réseaux Électriques Intelligents, si besoin en distribuant la co-simulation sur une architecture informatique répartie.

Dans ce chapitre, nous présenterons, dans un premier temps, la plate-forme DACCOSIM [1] que nous avons utilisée et améliorée au cours des trois ans de thèse. La première partie donnera un aperçu des principales caractéristiques ainsi que l'architecture fonctionnelle de DACCOSIM. Nous listerons ensuite les différents constituants utilisés pour développer ce logiciel, et valider ses résultats numériques. Dans la partie suivante, nous nous pencherons sur l'outil DacRun, qui permet de déployer une co-simulation DACCOSIM sur une architecture distribuée. Pour conclure ce chapitre, nous nous énumérerons les limites actuelles de cette plate-forme, et résumerons les améliorations apportées à cette plate-forme.

Motivations

Un *Smart Grid* se caractérise par un très grand nombre de composants physiques et de contrôle très imbriqués fonctionnellement et très distribués sur un territoire souvent très étendu. La co-simulation offre une vision système remplissant une fonction d'intégration indisponsable avant toute mise en service sur le terrain par les exploitants. Un *Smart Grid* est en fait un système cyber-physique complexe, qui fait appel à de nombreux modèles unitaires de nature différente, et fonctionnellement plus ou moins fortement couplés entre eux. Compte-tenu de l'hétérogénéité et du nombre parfois très important de ces modèles, il a fallu concevoir un logiciel capable de vérifier la pertinence et la cohérence des scénarios d'un *Smart grid*. Ce logiciel permettra de réaliser un réseau électrique intelligent, et d'assurer une synchronisation temporelle de toute la co-simulation.

1. https ://sourcesup.renater.fr/daccosim/

Ce sont ces ambitions portées par CentraleSupélec et EDF qui ont conduit au développement de la plate-forme de co-simulation distribuée DACCOSIM (*Distributed Architecture for Controlled CO-SIMulation*) ; une plate-forme qui permet de construire un simulateur de réseau de distribution potentiellement déployable sur une architecture distribuée multi-cœurs.

Progressivement au fil de la thèse, nous avons fait face à la complexité croissante des simulations de *Smart Grids*, et DACCOSIM a été éprouvé par le besoin de réaliser des co-simulations de grande taille.

2 Approche utilisée

Dans cette section, nous décrivons globalement les différentes étapes primordiales qu'il faut suivre pour réussir une co-simulation de type DACCOSIM :

1. **Fabrication des FMU**
   Tout d'abord, nous avons besoin d'encapsuler les différents modèles de simulation dans des composants atomiques et portables, pour qu'on puisse les importer dans DACCOSIM. Les composants de cette plate-forme se basent sur la norme FMI [2] (voir le chapitre 1). Afin de disposer des composants respectant les spécificités de cette norme, nous avons collaboré avec une des équipes d'EDF R&D, pour qu'elle nous construise des FMU-CS en Linux 64 bits avec un outil industriel appelé *Dymola* et qui va être abordé dans la section 4.3 de ce chapitre.

2. **Réalisation d'une simulation de *Smart Grid***
   Cette étape permet à un utilisateur de concevoir un simulateur de *Smart Grid* correspondant à un réseau de distribution réel, en se basant sur un modèle de description métier délivré par EDF. DACCOSIM peut s'utiliser soit directement via interface graphique, soit grâce à son langage de script (permettant d'assembler des milliers de FMU si besoin). En outre, la synchronisation des données entre ces différentes FMU s'appuie sur un modèle de temps global développé dans la plate-forme DACCOSIM qui permet de réaliser des co-simulations, c'est-à-dire indépendante de toute sémantique métier portée par exemple par les *Smart Grids*.

   L'utilisateur externe peut ensuite appliquer un algorithme de répartition automatique intégré dans DACCOSIM, pour calculer le placement et préparer le déploiement des composants hétérogènes de ce système sur une architecture distribuée.

2. *Functional Mockup Interface*

3. **Exécution**

   Les expérimentations des co-simulations de *Smart Grid* se réalisent soit sur une seule machine (mono-machine multi-coeurs) si la taille du système est petite, soit sur une architecture distribuée (multi-machines multi-coeurs) si le système est de grande taille.

La figure 1 illustre cette démarche avec ses trois étapes, qui sont réalisées par une ou plusieurs équipes de recherche spécifiques (à CentraleSupélec ou à EDF R&D).

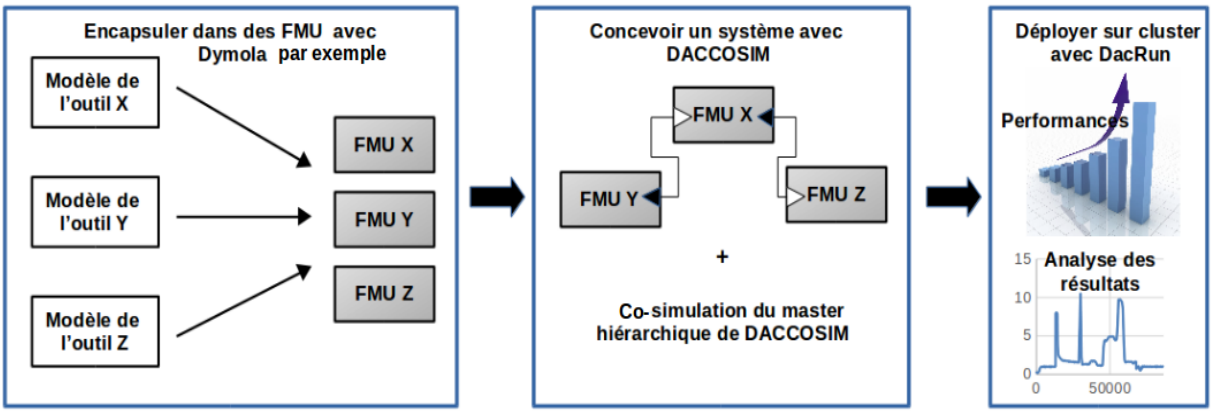


FIGURE 1 – Schéma de principe (origine : projet POMME d'EDF)

Les travaux réalisés dans cette thèse d'informatique se focalisent sur l'étape 2 et l'étape 3 de ce processus, sachant que l'étape 1 relève de la responsabilité d'EDF.

3 DACCOSIM

DACCOSIM est l'acronyme anglais de *Distributed Architecture for Controlled CO-SIMulation*. Il s'agit d'une plate-forme de co-simulation parallèle et distribuée, dédiée aux systèmes physiques à pas de temps, avec un maître de co-simulation hiérarchique. Cette plate-forme est basée sur la norme FMI (*Functional Mockup Interface*) présentée dans le chapitre 1, et supporte deux versions : la version 1.0 et la version 2.0.

De manière générale, DACCOSIM permet à un utilisateur de réaliser une co-simulation en interconnectant les différentes FMU entre elles : ces FMU représentent des dispositifs cyber-physiques indépendants comme des bâtiments, des réseaux électriques, etc. Les données échangées entres les composants du système sont synchronisées à l'aide d'un « master hiérarchique distribué », qui contrôle et orchestre toute la co-simulation. Ce master joue deux rôles importants : celui de master global, pour superviser l'ensemble de la co-simulation, et celui de master local, pour collecter les informations échangées

par les FMU se trouvant sur un même nœud de calcul [30]. Notons que cette plate-forme peut être parallélisée sur des processeurs multi-cœurs, et distribuée sur un ensemble de nœuds de calcul.

DACCOSIM permet de concevoir un réseau électrique en le représentant comme un graphe acyclique de FMU assez fortement couplées qui interagissent entre elles tout au long de la simulation. Autrement dit, cette plate-forme transforme le système à simuler en un graphe de calcul acyclique DAG (présenté dans le chapitre 2) où les nœuds représentent des FMU de type CS (Co-Simulation), et où les arêtes représentent les échanges de données entre les FMU. Ce graphe de FMU est ensuite traduit en code Java, prêt à être exécuté et déployé sur un cluster de PC.

DACCOSIM fonctionne aussi bien sous Windows 32/64-bits que sous Linux 32/64-bits. Notons qu'il n'est pas indispensable pour l'utilisateur final de DACCOSIM de comprendre les modèles de simulations intégrés dans chaque FMU du système (qui peut être assimilée à une boite grise).

Avant de réaliser toute co-simulation avec la plate-forme DACCOSIM, l'utilisateur doit construire deux types de graphes complémentaires : un graphe de ressources et un graphe de tâches [46] :

1. Le **graphe de ressources** modélise l'ensemble des ressources virtuelles qu'il faudra ensuite mapper sur un cluster de calcul (standard ou à haute performance). La figure 4 montre l'exemple d'un graphe de ressources qui comporte 8 noeuds, chacun muni d'un système d'exploitation Windows 32 bits.

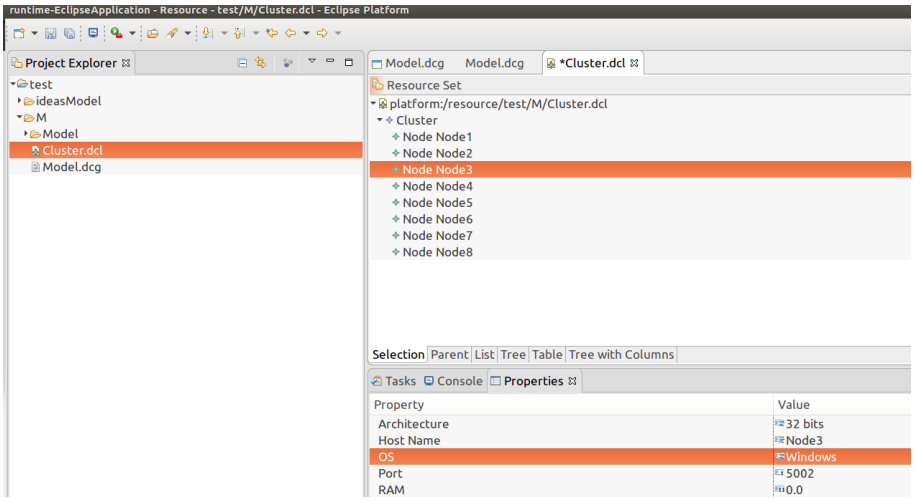


FIGURE 2 – Interface graphique de DACCOSIM (graphe de ressources

2. Le **graphe de tâches** regroupe un ensemble de modèles de simulation qui communiquent plus au moins fortement entre eux. Ces modèles forment un ensemble de tâches hétérogènes reliées entre elles par des liens logiques ; ce type de graphe est illustré dans la figure 3. Notons que chaque FMU réalise les calculs grâce à son propre solveur numérique intégré au modèle.

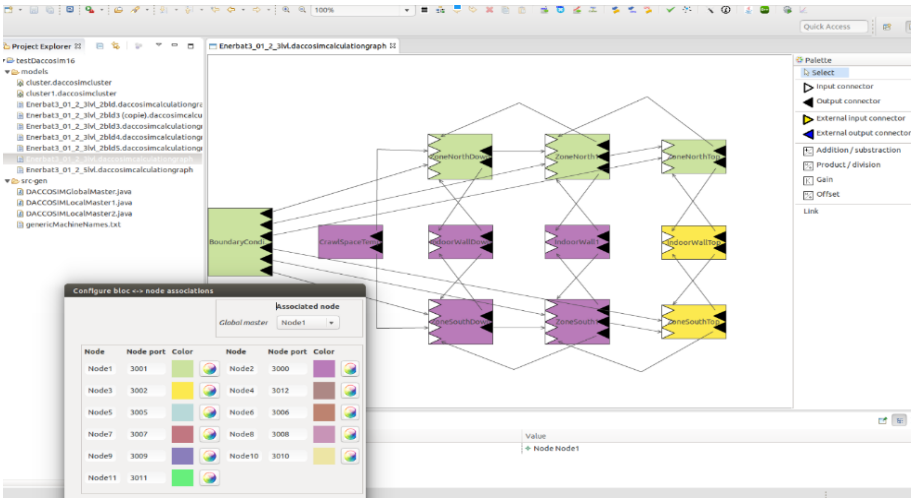


FIGURE 3 – Interface graphique de DACCOSIM (graphe de tâches)

Un des buts principaux de cette thèse est de mapper et déployer un graphe de FMU sur un graphe de ressources distribuées. Cependant, ces tâches sont exposées comme des boites grises hétérogènes qui communiquent fortement entre elles par des envois de nombreux et petits messages. La récupération d'informations individuelle sur l'exécution de ces FMU est quasi-impossible : nous ne pouvons accéder ni aux temps de calcul, ni aux coûts de communication. Malgré cela, notre travail consiste à proposer une démarche ou une méthodologie qui permet d'obtenir un placement plus au moins optimisé, c'est-à-dire qu'on doit résoudre un problème d'affectation de ces différentes boites sur des ressources de calcul, en prenant en compte les caractéristiques des FMU et celles des ressources. Cette proposition sera abordée dans le chapitre 5.

3.1 Principaux concepts de l'architecture de DACCOSIM

La structure générale de DACCOSIM se compose d'un ensemble de masters locaux. Chaque master local se trouve sur un unique nœud physique, et contrôle un ensemble de FMU situées sur ce nœud de calcul. Les communications de ces différents acteurs vont être supervisées et synchronisées par le master global. DACCOSIM [30] s'appuie donc sur une architecture hiérarchique à quatre niveaux (de 0 à 3) qui se résume comme suit (voir la figure 4) :

- Au niveau de la racine de DACCOSIM (niveau 3), nous trouvons le **master global** qui pilote et supervise l'ensemble de la co-simulation. Il joue en fait deux rôles majeurs, un rôle de master global qui gère et agrège les décisions de tous les masters locaux, et un rôle de master local qui centralise toutes les décisions des FMU situées sur sa machine.
- Au niveau 2, on trouve les **masters locaux**, à raison d'un par machine (un par noeud de cluster). Chacun master local collecte les décisions du master global, coordonne et synchronise les données envoyées par les FMU placées sur son noeud.
- Au niveau 1, on trouve un code « *wrapper* » connecté à la FMU en conformité aux capacités du standard FMI. Le *Wrapper* représente le thread qui lance les calculs de chaque FMU à chaque pas de temps (il est lui même multi-threads).
- Les feuilles de cette arborescence (niveau 0) représentent l'ensemble des **FMU** de notre système global (il y a normalement plusieurs FMU par noeud).

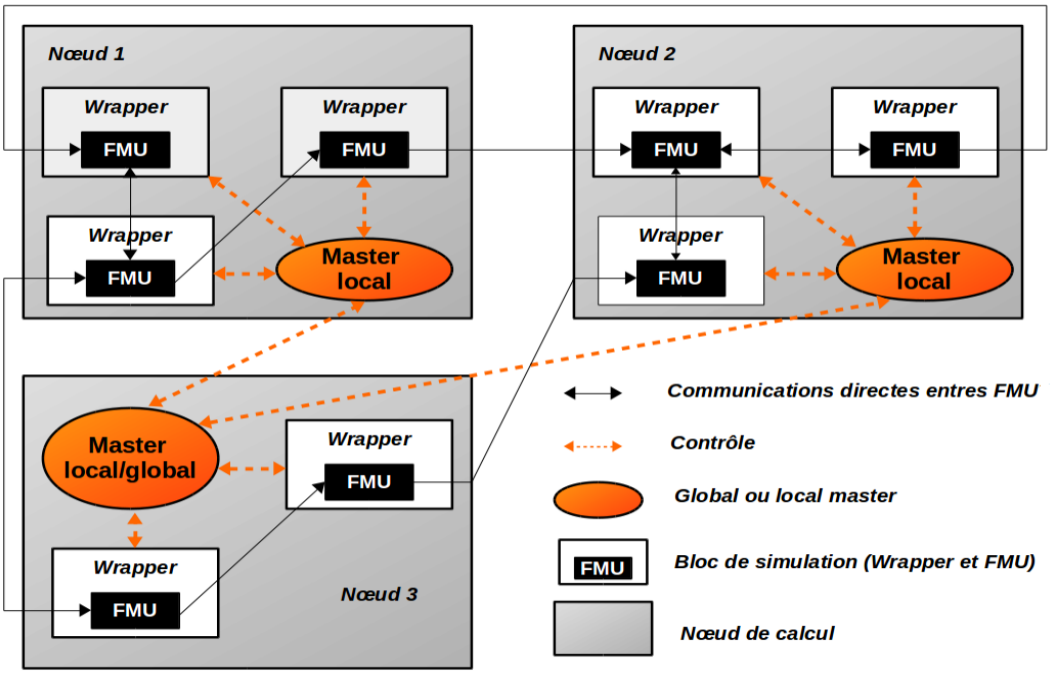


FIGURE 4 – Architecture de DACCOSIM distibuée sur 3 noeuds

Comme l'illustre la figure 4, un noeud physique comporte un code de co-simulation contenant un nombre fini de FMU supervisées par un seul master local qui est à son tour connecté à un master global de co-simulation. Par exemple, pour exécuter trois FMU se trouvant sur le même nœud, DACCOSIM lance trois *Wrapper* en parallèle, et chaque *Wrapper* exécute des threads en parallèle selon le mode d'orchestration.

Aspect *Mutli-threaded* de l'architecture DACCOSIM

DACCOSIM est une plate-forme *Multi-threaded* [30] dont chaque *Wrapper* de FMU contient trois types de *threads* exécutés à chaque pas de temps, ces derniers sont représentés ci-après :

— Un thread de calcul qui exécute et gère la phase de « calcul » de chaque FMU (le solveur de chaque FMU lance ses calcul en exécutant la primitive « doStep(step) »). À l'issue de la phase de calcul, ce même thread devient un thread de communication qui permet d'envoyer des résultats à d'autres FMU.

— Deux threads de réception : un thread type « ZMQ » (inter-nœuds ou intra-nœud), et un thread « inproc » (intra-noeud). Ces derniers permettent de recevoir les données des autres FMU connectées à la FMU en question.

La figure 5 ci-dessous montre l'exemple d'une co-simulation qui comporte deux FMU interconnectées par un seul lien. Toutes les FMU sont exécutées en parallèle, et chacune d'elles lance une instance de thread de calcul pour exécuter la phase de calcul qui devient après un thread de transmission, et deux instances de threads pour recevoir des données en intra- et inter-noeuds.

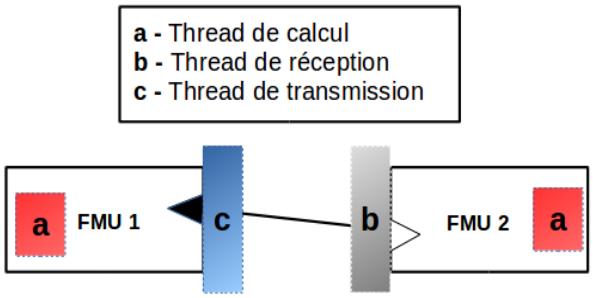


FIGURE 5 – Exemple d'une co-simulation DACCOSIM

Aspects distribués et parallèles de l'architecture DACCOSIM

De manière générale, DACCOSIM génère un ensemble de codes Java, correspondants aux codes des masters locaux localisés sur des nœuds de calculs différents (comme l'illustre la figure 4) dont chacun contient une ou plusieurs FMU associées à ces masters locaux. Un utilisateur DACCOSIM a la possibilité de décomposer manuellement son graphe de FMU en un ensemble de blocs de FMU, tout en les distribuant et les déployant sur une architecture répartie. Pour chaque pas de temps, ces FMU sont vues comme des boites grises autonomes intégrant leur propre solveur et s'exécutent en parallèle de manière indépendante. Ce sont les raisons pour lesquelles on dit que DACCOSIM est une plate-forme distribuée et parallèle.

Cependant, la taille et la complexité des systèmes complexes nous impose à intégrer un algorithme de répartition automatique des FMU, qui permet de distribuer au mieux les nombreuses boites grises du système sur une architecture distribuée ; cette partie sera détaillée dans le chapitre 5.

Principes d'exécution distribuée et parallèle

Dans son mode d'exécution le plus simple (mode *ordered*), le master global lance l'ensemble des FMU en parallèle à un même instant, et chaque FMU exécute sa phase de calcul indépendamment des autres FMU. À la fin de la phase de calcul, ce master synchronise toutes les FMU, et re-calcule le nouveau pas de temps s'il utilise une méthode à pas variable. À la fin de ce contrôle, le master global autorise toutes les FMU à effectuer leurs phases de communication simultanément.
Un autre mode d'exécution plus complexe (mode *Overlapped*) est conçu pour augmenter la vitesse de traitement dans certain cas (voir la section 3.2).

Type de communication dans DACCOSIM

Les communications de DACCOSIM se classent en deux catégories illustrées dans la figure 6 :

1. **Communications centralisées** : ce mode concerne des communications « verticales » effectuées à la fois entre chaque master local (niveau 2) et les *wrapper* de FMU (niveau 1), pour la prise de décisions au sein du même nœud, et entre le master global (niveau 3) et les masters locaux pour les prises de décisions concernant toute la simulation.
2. **Communication décentralisées** : ce mode concerne des communications « horizontales » effectuées entre les différentes FMU du système (niveau 0) sans passer par le master global ou le master local de la co-simulation. Ces communications peuvent être effectuées entre les FMU, en intra- ou inter-noeuds.

Note : La norme FMI ne spécifie pas ces communications directes entre FMU (communications décentralisées). C'est un choix de la plate-forme DACCOSIM qui a mis en place ces mécanismes pour traiter les communications directes, ce qui représente une innovation que l'on ne retrouve généralement pas dans les articles concurrents à DACCOSIM.

3.2 Caractéristiques détaillées de DACCOSIM

Dans cette section, nous listons les principales caractéristiques de la plate-forme DACCOSIM, définies comme suit :

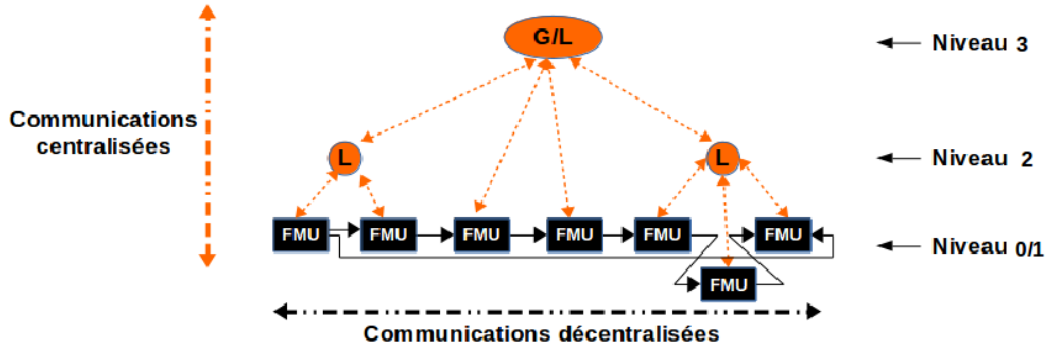


FIGURE 6 – Les communications dans DACCOSIM

Co-initialisation

Afin de démarrer le système avec un état de fonctionnement physiquement acceptable, il est primordial d'initialiser le système avec des valeurs cohérentes, en effectuant une initialisation distribuée entre FMU qu'on appelle co-initialisation. DACCOSIM a utilisé le même principe de co-initialisation que celui illustré dans l'article [14], permettant de construire un graphe de dépendance en s'appuyant sur les dépendances internes et externes des sorties et des entrées provenant des autres FMU. De manière générale, ces algorithmes collectent les informations de dépendance internes à partir du fichier *ModelDescription.xml* [3] associé à chaque FMU, et externes à partir du schéma de calcul (graphe de FMU de DACCOSIM).

DACCOSIM réalise une conversion automatique d'un graphe de causalité en graphe acyclique. Grâce à la co-initialisation, cette plate-forme génère un graphe de dépendance acyclique type DAG (*Directed Acyclic Graph*) fortement couplé, qui respecte la contrainte de dépendance pour initialiser les entrées et les sorties des FMU du système.
Dans le domaine des *Smart Grids*, la co-initialisation est souvent appelée « *local flow* » par les experts des réseaux électriques.

Dans la figure 7 ci-dessous, nous avons donné un exemple d'un graphe de dépendance décrit dans [47], avec les associations suivantes : ZoneA.Tin → wall.T-a, ZoneB.Tin → wall.T-b, SineFMU.y → ZoneA.Text, SineFMU.y → ZoneB.Text tel que FMU1.y → FMU2.x veut dire que le port d'entrée « x » de la FMU2 prend la valeur du port de sortie « y » de la FMU1.
Toutes ces informations sont utiles pour le processus de co-initialisation : il doit propager toutes les valeurs de sorties vers des entrées pour rendre le graphe cohérent avant de lancer la co-simulation. Notons que dans DACCOSIM la construction du graphe de dépendance effectue une vérification de la cohérence du graphe d'une manière automa-

3. Un fichier qui décrit le modèle métier exporté sous la forme d'une FMU.

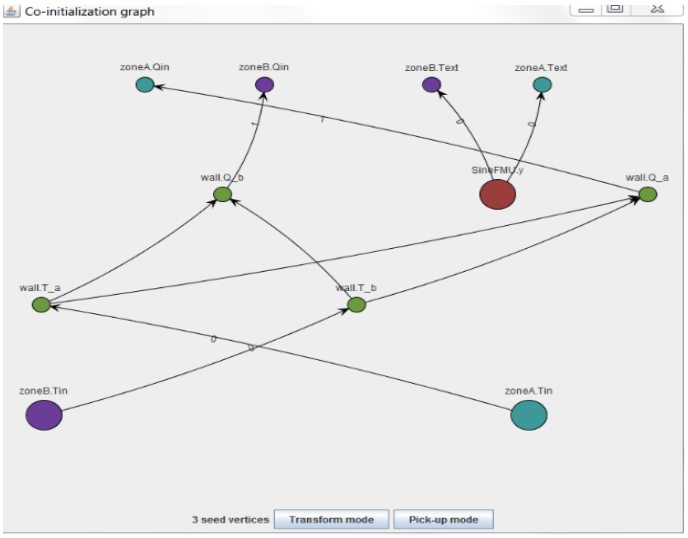


FIGURE 7 – La co-initialisation

tique.

DACCOSIM est basée sur l'algorithme de co-initialisation *NewtonRaphson* [68]. Cet algorithmes se fonde sur le principe de propagation des données selon les dépendances globales (externes et internes) matérialisées dans un article de l'UC *Berkeley* [14].

Visualisation du graphe

Comme l'illustre la figure 7, DACCOSIM offre deux représentations du graphe qui sont les suivantes :

— Le ***Transform mode*** est le mode par défaut ; il affiche l'ensemble du graphe, et permet d'agrandir ou de réduire les éléments du graphe de dépendance, pour améliorer la vue du graphe global.

— Le ***Pick-up mode*** permet de sélectionner et de déplacer les nœuds graphiques, afin d'améliorer le rendu graphique.

Temps continu discrétisé en pas de temps

Dans la plate-forme DACCOSIM, un pas de temps se compose de trois phases, à savoir la phase de calcul, la phase de contrôle et la phase de communication. La figure 8 représente un pas de temps avec deux modes d'orchestration qui vont être abordés par la suite.

Dans DACCOSIM, un pas de temps lancé pour chaque FMU est constitué de :

— **La phase de calcul**

Cette phase est déclenchée par le thread *Wrapper* qui permet de lancer les calculs

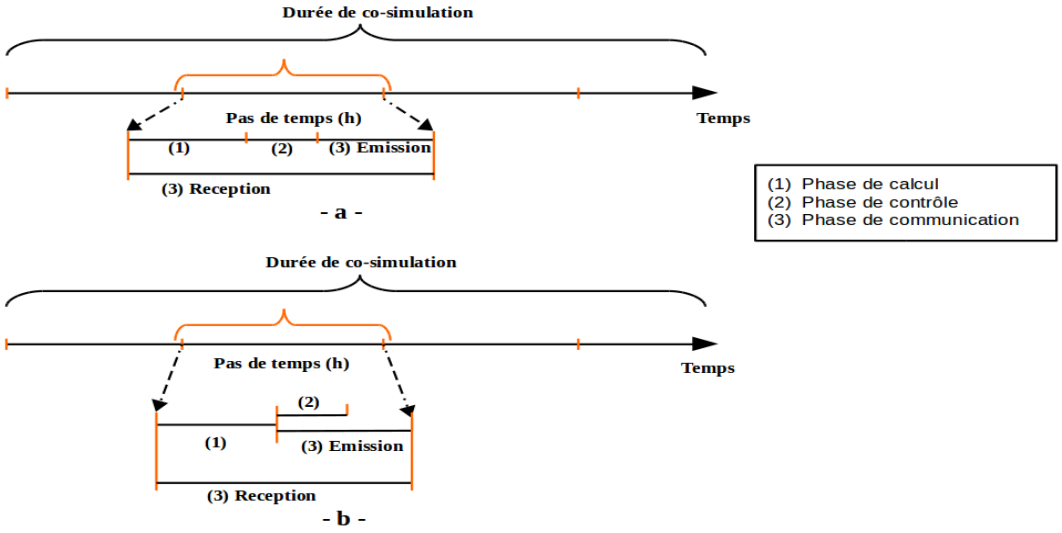


FIGURE 8 – (**a**) Une simulation avec le mode *Ordered* (**b**) Une simulation avec le mode *Overlapped*

de la FMU en exécutant la fonction *dostep*(h) avec le pas de temps (h), et de sauvegarder son état actuel en faisant un *checkpoint*[4].
Si la FMU utilise une méthode avec un pas de temps variable, le *Wrapper* de cette FMU contrôlera les dérivées de ses grandeurs, et définira un pas de temps plus petit à la fin de cette phase en cas de manque de précision dans ses calculs. À l'inverse, si les grondeurs évoluent peu, la contrainte de pas de temps sera relâchée.

— **La phase de contrôle** se place à deux niveaux :

1. Le premier contrôle est au niveau du master local, qui permet de collecter les décisions de ses FMU afin de les transférer au master global. Si nous utilisons une méthode de pas variable, le master local se trouvant sur chaque machine calcule le minimum des pas de temps proposé par ses FMU afin de l'envoyer au master global.
2. Le deuxième contrôle est au niveau du global master, ce dernier est un master central qui est relié à tous les masters locaux. Il contrôle l'ensemble de la co-simulation en agrégeant les décisions envoyées par les masters locaux, afin de prendre une décision unique applicable à toutes les FMU.

— **La phase de communication**
Dans cette phase, le master global reçoit les propositions des masters locaux. Après le calcul de la décision globale, il va la transmettre à chaque FMU en passant par les masters locaux. À la fin du pas de temps, toutes les FMU envoient des données simultanément (voir la section 3.1). Sachant que les threads de réception sont toujours actifs depuis le début de la co-simulation.

4. Un mécanisme qui sauvegarde l'état du système (toutes les variables...).

Méthode de calcul à pas contant ou adaptatif

L'exécution d'un pas de temps d'une multi-simulation DACCOSIM se fait soit en utilisant des méthodes à pas constants, soit des méthodes à pas auto-adaptatifs (variables).

— Avec la méthode à pas **constant**, le pas de temps est fixe pour les FMU de ce système pendant toute la co-simulation. Cette méthode ne vérifie pas la précision de ses calculs. La figure 9 montre une simulation quelconque avec un pas de temps constant (h). Notons qu'il est préférable d'adopter un petit pas de temps pour être sûr de faire toujours des calculs assez précis. Mais le réglage du pas de temps reste à la charge des concepteurs des modèles du système et dépend des constantes de temps des composants.

FIGURE 9 – simulation avec une méthode à pas constant

— Avec une **méthode à pas auto-adaptatif**, on utilise un pas de temps variable entre les itérations. Chaque FMU lance sa phase de calcul en premier, et à l'issue de cette phase, le *wrapper* de la FMU vérifie la précision des calculs en estimant les erreurs par des algorithmes codés dans DACCOSIM. Ces algorithmes vérifient si l'estimation de l'erreur est supérieure à un certain seuil, si c'est le cas, ils proposent un pas de temps plus petit donc plus précis en effectuant un *rollback*.

Comme illustré sur la figure 10, une co-simulation avec un pas de temps variable permet à une FMU d'augmenter, diminuer ou garder son pas de temps, en fonction du degré de précision de ses résultats mais ce mécanisme est incompatible avec le temps réel (HIL [5]). De ce fait, si le solveur du composant utilise un plus grand pas de temps h' tel que (h<h'), les performances de la co-simulation seront améliorées. Par contre, si il est nécessaire d'utiliser un plus petit pas de temps h'' ($h'' < h'$), la co-simulation globale du système sera localement ralentie pour garder des résultats suffisamment précis. Autrement dit, les solveurs qui utilisent des méthodes auto-adaptatives ou variables ajustent dynamiquement leur pas de temps sur la base d'une analyse du signal associé à chaque variable d'état calculée.

En général avec les plate-formes du marché, il n'est pas souvent possible d'utiliser des méthodes à pas variables contrairement à notre plate-forme. En fait, DACCOSIM propose trois algorithmes d'adaptation du pas de temps pour estimer les

5. Hardware-In-the-Loop.

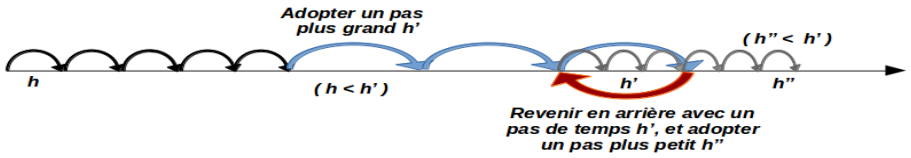


FIGURE 10 – simulation avec une méthode auto-adaptative (variable)

erreurs produites : la méthode d'*Euler* [26] qui s'appuie sur une résolution des équations différentielles du premier ordre, la méthode de *Richardson* [63] qui permet d'effectuer deux calculs en parallèle avec le pas de temps normal (h) et son demi pas ($\frac{h}{2}$), et la méthode d'*Adam-Bashforth* ([36] dans le chapitre 3) à pas multiples. Cette dernière est plus précise que la méthode d'*Euler* car elle est multi-pas, c'est-à-dire qu'elle utilise plusieurs pas de temps précédents pour évaluer le point courant, mais elle consomme beaucoup de mémoire en sauvegardant l'état du système sur plusieurs pas consécutifs.

Détection d'un événement dans DACCOSIM

De manière générale, DACCOSIM permet de coupler des composants physiques avec des fonctions de contrôle basées sur des événements de type alarmes ou commandes. Dans cette plate-forme, un événement correspond à un changement de valeurs d'une grandeur discrète (un entier, un booléen) d'une FMU. Vu que le standard FMI-CS 2.0 n'envisage pas la détection précise et immédiate des événements, une manière d'y arriver consiste simplement à observer les variables aux instants des phases de communications et de détecter des changements de valeurs. Ensuite, on peut cerner cet événement grâce au mécanisme de *rollback* en appliquant une méthode dichotomique.

Mécanisme de *Rollback* dans DACCOSIM

Dans le cas de DACCOSIM, si nous utilisons une méthode auto-adaptative (voir la section 3.2), toutes les FMU démarrent leur pas de temps (h) en parallèle, et chacune de ces FMU exécute en premier sa phase de calcul. À la fin de cette phase, le *wrapper* de chaque FMU vérifie la précision des calculs et propose d'augmenter ou de diminuer le pas de temps au master local dont il dépend. Ensuite, le master local calcule le minimum des pas de temps proposés par les *wrapper* de FMU qu'il gère, puis il envoie à son tour une proposition au master global qui décide soit d'utiliser un pas de temps plus grand h'' pour toutes les FMU, soit de revenir en arrière avec un pas de temps plus petit h'.
Si le master global propose un pas de temps plus petit que le pas de temps actuel, les masters locaux obligent leurs FMU à refaire leurs calculs plus finement avec ce nouveau pas de temps proposé ($h' < h$). Ce mécanisme représente le *rollback*, comme l'illustre la

figure 11.

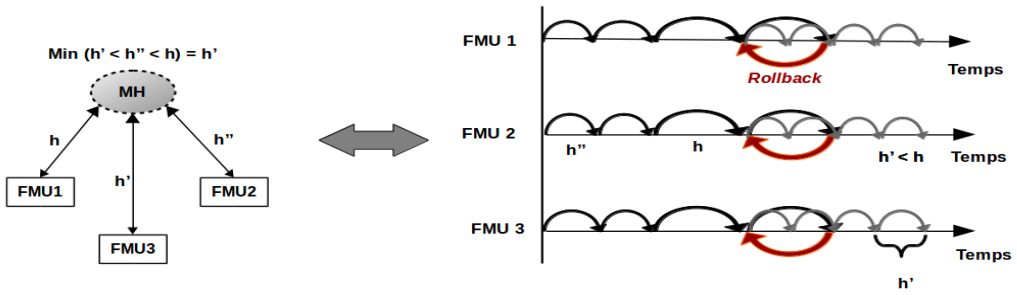


FIGURE 11 – Mécanisme du retour en arrière dans DACCOSIM

Pour effectuer un *rollback*, DACCOSIM passe par les étapes suivantes :

1. Avant que les FMU exécutent une phase de calcul, elles sauvegardent leurs états actuels en exécutant la fonction *checkpoint*. Ensuite, elles lancent leurs calculs avec la fonction *doStep*. À la fin de cette phase, les *wrappers* de FMU vérifient la précision de leur calculs en estimant localement leurs erreurs par rapport à une tolérance [6]. Si un *wrapper* détecte des erreurs trop importantes au niveau du calcul de sa FMU, il envoie au master local un message avec une suggestion d'un nouveau pas de temps plus petit que le pas courant.

2. Le master local calcule le minimum de toutes les suggestions de ses FMU afin de l'envoyer au master global.

3. Le master global collecte tous les messages envoyés par les masters locaux et calcule le minimum global de toutes ces valeurs. En cas de *rollback*, ce master global demande à tous les masters locaux de refaire leur phase de calcul avec un pas de temps plus petit (h').

Cependant, si les FMU du système ne détectent pas d'erreurs ($erreur < tolérance$), le master global pourra augmenter le pas de temps à la prochaine itération. Cette augmentation de pas de temps sera d'autant plus importante que l'erreur sera faible (par rapport à la tolérance admise).

On remarque que DACCOSIM impose le même pas de temps à toutes les FMU de la co-simulation, c'est-à-dire que l'outil fait l'hypothèse d'un couplage fort avec un pas de temps commun à toutes les FMU. Le tableau 3.1 ci-dessous illustre les différents cas implantés dans la plate-forme DACCOSIM.

6. Tolérance relative, donnée par l'utilisateur en fonction du modèle.

h	**Constant**	**Auto-adaptatif**
Commun	oui	oui (*rollback*)
Différencié	non	non car cela ne supporte pas de *rollback*

TABLE 3.1 – Les différents cas envisagés dans DACCOSIM

Dans la plate-forme, on a utilisé un pas de temps commun pour toutes les FMU afin (1) de faciliter et simplifier les échanges des données entre FMU : la phase de communication se fait une seule fois à la fin de ce pas de temps commun contrairement au deuxième mode où les communications sont aléatoires, et (2) d'effectuer des *rollbacks* en cas de besoin : si on adopte une méthode variable, le master global oblige toutes les FMU à revenir en arrière au même point, contrairement au pas de temps différencié qui ne peut pas tolérer ce mécanisme de *rollback*, car les FMU ont chacune leur propre pas de temps (elles ne reviennent pas au même point).

Modes d'orchestration

DACCOSIM lance toutes les FMU en parallèle, et chacune de ses FMU exécute ses pas de temps en appliquant un des deux modes de synchronisation qui sont détaillés ci-dessous.

— **Mode *Ordered* ou *Control First Send Last*** :
Durant un pas de temps, chaque FMU lance ses trois phases l'une après l'autre dans l'ordre « calcul-contrôle-communication », comme illustré sur la figure 12.

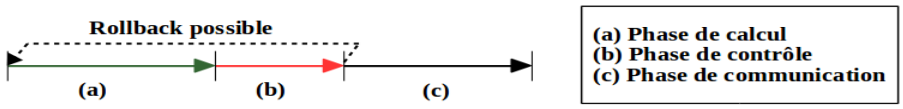


FIGURE 12 – Séquencement d'un *wrapper* de FMU avec le mode *Ordered*

Dans ce mode, toutes les FMU s'exécutent en parallèle et démarrent simultanément leur phase de calcul. À la fin de cette étape, chaque FMU vérifie la précision de ses calculs afin de proposer un pas de temps plus petit que le précédent si les calculs sont entachés d'erreurs. Les masters collectent les informations de toutes les FMU et autorisent ensuite les communications sauf si un *rollback* est nécessaire. Dans cette méthode, chaque FMU doit attendre la réponse et l'autorisation du master global avant d'échanger ses données avec les autres FMU. Le diagramme ci-dessous décrit le principe général du fonctionnement de DACCOSIM avec la méthode *Control First Send Last* en utilisant un pas de temps variable. Notons que le fonctionnement de DACCOSIM avec une méthode constante représente un

cas particulier du pas de temps variable dans lequel l'implémentation est simplifiée.

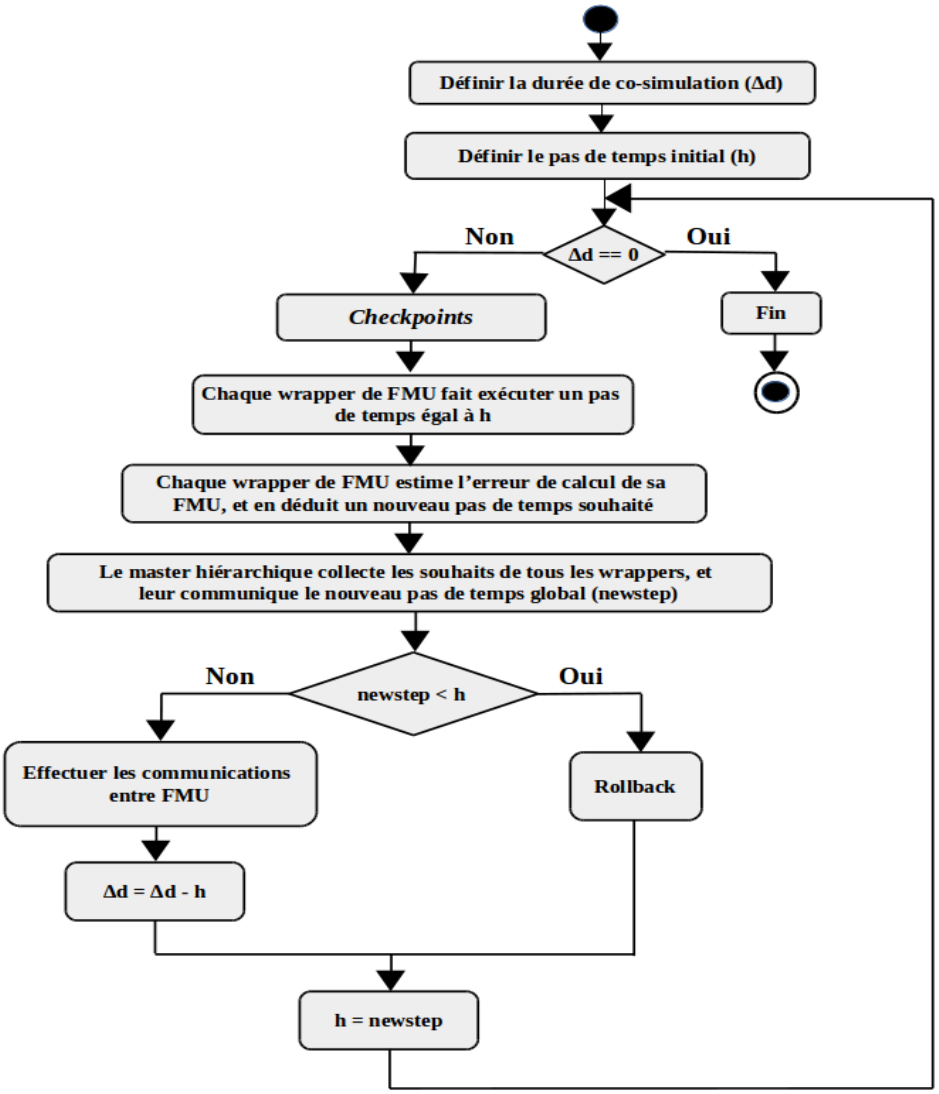


FIGURE 13 – Fonctionnement de DACCOSIM avec le mode *Ordered*

Notons que le mode *Ordered* est favorable lorsque la co-simulation est susceptible de comporter beaucoup de *rollbacks*.

— **Mode *Overlapped* ou *Send First Control Last*** :
Durant un pas de temps, chaque FMU exécute en premier sa phase de calcul, ensuite elle entre dans la phase de contrôle et dans la phase de communication en parallèle. Ce mode permet de recouvrir les communications inter-FMU avec les échanges entre les FMU et le master, ce qui permet à la fois de masquer les effets des temps de latence, et d'accélérer la co-simulation du système global (voir la figure 14).

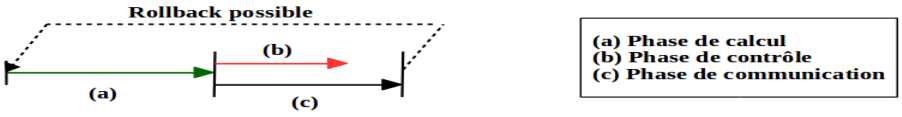


FIGURE 14 – Séquencement d'un *wrapper* de FMU avec le mode *Overlapped*

Comme indiqué dans le figure 15 ci-après, chaque FMU commence par lancer sa phase de calcul, ensuite, après la vérification de ses calculs, elle envoie et reçoit ses données avant la réception de la réponse du master global. Dans le cas où on a besoin d'un *rollback*, le master global interrompe les échanges entre FMU afin de refaire les calculs des FMU du système global.

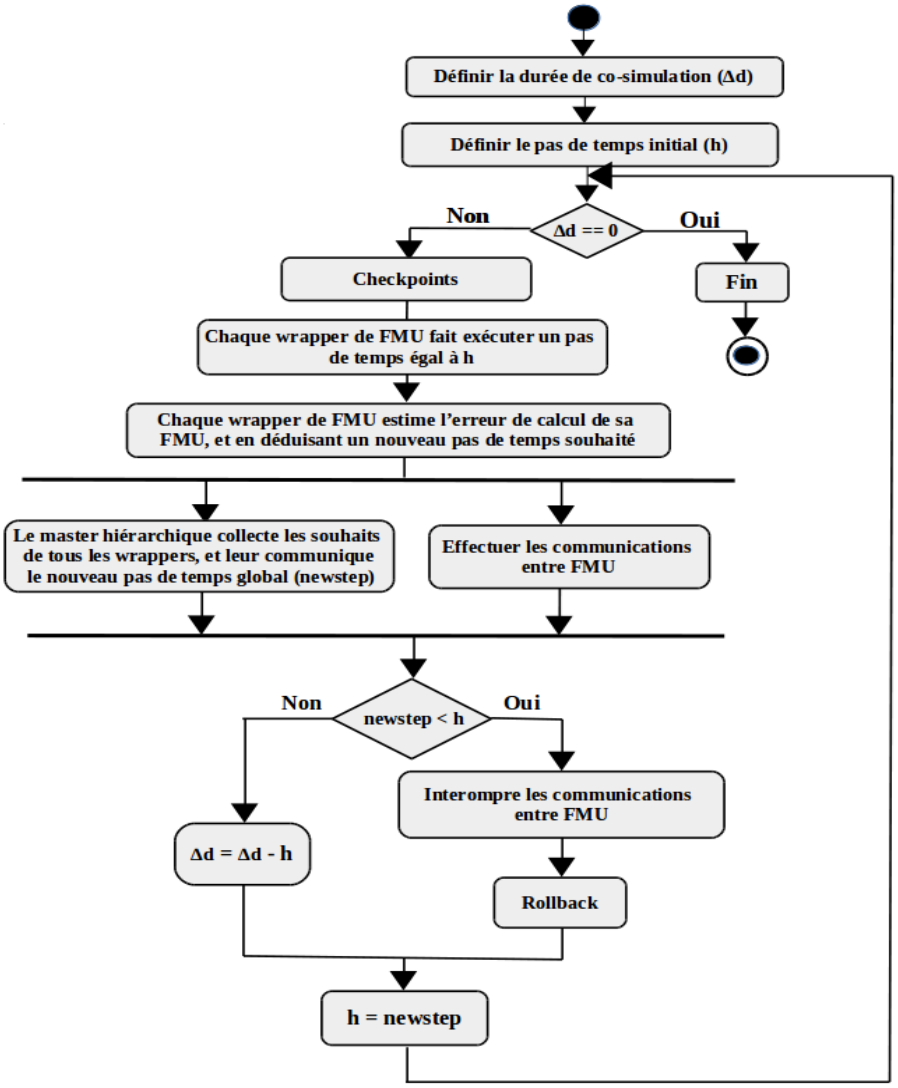


FIGURE 15 – Fonctionnement de DACCOSIM avec le mode *Overlapped*

Notons que le mode *Overlapped* est favorable lorsque la co-simulation est susceptible de comporter peu de *rollbacks*.

Grâce au parallélisme de ce mode, on peut espérer avoir de bonnes performances comparé au mode *Ordered*, car toutes les FMU effectuent leurs communications en parallèle avant même de recevoir la décision globale du master central. Tout dépend du nombre de *rollbacks* de la co-simulation ce qui est imprévisible à prévoir à l'avance. Notons que le mode d'orchestration *Overlapped* n'a pas de sens avec un pas de temps fixe.

Extrapolation des entrées

DACCOSIM s'appuie sur la norme FMI 2.0 : cette version offre la possibilité d'accéder non seulement aux valeurs d'entrées de la FMU, mais également aux dérivées de ces dernières. Cette caractéristique permet d'extrapoler les entrées afin d'améliorer la précision des résultats [30], ce qui permet à la co-simulation de gagner en précision de calcul même si cette dernière perd un peu en performance (elle augmente la charge de calcul de chacune de ses FMU).

Protocoles de communication

Nous trouvons dans DACCOSIM deux niveaux de communication : des communications directes ou horizontales entre FMU connectées, et des communications indirectes ou verticales passant par la hiérarchie des masters. Dans DACCOSIM, les FMU qui se trouvent sur la même machine utilisent le même protocole de communication qui est soit $InProc$ (*inside-process*), soit ZMQ[7]. Les FMU qui se trouvent sur des machines différentes s'appuient sur le protocole de communication ZMQ (voir la figure 16). À l'opposé, le mode $InProc$ propre à DACCOSIM est basé sur le passage de message entre *threads*. Ce mode évite de passer par le réseau de communication, alors que le mode ZMQ est fondé sur le protocole TCP pour les communications entre nœuds de calcul. Le fonctionnement du mode ZMQ est basé sur la création de *sockets* pour chaque entité.

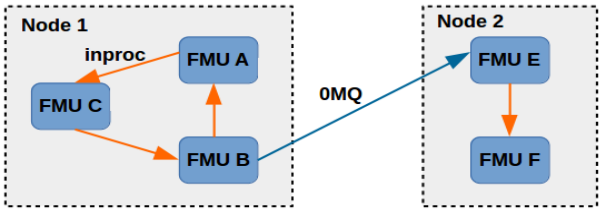


FIGURE 16 – Exemple de communications DACCOSIM en ZMQ et $InProc$ sur cluster de PC multi-coeurs

En outre, sur un cluster de machines, il est possible d'utiliser ZMQ et $InProc$ (voir la

7. ZeroMQ, 0MQ ou ZMQ est une bibliothèque qui permet de créer des applications distribuées types client-serveur.

figure 16), ou uniquement ZMQ. Sur une seule machine, il est possible d'adopter $InProc$ seul ou ZMQ seul.

4 Co-simulation avec DACCOSIM

4.1 Fonctionnement de DACCOSIM

Comme présenté dans les sections précédentes, au début d'un pas de temps, le master global de DACCOSIM autorise tous les masters locaux à commencer leurs simulations, chacun de ces derniers lance ses FMU en parallèle, pour un pas de temps donné. L'ensemble de ces FMU exécutent leur pas de temps selon le mode d'orchestration défini par l'utilisateur. À l'issue de ce pas de temps, certaines FMU échangent des informations avec d'autres FMU du système.
DACCOSIM a été spécialement conçu pour co-simuler les systèmes complexes qui fonctionnent sur un cluster de nœuds multi-coeurs, mais la distribution automatique des différentes FMU sur une telle architecture reste difficile à réaliser.

Afin de déployer une co-simulation DACCOSIM sur une architecture distribuée, nous avons suivi une démarche s'appuyant sur trois grandes étapes illustrées par la figure 17.

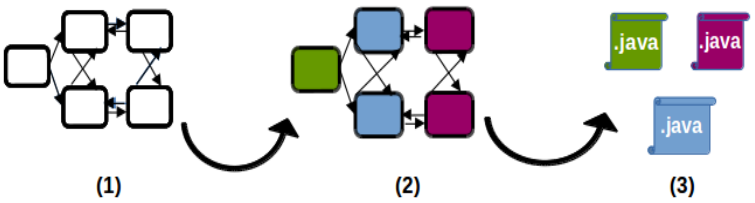


FIGURE 17 – Pricipales étapes de construction d'une co-simulation avec DACCOSIM

Les étapes présentées sur cette figure sont les suivantes :

1. **Définition d'un modèle de *Smart Grid*** :
   Un utilisateur peut construire graphiquement son schéma de calcul soit en utilisant directement l'interface graphique de DACCOSIM (voir la figure 3) mais uniquement si la taille du problème est raisonnable (peu de FMU), soit en utilisant un script DSL (*DACCOSIM Script Language*) qui permet de générer automatiquement tout le schéma de calcul d'une co-simulation de grande taille (voir la figure 18). Un script DSL définit automatiquement un système en décrivant toutes les FMU ainsi que les connexions reliant ces dernières.

2. **Placement des tâches hétérogènes sur des noeuds virtuels** :
   Après la première étape, l'utilisateur peut choisir comment placer les différentes FMU sur des nœuds virtuels avant de les exécuter sur une architecture physique

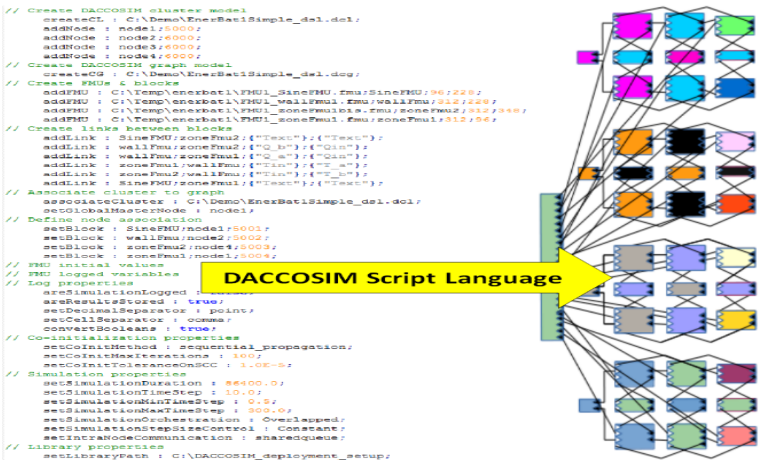


FIGURE 18 – Un script DSL et son équivalent graphique.

distribuée ; cette étape se fait de manière manuelle ou automatique. Notons que les FMU qui ont la même couleur dans le graphe de l'interface de DACCOSIM se retrouveront sur le même nœud de calcul.

3. **Génération des codes de simulation** :
   Cette dernière étape prépare et génère les codes Java, qui représentent les *wrappers* de FMU, les masters locaux et le master global. Chaque master local situé sur un nœud physique fait appel à des méthodes qui lui permettent de communiquer à la fois avec le master global de la co-simulation et avec ses FMU.
   De manière générale, le code généré par DACCOSIM contient :

   (a) **Une phase d'initialisation** : dans cette phase, DACCOSIM initialise et teste les chemins d'accès aux bibliothèques, crée le master local du nœud et l'ensemble des *Wrapper* de chaque FMU, et initialise les variables de la co-simulation globale (pas de temps (h)) et la durée globale de la co-simulation.

   (b) **Une phase de co-initialisation** : cette phase est essentielle pour le fonctionnement correct et cohérent du système. Ce processus permet l'initialisation de chaque entrée et sortie des FMU, en appliquant des algorithmes de co-initialisation qui ont été abordés dans la sous-section 3.2 de ce chapitre.

   (c) **Une phase de co-simulation** : cette phase permet de lancer en parallèle les différentes FMU qui se trouvent sur le même nœud et sur l'ensemble des nœuds du cluster.

   (d) **Une phase de clôture** : la phase de terminaison de la co-simulation, qui permet d'arrêter l'ensemble des *wrapper* des FMU du système et de fermer les *sockets*.

   (e) **La récupération des résultats** : elle permet de collecter tous les résultats

de chaque master local ainsi que les résultats de leur FMU, éventuellement stockés sur des disques locaux aux différents nœuds de calcul.

Lorsque nous souhaitons exécuter le modèle sur une seule machine, DACCOSIM génère un seul code de co-simulation qui inclut à la fois le master global et l'ensemble des FMU du système ; ce master joue également le rôle d'un master local pour un seul nœud de calcul. Dans le cas où on exécute le modèle sur une architecture distribuée, nous générons plusieurs codes de co-simulations (un par master local) dont chacun d'eux peut être exécuté ensuite sur une machine différente.

4.2 Briques de base utilisées dans DACCOSIM

L'implémentation de DACCOSIM et l'exécution d'une co-simulation sur une architecture distribuée repose sur les composants suivants :

— DACCOSIM est implémenté avec un langage orienté objet, dit Java. Il rend la programmation portable sur plusieurs systèmes d'exploitations (Linux, Winsdows) en utilisant un JRE [8].

— DACCOSIM repose principalement sur JavaFMI [9] ; une bibliothèque Java *open source* qui est dévelopée et maintenue par *SIANI* [10] (partenaire académique d'EDF pour la co-simulation). Cette bibliothèque fournit un ensemble de composants pour exploiter la norme FMI. En outre, JavaFMI est portable sur deux systèmes d'exploitations : *Windows* et *Linux*, en 32 bits ou 64 bits.

— La création des FMU se base sur des environnements de développement de modèles conformes au standard FMI. Il existe de nombreux outils qui utilisent le FMI comme par exemple *Dymola*, *OpenModelica*, *MathModelica*, *Papyrus*, *ControlBuild*...

— Le langage de script DSL (*DACCOSIM Script Langage*) est utilisé dans DACCOSIM pour spécifier et construire des modèles de grandes tailles.

— Le déployeur « DacRun » est développé avec le langage de programmation *Python2.7* [11] et la bibliothèque *ppserver* [12]. Il permet de déployer les codes Java sur des ressources informatiques distribuées à raison d'une JVM [13] par nœud de calcul.

8. *Java Runtime Environment.*
9. https ://bitbucket.org/siani/javafmi/wiki/Home.
10. Département informatique de l'Université de *Las Palmas*, Espagne.
11. https ://docs.python.org/2.7/.
12. *Parallel Python Network Server.*
13. Java virtual machine.

4.3 Validation des résultats obtenus par DACCOSIM

Afin d'analyser la pertinence des résultats numériques trouvés avec le logiciel DACCOSIM, nous avons utilisé un outil de référence appelé *Dymola*[14] développé par Dassault Systèmes. Il représente à la fois un outil de modélisation et de simulation basé sur le langage *Modelica*. *Dymola* ne fait pas vraiment de la co-simulation et ses performances sont limitées : pas de parallélisme, pas de distribution des calculs. Cet outil permet cependant de concevoir et de simuler de petits systèmes multi-physiques et multi-domaines. Pour vérifier la cohérence des résultats numériques de DACCOSIM, nous avons testé les mêmes scénarios fournis par EDF avec la plate-forme DACCOSIM et l'outil *Dymola*. Ensuite, nous avons analysé et comparé les résultats numériques obtenus avec ses deux environnements.

Grâce à *Dymola*, nous avons pu valider expérimentalement le fonctionnement des systèmes physiques ainsi que les résultats numériques de la plate-forme DACCOSIM, au moins sur des systèmes de petite taille capables de tourner sans *Dymola*. En fait, ce qui est vraiment intéressant avec *Dymola*, c'est qu'il permet de traduire un code *Modelica* en une FMU en Linux ou en Windows, disponible en 32 et 64 bits.

Dans la figure 19, nous avons analysé la précision des résultats des températures d'une zone d'un *use-case* fourni par EDF en utilisant Dymola Modélica (sans FMU) et Dymola avec FMU. Les résultats obtenus par Dymola FMU sont semblables à ceux obtenus par *Dymola* Modelica. Cela montre que nous pouvons considérer les FMU comme des modèles de base sans *Dymola* qui est lui-même considéré comme un outil de référence au niveau mondial.

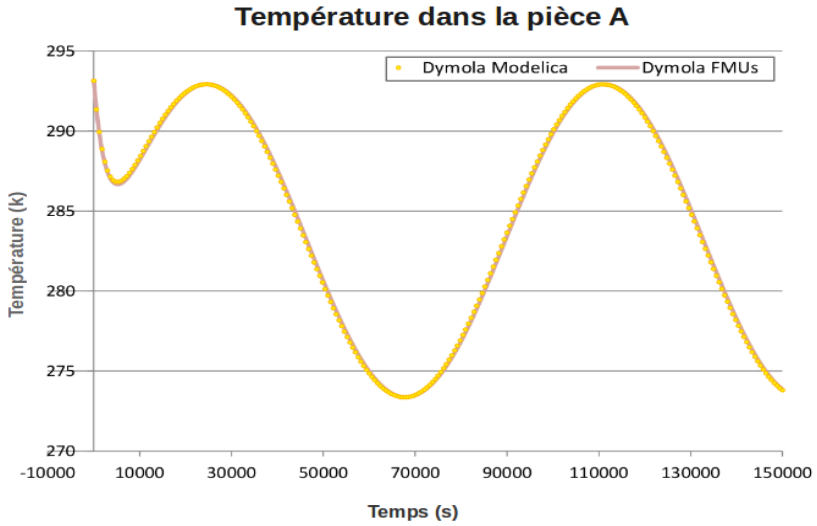


FIGURE 19 – Comparaison de résultats entre *Dymola* avec FMU et *Dymola* sans FMU

14. https ://org-www.3ds.com/products-services/catia/products/dymola/latest-release/

5 Déploiement sur cluster

Afin de faciliter la compilation et l'exécution des différents codes de co-simulation sur une architecture distribuée, nous avons développé, dans un premier temps, un script en *bash*[15] permettant de déployer l'ensemble de la co-simulation DACCOSIM sur une telle architecture. Ce premier script a ensuite été converti en *Python* et enrichi par d'autres membres de l'équipe pour devenir l'outil « DacRun ».
DacRun est un outil qui facilite la compilation et l'exécution d'un ensemble de code de co-simulation sur une architecture distribuée. Il permet notamment :

— de lancer la compilation puis l'exécution des différents fichiers Java, et sauvegarder ensuite les résultats de chaque FMU.

— de contrôler l'ensemble de la co-simulation, et évaluer les performances de chaque local master.

— de déboguer toutes les opérations exécutées sur un nœud, en générant des fichiers d'erreurs et des fichiers traces pour localiser les problèmes d'exécution.

— d'agréger les résultats numériques produits par chaque FMU du système pour construire un unique fichier csv par nœud de calcul.

— d'afficher les performances de la co-simulation globale.

De plus, DacRun est compatible avec deux environnements classiques de gestion de cluster : Oar [33] disponible sur les machines de CentraleSupélec, et Slurm [72] utilisé à EDF.

5.1 Clusters d'expérimentation

Nous avons testé un ensemble de *use-cases* sur des clusters de calcul, qui comportent des noeuds homogènes reliés par un réseau de communication standard ou *Infiniband*[16]. Ces environnements sur lesquels nous avons réalisé nos expérimentations sont les suivants :

1. **Cluster *Cameron*** : un cluster qui comporte 17 nœuds dont chacun contient un Héxa-coeurs type *Sandy-bridge hyper-threaded*[17], avec l'utilisation d'un réseau Ethernet de 10 Gbit/s. Ce cluster possède une architecture UMA (*Uniform Memory Access*).

2. **Cluster *Skynet*** : un cluster qui contient 17 PC type *Nehalem* quadri-coeurs physiques *hyper-threaded*, avec un réseau d'interconnexion Ethernet 1 Gbit/s.

15. *Bourne-Again shell.*
16. Un cluster de PC performant qui néglige les communications évoquées en inter-noeuds (des communications qui traversent le réseau d'interconnexion).
17. Il crée deux cœurs logiques d'un seul cœur physique sur un seul processeur.

3. **Cluster *Sarah*** : un cluster qui comporte 32 nœuds dont chaque nœud contient deux processeurs, et chaque processeur a 4 cœurs physiques *hyper-threaded*. Ces nœuds se basent sur des architectures NUMA (*Non Uniform Memory Access*).

4. **Cluster *Porthos*** : un cluster à haute performance *Infiniband* géré par EDF. La figure 20 illustre l'architecture d'un nœud de type *Porthos* qui est un bi-processeurs, chaque processeur comportant 14 cœurs physiques *hyper-threaded* avec l'architecture NUMA. Le réseau de communication *infiniband* présente une bande passante élevée et une latence fixe, faible et constante entre deux nœuds quelconques du cluster (voir le chapitre 6).

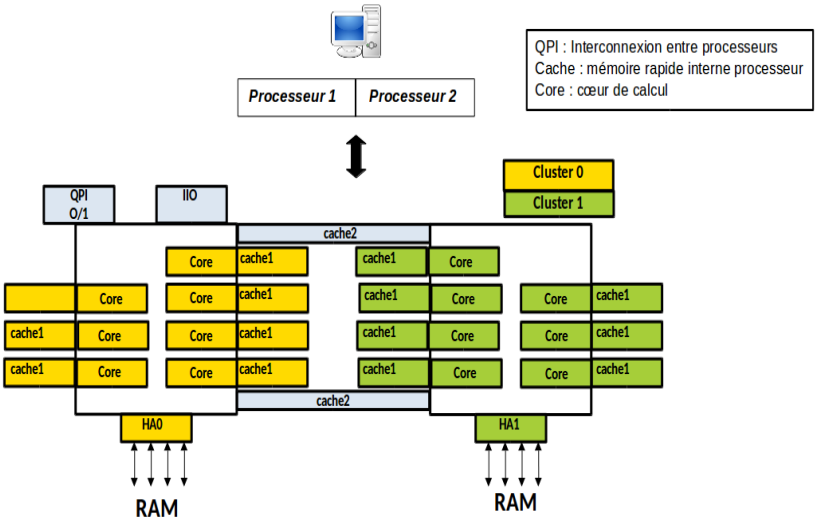


FIGURE 20 – Architecture d'un noeud Porthos
(d'après une présentation interne à EDF R&D)

5.2 Structure et utilisation de DacRun

Après avoir obtenu un graphe de FMU dans DACCOSIM (en général avec un script DSL), on peut lancer l'algorithme de répartition pour affecter ces FMU à des noeuds virtuels appelés « vnode » (les résultats des algorithmes/stratégies de placement seront présentés dans le chapitre 6). Le résultat obtenu par cet algorithme est un fichier de *mapping* qui permet d'associer chaque FMU (programme Java [18] généré par DACCOSIM) à un noeud virtuel (vnode). Ensuite, DACCOSIM récupère à la fois ce fichier de *mapping* et la liste des machines physiques allouées (par le gestionnaire du cluster Oar ou Slurm) appelée « pnode ». À partir de ces deux fichiers, DacRun attribue chaque « vnode » à un « pnode ».

Au final, on peut exprimer le fonctionnement de DacRun par l'association des noeuds

18. celui du master local et les *wrappers* situés sur le même noeud.

virtuels (vnode) à des noeuds physiques (pnode).

Afin de déployer une co-simulation DACCOSIM distribuée depuis un terminal Linux, l'utilisateur peut utiliser des options illustrées dans le tableau 3.2, afin de lancer le déploiement sur un cluster (standard ou performant) géré par Oar[33] ou Slurm[72].
Voici ci-après un exemple d'utilisation de DacRun :

<Path># **dacrun.py** -p . **-vl** chemin_pour_sauvegarder_Logs_FMU/ **-bl** chemin_pour_sauvegarder_process/ **-br** chemin_pour_sauvegarder_resultats/ **-fp** chemin_pour_récuperer_FMU/ **-cm** oar –trace **-m** Constant **-om** *Overlapped* **-iss** 40 **-sd** 86400 **-V** 4

Dans l'exemple ci-dessus, nous avons lancé une multi-simulation distribuée d'une journée avec un pas de temps constant égale à 40 s, sur un cluster de PC multi-coeurs dont le gestionnaire est un *Oar*, avec le mode d'orchestration *Overlapped*.

Option	**Définition**
-m[*]	méthode de pas de temps : constante ou variable.
-e[*]	permet l'extrapolation des résultats (si Extrapolation alors 1, sinon 0).
-om[*]	mode d'orchestration (Ordered ou Overlapped).
-sd[*]	durée de la co-simulation.
-bl	sauvegarder les fichiers trace de chaque FMU.
-br	sauvegarder les résultats numériques de chaque FMU.
-vl	sauvegarder les résultats de performances de chaque FMU.
-cm[*]	préciser le gestionnaire du Cluster (« Slurm » ou « Oar »).
-minss	imposer le pas de temps minimal si la méthode utilisée est variable.
-maxss	imposer le pas de temps maximal si la méthode utilisée est variable.
-iss[*]	initialisation du pas de temps.
-V	Verbose ; affiche les informations du déploiement.
-h	Affiche l'aide sur l'appel du script

TABLE 3.2 – Les options du déployeur DacRun

Remarque :
* : des options et des arguments obligatoires.

6 Limitations actuelles de DACCOSIM

Malgré tous les avantages que procure cette plate-forme, DACCOSIM présente un certain nombre de limitations que nous allons résumer succinctement dans ce qui suit.

— **Détection d'un événement imprévu** : Comme indiqué précédemment, DACCOSIM traite les événements avec un léger décalage même en employant le *rollback* et une méthode dichotomique. Autrement dit, la version actuelle de DACCOSIM ne permet pas la détection instantanée de ces événements. Cette limitation étant dûe à des insuffisances de la norme FMI, nous avons proposé des améliorations de l'API JavaFMI et la norme FMI dans le chapitre 4, pour qu'elle puisse détecter et traiter les événements dès leur apparition.

— **Performances** : En cas d'utilisation de pas de temps variable, DACCOSIM peut faire beaucoup de *rollbacks* ce qui amoindrit et impacte négativement les performances de la co-simulation globale. Pour remédier à ce problème, nous avons proposé des évolutions des méthodes à pas variables.

— **Couplage** : DACCOSIM ne traite que le couplage fort et fréquent où les FMU ont un pas temps commun, en utilisant deux méthodes de pas de temps : constantes et auto-adaptatives. Toutefois, il serait peut-être envisageable de prendre en compte le couplage où le pas de temps est différencié entre toutes les FMU du système et proposer des mécanismes de *rollback* supportés par ce type de couplage .

7 Améliorations apportées dans DACCOSIM

D'une manière générale, les performances sont strictement liées à la nature des calculs effectués dans les composants du *Smart Grid*. Dans notre cas, nous nous concentrons sur l'analyse des tâches les plus gourmandes en calcul, c'est-à-dire les composants physiques (qui intègrent des équations différentielles) puisque les composants discrets ont un poids de calcul quasi-négligeable.

La présente thèse avait notamment pour objectif de se focaliser sur le placement des tâches hétérogènes physiques sur un cluster de PC multi-cœurs. Notre projet intervient donc au niveau de l'étape 2 présentée dans la section 4.1. De manière générale, cette étape permet de placer les différentes FMU sur un ensemble de nœuds virtuels (vnodes) afin de les déployer sur un cluster contenant un ensemble de machines physiques multi-cœurs (pnodes). Le but ultime est de trouver une solution automatique et rapide permettant de mieux répartir les FMU sur des PC multi-cœurs, afin d'obtenir une co-simulation réalisable dans un temps raisonnable (voir chapitre 5).

Un autre objectif est d'augmenter les performances d'un *Smart Grid* en limitant les *rollbacks* en cas de pas adaptatif, et en proposant de traiter précisément les événements de type contrôle-commandes. Ce travail porte sur une analyse de la notion d'événements dans le contrôle-commande, sur des évolutions de la norme FMI et donc à terme sur la

plate-forme DACCOSIM.

8 Conclusion

Ce chapitre a été axé sur le logiciel DACCOSIM et son utilisation. L'évolution qu'a connu cette plate-forme durant ces trois années de thèse a permis l'apparition de nouvelles fonctionnalités et options, qui offrent à cet outil un avantage concurrentiel par rapport aux autres plate-formes de co-simulation. DACCOSIM offre la possibilité de co-simuler des systèmes complexes à très grande échelle.

Le chapitre 5 sera consacré aux placements automatiques de ces FMU hétérogènes sur une architecture distribuée multi-coeurs en utilisant trois approches différentes, ce qui représente l'un des principaux défis de la co-simulation distribuée. Le choix entre ces approches sera directement lié à la taille du problème à traiter.

Deuxième partie

Contribution à un environnement de Co-Simulation distribuée pour *Smart Grids*

CHAPITRE 4

MODÈLE D'INTÉGRATION ET DE SYNCHRONISATION D'UN SYSTÈME COMPLEXE

Sommaire

1 Introduction

Les *Smart Grids* représentent un exemple concret de systèmes cyber-physiques, et un enjeu industriel pour EDF et sa filiale ENEDIS [1]. La co-simulation de ce type de systèmes fait appel à la fois à des simulateurs évènementiels pour simuler les réseaux de télécom et les systèmes d'informations, et à des simulateurs à temps continu discrétisé pour simuler les parties physiques (électromagnétisme, transfert thermique, processus industriels...). Ces simulateurs sont spécifiques à leurs domaines d'applications métiers, et doivent s'échanger des données par l'intermédiaire d'un intergiciel [2] qui contrôle et supervise l'ensemble des composants. Dans cette thèse, nous traitons deux types d'événements : le premier dit *time event* qui sera déclenché par des composants événementiels et envoyé vers les composants continus, et le second dit *state event* qui sera déclenché par les composants physiques et envoyé vers les composants événementiels. Pour traduire l'hétérogénéité d'un système complexe mélangeant ces deux types de données (événements et variables d'états continues), le qualificatif de co-simulation hybride est souvent employé. Dans la première partie de ce chapitre, nous proposons un modèle hybride qui n'a pas la même vision du temps et des événements que les modèles classiques, un modèle qui (1) traite précisément les événements apparus durant une co-simulation donnée, et (2) prend en compte la précision et la qualité des résultats des composants physiques en utilisant des pas de temps auto-adaptatifs. La seconde partie s'oriente vers une amélioration de la norme FMI, dans le but d'obtenir des co-simulations plus précises en prenant en compte les divers types d'événements qu'on peut trouver dans la réalité (par exemple au sein du contrôle-commande), et d'augmenter les performances en réduisant le temps global de la co-simulation.

Nous présentons, à travers ce chapitre, la méthodologie que nous avons suivie pour proposer et concevoir un modèle hybride qui gère et synchronise des événements standards parfaits (de types *Dirac*) avec des pas de temps. Ensuite, nous nous penchons sur le traitement des événements réels de type contrôle-commande en proposant une amélioration du standard FMI 2.0 et de l'API JavaFMI qui l'implante logiciellement.

2 Définitions et approche de couplage

La co-simulation des systèmes complexes hybrides agit sur des modèles de simulation continus en temps discrétisé, et sur d'autres modèles de simulation événementiels. Tous ces composants sont exécutés soit sur la même machine, soit sur des architectures

1. ENEDIS : une entreprise chargée de la gestion du réseau de distribution d'électricité en France.
2. Certains intergiciels sont bâtis autour du concept de bus à événements.

distribuées.

2.1 Notations

Dans cette section, nous définissons quelques concepts de base valables pour une co-simulation quelconque :

— Un **modèle** M_i est défini comme une représentation permettant de reproduire le comportement dynamique d'un système quelconque S_i, qui s'appuie sur son propre modèle de temps. On peut trouver deux types de modèles de temps : un modèle de temps discret ou événementiel nommé Δ, et un modèle de temps continu nommé Φ discrétisé à pas de temps éventuellement variables ; ces derniers sont définis dans le chapitre 1 section 3.3.2.

— Un **point de sauvegarde** (*checkpoint*) est un mécanisme qui permet à un composant d'enregistrer son état à un instant donné (notamment toutes ses variables d'états).

— Un mécanisme de **restauration** (*restore*) est un mécanisme qui permet à un simulateur de récupérer l'ensemble des données nécessaires pour reprendre la simulation à partir de son dernier point de sauvegarde.

— Dans notre cas, nous proposons d'utiliser deux types de *checkpoint/restore* :

1. Un *checkpoint/restore* **local** (*rollback*) qui permet d'enregistrer l'état des variables d'états d'un seul composant afin de le récupérer (ex : améliorer la précision des calculs en diminuant les pas de temps).

2. Un *checkpoint/restore* **global** qui permet de sauvegarder l'état global du système et de le récupérer en totalité après un *rollback* global généré par un *rollback* local ou à l'émergence d'événements dans le système.

   Ce type de *checkpoint/restore* est représenté comme étant un ensemble de *checkpoint/restore* locaux à travers tout le système.

— Un **point de synchronisation** [42] (point de rendez-vous) est une date où les composants de la partie évènementielle Δ, et les dispositifs de la partie continue Φ peuvent échanger leurs résultats. On le notera par $TimeSynch^i$ à l'itération i.

— Un ***Time event*** est un évènement émis par un simulateur de la partie événementiel Δ à destination de la partie continue Φ, il est qualifié comme un évènement descendant. Notons que dans notre cas, on ne traite pas les *time events* qui sont à destination d'autres composants événementiels (traitement interne à la partie événementielle).

Soit e_i un évènement de type *time event* (exemple : un passage de commande), on le représente par le formalisme suivant :
$e_i \longmapsto <val, port, TimeStamp>$ tel que : le « *port* » représente une des sorties du système discret avec $port \in Outputs$ (l'ensemble des sorties), « *val* » représente la valeur modifiée d'une variable de sortie (exemple : $val \in \mathbb{R}$ si la variable est réelle), et le « *TimeStamp* » est la date d'apparition de ce *time event*.

— Un ***State event*** est un événement émis par un simulateur de la partie continue Φ à destination de la partie événementielle (exemple : une alarme), il est caractérisé comme un événement montant vers Δ. Un *state event* est représenté par le quadruplet suivant :
$se_i \longmapsto <val, port, TimeStamp, urgence>$ tel que : « *val* » représente la valeur modifiée d'une variable de sortie, le $port \in Outputs$ (l'ensemble des sorties), le « *TimeStamp* » est la date d'apparition de l'evènement détecté au mieux par le solveur utilisé, et « *urgence* » représente une variable booléenne (un *flag*) qui indique l'importance de ce *state event* (il est égal à 1 si l'événement est urgent, sinon 0). Le temps d'envoi t_{event}^{env} d'un *state event* est défini comme suit :

$$t_{event}^{env} = \begin{cases} TimeStamp & \textbf{si} \quad urgence = 1 \\ TimeSynch & \textbf{sinon} \quad (\textbf{si } urgence = 0) \end{cases} \tag{4.1}$$

Le *State event* sera envoyé et traité selon les deux cas illustrés dans le système d'équation 4.1 ci-dessus :

— Si cet événement est urgent ($urgence = 1$), il faut l'envoyer tout de suite, à sa date d'apparition ($TimeStamp$) afin d'assurer sa diffusion vers les autres composants du système. Autrement dit, le traitement de l'événement ne peut pas être différé même si cela perturbe l'exécution du pas de temps en cours.

— Si cet événement n'est pas important, il suffit de l'envoyer au premier point de synchronisation ($TimeSynch$). Dans certains cas, il est même possible de le traiter encore plus tard par exemple à la fin de la co-simulation (exemple : l'écriture de *logs*).

2.2 Approche de couplage

La cohabitation de pas de temps et d'événements horodatés dans un même co-simulateur de *Smart Grid* rend ce type de système non classique (voir la figure 1). Ce système hybride se conçoit naturellement autour d'un bus de co-simulation qui se base sur un modèle de temps événementiel : un bus qui assure les communications et synchro-

nise les données échangées entre les composants hétérogènes du système. Toutefois, ce bus devra aussi s'interfacer avec des composants évoluant selon des pas de temps.

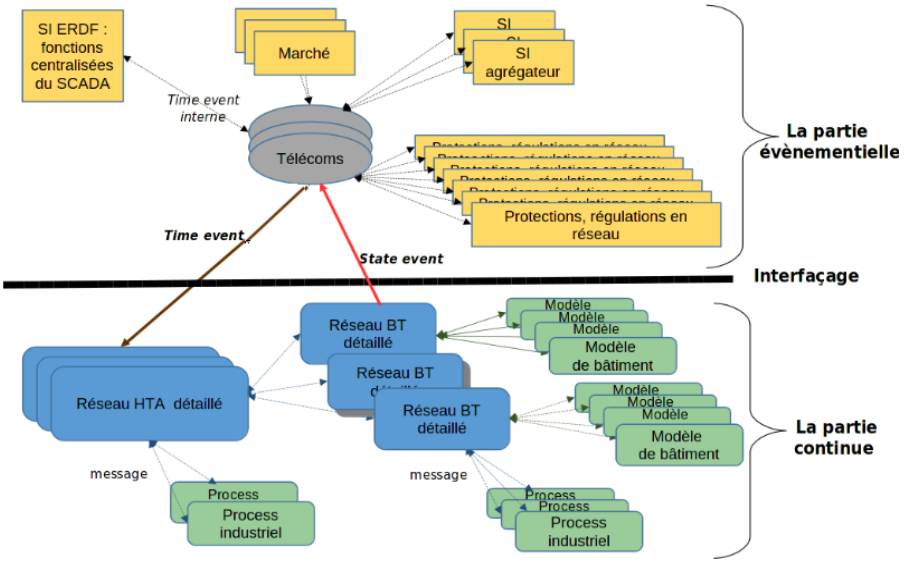


FIGURE 1 – Exemple d'un modèle de *Smart Grid* proposé par EDF R&D

Dans la littérature, nous avons trouvé des modèles de temps hybrides qui s'appuient généralement sur une démarche classique [2]. Une telle démarche consiste à interfacer individuellement chaque simulateur d'un domaine spécifique sur un bus de co-simulation qui gère entièrement la progression des calculs et le traitement de ces derniers. Contrairement à cette approche classique, notre démarche consiste à interfacer (ou accrocher) une grappe[3] de composants physiques sur un bus à événements, comme l'indique la figure 2. Cette combinaison permettra (1) de garantir le parallélisme d'un coté en exécutant tous les composants physiques (de la grappe) simultanément, (2) de permettre aux composants physiques de revenir en arrière (*rollback*) en cas de problème, tout en gardant une certaine transparence vis-à-vis de la partie événementielle, et (3) de synchroniser les données entre ces deux parties. Concrètement, les composants événementiels (télécoms et SI) communiquent avec les composants physiques (FMU) en envoyant des *time events* ($\Delta \longmapsto \Phi$), et les composants physiques génèrent des *state events* (exemple : un franchissement de seuil, une alarme) qu'ils envoient vers les composants événementiels ($\Phi \longmapsto \Delta$).

À cet effet, nous considérons un système complexe comme étant un système global qui comporte deux grandes parties : la partie discrète Δ qui regroupe tous les composants événementiels (SI et Télécom), et la partie continue Φ qui agrège tous les composants modélisant des dispositifs physiques dans une grappe de FMU physiques (voir la figure 2).

3. une grappe contient plusieurs composants physiques fortement couplés et interconnectés entre eux.

Ces deux parties sont connectées et interfacées entre elles via un bus de co-simulation, ce dernier traite les requêtes en respectant l'ordre chronologique d'arrivée des événements. Initialement, nous avons prévu de simuler la partie continue avec la plate-forme DACCOSIM, et la partie événementielle (aussi que le bus d'événements) avec un environnement comme HLA [4] ou MECSYCO [5] s'appuyant sur une méthode conservative.

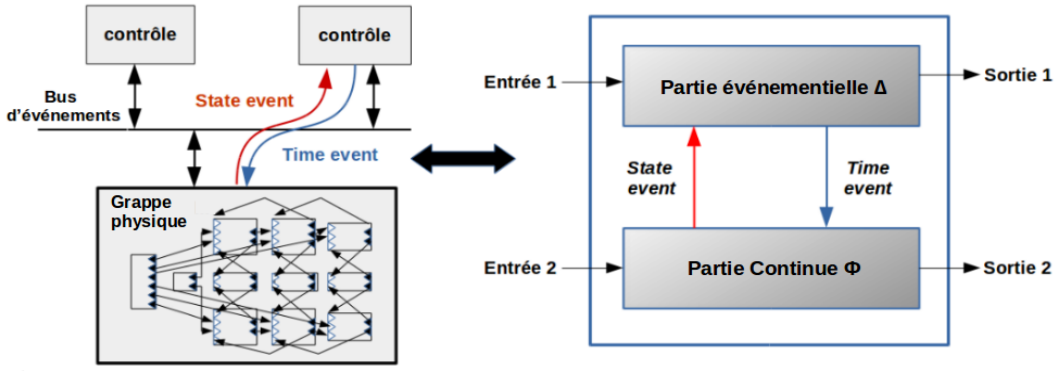


FIGURE 2 – Notre approche de couplage

Cette approche garantit un compromis entre précision et performance : *précision* car le master de la grappe physique peut toujours s'intéresser à la qualité des résultats en minimisant l'erreur produite dans les calcul par le mécanisme de *rollback* et router les événements urgents vers les composants concernés dès leur apparition, et *performance* car elle exécute les différents composants de manière simultanée quelle que soit la taille de la grappe en réduisant le temps global de la simulation.

2.3 Hypothèses sur nos plate-formes à coupler

Dans le but de définir un modèle de temps efficace et réaliste, les contraintes d'une co-simulation citées dans le chapitre 1 et les contraintes imposées par nos deux plate-formes nous amènent à poser quelques hypothèses que nous pouvons résumer comme suit :

1. La partie événementielle Δ est considérée très rapide en temps de calcul par rapport à la partie continue Φ qui, à l'inverse, réclame souvent des calculs importants notamment pour intégrer les équations différentielles.
2. Il existe un mécanisme de détection des *state events* intégré dans chaque FMU, qui permet de détecter aussi leur date d'apparition.
3. La partie continue envoie des *states events* à la partie événementielle (exemple :

4. *High Level Architecture.*
5. http ://www.mecsyco.com/fr/.

des alarmes), qui va lui envoyer à son tour un ou plusieurs *time events* (exemple : des commandes).

4. Nous accrochons une grappe de FMU physique sur un bus d'événements qui s'appuie sur une méthode conservative (cas le plus fréquent des plate-formes événementielles).

5. Les composants de la partie continue peuvent s'exécuter en parallèle de manière distribuée. Les solveurs de ces dispositifs peuvent utiliser soit des méthodes à pas constants, soit des méthodes auto-adaptatives à pas variables. On verra plus loin qu'en cas de pas de temps adaptatif, seule la partie continue Φ aura besoin de faire des *rollbacks*.

6. Le comportement de nos FMU est reproductible ; il n'y a pas de génération aléatoire de *state events* non-déterministes. En cas de *rollback*, on retrouvera bien les mêmes *state events* aux mêmes instants de la simulation.

Afin de proposer un modèle hybride qui prend en compte les contraintes abordées auparavant, nous proposons d'améliorer les modèles classiques à base de bus d'événements en leur ajoutant deux sous-couches logicielles, ces dernières vont être détaillées dans la suite de ce chapitre.

3 Scénarios réalistes de co-simulation d'un *Smart Grid*

Nous avons défini et construit des scénarios de co-simulations réalistes, adéquats à des cas d'utilisations d'un vrai *Smart Grid*, en accord avec le contexte de collaboration industrielle de cette thèse dans l'institut *RISEGrid*. Pour cela, nous avons pu résumer les principaux cas possibles du fonctionnement d'un *Smart Grid* en deux scénarios illustrés dans la figure 3 :

1. Le premier scénario comprend des *time events* envoyés par la partie événementielle Δ (voir la figure 3-a) à destination de la partie continue Φ. On peut prendre comme exemple la simulation d'un réseau électrique avec des événements de type contrôle-commande (exemple : ouverture/fermeture de disjoncteurs).

2. Le second scénario englobe les deux types d'évènements : *time event* et *state event* (voir la figure 3-b). On prend comme exemple la simulation d'un réseau électrique avec des événements de type contrôle-commande et des défauts électriques au niveau de la partie physique (exemple : des défauts monophasés qui sont très fréquents en moyenne tension sur les lignes aériennes).

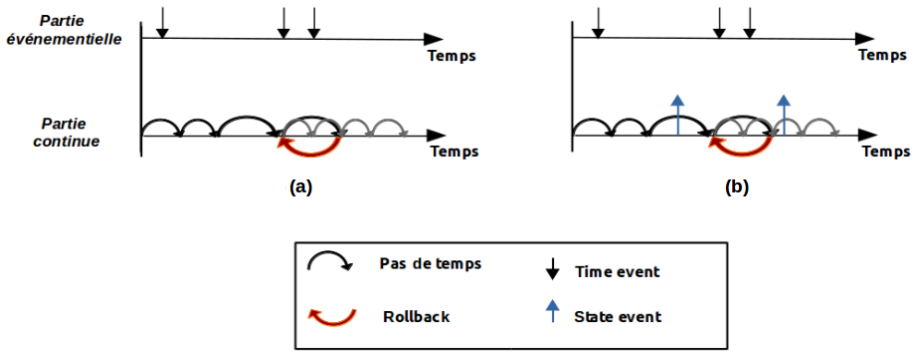


FIGURE 3 – Les différents scénarios retenus pour la co-simulation d'un *Smart Grid*

4 Architecture et fonctionnement d'un co-simulateur hybride classique

Dans un *Smart Grid* réel, les différents dispositifs physiques interagissent et communiquent à tout instant avec d'autres composants physiques ainsi qu'avec des unités de contrôle du même système. Afin de modéliser au mieux ces différents échanges, il est important de définir un modèle global de temps qui prend en considération les critères du modèle événementiel (basé sur des événements), et du modèle continu (discrétisé en pas de temps). Dans cette section, nous définissons et décrivons un modèle hybride générique dédié à la simulation des systèmes complexes : un modèle de temps qui rend la partie continue interopérable avec la partie événementielle. En effet, ce modèle hybride prend en compte une méthode optimiste qui accepte le mécanisme de *rollback* (partie continue), et une méthode conservative en respectant la contrainte de causalité (partie événementielle), voir le chapitre 1.

Dans cette approche, nous traitons des événements standards parfaits, présentés sous forme d'impulsions de *Dirac* (voir la figure 4), apparus à des instants t_i quelconques ($t_i \in \mathbb{R}$). La plupart des simulateurs événementiels utilisent des modèles discrets en s'appuyant sur ce type de représentation des événements. Un événement est vu comme une requête envoyée à un instant « t_i », avec un temps de transmission et un temps de traitement qui sont nuls, à moins que certains composants modélisent et simulent explicitement des retards et des délais. La figure 4 illustre une séquence chronologique d'événements. Cette représentation modélise en fait l'état du système à n'importe quel instant, car elle matérialise les changements d'états successifs du système.

FIGURE 4 – Séquence d'événements parfaits

4.1 Principe d'un co-simulateur hybride classique

Comme l'illustre la figure 5, le modèle classique d'un co-simulateur hybride comporte trois éléments importants : des composants physiques qui représentent les simulateurs à base de temps continu, des composants événementiels qui sont des unités de contrôle et un bus à événements classique (*middleware*) qui interconnecte les différents composants du système entre eux.

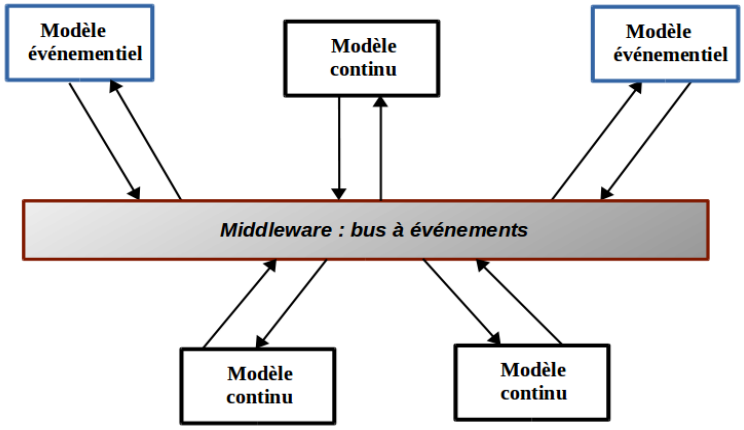


FIGURE 5 – Architecture classique d'un co-simulateur hybride

Propriétés utiles et classiques de ce type de *middleware*

Les principaux objectifs de ce bus de co-simulation sont :

— L'interconnexion des composants hétérogènes entre eux.

— La synchronisation des données entre les composants du système.

— L'envoie des événements entre les composants.

— La progression du temps global de la co-simulation en définissant une horloge globale unique pour l'ensemble du système.

En revanche, ces bus à événements adoptent très souvent des méthodes conservatives permettant d'avancer dans le temps sans revenir en arrière. C'est le cas de certaines implantations du bus RTI de la norme HLA, et du bus de la plate-forme MECSYCO [16, 23] (initialement choisie par EDF pour ce projet). Mais dans notre cas, on doit interfacer

ce type de bus avec une grappe de FMU. Pour que cette grappe physique soit efficace, elle doit pouvoir effectuer des *rollbacks*, c'est-à-dire imposer à ses FMU de refaire leurs calculs avec des pas plus petits en revenant en arrière.

4.2 Notre modèle de co-simulation hybride classique

Dans cette section, nous proposons un modèle hybride de co-simulation qui reprend les principes des modèles classiques de la section précédente, et les adapte à notre approche de couplage en deux parties (événementielle Δ et continue Φ) présentée dans la section 2.2. Le grand défi de ce modèle est de synchroniser les données envoyées entre des composants hétérogènes, en traitant à la fois des données numériques échangées entre des composants physiques, et des événements descendants (*time events*) et ascendants (*state events*) échangés entre les deux parties. De plus, les traitements de ces événements peuvent être urgents ou différés (cas de signaux d'alarme devant remonter rapidement jusqu'aux unités de contrôle). Pour répondre à ces besoins, nous concevons un modèle hybride en ajoutant deux sous-couches logicielles à l'architecture classique : l'une au niveau de la partie continue et l'autre au niveau du *middleware* (bus d'événements).

4.2.1 Le besoin d'explorer le futur des FMU

Comme indiqué précédemment, on ne peut ni modifier ni arrêter l'exécution d'un pas de calcul géré par le solveur numérique d'une FMU, c'est-à-dire qu'aucune intervention ne peut être effectuée sur ces FMU entre deux points de synchronisation. Cette contrainte nous impose de réaliser une pré-phase qui explore le futur de la grappe physique dans le but de faire coïncider au mieux les points de synchronisation avec les événements qui apparaissent pendant le pas de temps. Nous avons travaillé initialement avec les auteurs de l'article [15], et partageons leur conclusion qui impose aux différentes FMU d'explorer leur futur afin de découvrir tous les *state events* jusqu'au prochain point de synchronisation $TimeSynch^{i+1}$ de l'itération $i+1$. Mais, leur travaux se sont concentrés sur l'implantation d'une solution dans la plate-forme MECSYCO, alors que nous avons préféré en concertation avec EDF concevoir une solution capable d'adapter DACCOSIM à une grande variété de modèles événementiels.

4.2.2 Architecture générale de notre co-simulateur hybride classique

L'objectif de notre modèle est d'assurer une synchronisation des données entre les composants physiques et les composants de la partie événementielle, en prenant en compte à la fois le traitement des *state events* et des *time events*. Néanmoins, dans notre cas, nous sommes contraints par deux plate-formes de simulation : (1) une plate-

forme qui traite des modèles de simulations dont les solveurs utilisent des pas de temps constants ou bien des pas de temps variables avec le mécanisme de *rollback* en cas de besoin, et (2) une plate-forme événementielle qui utilise des méthodes conservatives dont le modèle de temps permet juste d'avancer dans le temps (pas de *rollback* possible). Nous avons gardé l'architecture classique établie précédemment mais composée de trois parties seulement :

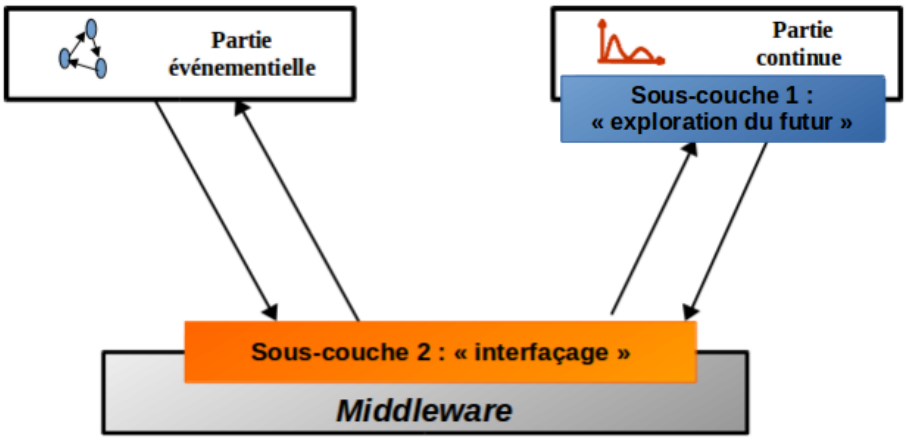


FIGURE 6 – Notre modèle hybride

— Une **partie continue :** la grappe physique, qui contient plusieurs FMU hétérogènes capables d'être exécutées en parallèle, en utilisant soit des pas de temps constants, soit des pas de temps variables avec la possibilité de faire des *rollbacks*. Notons que l'exécution des composants de la grappe est transparente par rapport aux autres composants événementiels.

— Une **partie événementielle** qui représente toutes les unités de contrôle, destinées à superviser le système et à assurer sa fiabilité.

— Un ***middleware*** qui supervise l'ensemble des données échangées entre ces deux parties (voir la sous section 4.1).

Comparé au modèle basique d'un co-simulateur hybride, notre modèle greffe deux sous-couches autour du *middleware*. La première sous-couche permet à la grappe physique (des FMU) d'explorer le futur afin de détecter les événements montants (*state events*) vers la partie événementielle tout en préservant une certaine transparence vis-à-vis du *middleware* qui s'appuie sur une méthode conservative. La deuxième sous-couche interface la grappe physique à cette architecture, et se charge des envois d'événements descendants de la partie événementielle avec le *middleware*. Ces deux sous-couches seront définies en détail par la suite.

Notations et fonctions

Afin de prendre en compte les contraintes des deux parties Φ et Δ, nous avons rajouté quelques notations et fonctions qui sont :

- un pas de temps constant ou variable à l'itération i, noté h_i.
- un prochain point de synchronisation proposé par la partie continue, noté $TimeSyn_{Cont}^{i+1}$,
- une date de prochaine synchronisation de la partie événementielle calculée par le *middleware*, notée $TimeSyn_{Disc}^{i+1}$.
- une liste L qui contient l'ensemble des *state events* survenus entre deux points de synchronisations $TimeSynch^{i}$ à l'itération i et $TimeSynch^{i+1}$ à l'itération $i+1$.
- une durée globale de la co-simulation notée Δd définie par $\Delta d = t_{fin}$ - $t_{début}$ tel que $t_{début}$ est l'instant de début et t_{fin} est l'instant de fin de la co-simulation globale.
- la fonction *globalCheckpoint()*, qui permet à la partie continue (Φ) de sauvegarder l'état global du système.
- la fonction *globalRestore()*, qui permet à la partie continue (Φ) de retourner à son état précédant en récupérant les valeurs de ces variables d'état.
- la fonction *doStep*(h_i), qui permet de lancer les calculs d'une FMU pendant le pas de temps h_i.
- la fonction *getDiscretEventList*(), qui permet de récupérer la liste des *time events* de la partie événementielle.
- la fonction *getNextDiscretEvent*(), qui permet de récupérer le *TimeStamp* du prochain *time event* connu de la partie événementielle (Δ).
- la fonction *getCurrentTime*(), qui permet d'obtenir l'instant actuel de la co-simulation.
- la fonction *detectionStateEvent*(), qui permet de détecter les *state events* apparus pendant un pas de temps. Cette fonction retourne une liste des *state events* s'il y a des événements, ou une liste vide.
- la fonction *hasVarStateChange*(), qui retourne un $True$ si elle détecte un changement d'état dans une des variables discrètes, sinon un $False$.

4.2.3 Fonctionnement de notre modèle hybride classique

La synchronisation des deux parties Δ et Φ, et de leurs modèles de temps, est réalisée par à un *middleware*, désignant en fait un bus de co-simulation. Ce *middleware* permet de gérer les évènements selon la date de leur arrivée, et d'échanger les données périodiquement entre ces deux parties. L'idée principale de ce modèle est donc de donner

la possibilité aux deux parties (Δ et Φ) de proposer un point de synchronisation qui est corrélé à la date d'apparition du prochain événement. Pour se faire, le *middleware* commence par communiquer aux différentes parties du système le prochain point de synchronisation qui se situe à un pas de temps du précédent ($TimeSynch^{i+1}$ par défaut). Ensuite, chacune des deux parties propose à son tour un nouveau point de synchronisation qui est différent de celui du *middleware* en cas d'apparition d'événements, soit en passant par une exploration du futur (pour la partie continue), soit en interrogeant la liste des événements prévus (pour la partie événementielle). La partie continue et la partie événementielle propagent leurs propositions auprès du *middleware*, qui va calculer à son tour la valeur minimale de toutes les propositions. Nous avons choisi la valeur minimale des toutes les propositions car elle permet (1) de garantir le traitement de tous les événements à temps de manière chronologique, et (2) d'éviter l'accumulation des événements et leur traitement tardif (comme dans le cas de la plate-forme EPOCHS [41]).

La figure 7 représente une vue globale de notre modèle hybride, et elle liste les différents points à réaliser pour appliquer notre proposition sur un système complexe respectant les hypothèses présentées dans la section 2.3.

Dans cette section, nous définissons les principales étapes illustrées dans la figure 7 : la phase d'initialisation pour initialiser toutes les données, la phase de préparation qui aide les deux parties à proposer des points de synchronisation et la phase de co-simulation qui déclenche la simulation globale du système pour un pas de temps donné. Le but principal de notre modèle est de calculer le prochain point de synchronisation : un point de rendez-vous où les deux parties peuvent échanger les données entre elles tout en traitant les différents événements apparus entre les deux points de synchronisation initialement prévus.

- (0) **Phase d'initialisation** :
  - Définir la durée de co-simulation Δd (spécifiée par l'utilisateur).
  - Initialiser le point de synchronisation courant et le temps actuel à zéro : $TimeSynch^i = T_{current} = 0$.
  - Initialiser le compteur de pas de temps (itération) à zéro : $i = 0$.
  - Initialiser le pas de temps h_i (pas de temps constant ou auto-adaptatif), la valeur initiale de ce paramètre étant spécifiée par l'utilisateur.

- (1) **Phase de préparation** : avant de lancer la co-simulation
  - Initialiser le nouveau point de synchronisation par défaut à $TimeSynch^{i+1} = T_{current} + h_i$.

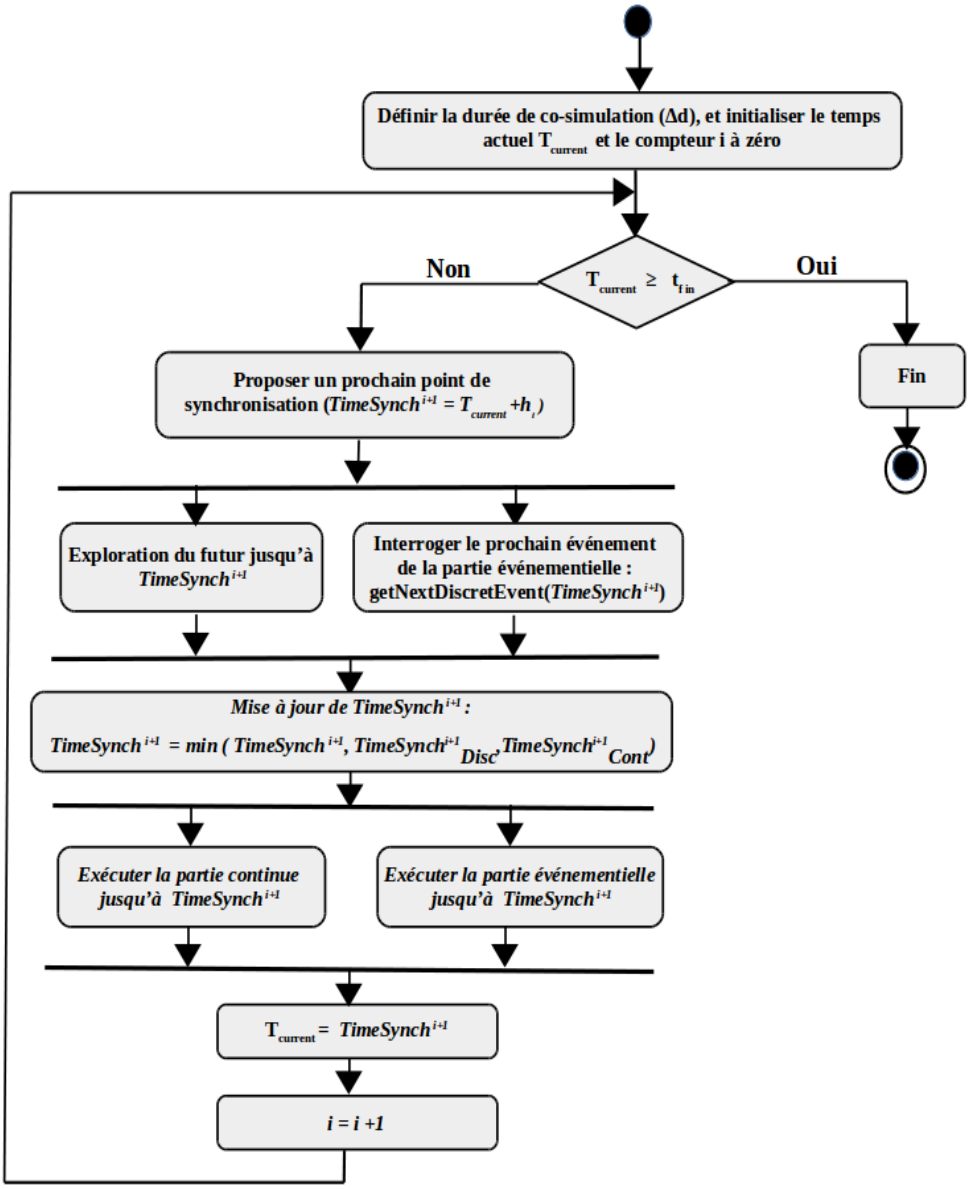


FIGURE 7 – Diagramme d'activité de notre modèle hybride classique

— Appeler la fonction *GetNextDiscretEvent()* pour déterminer $TimeSyn_{Disc}^{i+1}$ proposé par la partie événementielle :
$TimeSyn_{Disc}^{i+1} = GetNextDiscretEvent(TimeSynch^{i+1})$.

— Explorer la partie continue jusqu'à $TimeSynch^{i+1}$ en appelant la fonction $ExplorateContPart()$ afin de définir $TimeSyn_{Cont}^{i+1}$:
$TimeSyn_{Cont}^{i+1} = ExplorateContPart(TimeSynch^{i+1}, h_{min}^{i})$.

— Mettre à jour le prochain point de synchronisation $TimeSynch^{i+1}$ par appel de la fonction $NewTimeSynch()$ pour définir le prochain point de synchronisation :

$$TimeSynch^{i+1} = NewTimeSynch(TimeSynch^{i+1}, TimeSyn_{Cont}^{i+1}, TimeSyn_{Disc}^{i+1}).$$

— (2) **Phase de co-simulation** :

  — Lancer la co-simulation jusqu'au nouveau $TimeSynch^{i+1}$ en exécutant la partie événementielle et la partie continue en parallèle.

  — Mettre à jour le temps courant $T_{current}$ à $TimeSynch^{i+1}$ et le compteur $i = i + 1$ pour passer à la prochaine itération.

  — Revenir à la phase de préparation (1).

Dans les sections qui suivent, nous présentons de manière détaillée les deux sous-couches à savoir le traitement effectué dans la grappe physique et celui réalisé par le *middleware* pour la co-simulation.

4.2.4 Fonctionnement de la sous-couche 1 « Exploration du futur »

Puisque les composants de la partie continue ne peuvent pas connaître à l'avance leurs *state events*, on est obligé de passer par une solution qui permet à chaque FMU d'explorer son futur afin de connaître l'ensemble des *state events* qui apparaîtront dans le prochain pas de temps. Cette exploration doit être transparente et invisible vis-à-vis des autres éléments du système car elle ne respecte pas la contrainte de causalité, et que la partie événementielle ne peut en principe pas explorer son futur et faire des *rollback*.

Afin d'explorer le futur de la partie continue, le master global de la grappe physique enregistre l'état actuel de tout le sous-système en utilisant le mécanisme de *checkpoint* global (fonction *globalCheckpoint()*). Ensuite, chaque FMU exécute sa phase de calcul jusqu'au bout du pas de temps, et vérifie ses sorties discrètes (booléennes, entières ou réelles) dont les changements de valeur représentent des *state events* (exemple : une variable booléenne correspondante à un seuil d'alarme : b = $(Var > 50)$). Si au moins une variable discrète a changé d'état, le master global de la co-simulation oblige la grappe continue à retourner à son état précédent (au début du pas de temps) en utilisant le mécanisme de restauration globale (*globalRestore()*). Autrement dit, le master global oblige la FMU à ré-exécuter ses calculs depuis le début du pas de temps, en appelant sa primitive (*doStep()*) soit avec un tout petit pas de temps défini par l'utilisateur (h_{min}^{i}), soit avec une méthode dichotomique [15], afin de trouver la position à peu près exacte du *state event* apparu. Cet algorithme d'exploration du futur retourne le $TimeStamp$ du premier *state event* qui exige un traitement urgent ($urgence = 1$) s'il y en a un. A la fin de cette recherche d'un *state event* urgent, le master global oblige la grappe continue à revenir au début du pas de temps initial par un (*globalRestore()*).
Rappelons que dans notre cas les *state events* sont considérées comme déterministes.

L'algorithme 1 représente une exploration du futur en utilisant une méthode d'intégration avec un petit pas de temps.

Algorithm 1 ExploreContPart($TimeSynch^{i+1}$,h^i_{min}) $\rightarrow$ $TimeSyn^{i+1}_{Cont}$

Paramètre d'entrée *: $TimeSynch^{i+1}$ représente le prochain point de synchronisation suggéré par défaut*
Paramètre d'entrée *: h^i_{min}, le pas de temps (le plus petit) défini par l'utilisateur*
Valeur retournée *: $TimeSyn^{i+1}_{Cont}$, la proposition de la partie continue*

Type *: $StateEvent$ = <val, port, $TimeStamp$, urgence>*
Variables locales :
entier $j \leftarrow 0$
Double $Time_{current} \leftarrow getCurrentTime()$
List<$StateEvent$> L
Boolean $UrgentDetection \leftarrow False$

Instructions :
$TimeSyn^{i+1}_{Cont} \leftarrow TimeSynch^{i+1}$
globalCheckpoint()
doStep($TimeSynch^{i+1}$)
if ($hasVarStateChange() == True$) **then**
  globalRestore() // on suppose que cette primitive ne détruit pas les
                    // sauvegardes effectuées par*globalCheckpoint()*
  while (($Time_{current} \leqslant TimeSynch^{i+1}$) && ($UrgentDetection == False$)) **do**
    doStep($Time_{current} + h^i_{min}$)
    if ($hasVarStateChange() == True$) **then**
      $L[k] \leftarrow detectionStateEvent()$
      if ($L[k].urgence == 1$) **then**
        $UrgentDetection \leftarrow True$
        $TimeSyn^{i+1}_{Cont} \leftarrow L[k].TimeStamp$
        $k \leftarrow k + 1$
      end if
    end if
    $Time_{current} \leftarrow getCurrentTime()$
  end while
else
  $TimeSyn^{i+1}_{Cont} \leftarrow TimeSynch^{i+1}$
end if
globalRestore()
Return $TimeSyn^{i+1}_{Cont}$

4.2.5 Fonctionnement de la sous-couche 2 « Interfaçage »

Cette sous-couche permet à la fois de calculer le besoin de synchronisation de la partie événementielle et de synchroniser les deux parties de la co-simulation globale.

— **Accès à la liste des *time events***

La simulation de la partie événementielle se base sur un modèle discret, qui classe l'ensemble des *time events* dans une liste d'événements estampillés avec un ordre chronologique croissant. Comme l'indique la figure 7, la partie événementielle interroge sa liste d'événements avec la fonction $getNextDiscretEvent()$ afin de déterminer le $TimeSyn_{Disc}^{i+1}$. Ce temps représente le temps de synchronisation proposé par la partie événementielle. On suppose que les *time events* de la partie événementielle sont déterministes.

Algorithm 2 GetNextDiscretEvent($TimeSynch^{i+1}$) $\rightarrow TimeSyn_{Disc}^{i+1}$

1: ***Paramètre d'entrée*** *: $TimeSynch^{i+1}$ représente le prochain point de synchronisation suggéré par défaut*
2: ***Valeur retournée*** *: $TimeSyn_{Disc}^{i+1}$, la proposition de la partie événementielle*

3: ***Type*** *: $TimeEvent$ = <val, port, TimeStamp>*
4: **Variables locales :**
5: List<$TimeEvent$> L

6: **Instructions :**
7: $L \leftarrow getDiscretEventList()$
8: **if** $(L.size == 0)$ **then**
9: $\quad TimeSyn_{Disc}^{i+1} \leftarrow TimeSynch^{i+1}$
10: **else**
11: $\quad TimeSyn_{Disc}^{i+1} \leftarrow L[0].TimeStamp$
12: **end if**
13: ***Return*** $TimeSyn_{Disc}^{i+1}$;

L'algorithme 2 décrit la fonction GetNextDiscretEvent($TimeSynch^{i+1}$). S'il y a des *time events* définis au préalable entre deux points de synchronisation $TimeSynch^{i}$ et $TimeSynch^{i+1}$, le *middleware* interroge sa liste d'événements pour déterminer un prochain point de synchronisation ($TimeStamp$), sinon, il retourne le point de synchronisation proposé au début.

— **Synchronisation de la co-simulation**

Comme l'illustre le diagramme de la figure 7, le *middleware* est aussi responsable de la synchronisation des données entre les deux parties Δ et Φ. Il récupère, dans un premier temps, les deux dates proposées par les deux parties du système global, puis met à jour le nouveau point de synchronisation en calculant le minimum de ces trois valeurs (par un appel de la fonction $NewTimeSynch()$), voir l'algorithme 3. Ensuite, le *middleware* communique le nouveau point de synchronisation ($TimeSynch^{i+1}$) aux deux parties, et autorise la co-simulation globale de tout le système en exécutant en parallèle les événements de la partie événementielle et la

grappe de FMU jusqu'au point de synchronisation $TimeSynch^{i+1}$.

Algorithm 3 NEWTIMESYNCH($TimeSynch^{i+1}$, $TimeSyn^{i+1}_{Cont}$, $TimeSyn^{i+1}_{Disc}$) $\rightarrow$ $TimeSynch^{i+1}$

1: ***Paramètre d'entrée*** *: $TimeSynch^{i+1}$, la proposition du prochain point de synchronisation par défaut*
2: ***Paramètre d'entrée*** *: $TimeSyn^{i+1}_{Cont}$, la proposition de la partie continue*
3: ***Paramètre d'entrée*** *: $TimeSyn^{i+1}_{Disc}$, la proposition de la partie événementielle*
4: ***Valeur retournée*** *: $TimeSynch^{i+1}$, le nouveau point de synchronisation*

5: **Instructions :**
6: $TimeSynch^{i+1} = min(TimeSyn^{i+1}_{Continu}, TimeSyn^{i+1}_{Disc}, TimeSynch^{i+1})$
7: ***Return*** $TimeSynch^{i+1}$;

Exemple

Comme illustré dans la figure 8, le *middleware* propose par défaut aux deux parties un point de synchronisation à t_{15} (correspondant à une avancée d'un pas de temps). La partie continue du système explore son futur et envoie au *middleware* le *TimeStamp* du *state event* apparu à t_4, qui est inférieur au point de synchronisation proposé. En parallèle, le *middleware* interroge la liste des *time events* pour déterminer le prochain événement de la partie événementielle, représenté par le *time event* apparu à l'instant t_{10}. À la fin, le *middleware* calcule le minimum des trois valeurs proposées, et il lance la co-simulation globale du système jusqu'au nouveau point de synchronisation qui est maintenant égal à t_4. À l'issue de ce point de synchronisation, les deux parties échangent des données entre elles.

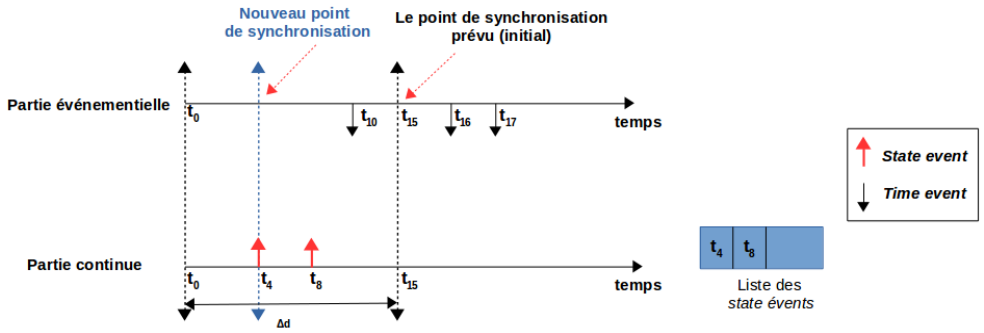


FIGURE 8 – Déroulement d'un exemple en utilisant notre modèle hybride classique

4.3 Optimisation de notre modèle de temps

Dans cette section, nous proposons d'optimiser notre modèle, en améliorant la structure de donnée définie auparavant, plus particulièrement en remplaçant le *flag* d'urgence

d'un événement par un paramètre δ_t, qui représente maintenant un délai maximal d'envoi de cet événement ($\delta_t \in \mathbb{R}$). Donc selon le degré d'urgence d'un *state event* apparu à $TimeStamp$, nous pouvons calculer sa date d'envoi comme comme suit :

$$t_{event}^{env} \begin{cases} = TimeSynch & \textbf{si} \quad TimeStamp + \delta_t \geq TimeSynch \\ \in [TimeStamp, TimeStamp + \delta_t] & \textbf{si} \quad TimeStamp + \delta_t < TimeSynch \end{cases} \tag{4.2}$$

Si ce *state event* n'est pas urgent ($TimeStamp + \delta_t \geq TimeSynch$), on le traite au prochain point de synchronisation, à l'instant ($TimeSynch$). Sinon, on rapproche le point de synchronisation pour traiter l'événement avant ($TimeStamp + \delta_t$).
Deux cas extrêmes sont alors possibles :

— Si ce *state event* est très urgent ($\delta_t = 0$), on doit le traiter à l'instant de son apparition ($TimeStamp$).

— Si cet événement n'est pas du tout urgent ($\delta_t = \infty$), alors on le traite au prochain point de synchronisation à l'instant ($TimeSynch$) voire à la fin de la co-simulation.

La solution de la section 2 représente un cas particulier de cette optimisation, elle correspond au traitement des événements non urgents ($urgence = 0$) traités à $TimeSynch$ dans le cas où ($TimeStamp + \delta_t >= TimeSynch$), sinon au traitement des événements urgents ($urgence = 1$) traités à $TimeStamp$ dans le cas où ($\delta_t = 0$).

4.4 Limitations de notre modèle hybride classique

La synchronisation des données, dans la simulation des systèmes complexes tels que les *Smart Grids* posent des problèmes de performance, qui sont essentiellement dus à l'hétérogénéité des composants. Dans la section précédente, nous avons proposé un modèle hybride combinant les deux parties du système. La partie continue de ce système réalise une exploration du futur qui est nécessaire pour définir ses *state events*. Mais malgré les bénéfices que cette pré-phase peut procurer, elle ralentit la co-simulation globale du système. Autrement dit, notre proposition rajoute de la précision puisqu'elle traite précisément les *state events* et les *time events*, mais elle réduit les performances de la co-simulation car la partie continue doit exécuter deux fois une partie de ces calculs.

Dans notre travail, nous avons traité des *state events* sous forme d'impulsions de *Dirac*, alors qu'en réalité, nous trouvons d'autres types de *state events* qui vont être abordés dans la section suivante. En outre, pour éviter les problèmes rencontrés par notre modèle hybride, nous proposons dans la partie qui suit quelques améliorations de la norme FMI et de l'API JavaFMI, pour qu'elles soient plus flexibles et applicables sur n'importe quel

système complexe.

À ce stade, notre solution et son modèle hybride devraient être transposables dans un formalisme de type DEVS&DESS [80] afin de le valider.

Mais, avec notre partenaire EDF, nous avons finalement choisi d'explorer une autre voie pour des co-simulations hybrides, qui nous permettra de ne plus utiliser de bus d'événements, et donc de simplifier notre architecture de co-simulateur. Cette nouvelle approche sera abordée dans les sections suivantes.

5 Concepts d'événements dans le contrôle-commande

Les systèmes complexes que nous considérons génèrent des commandes de type *time event* pour agir à distance sur les dispositifs physiques, et des signaux de type *state event* déclenchés et envoyés auprès des unités de contrôle en cas d'événements dans la partie continue. Jusqu'ici nous avons représenté et modélisé tous ces événements comme étant des impulsions de *Dirac*. Dans la réalité, les dispositifs physiques produisent essentiellement des signaux continus et différentiables porteurs d'informations, dont les ruptures de continuité et/ou de différentiabilité matérialisent les événements. Ces aspects des signaux des composants de la partie continue Φ sont incomplètement intégrés dans la norme FMI et donc dans l'API JavaFMI. C'est pour ces raisons que nous commençons dans cette partie par identifier divers types de signaux que l'on trouve dans un *Smart Grid* [70], et qui « contiennent des événements ». Dans les sous-sections suivantes, nous listons les différentes représentations d'événements que l'on trouve « sur le terrain », tantôt pour la partie continue, tantôt pour la partie événementielle.

Signaux continus mais dérivables par morceaux

On dit que la fonction « f » est continue et dérivable par morceaux lorsque « f » est continue en tout point de $\mathbb{R}$ et non dérivable en des points t_i ($t_i \in \mathbb{R}$), c'est-à-dire qu'il existe des points où la dérivée « f' » de la fonction « f » contient une limite finie à droite qui n'est pas égale à sa limite finie à gauche ($\lim_{\varepsilon \to 0} f'(t_i - \varepsilon) \neq \lim_{\varepsilon \to 0} f'(t_i + \varepsilon)$), comme l'illustre la figure 9.

Chaque point du signal où se trouve une rupture de la dérivée première est en fait un événement.

Signaux constants par morceaux

Dans la figure 10, le signal a une évaluation constante par morceaux, c'est-à-dire que le signal est :

— Continu et constant dans des intervalles de temps I^i tel que : $\cup_i I^i = \mathbb{R}^+$.

— Discontinu en des points précis (par exemple aux instants t_1, t_2 et t_3).

Chaque discontinuité du signal est de fait un événement.

Signaux dérivables et continus par morceaux

Ce type de signal représente un signal dérivable et continu par morceaux, et discontinu sur quelques points (voir la figure 11). En effet, il suffit que le signal de la fonction « f » soit :

— Continu et dérivable sur des intervalles de temps I^i tel que : $\cup_i I^i = \mathbb{R}^+$.

— non dérivable et non continu en certains points où les limites à droite et à gauche de la fonction f existent telles que : $\lim_{\varepsilon \to 0} f(t_i - \varepsilon) \neq \lim_{\varepsilon \to 0} f(t_i + \varepsilon)$

Chaque discontinuité du signal est de fait un événement.

Signaux discrets

Le signal discret est le plus connu dans les simulateurs événementiels, il représente un ensemble dénombrable d'impulsions de *Dirac* espacé par des Δt aléatoires, comme illustré sur la figure 12.

Chaque impulsion est naturellement un événement, mais cette illustration n'est pas celle des signaux du contrôle-commande. La réalité d'un *Smart Grid* correspondant surtout aux trois premières représentations (figures 9 à 11).

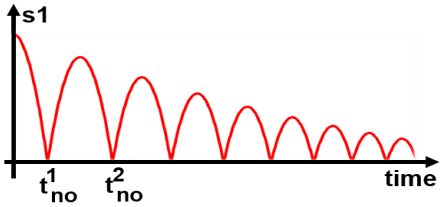


FIGURE 9 – Signal continu et dérivable par morceaux

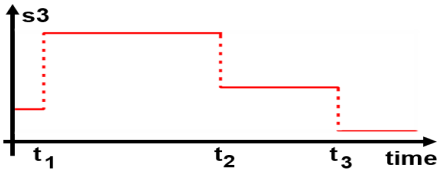


FIGURE 10 – Signal continu et constant par morceaux

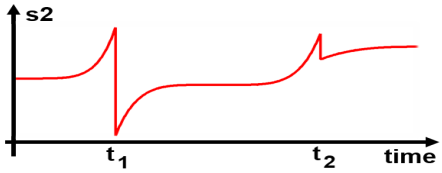


FIGURE 11 – Signal dérivable et continu par morceaux

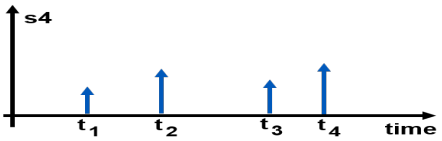


FIGURE 12 – Signal purement discret

6 Nouvelle approche de co-simulation hybride

6.1 Motivation pour une co-simulation entièrement FMI

Comme mentionné dans la section précédente, les composants du *Smart Grid* produisent des signaux continus et dérivables par morceaux dont les discontinuités représentent des événements (voir la section précédente). Dans la réalité, aucun signal n'est absent par morceaux. Ce constat nous encourage à utiliser et à exploiter la norme FMI, en encapsulant tous les composants dans des FMU (les composants discrets comme les composants physiques). Grâce à cette approche, nous pouvons représenter tous les composants du *Smart Grid* par des FMU, et nous pouvons utiliser la plate-forme de co-simulation DACCOSIM pour simuler l'ensemble des composants d'un *Smart Grid*. Autrement dit, nous n'avons plus besoin d'intégrer un bus d'événements dans notre architecture de co-simulation, y compris pour interfacer des simulateurs événementiels comme des simulateurs télécoms dans la mesure où les *smart grids* ne nécessitent pas de simuler finement les aspects télécoms.

Dans la section suivante, nous proposons une amélioration de la norme FMI et de l'API JavaFMI afin qu'elles puissent prendre en compte les différents cas d'interactions avec des FMU de composants discrets.

6.2 Proposition d'évolution de la norme FMI

6.2.1 Besoin d'une évolution de la norme FMI

Nous avons vu qu'il est possible de rencontrer des événements de type *state events* et/ou de type *time events* dans un système complexe. L'actuelle version 2.0 de la norme FMI ne détecte ces événements qu'à la fin d'un pas de temps, alors que nous avons vu que ces événements peuvent être urgents et nécessiter un traitement immédiat. La norme FMI 2.0 suit le même principe qu'EPOCHS, et engendre des retards de traitement des événements (surtout si le pas de temps est grand) en les prenant en compte seulement au début du prochain pas de temps. Ce qui signifie que la simulation n'est pas précise et ne reflétera pas la réalité.

Dans notre modèle (section 4.2), nous avons proposé de procéder par une exploration du futur suivie d'un *rollback* pour détecter les événements imprévisibles, au milieu d'un pas de temps. Dès lors, nous pouvons réduire le pas de temps et co-simuler au mieux le système dès l'émergence du premier événement, dans le but de le traiter avec un délai le plus réduit possible. Néanmoins, les retours en arrière générés ralentissent la co-simulation globale de notre système, et il est préférable d'en faire le moins souvent possible.

C'est pour cela que nous proposons d'améliorer la norme FMI et l'API JavaFMI, plus particulièrement en ajoutant les primitives suivantes : *fmi21DoStep()*, *fmi21GetNextEventTime()*, et *fmi21breakPendingStep()* [71]. Cet enrichissement [6] de la norme FMI permettra de traiter les événements sans délais avec un surcoût de temps d'exécution réduit au strict minimum.

6.2.2 Principe de la primitive *fmi21DoStep()*

Dans DACCOSIM, chaque dispositif physique (FMU) effectue un pas de simulation de taille h_i en appelant la primitive *fmi2doStep()* définie dans la version 2.0 du standard FMI, afin d'exécuter ses calculs à l'itération i. Néanmoins, cette fonction ne traite pas les événements intervenants au milieu d'un pas de temps et ne le signale qu'à la fin de ce pas de temps. Pour remédier à cela, nous proposons d'améliorer la primitive *fmi2DoStep()* par la primitive *fmi21DoStep(stepSize, ↑*[7] *nextEventTime)* qui permet de détecter avec précision un *state event* sans exploration du futur (solution proposée jusqu'ici dans notre modèle) : cette nouvelle primitive autorise le solveur numérique de la FMU à s'arrêter exactement à la première apparition d'un *state event*, ce qui semble simple à réaliser

6. Ces propositions ont été publié dans la conférence ETFA'2016 (*Emerging Technologies And Factory Automation*) dans la catégorie « *Work in progress* », et ont été récompensées par un « *Best Paper Award* ».

7. ↑ indique un paramètre de sortie.

compte-tenu des solveurs numériques qui fonctionnent en interne avec une succession de micro pas de temps (des pas de temps internes).

La figure 13 illustre le comportement de ces deux primitives. La nouvelle primitive *fmi21DoStep()* interrompt automatiquement son pas de temps dès qu'elle détecte son premier *state event* (s_e) afin de le signaler au master de co-simulation.

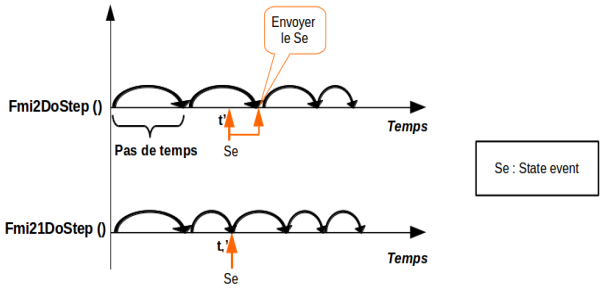


FIGURE 13 – Comportement des primitives *fmi2DoStep()* et*fmi21DoStep()* dans l'intégration d'un pas de calcul d'un composant physique

Comme le résume le diagramme de la figure 14, si un événement est bien détecté, cette primitive retourne le code $fmi21Event$ et fournit le *timestamp* de l'événement par la variable de sortie $nextEventTime$ (paramètre passé par référence) qui retourne sa date d'apparition. Dans le cas où il n'apparaît pas de *state event*, la valeur de la variable de sortie $nextEventTime$ correspond à la fin du pas de temps sans interruption et retourne le code $fmi21NoEvent$. L'utilisation de cette fonction évite l'exploration du futur qui implique des *rollbacks* et un ralentissement excessif de la co-simulation globale.

6.2.3 Principe de la primitive *fmi21BreakPendingStep()*

Dans la version actuelle de la norme FMI, quand une des FMU s'arrête prématurément, les autres FMU du système continuent leurs calculs jusqu'à la fin du pas de temps proposé par le master. Cela entraîne des calculs inutiles et réduit les performances de la co-simulation globale. Pour cette raison, nous proposons une nouvelle primitive appelée *fmi21BreakPendingStep(earlierTime)* permettant d'interrompre l'exécution des autres FMU qui n'ont pas encore fini d'intégrer leur pas de temps. Ces exécutions seront arrêtées par un appel asynchrone de cette primitive. Pour ce faire, les FMU du système s'arrêteront au $earlierTime$ si elles n'ont pas encore atteint la fin du pas de temps (leur objectif initial était bien au delà de $earlierTime$). Dans le cas contraire, les FMU s'arrêtent immédiatement pour effectuer un *rollback* et reprendre leurs calculs jusqu'à atteindre ce nouveau point ($earlierTime$).

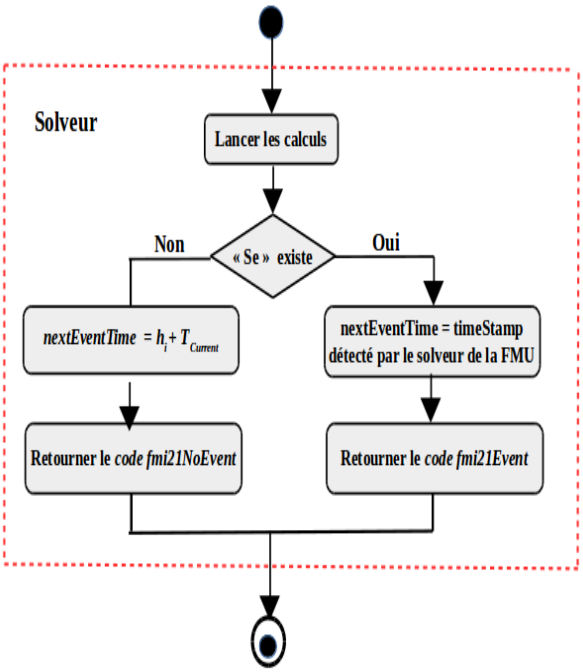


FIGURE 14 – Diagramme du fonctionnement de la primitive *fmi21DoStep()*

6.2.4 Principe de la primitive *fmi21GetNextEventTime()*

Afin de connaître à l'avance la date du prochain *time event* prévisible dans la partie événementielle, nous proposons une nouvelle primitive *fmi21GetNextEventTime()* qui permet à la partie événementielle (Δ) de signaler à l'avance la date précise de son prochain événement connu. En fait, le master du co-simulateur utilisé (le master dans DACCOSIM) doit exploiter aussi cette information pour définir à l'avance la valeur du prochain pas de temps calé à la date exacte du prochain *time event* prévisible.

Comme le montre la figure 15, cette primitive permet de définir la date des événements prévisibles qui apparaissent pendant la co-simulation.

Exemple

La figure 16 illustre un exemple d'un système comportant trois FMU, ces dernières devraient exécuter leurs calculs pendant un pas de temps h_0, de manière simultanée sur différents nœuds du cluster. Comme le montre cette figure, FMU1 génère un événement de type *state event* au temps simulé t'_i et s'arrête en exécutant la primitive *fmi21DoStep*(t_i) (qui détecte avec précision la date de ce *state event*). La FMU1 en question signale aux autres FMU (FMU2 et FMU3) d'arrêter leur calcul à t'_i en appelant la primitive *fmi21BreakPendingStep* (t'_i). Vu que la FMU2 réalise des calculs légers, elle a déjà atteint la fin du pas de temps initial (t_i), du coup, elle doit retourner à t_{i-1} en réalisant un *rollback*

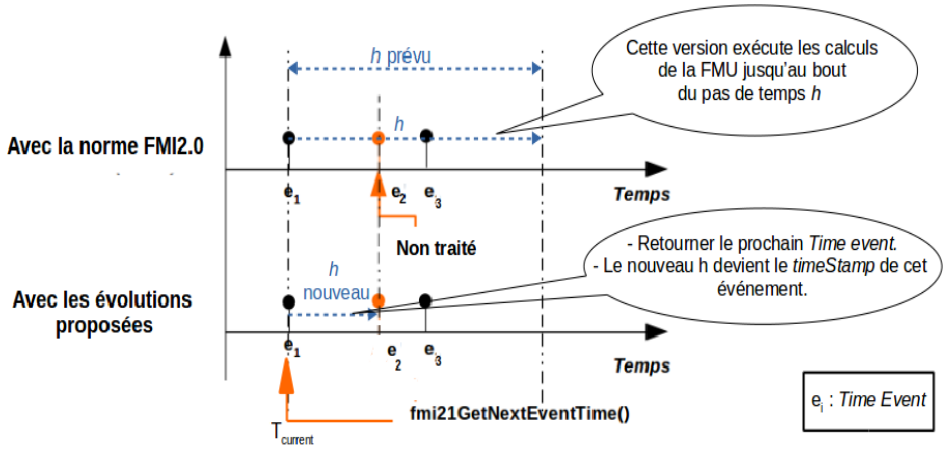


FIGURE 15 – Illustration de la primitive *fmi21GetNextEventTime()* dans la partie événementielle

afin de ré-exécuter son calcul jusqu'à t'_i. À l'inverse, FMU3 est plus gourmande en calcul et n'a pas encore atteint (t_i). Le solveur de cette FMU s'arrêtera ainsi à t'_i à l'effet de la primitive *fmi21BreakPendingStep()*. Au final, dans cet exemple, une seule FMU aura effectué un *rollback* pour traiter au plutôt le *state event* à t'_i.

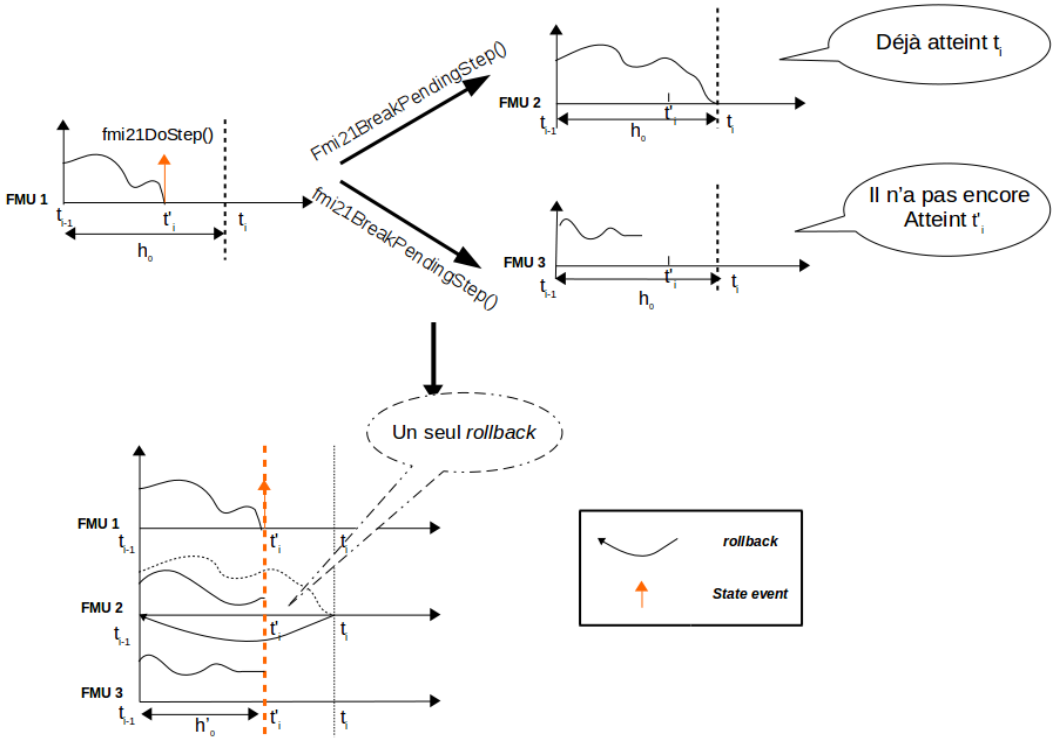


FIGURE 16 – Exemple de l'utilisation des primitives *fmi21DoStep()* et*fmi21BreakPendingStep()*

6.3 Architecture de co-simulation hybride entièrement DACCOSIM

Grâce aux propositions présentées dans la section précédente, nous pouvons réaliser une co-simulation hybride efficace et précise en utilisation notre plate-forme de co-simulation DACCOSIM. Il s'agit d'une co-simulation qui s'appuie sur la norme FMI 2.0 avec des évolutions limitées pour encapsuler les composants physiques et les composants de type contrôle-commande dans des FMU. Comme l'indique la figure 17, la nouvelle architecture d'un co-simulateur n'a donc plus besoin d'un bus d'événements pour synchroniser les échanges de données entre les différents composants du système. Le master de DACCOSIM se charge d'orchestrer et de contrôler l'ensemble de la co-simulation.

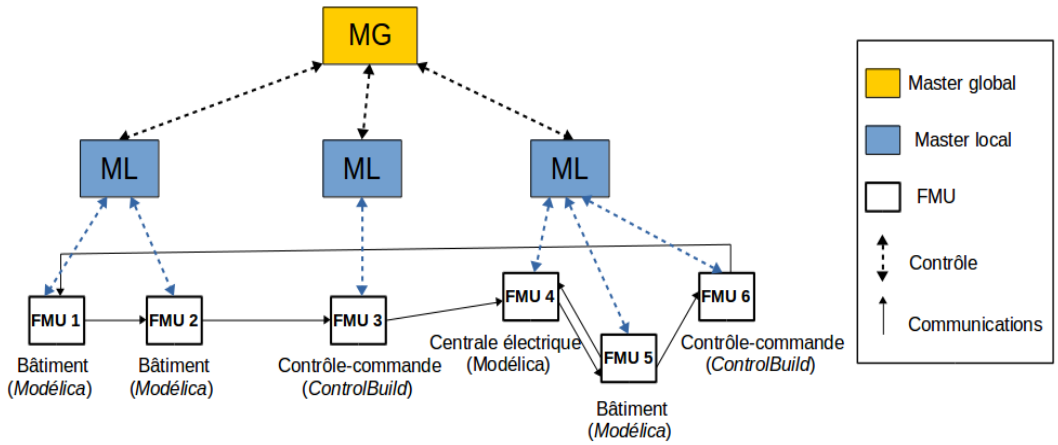


FIGURE 17 – Exemple d'une co-simulation hybride entièrement DACCOSIM

6.4 Travaux de normalisation au sein des *FMI Working Groups*

En parallèle des travaux engagés dans le cadre de la présente thèse et qui ont abouti à une proposition d'évolution de la norme FMI, EDF *Lab Saclay* a décidé de participer au développement du projet FMI à partir de début 2016. La maintenance et les évolutions de ce projet sont portées par l'Association Modelica [8].

Pour mémoire, le projet FMI a été créé pour développer, standardiser et promouvoir l'interface FMI qui est l'un des principaux résultats de l'ancien projet européen ITEA2 MODELISAR. On rappelle également que l'objectif du FMI est que des modèles de systèmes dynamiques provenant de différents systèmes logiciels puissent être utilisés ensemble, notamment pour la simulation des systèmes complexes de type cyber-physiques. Les versions 1.0 puis 2.0 du FMI ont été respectivement publiées en 2010 et en 2014.

Pour parvenir à une nouvelle version de cette norme, le projet FMI s'est constitué en plusieurs groupes de travail (*Working Groups*) collaborant en web conférences. Les experts

8. https ://www.modelica.org/.

de ces groupes de travail sont essentiellement européens (surtout allemands et suédois). Tous les groupes de travail se synchronisant une à deux fois par an lors de réunions présentielles appelées « *FMI Design Meetings* ». A titre indicatif, la dernière réunion de ce type s'est tenue à Munich fin novembre 2017. A noter également qu'EDF est pratiquement le seul utilisateur du FMI participant à l'élaboration de la norme, les autres participants étant presque tous des éditeurs de logiciels comme Dassault Systèmes qui commercialise les outils *Dymola* et *ControlBuild*.

EDF participe activement aux WG dans la partie « *clock and hybrid co-simulation* » en portant le double besoin :

— d'une détection exacte des événements internes à une FMU (triggered events) qui ne peut s'obtenir qu'avec une évolution du calcul d'un pas de temps sur le principe de la primitive *fmi21DoStep()*.

— d'une meilleure efficacité de la synchronisation des FMU dès lors qu'un événement a été détecté par une FMU qui s'est arrêté prématurément à un instant correspondant à son *time event*. Il s'agit ici du besoin porté par la primitive *fmi21BreakPendingStep()*.

Comme toujours en normalisation, les travaux avancent très lentement car il faut tenir compte en même temps du point de vue de chaque partie prenante, de la compatibilité ascendante avec la version 2.0 du standard et de la nécessaire cohérence à assurer entre les deux branches du FMI (correspondant aux modes de fonctionnements : *Model Exchange* et *Co-Simulation*).

À ce stade d'avancement des discussions, il semble peu probable qu'une publication soit effective en 2018, sauf peut-être pour une version de travail (*draft proposal*). Néanmoins, les idées présentées dans les sections 5 et 6 de ce chapitre sont étudiées attentivement dans les *FMI Working Groups*.

7 Conclusion

Dans ce chapitre, nous avons présenté les lignes directrices d'une approche hybride classique appliquée aux systèmes complexes quelconques (et son modèle de temps hybride). Cette approche combine le modèle événementiel à base d'événements et le modèle continu à pas de temps. Le premier modèle hybride proposé a la responsabilité de gérer et synchroniser les données échangées entre les FMU dans tout l'environnement de la co-simulation en enrichissant une architecture classique autour d'un bus d'événements, et d'interfacer une grappe de FMU sur ce bus. Nous avons également décrit le fonctionnement du modèle proposé, à l'aide de diagrammes et d'algorithmes. Nous avons montré

que malheureusement cette approche générique reste coûteuse en temps de calcul.

Dans la seconde partie de ce chapitre, nous avons proposé une solution plus originale qui ne nécessite plus d'utiliser un bus d'événements en simplifiant l'architecture du co-simulateur, et en allégeant les traitements. Ceci a été possible suite à une analyse des besoins concrets de co-simulation de *Smart Grids* et de la nature des événements portés par des signaux réels. Cette approche nécessite cependant une évolution de la norme FMI qui fait l'objet de travaux d'EDF au sein des *FMI Working Groups*.

Dans la deuxième partie de cette thèse, nous nous intéressons aux performances des *Smart Grid* dans différentes circonstances. En fait, la simulation de la partie événementielle est négligeable en temps de réponse par rapport aux dispositifs physiques du *Smart Grid*. Pour cette raison, nous nous focaliserons dans le chapitre suivant sur la partie la plus gourmande en calcul qui est la partie continue contenant un ensemble de composants physiques. Des approches seront proposées pour répartir au mieux ces composants hétérogènes sur un cluster de machines multi-cœurs, en utilisant la plate-forme DACCOSIM présentée dans le chapitre 3.

CHAPITRE 5

MAPPING DES TÂCHES DE TYPE « FMU » SUR CLUSTER DE PC MULTI-COEURS

Sommaire

1 Introduction

1.1 Objectifs

La co-simulation des *Smart Grids* fait appel à la fois à des simulateurs qui exécutent une partie continue et une partie événementielle. Néanmoins, la simulation de la partie événementielle est habituellement très rapide par rapport à celles des composants physiques de la partie continue. Étant donné que cette thèse se préoccupe fortement des performances de la co-simulation, il est donc judicieux de s'intéresser à la partie la plus gourmande en calcul amenant un ralentissement de la co-simulation globale, c'est-à-dire celle des dispositifs physiques. Pour répondre à ce besoin, nous utilisons la plate-forme DACCOSIM pour simuler un réseau de distribution électrique. Cette plate-forme représente ce type de réseau sous la forme d'un graphe de FMU qui devient ensuite un graphe de tâches hétérogènes communicant fréquemment. Toutefois, si la taille du graphe est importante, on ne peut pas exécuter ses tâches sur une seule ressource informatique (avec une taille mémoire et une puissance de calcul limitées), mais plutôt sur un cluster de PC multi-cœurs. La solution la plus simple est de placer une FMU sur un nœud de calcul, mais cela représente un gaspillage de ressources car on n'exploite pas totalement les cœurs disponibles sur chaque nœud de calcul, surtout si on n'a pas assez de ressources informatiques.
L'objectif de ce chapitre est de chercher à répartir au mieux les différents composants physiques (FMU) d'une même co-simulation (graphe de FMU) sur des clusters de machines multi-cœurs. Durant ces trois années de thèse, nous avons ainsi adapté DACCOSIM à ce type d'architecture en améliorant ses capacités d'exécution.

Nous évoquons, dans ce chapitre, trois approches de placement différentes : une approche par heuristiques élémentaires s'appuyant uniquement sur le graphe DACCOSIM, une approche par heuristiques complexes permettant d'analyser finement le comportement d'une co-simulation sur une architecture distribuée multi-cœurs, et une approche automatique ne nécessitant aucune pré-étude à l'avance. Mais avant cela, nous étudions quelques expérimentations décrivant le comportement de nos FMU sur une architecture distribuée multi-cœurs, afin de cadrer nos approches.

Pour finir, nous présentons un modèle prédictif qui permet d'estimer de manière approximative nos phases de calculs et nos phases de communications, afin de les utiliser dans les approches qui nécessitent des connaissances préalables à chacune de ces deux phases.

1.2 Le *mapping* dans la chaîne de co-simulation

Dans cette section, nous décrivons d'une manière générale la démarche effectuée pour notre projet de co-simulation. Comme l'illustre la figure 1, notre démarche comporte quatre étapes importantes pour co-simuler un système cyber-physique sur un cluster de PC multi-cœurs :

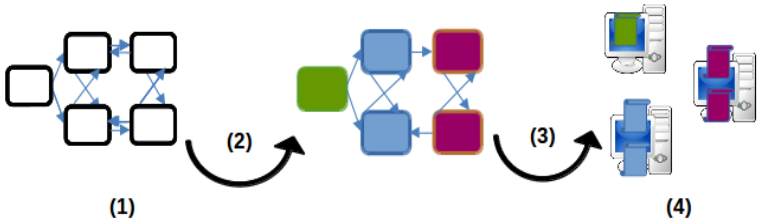


FIGURE 1 – Approche suivie dans notre projet de co-simulation

1. Réalisation d'un graphe de co-simulation d'un système cyber-physique avec la plate-forme DACCOSIM qui a été présentée dans le chapitre 3.
2. Répartition des « k » FMU (tâches) sur « n » nœuds virtuels (vnode) définis dans le graphe de ressources DACCOSIM (voir le chapitre 3), tels que (k >= n), $k \in \mathbb{N}^*$ et $n \in \mathbb{N}^*$.
3. Création de « n » codes Java correspondant aux différents codes de co-simulation (un fichier par vnode).
4. Déploiement des « n » nœuds virtuels (vnodes) sur « n » nœuds physiques (pnodes), en appliquant un des algorithmes de *mapping* proposés dans les sections qui suivent.

Notre travail se focalise sur l'étape 2, qui vise à attribuer chaque tâche indépendante à une ressource informatique, en cherchant à minimiser le temps d'exécution global de la co-simulation.

En accord avec EDF, nous faisons l'hypothèse que les tâches (FMU) de nos applications ne changent pas dans le temps. Comme de plus les graphes de co-simulation sont statiques, on peut utiliser des approches « *off-line* » qui traitent des tâches fixes contrairement aux approches « *on-line* » qui traitent des applications dynamiques. Pour cela, nous avons proposé des solutions permettant de calculer le placement avant l'exécution des tâches (voir le chapitre 2). Dans ce cadre, les deux premières approches s'appuient sur une démarche partiellement manuelle qui a comme but d'analyser et d'évaluer le comportement des FMU, et la troisième est une approche automatique appliquée aux modèles à grande échelle.

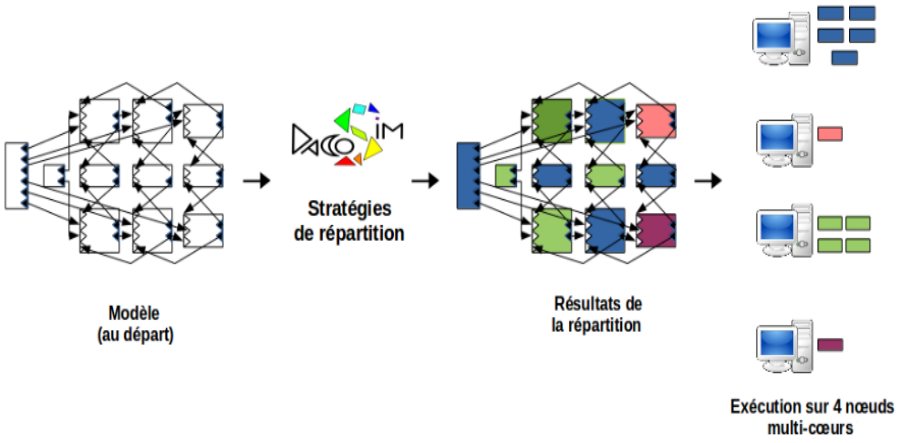


FIGURE 2 – Les étapes de la répartition sur cluster

2 Modèle d'exécution d'une co-simulation DACCOSIM

2.1 Modèle en trois étapes

Dans notre cas, nous avons un graphe de tâches contenant des boites grises fortement connectées entre elles. Il s'agit d'un graphe différent des graphes présentés habituellement dans la littérature (voir le chapitre 2). C'est un graphe hybride dégénéré qui se comporte à la fois comme un graphe de tâches indépendantes pour exécuter la phase de calcul des FMU, et comme un graphe de tâches dépendantes de la fin des calculs jusqu'au début du pas de temps suivant. De manière générale, durant le pas de temps (h_i) de l'itération i, toutes les FMU du graphe DACCOSIM exécutent leur phase de calcul en parallèle selon le mode d'orchestration défini auparavant (graphe de tâches indépendantes). À l'issue de ce pas de temps, les FMU font circuler les résultats de leurs sorties vers les entrées des FMU connectées avec elles (graphe de tâches dépendantes). Notons que pour la suite, nous allons utiliser le mode d'orchestration «*ordered*» car ce dernier nous permet de mesurer correctement et précisément les temps de communication contrairement au mode «*overlapped*» qui fausse ces mesures en exécutant en parallèle la phase de communication et de contrôle.

Dans la figure 3, nous avons un exemple qui propose quatre tâches hétérogènes indépendantes qui s'exécutent en parallèle pendant un même pas de temps. Avec DACCOSIM, les *wrappers* des quatre FMU lancent leur phase de calcul en parallèle, cette caractéristique représente un graphe de tâches indépendantes. Ensuite, le master hiérarchique (MA [1]) de ce graphe attend la fin de ces calculs pour déterminer le prochain pas de temps de toute la co-simulation (voir le chapitre 3), cela représente une dépendance entre la phase de calcul (voir Fig. 3-a) et la phase de communication (voir Fig. 3-b) dans

1. MA : englobe les masters locaux et le master global de la co-simulation.

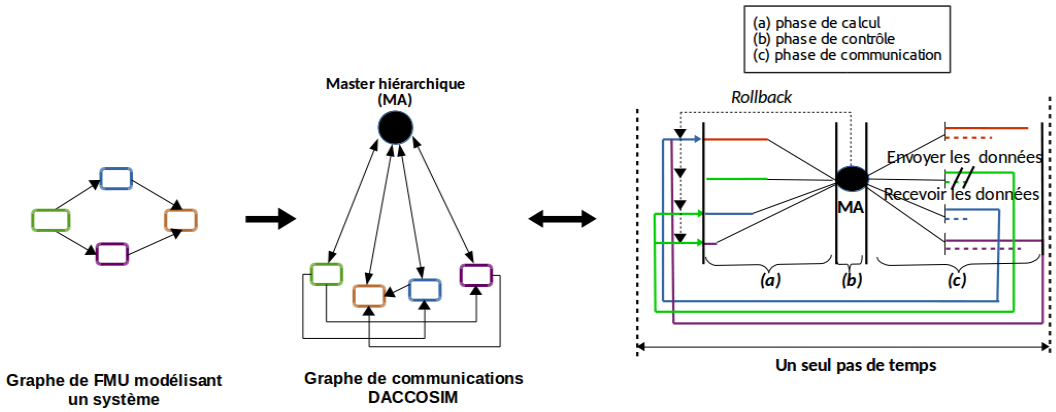


FIGURE 3 – Exécution abstraite de DACCOSIM

le graphe DACCOSIM : on ne lance pas la phase de communication tant que les FMU n'ont pas encore finies leurs calculs (synchronisation forte dans le mode *ordered*), et que le master global n'a pas pris la décision globale soit de poursuivre la simulation ou de revenir en arrière. Ensuite, chaque FMU doit attendre les résultats des sorties des autres tâches connectées avec elle avant de commencer un nouveau calcul (un nouveau pas de temps), cela indique un graphe de tâches dépendantes. Notons que les FMU effectuent en fait leurs communications en parallèle, cette dernière étape représente donc à nouveau un graphe de tâches indépendantes. En outre, on peut considérer ce graphe de calcul DACCOSIM comme un modèle de tâches périodiques avec une période qui est égale à un pas de temps (voir la partie droite de la figure 3).

La figure 4 illustre une représentation détaillée du modèle d'exécution de l'exemple précédent pendant toute la co-simulation. Les communications échangées entre les différentes FMU du graphe sont effectuées à la fin du pas de temps actuel (itn), mais ne vont être prises en compte qu'à partir de l'itération suivante ($it(n+1)$) (au prochain pas de temps).

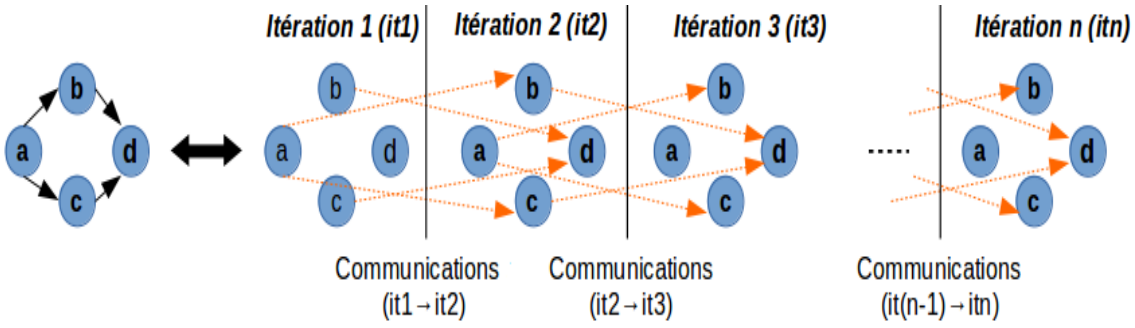


FIGURE 4 – Modèle d'exécution de notre graphe de FMU

2.2 Modélisation du temps d'exécution

Comme l'indique la figure 3 de la section précédente, nous avons modélisé le graphe DACCOSIM en utilisant le mode (*ordered*) car il assure une exécution séquentielle pour chaque FMU au sein du même pas de temps, ce qui permet de mesurer précisément les temps des différentes phases de l'exécution. En général, le temps d'exécution de chaque FMU comprend le temps de calcul (T_{Calcul}) du solveur de la FMU, le temps de contrôle ($T_{Control}$) nécessaire pour superviser la co-simulation globale et débloquer la barrière de synchronisation qui existe entre les FMU à la fin des calculs, et le temps des communications (T_{Comm}) entre la FMU et les autres FMU qui lui sont connectées. Nous pouvons formaliser le temps d'exécution de la FMU j comme suit :

$$T_{FMU}(j) = T_{Calcul}(j) + T_{Control}(j) + T_{Comm}(j) \tag{5.1}$$

Dans DACCOSIM, nous avons deux synchronisations : une synchronisation au niveau du master local pour synchroniser les calculs des FMU qui se trouvent sur le même nœud physique, et une autre synchronisation au niveau du master global pour synchroniser les masters locaux et assurer un contrôle global cohérent. Le coût de ces synchronisations est important dans nos applications parallèles car il peut accroître le temps global de la co-simulation. C'est cependant le prix à payer pour disposer d'une supervision distribuée.

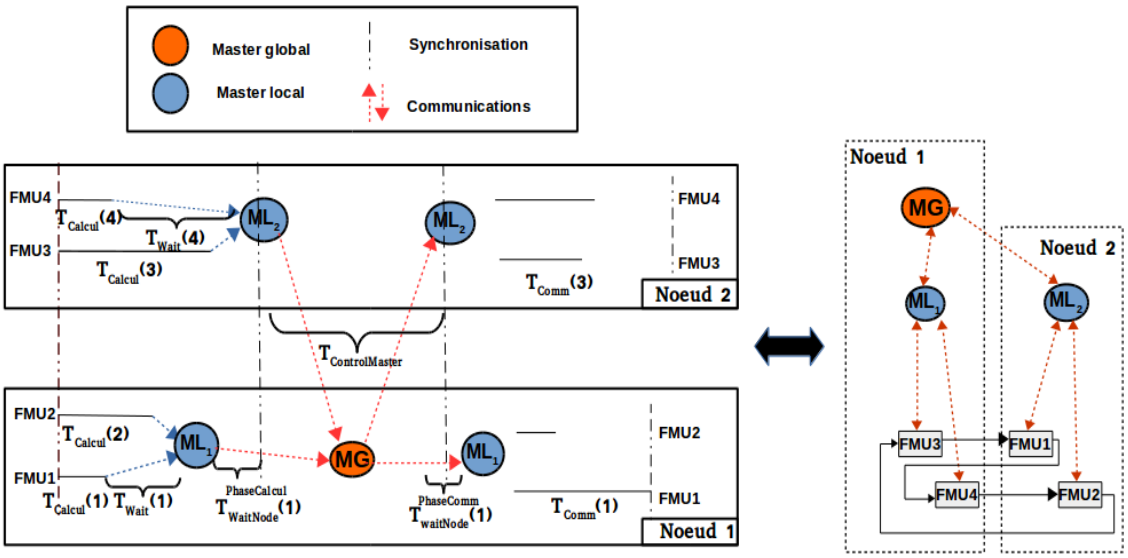


FIGURE 5 – Schéma détaillé de la modélisation d'une co-simulation DACCOSIM

Comme l'indique la figure 5, la première étape de contrôle inclut une synchronisation

des FMU se trouvant sur le même nœud de calcul : chaque master local du graphe DAC-COSIM attend d'avoir reçu un message de ses FMU avant de décider de la suite des opérations soit en les laissant poursuivre leur traitement (et accomplir leur communication), soit en les obligeant à refaire leur calcul. Par la suite, les FMU qui se trouvent sur le même nœud de calcul attendent une réponse de leur master local, qui lui même attend un message de la FMU la plus longue à faire ses calculs. Après la collecte des souhaits proposés par leur FMU, chaque master local transmet une demande unifiée au master global et attend sa réponse. Le temps de contrôle perçu par une FMU ne dépend donc pas que de cette FMU, mais plutôt des comportements de toutes les FMU du système, et cela rend la modélisation difficile.

En résumé, le temps de contrôle d'une FMU j comprend le temps de synchronisation entre l'ensemble des FMU du même nœud de calcul et le master hiérarchique (master local et master global). Il regroupe donc le vrai temps de contrôle effectué au niveau du master ($T_{ControlMaster} = c^{ste}$ pour toutes les FMU du système), et un temps d'attente $T_{Wait}(j)$ variable pour chaque FMU du système, ce qui donne :

$$T_{Control}(j) = T_{ControlMaster} + T_{Wait}(j) \tag{5.2}$$

Cependant, puisque le processeur sur lequel nous travaillons contient plusieurs cœurs physiques, nous pouvons exploiter leur utilisation en exécutant un ensemble de FMU simultanément sur un ensemble de cœurs du même nœud physique. Si on modélise le temps de co-simulation d'un nœud de calcul, on doit rajouter le coût de synchronisation $T_{WaitNode}$ qui est un temps d'attente du master local du nœud concerné vis-à-vis du master local du nœud le plus long à faire ses calculs. Ce temps d'attente est nécessaire pour estimer le vrai temps de contrôle du nœud visé, dans le cas où la plus longue FMU qui se trouve sur ce nœud physique a fini ses calculs alors que d'autres FMU sur d'autres nœuds n'ont pas encore fini les leurs. Le temps de co-simulation d'un nœud n multi-cœurs sera donc calculé comme suit :

$$\begin{aligned} T_{CoSimNode}(n) = \max_{1\leq k\leq K_n} T_{Calcul}(k) + T_{WaitNode}^{PhaseCalcul}(n) + T_{ControlMaster} \\ + T_{WaitNode}^{PhaseComm}(n) + \max_{1\leq k\leq K_n} T_{Comm}(k) \end{aligned} \tag{5.3}$$

Où « K_n » est le nombre de FMU se trouvant sur le nœud physique « n ». On peut aussi exprimer le $T_{WaitNode}^{PhaseCalcul}$ de chaque nœud de calcul à partir des temps d'attentes de ses FMU :

$$T_{WaitNode}^{PhaseCalcul}(n) = \min_{1\leq k\leq k_n} T_{Wait}(k) \tag{5.4}$$

En revanche, vu du master global, si on modélise le temps de co-simulation de tout le système, le temps d'attente de chaque master local va être éliminé car le master global de cette co-simulation attend la fin des calculs du nœud le plus long à calculer. Il prend donc le maximum des temps de calcul de tous les masters locaux, ce qui absorbe le temps d'attente qui apparaît suite à cette synchronisation. On trouve ce même raisonnement dans la phase de communication (les temps d'attente sont masqués). Le temps de co-simulation de tout le système, pour un seul pas de temps d'une itération i, est donc calculé comme ci-après :

$$T_{CoSimOneStep}(i) = \max_{1 \leq n \leq N} T_{CalculNode}(n) + T_{ControlMaster} + \max_{1 \leq n \leq N} T_{CommNode}(n) \quad (5.5)$$

Tel que le $T_{CommNode}(n)$ est le temps maximum des communications des FMU qui interviennent au sein du même nœud physique, et le N est le nombre de nœuds de calcul. Notons que dans la section 5 de ce chapitre, nous ferons une modélisation plus fine de la co-simulation qui prendra en compte les deux temps d'attente : $T_{WaitNode}^{PhaseCalcul}$ et $T_{WaitNode}^{PhaseComm}$.

Et bien sûr le temps global de toute la co-simulation se calcule comme ci-après :

$$T_{TotalCosim} = \sum_{i=1}^{iterations} (T_{CoSimOneStep}(i)) \quad (5.6)$$

Tel que « *iterations* » représente le nombre d'itérations de toute la co-simulation. Dans le cas où on adopte un pas de temps constant ($h_i = h$), le nombre d'itérations sera calculé par :$iterations = \frac{t_{duration}}{h}$. Mais dans le cas où on utilise un pas de temps variable, le nombre d'itérations ne sera connu qu'à la fin de la co-simulation car il dépend du nombre de *rollback* effectués.

3 Investigations expérimentales

Un des objectifs de cette thèse est de proposer des approches et des méthodes de placement de tâches sur un cluster de PC multi-cœurs, permettant d'aboutir à des co-simulations à la fois précises et performantes. Les tâches traitées sont représentées sous forme de boites grises dont on ne connaît pas certaines caractéristiques, par exemple la charge de calcul et le coût des communications de chaque FMU. La répartition de telles boites grises, de natures hétérogènes et échangeant fréquemment des petits messages, reste difficile. Pour combler le manque d'informations de nos boites grises, nous évaluons dans les sections qui suivent le comportement de nos FMU sur un cluster de PC multi-

cœurs.

3.1 Comportement des FMU sur un nœud multi-cœurs

La problématique de cette étape est d'étudier le comportement des FMU sur un nœud multi-cœurs, et de déterminer expérimentalement un paramètre qui définisse le nombre de FMU supportées par ce nœud de k cœurs. Pour réaliser cela, nous mettons une seule FMU sur un nœud multi-cœurs et au fur et à mesure nous augmentons le nombre de FMU jusqu'à atteindre le nombre de cœurs physiques de ce processeur. Et à la fin de chaque opération, nous vérifions si le temps d'exécution de ce nœud reste plus ou moins constant. Cette section englobe deux parties d'expérimentation, la première qui permet de positionner le comportement de nos applications basées sur des FMU par rapport à des calculs classiques de type HPC [2] qui constituent souvent une référence de bonne utilisation des ressources de calculs. La deuxième partie de ces expérimentations a comme but de déterminer le nombre maximal de FMU conservant le même temps de co-simulation d'un nœud multi-cœurs.

3.1.1 Comportement des FMU par rapport aux calculs classique HPC

Pour répondre au premier objectif de ces expérimentations, nous utilisons une bibliothèque de calcul optimisée (où les données sont « bloquées en cache ») dite BLAS (*Basic Linear Algebra Subroutines*), qui fournit des routines pour réaliser des opérations classiques sur les matrices et les vecteurs (factorisations, multiplications...). Dans notre cas, nous prenons une méthode BLAS appelée *DGEMM* (*Double-precision General Matrix Multiply*) permettant de calculer la matrice C sur la base du produit de la matrice A et B et de la valeur précédente de la matrice C ($C \longleftarrow \alpha \times A \times B + \beta \times C$). Dans notre cas, nous prenons $\alpha = 1$ et $\beta = 0$ et nous évaluons les performances d'une même fonction *DGEMM* appliquée à trois tailles de matrices différentes :

— **DGEMM de grande taille (DGEMM-GT)** calculera un produit de matrices denses (4096×4096) avec une taille de données de 128 MB par matrice (voir la Fig 6 couleur verte).

— **DGEMM de taille moyenne (DGEMM-MT)** calculera un produit moins important (2048×2048) que la première catégorie, avec une taille de 32 MB par matrice (voir la Fig 6 couleur bleue).

— **DGEMM de petite taille (DGEMM-PT)** calculera un produit léger avec une dimension de matrice de (1024×1024), et une taille de donnée de 8 MB par matrice

2. High Performance Computing

(voir la Fig 6 couleur orange).

Dans un nœud multi-cœurs, l'interférence des threads sur les ressources partagées (les caches et la mémoire principale) et la surcharge des cœurs du processeur peuvent ralentir les exécutions des applications parallèles du système. Pour palier à ce problème, nous étudions à travers des courbes de ralentissement la qualité du parallélisme de ces routines exécutées concurremment sur un nœud de calcul multi-cœurs. Le ralentissement d'une application est défini ici comme le rapport entre son temps d'exécution lorsqu'elle est exécutée avec d'autres applications (parallèles) et son temps d'exécution lorsqu'elle est exécutée seule sur un nœud multi-cœurs.

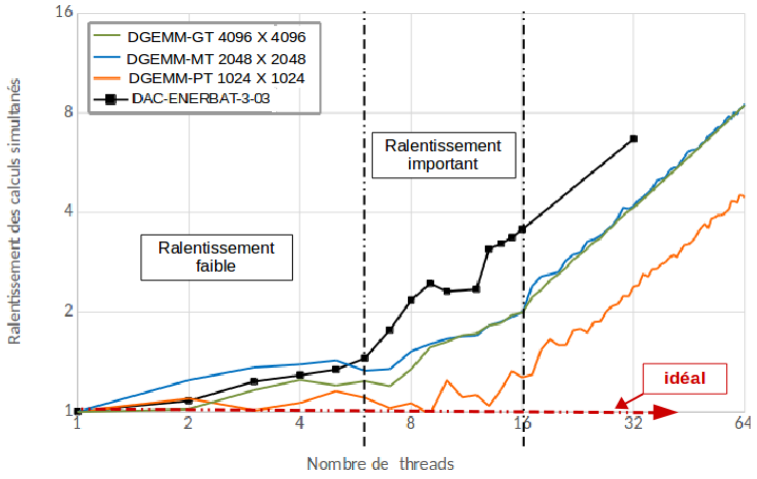


FIGURE 6 – Comparaison du comportement des FMU avec ceux de calculs HPC.

Afin de positionner le comportement de nos applications basées sur des FMU par rapport à des calculs HPC classiques, nous utilisons un cas industriel simplifié fourni par EDF R&D. Il représente en fait des transferts de chaleur dans un bâtiment à trois étages et il comporte onze FMU physiques. La figure 6 illustre les temps de calcul mesurés par des calculs HPC concurrents et de tailles différentes, et par un graphe de FMU en fonction du nombre de threads se trouvant sur le même nœud multi-cœurs. Pour chaque taille de problème, nous exécutons simultanément de 1 à 64 threads sur la même machine multi-cœurs dont chaque thread est soit un appel complet de la routine DGEMM avec ses propres structures de données allouées, soit un appel d'un *wrapper* d'une FMU. Une campagne d'expérimentations a été réalisée pour mesurer le ralentissement de ces calculs concurrents sur des nœuds multi-cœurs en vérifiant si ces applications permettent une bonne concurrence des threads ou non.

Dans le cas idéal, on obtient une ligne plate dont la valeur est constante et vaut 1, cela montre une exécution concurrente parfaite des threads sur les différents cœurs du nœud.

Néanmoins, dans la pratique, les tâches qui s'exécutent simultanément peuvent se gêner et se perturber les unes avec les autres lors de l'accès à la mémoire, et cela peut entraîner un ralentissement important. Afin d'évaluer notre application à base de FMU avec celles à base de calculs HPC présentées ci-dessus, nous utilisons un nœud *bi-sockets* comportant 2×4 cœurs physiques (2×8 cœurs logiques).

À travers cette figure, nous observons que les FMU évoquées se comportent presque comme des tâches optimisées de grande taille (DGEMM-GT). Nous remarquons, sur cette dernière, que lorsqu'on a exécuté un ensemble de FMU (moyenne) sur plusieurs cœurs physiques, on obtient un ralentissement faible et acceptable jusqu'à arriver au nombre de cœurs physiques moins 2 (on soupçonne que ces deux cœurs physiques sont occupés par des threads de communication et des appels systèmes). **Mais si le nombre de threads dépasse le nombre de cœurs physiques moins 2, le temps de co-simulation augmente, ce qui mène à un ralentissement important et à une parallélisation dégradée**. Au delà des cœurs logiques, on observe un ralentissement normal car le nombre de threads à exécuter simultanément dépasse le nombre d'unités de calculs disponibles.

3.1.2 Détermination du nombre de FMU supportées par un nœud multi-cœurs

Comme déjà expliqué précédemment, nous mettons une seule FMU sur un nœud de k-cœurs, et nous rajoutons progressivement d'autres FMU sur ce même nœud jusqu'à arriver à k FMU sur le même nœud. L'objectif de ces expérimentations est d'analyser la qualité du parallélisme des tâches en conservant le temps de calcul constant jusqu'à arriver à « k » FMU sur le même nœud k-cœurs. Dès lors, nous avons exécuté le même *use-case* que la section précédente sur un des nœuds *quadri*-cœurs (4 coeurs physiques/ 8 cœurs logiques) et *hexa*-cœurs (6 cœurs physiques/ 12 cœurs logiques) respectivement. À la fin de chaque expérience, nous mesurons le temps maximum de calcul de toutes les FMU qui se trouvent sur le même nœud de calcul. Notons que nous avons réalisé ces tests sous des conditions et des configurations réelles : le graphe de co-simulation global était celui d'une co-simulation de consommation énergétique de bâtiments.

Sur la figure 7, nous observons que le temps de calcul augmentent légèrement jusqu'à arriver à 3 et 4 FMU respectivement sur un nœud quadri-cœurs et hexa-cœurs (une augmentation de 20%) et au delà l'augmentation devient très significative. Cette évaluation montre que l'équilibrage de charge d'un graphe de FMU sur un cluster de PC multi-coeurs n'est donc pas évidente surtout dans le cas où les tâches sont hétérogènes.

À travers l'ensemble de ces expérimentations, nous avons défini de manière générale le

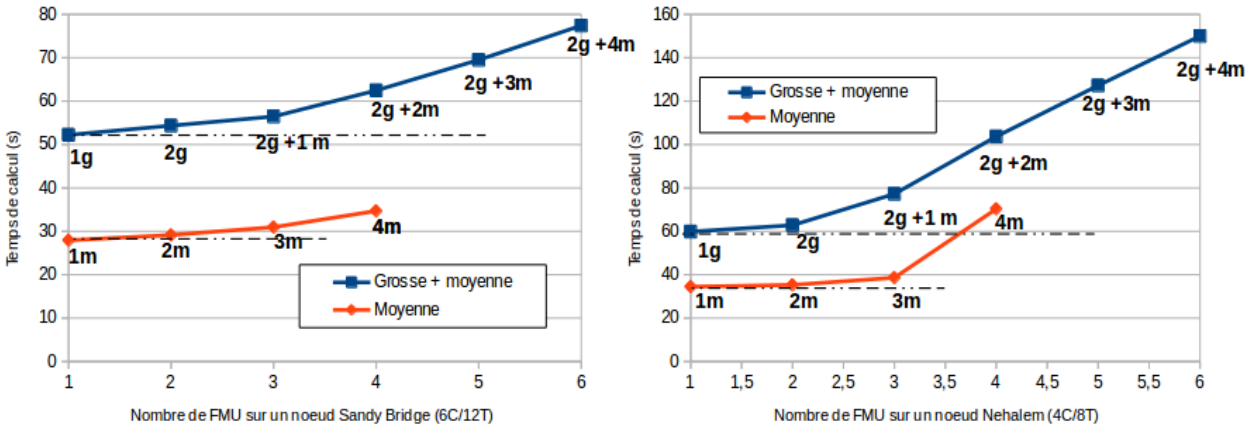


FIGURE 7 – Comportement de nos FMU sur un nœud physique de multi-cœurs.

paramètre $nbrFmuOnOneNode$, qui définit le nombre de FMU qu'on peut mettre sur le même nœud de calcul sans entraîner de gros ralentissements (voir l'équation 5.7) :

$$nbrFmuOnOneNode = \begin{cases} nbrPhysicalCores - 2 & \textbf{si la charge des FMU est proche} \\ \lceil \frac{nbrPhysicalCores}{2} \rceil & \textbf{sinon} \end{cases} \tag{5.7}$$

Tel que $nbrPhysicalCores$ est le nombre de cœurs physiques d'un nœud multi-cœurs.

Pour garantir de bonnes performances, on est arrivé expérimentalement à la conclusion suivante : selon le type du nœud multi-cœurs et l'application à traiter, on doit mettre soit un nombre de FMU égal au nombre de cœurs physiques sur deux (cas idéal) ou un nombre de cœurs physiques moins deux sur un même nœud. Mais cela suppose que l'utilisateur peut allouer suffisamment de nœuds de calculs (dans le cas idéal).

3.2 Sensibilité de la co-simulation au réseau d'interconnexion

Dans cette section, nous voulons analyser le comportement d'un ensemble de modèles de tailles différentes sur des réseaux standards de type Ethernet afin de voir si le type du réseau peut influencer et impacter le temps global de la co-simulation. Pour se faire, nous avons réalisé différents tests sur deux types de clusters : avec un réseau *Ethernet* de 1 Gbit/s et avec un réseau *Ethernet* de 10 Gbits/s, en agrandissant la taille du problème élémentaire de 1 à 5 fois sur chaque cluster.

La figure 8 illustre des exécutions de problèmes de plus en plus gros sur de plus en plus de machines multi-cœurs. Nous avons exécuté des simulations de 1, 2, 3, 4 et 5 bâtiments sur un cluster de PC de 3, 6, 9, 12 et 15 nœuds multi-cœurs, en augmentant à la fois le nombre de FMU et le nombre de PC muti-cœurs dans les mêmes proportions.

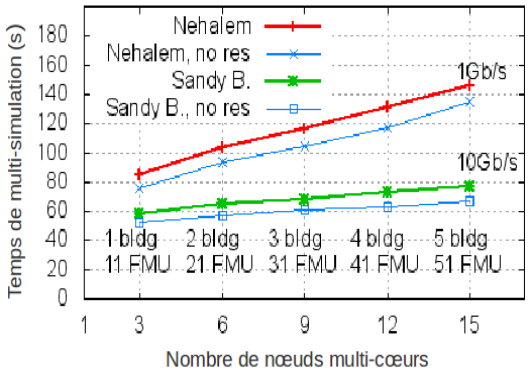


FIGURE 8 – Comportement des FMU sur des réseaux de communications standards

Dans le cas idéal, sur le même réseau de communication, on devrait obtenir des temps de co-simulation identiques pour tous les problèmes traités afin de garder une parallélisation parfaite. Nous observons, à travers cette figure, qu'à chaque fois qu'on augmente la taille du problème, le temps devient plus long que le précédent, surtout si on utilise un réseau de communication peu performant (1 Gbit/s). **Cette dégradation est due aux communications importantes effectuées au sein de ces co-simulations : les communications inter-FMU ne sont pas du tout négligeables dans le temps d'exécution global (pourtant ces communications se font bien directement entre les nœuds concernés, sans passer par un nœud central)**.

D'autre part, le réseau de communication de 10 Gbits/s donne de meilleures performances par rapport au réseau d'un Gbit/s, ce qui montre qu'on peut augmenter les performances des d'applications à base de FMU si on utilise un réseau de communication performant et de qualité. On peut conclure que la co-simulation à base de FMU nécessite un réseau rapide et performant, comme par exemple les réseaux *infinibands* qui fournissent une petite latence et une bande passante importante. Des expérimentations sur ce type de réseau seront présentées dans la suite de ce mémoire.

3.3 Expérience de passage à l'échelle

Au début de la thèse, nous n'avions pas assez d'informations ni sur les FMU ni sur leur comportement sur un cluster de PC multi-cœurs. Pour pallier à cette pénurie, nous avons suivi une *méthodologie* expérimentale de placement manuel pour vérifier si ce type d'applications qui sont à base de FMU passeront à l'échelle.

Mais avant de suivre notre méthodologie, nous définissons deux placements de références que nous avons utilisés soit dans cette méthodologie soit pour calculer les performances : un placement totalement localisé *AllOnOneNode* et un placement distribuée à l'extrême *OneFMUOneNode* .

— ***AllOnOneNode***

C'est un placement localisé consistant à mettre toutes les FMU du système sur un unique nœud multi-cœurs. Toutefois, on pourrait avoir un nombre de FMU k qui dépasse le nombre de cœurs physiques c du nœud tel que $(k > c)$, ce qui mène à avoir de mauvaises performances dû à la surcharge des cœurs du processeur. Notons que *AllOnOneNode* nous aidera à déterminer le *speedup* des programmes parallèles dans une étude décrite au chapitre 6.

— ***OneFMUOneNode***

OneFMUOneNode est un placement qui consiste à mettre une seule FMU par nœud multi-cœurs. Ce placement met tous les cœurs du processeur à la disposition des quelques sous-tâches de cette FMU. Grâce à ce placement, on pourra à la fois mesurer la charge exacte des calculs et les communications exactes des FMU car on annule la gêne des FMU entre elles, et on minimise la gêne mutuelle des threads attachés à une même FMU, comme l'illustre la figure 9.

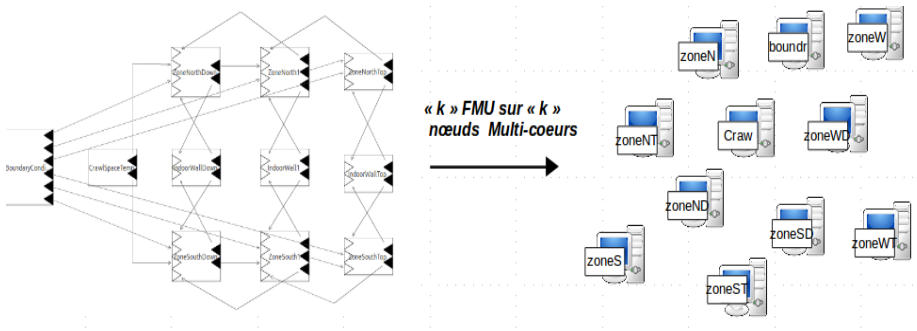


FIGURE 9 – Placement *OneFMUOneNode*

Le but de cette section est d'évaluer la capacité de passage à l'échelle de ce type d'applications en exploitant les nœuds de calcul du cluster visé. Pour réaliser cela, nous avons suivie une démarche expérimentale complète [22], qui comporte pour chaque co-simulation trois étapes :

1. **Caractérisation des FMU**

   Vu que ces FMU sont considérées comme des boîtes grises : leurs structures et leurs fonctionnements internes ne sont pas connus ni par le développeur ni par l'utilisateur de la co-simulation. Néanmoins, on peut avoir quelques connaissances sur ces FMU (informations limitées), soit par l'intermédiaire d'EDF (connaissances métiers concernant la charge de calcul de chaque FMU), soit en parcourant les fichiers XML de ces FMU pour connaître les variables échangées par chaque FMU (notamment leur nombre). Nous caractérisons chaque FMU du système par un

triplet <*temps de calcul*, *temps de contrôle*, *temps de communication*>, tel que le temps de calcul représente le temps mesuré du *DoStep* lancé par le solveur, le temps de contrôle est le temps des échanges verticaux (communications verticales, voir le chapitre 3), et le temps de communications entre FMU (communications horizontales). On a mesuré ces temps en distribuant les « n » FMU du système sur les « n » nœuds multi-cœurs du cluster visé. Ce déploiement est effectué selon le placement de référence *OneFMUOneNode* défini précédemment pour s'assurer que toutes les FMU du système ne se perturbent pas entre elles dans un même nœud, et les threads associés aux différentes FMU ne se gênent pas entre eux. De plus, nous utilisons dans nos expérimentations le mode d'orchestration «*ordered*» (voir la le chapitre 3) pour mesurer avec précision les temps de communication car le deuxième mode «*overlapped*» lance les communications et le contrôle en parallèle ce qui fausserait nos mesures de performances.

2. **Optimisation expérimentale de la distribution d'un problème donné**
   Cette étape s'intéresse aux comportements de l'ensemble des FMU du système sur un cluster de PC multi-cœurs, en prenant en compte le nombre de FMU se trouvant sur le même nœud de calcul, et en appliquant diverses stratégies de répartitions. Après plusieurs essais, nous avons trouvé des répartitions qui ont donné de bonnes performance pour un problème donné (ces dernières vont être abordés dans les sections suivantes). Cette étape a donc permis de trouver des solutions pour un problème élémentaire de petite ou de moyenne taille.

3. **Expérience de passage à l'échelle par réplication**
   Cette étape s'intéresse aux applications de grandes tailles. Mais, EDF ne nous avait alors fourni que de moyens *use-cases* qui ne dépassaient pas une vingtaine de FMU. Pour remédier à cette limitation, nous avons procédé par une stratégie de réplication pour agrandir la taille de nos *use-cases*. Cette stratégie de réplication nous a permis d'augmenter à la fois la taille des problèmes élémentaires de « 1 » à « n » problèmes (par exemple : le nombre de bâtiments simulés), et dans les mêmes proportions le nombre de ressources informatiques réservées. Afin de réaliser un passage à l'échelle, on cherche à conserver le temps de toute la co-simulation constant quelque soit la taille de problème (un *size up* idéal).

   Soit T($1 \times q_1$, $1 \times n_1$) la meilleure répartition d'un problème élémentaire q_1 sur n_1 nœuds physiques. Pour réaliser un passage à l'échelle, on doit distribuer des problèmes de taille $(k \times q_1)$ sur $(k \times n_1)$ nœuds physiques ($n_1 \in \mathbb{N}$, $k \in \mathbb{N}$), tout en essayant de maintenir des temps d'exécution constants. Cela nous mène à garantir

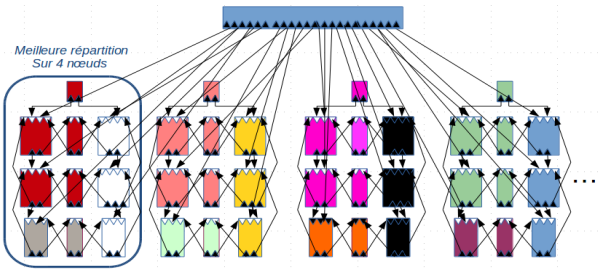


FIGURE 10 – Stratégie de réplication appliquée sur un *use-case* fourni par EDF.

la condition de passage à l'échelle sur nos clusters qui est :

$$T(k \times q_1, k \times n_1) = T(1 \times q_1, 1 \times n_1) = C^{ste} \tag{5.8}$$

Cette étude manuelle a confirmé la faisabilité d'un tel passage à l'échelle de nos applications basées sur des FMU sur des cluster de PC multi-cœurs. Toutefois, il est apparu nécessaire d'automatiser la répartition des FMU, en développant des algorithmes de *mapping*, car le temps écoulé pour réaliser cette démarche expérimentale a été très long.

4 Approches proposées de *mapping* de FMU sur cluster

4.1 Approche par heuristiques élémentaires

Nécessitant seulement des pré-études des graphes de FMU.

Au début de la thèse, nous avons testé quelques heuristiques sur deux premiers *use-cases* de petites tailles. Cette présente approche exploite les données et les informations que le graphe DACCOSIM contient, en équilibrant la charge entre les différents nœuds de calcul réservés. Ces heuristiques étaient les suivantes :

— ***nbIO*** (nombre d'entrées/sorties) : le poids d'une FMU est représenté par le nombre total de ses connecteurs d'entrées et de sorties présents dans le modèle DACCOSIM. En gros, cette heuristique commence à attribuer les plus grosses FMU (qui ont un *nbIO* élevé) à des nœuds multi-cœurs. Les différentes FMU déjà placées seront retirées ensuite de la liste initiale. Le processus se répète tant que la liste initiale n'est pas vide, en surchargeant les nœuds déjà utilisés. Dans ce cas, on considère que les *nbIO* représentent le coût de communication de chaque FMU.

— **L** (localité) : On met sur un même nœud multi-cœurs des FMU qui s'échangent beaucoup de données entre elles (on regroupe les FMU qui communiquent beaucoup), ce qui demande d'étudier le graphe de FMU (voir la solution de *clustering*

dans la sous-section suivante).

— **R** (*aléatoire*) : C'est une stratégie aléatoire, elle permet de placer les FMU sur des nœuds multi-cœurs de façon arbitraire (généralement elle suit l'algorithme *Round Robin* appliqué à une liste sans ordre).

Nous nous sommes appuyés sur des séries de *benchmarks* pour évaluer les solutions proposées. Les résultats de ces heuristiques sur les petits *use-cases* fournis par EDF montrent que l'heuristique ***nbIO*** donne les meilleures performances (voir le chapitre 6).

4.2 Approche par heuristiques complexes

Nécessitant des pré-études des graphes de FMU et des *benchmarks* sur le cluster visé.

Nous nous sommes intéressés, par la suite, à étudier le passage à l'échelle des applications parallèles basées sur des FMU en améliorant les heuristiques présentées précédemment par la prise en compte de leur comportement sur un cluster de PC multi-cœurs. Pour réaliser cela, EDF nous a fabriqué des *use-cases* de moyennes et de grandes tailles en augmentant à la fois le poids de calcul (nombre d'équations de chaque FMU) et le coût de communication de chaque FMU (nombre de communications entre les différentes FMU). En fait, nous avons mesuré précisément le poids de calculs et le temps de communications de chaque FMU en appliquant le placement de référence *OneFMUOneNode* qui met une seule FMU sur un nœud multi-cœurs en mode *ordered* où il n'y a pas de chevauchement entre les phases d'exécution. Grâce à ces expérimentations, nous avons pu conclure que l'augmentation des performances dépend principalement de l'équilibrage de charge des calculs et de la limitation du coût des communications en inter-nœuds.

Mais avant de détailler nos solutions de répartition d'un graphe de FMU sur une architecture distribuée de PC multi-cœurs, nous décrivons ci-dessous les différentes fonctions que nous allons utiliser dans nos algorithmes :

♢ **sortD(List<FMU>)** permet de retourner une liste triée de façon descendante selon le temps de calcul (*ComputTime*).

♢ **sortA(List<FMU>)** permet de retourner une liste des nœuds triée selon la charge des nœuds de façon ascendante. Cette fonction retire les nœuds saturés de la liste triée.

♢ **getNbrFMUOnOneNode()** retourne le nombre maximal de FMU qu'on peut mettre sur le même nœud multi-cœurs, elle est calculée selon l'équation 5.7 présentée dans la section 3.1.

♢ **max(List<FMU>)** retourne un couple $< indice, card_{FMU} >$ où « *indice* » est l'indice de la FMU dans la liste initiale des FMU, et où $card_{FMU}$ contient la cardinalité [3] la plus élevée de toutes les FMU.

♢ **remove(List<FMU>, FMU)** retourne une liste des FMU (tâches) mise-à-jour sans la FMU spécifiée.

♢ **search(i, List<FMU>)** vérifie si la FMU de l'indice i est déjà placée sur un nœud de calcul ou non. Si cette fonction trouve une FMU de type $FMU_{default}$, elle retourne $False$ (tâche déjà assignée), sinon elle retourne un $True$ (FMU pas encore placée). Notons que $FMU_{Default}$ représente une FMU vide avec un poids nul.

♢ **sortFamily(List<FMU>)** permet de trier la liste initiale des FMU selon leur type (famille). Elle retourne un object de type ($List < List < FMU >>$).

♢ **append(E, List<FMU>)** permet d'insérer l'élément E à la fin de la liste.

♢ **getFirst(List<FMU>)** pour récupérer et supprimer le premier élément de la liste initiale des FMU.

♢ **residualComm(List<List<FMU»)** retourne $True$ si les FMU placées dans des nœuds de calcul sont connectées avec d'autres FMU non placées. Dans le cas où il n'y a plus de communications, cette fonction retourne $False$.

Après plusieurs expérimentations de différentes heuristiques, nous avons pu déterminer et extraire, dans cette section, les heuristiques que nous trouvons efficaces et rapides à établir pour traiter nos *use-cases* à base de FMU. D'autre part, dans le chapitre suivant, nous évaluons la capacités de ces heuristiques sur de grands *use-cases* métiers. Ces heuristiques sont définies comme suit :

— ***Load Balancing (LB)***

Load Balancing regroupe les deux heuristiques classiques présentées dans le chapitre 2 : OLB [4] et max-min. Cette heuristique se base sur le temps de calcul de chaque FMU. Autrement dit, on doit d'abord exécuter les « k » FMU du système sur les « k » nœuds multi-cœurs du cluster en appliquant le placement de référence *OneFMUOneNode* (une FMU par un nœud multi-cœurs) afin de mesurer le temps de calcul de chaque FMU. L'heuristique LB permet ensuite d'équilibrer la charge entre les différents ressources en commençant par assigner les « n » grosses FMU aux « n » nœuds multi-cœurs afin de ne pas retarder l'exécution de tout le système. Elle prend en considération le nombre de ressources de calcul réservées et le nombre

3. Une cardinalité est le nombre total de données envoyées sur le même lien de communication.
4. Opportunistic Load Balancing

de FMU qui se retrouveront sur un même nœud, comme l'illustre la figure 11.

Algorithm 4 ALGORITHME *Load Balancing* (BOOLEAN *ImposedNode*, LIST<FMUTYPE> *ListFMU*) → *List<List<FMUType> > ListNodeFMU*

Paramètre d'entrée *: ImposedNode qui indique si l'utilisateur a imposé le nombre de ressources (True) ou non (False)*
Paramètre d'entrée *: ListFMU qui représente la liste initiale des FMU*
Paramètre retourné *: ListNodeFMU représente une liste des nœuds dont chacune contient une liste des FMU*

Type *: FMUType est un couple <Entier rangFMU, Double ComputTime>*
Variables locales :
Entier $j \leftarrow 0$, $k \leftarrow 0$, *nbrFMUOneNode*, *nbrNode*, *n*
List<double *ComputNodeLoad*, entier *index*> *LoadNode* /* la charge des nœuds */

Instructions :
/* **Calculer le nombre de nœuds de calcul** */
/* **Calculer le nombre de FMU qui se trouvent sur le même nœud** */
if $(ImposedNode = false)$ **then**
  $nbrFMUOneNode \leftarrow getNbrFMUOnOneFMU()$
  $nbrNode \leftarrow \lceil \frac{ListFMU.Size}{nbrFMUOneNode} \rceil$
else
  $nbrFMUOneNode \leftarrow \lceil \frac{ListFMU.Size}{nbrNode} \rceil$
end if
$ListFMU \leftarrow sortD(ListFMU)$
/* **Placer les *nbrNode* grosses FMU sur les *nbrNode* nœud multi-cœurs** */
while $(k < nbrNode)$ **do**
  $ListNodeFMU[k].append(ListFMU[0])$
  $LoadNode[k].ComputNodeLoad \leftarrow ListFMU[0].ComputTime$
  $LoadNode[k].index \leftarrow k$
  $remove(ListFMU[0])$
  $k++$
end while
/* **Placer le reste des FMU sur les *nbrNode* nœud multi-cœurs** */
while $(ListFMU.Size \neq 0)$ **do**
  $LoadNode \leftarrow sortA(LoadNode)$
  /* **Équilibrer la charge entre les nœuds de calcul** /*
  $n \leftarrow j \mod nbrNode$
  if $(ListNodeFMU[LoadNode[n].index].size < nbrFMUOneNode)$ **then**
    $ListNodeFMU[LoadNode[n]].append(ListFMU[0])$
    $LoadNode[n].ComputNodeLoad \leftarrow LoadNode[n].ComputNodeLoad + ListFMU[0].ComputTime$
    $remove(ListFMU[0])$
  end if
  $j++$
end while

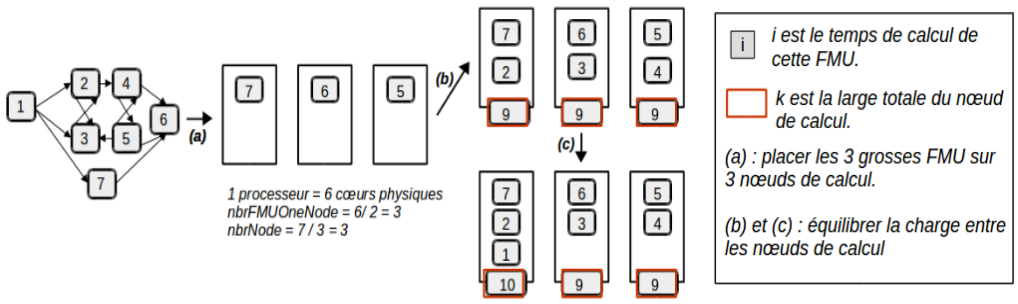


FIGURE 11 – Exemple d'une répartition sur trois nœuds multi-cœurs du cluster *Sarah* de CentraleSupélec en appliquant la stratégie LB et un $nbrFMUOneNode = nbrPhysicalCores/2$

Comme l'indique l'algorithme 4, si l'utilisateur définit à l'avance le nombre de ressources « $nbrNode$ », l'algorithme peut calculer le nombre maximal de FMU qui peuvent se mettre sur la même machine. Mais dans le cas contraire, l'algorithme détermine, tout d'abord, le nombre de FMU supportées efficacement par un nœud multi-cœurs et en déduit le nombre optimal de ressources qu'on doit réserver pour traiter un problème donné qui comporte « k » FMU.

Cet algorithme commence par répartir les « n » premières grosses FMU sur les « n » nœuds de calcul multi-cœurs en retirant de la liste initiale ($ListFMU$) les FMU qui sont déjà placées sur les nœuds de calcul. Ensuite, nous assignons les FMU restantes et non placées sur les nœuds les plus légers en poids de calcul. Le but dans cette deuxième partie de l'algorithme est d'équilibrer la charge entre ces ressources réservées tout en garantissant sur le même nœud de calcul un nombre de FMU égal à « $nbrFMUOneNode$ ». En outre, dans le cas où le nœud est saturé, nous passons au prochain nœud non saturé léger en poids, cette partie s'exécute de manière itérative tant que la liste initiale des FMU n'est pas vide, et en calculant à chaque itération le rang (ou l'indice) de la machine la plus légère en calcul.

— **Regroupement des communications (*ClusteringComms*)**

Nos applications sont des graphes de FMU où chaque FMU interagit avec d'autres FMU. Le but de cette heuristique est de regrouper et de placer les FMU qui communiquent entre elles sur le même nœud multi-cœurs. Autrement dit, elle permet de réduire le temps global d'une co-simulation par la minimisation des communications entre nœuds de calcul (les communications inter-nœuds).

Le principe de cet algorithme se fonde sur un regroupement hiérarchique, c'est-à-dire qu'on commence par placer de manière aléatoire « n » FMU du système sur les « n » nœud de calcul multi-cœurs. Ensuite, on vérifie pour chaque FMU déjà placée si elle est connectée avec d'autres FMU du système grâce à la matrice des communications. Si une FMU x est connectée avec plusieurs FMU du système, on

prend la première FMU qui lui est connectée et n'est pas encore placée, dans le but de la mettre sur le même nœud multi-cœurs non saturé que la FMU x. Ce processus se fait de manière itérative sur les nœuds de calcul. Pour l'itération suivante $(i+1)$, on traite les FMU restantes dans la liste initiale, qui sont connectées avec les FMU déjà placées lors de l'itération i. Pour détecter les connexions entre FMU, on se sert d'une matrice de communications (ou de cardinalité) que l'on a récupérée de l'étape de co-initialisation ; elle construit un graphe de dépendance en s'appuyant sur les dépendances internes et externes des sorties et des entrées de l'ensemble des FMU. Cette heuristique *ClusteringComms* est détaillée sur l'algorithme 5 et sur la figure 12.

Algorithm 5 ALGORITHME DE *ClusteringComms*(BOOLEAN *ImposedNode*, LIST<FMUTYPE> *ListFMU*, ENTIER [ ][ ] CARD) → *List<List<FMUType> > ListNodeFMU*

Paramètre d'entrée *: ImposedNode qui indique si l'utilisateur a imposé le nombre de ressources (True) ou non (False)*
Paramètre d'entrée *: ListFMU qui représente la liste initiale des FMU.*
Paramètre d'entrée *: Card qui représente une matrice de communications.*
Paramètre retourné *: ListNodeFMU représente une liste des nœuds dont chacune contient une liste des FMU*

Type *: FMUType est un couple <Entier rangFMU, Double ComputTime>*
Variables locales :
Entier $j \leftarrow 0$, $i \leftarrow 0$, $k \leftarrow 0$, $n \leftarrow 0$, $nbrFMUOneNode$, $nbrNode$
Boolean $NoComm$
List<FMUType> FMU /* liste des FMU se trouvant sur le même nœud */

Instructions :
$NoComm \leftarrow False$
/* **Calculer le nombre de nœuds de calcul** */
/* **Calculer le nombre de FMU qui se trouvent sur le même nœud**/
if $(ImposedNode = false)$ **then**
  $nbrFMUOneNode \leftarrow getNbrFMUOnOneFMU()$
  $nbrNode \leftarrow \lceil \frac{ListFMU.Size}{nbrFMUOneNode} \rceil$
else
  $nbrFMUOneNode \leftarrow \lceil \frac{ListFMU.Size}{nbrNode} \rceil$
end if
/* **Répartir une FMU sur un nœud de calcul de façon aléatoire** */
while $(k < ListNodeFMU.size)$ **do**
  $ListNodeFMU[k].append(ListFMU[0])$
  $ListFMU[k] \leftarrow FMU_{Default}$
  $k++$
end while
***/ Placer les FMU tant que la liste initiale n'est pas vide */**

```
while (k < ListFMU.Size) do
  n ← 0
  if (ListNodeFMU[k mod nbrNode].size < nbrFMUOneNode) then
    i ← 0
    while (i < ListNodeFMU[k mod nbrNode].size)) do
      j ← ListNodeFMU[k mod nbrNode].append(FMU[i].rangFMU)
      */ Placer les FMU qui sont connectées avec celles déjà placées */
      while (n < ListFMU.size)) do
        */ Traiter une communication de toutes les FMU du nœud */
        if ((Card[j][n] ≠ 0) || (NoComm = True) then
          ListNodeFMU[k mod nbrNode].append(ListFMU[k])
          ListFMU[k] ← FMU_Default
          Card[j][n] ← 0
          i ← ListNodeFMU[k mod nbrNode].size
        else
          i++
        end if
        n++
      end while
    end while
    */ Vérifier s'il y a des FMU non placées qui ne sont pas connectées */

    if ((residualComm(ListNodeFMU[nbrNode]) == False) && (k <
    ListFMU.Size)) then
      NoComm ← True
    else
      k++
    end if
  end if
end while
```

Cet algorithme ne peut mettre au maximum que $nbrFMUOneNode$ FMU dans le même nœud multi-cœurs. Dans le cas où il y a plus de FMU connectées aux FMU déjà placées, l'algorithme assigne finalement les FMU non placées à des nœuds qui ne sont pas encore saturés ($nbrFMU < nbrFMUOneNode$).

— ***LoadBalancingAndClusteringComms*** (LBCComms)

Le temps global du placement ne dépend pas uniquement des coûts des calculs, mais des coûts des communications. C'est pourquoi nous proposons cette heuristique qui regroupe les avantages des deux propositions présentées précédemment : celle de *Load Balacing* et celle de *ClusteringComms*. En fait, l'heuristique $LBCComms$ permet de répartir les FMU sur l'ensemble des nœuds pour équilibrer au mieux leurs charges de calculs, mais en regroupant les FMU qui communiquent entre elles pour minimiser au mieux les communications inter-nœuds. Cette heuristique tient donc

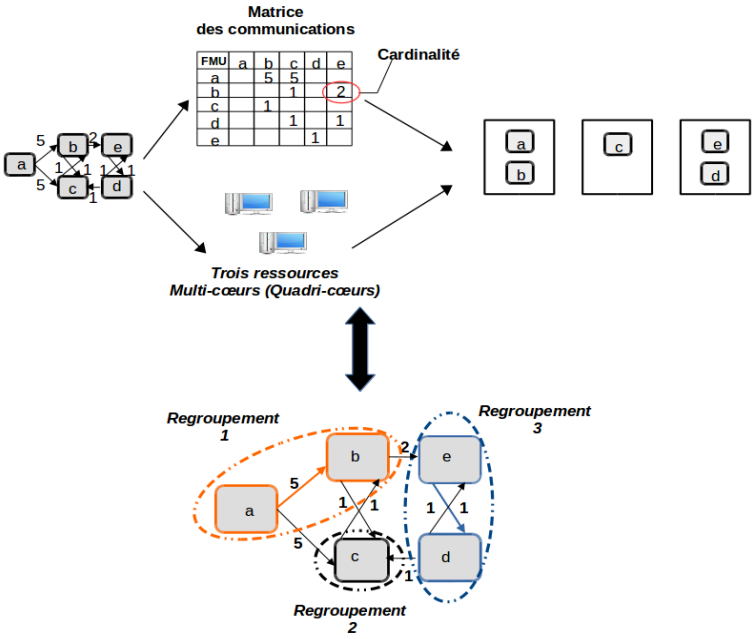


FIGURE 12 – Exemple de placement en appliquant l'heuristique complexe de *ClusteringComms* et un $nbrFMUOneNode = nbrPhysicalCores/2$.

compte de la cardinalité des messages échangés entre FMU. Après avoir répartit les FMU les plus gourmandes en calcul, nous balayons le graphe DACCOSIM en cherchant à chaque étape à regrouper les FMU restantes non placées qui sont connectées avec celles déjà placées et qui ont une cardinalité élevée. Le détail de cette heuristique complexe se fait dans l'algorithme 6, et est illustré sur la figure 13. Comme la stratégie de *Load Balancing*, cet algorithme commence par attribuer les « n » grosses FMU sur les « n » ressources informatiques (de la $17^{ème}$ ligne à la $22^{ème}$ ligne). Ensuite, il regroupe les FMU restantes qui échangent beaucoup de données entre elles (qui ont une cardinalité élevée) afin de les mettre sur le même nœud de calcul (une FMU par un nœud multi-cœurs). En fait, cette stratégie permet de regrouper les communications importantes afin de minimiser les communications qui traversent le réseau d'interconnexion, tout en équilibrant la charge entre les machines.

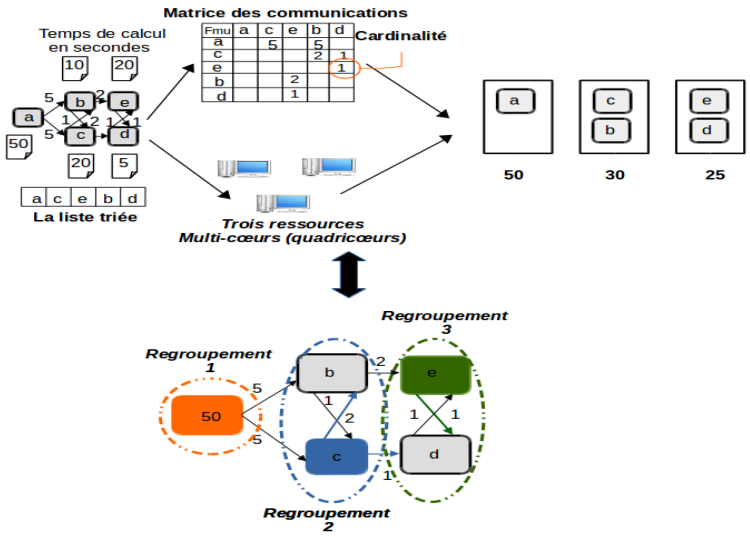


FIGURE 13 – Heuristique complexe *LBCComms*

4.3 Approches automatiques

Ne nécessitant aucune pré-étude de graphe ni aucun *benchmarking*.

Pour une co-simulation DACCOSIM, nous avons d'un côté un graphe de boîtes hétérogènes qui communiquent fréquemment entre elles, et d'un autre coté un graphe de ressources (nœuds) comportant chacune un ensemble fini de cœurs physiques. Avec les heuristiques présentées précédemment, on était obligé de réaliser une campagne d'expérimentations afin de déterminer le poids de chaque FMU du système pour réaliser ensuite un placement fin des FMU sur un cluster de PC multi-cœurs. Avec l'approche automatique, on souhaite au contraire mettre au point des stratégies qui donnent des solutions avec des temps de co-simulation non optimaux mais rapides sans avoir à prè-exécuter le problème sur cluster. Pour réaliser cela, nous nous appuyons sur des algorithmes qui nécessitent moins d'informations sur les FMU.

— ***Round Robin*** : Cet algorithme permet d'assigner chaque tâche à une ressource informatiques multi-coeurs selon un ordre arbitraire (sans se soucier des poids des FMU). Ce processus itère jusqu'à ce que la liste initiale des FMU soit vide.

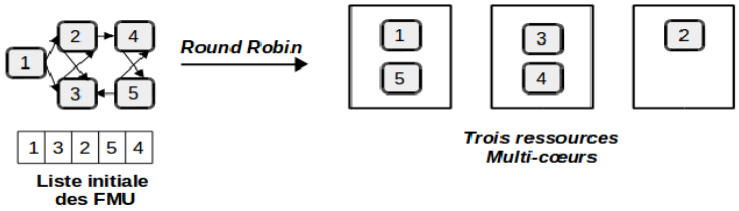


FIGURE 14 – Stratégie de *Round Robin*.

Comme l'illustre la figure 14, nous avons un exemple d'un graphe de tâches comportant cinq tâches hétérogènes, et un graphe de ressources multi-coeurs contenant trois machines informatiques. Quand on applique l'algorithme itératif de *Round Robin* (RR), on assigne chaque FMU à une ressource informatique et on repart de la première ressource quand on a atteint la dernière. On s'arrête quand on n'a plus de FMU dans la liste initiale des FMU à placer.

L'algorithme *Round Robin* (algorithme 7) présente le placement des FMU sur un cluster de PC multi-coeurs. À partir de la ligne 15, cet algorithme attribue chaque FMU à une ressource informatique, tant que la liste initiale $ListFMU$ n'est pas vide. À chaque itération, il place une FMU puis il l'a retire de la liste initiale des FMU non placées (les lignes 16 et 17).

— **Family Round Robin (FRR)** : Cet algorithme s'appuie sur des connaissances « métiers » des développeurs et utilisateurs du graphe de co-simulation. Par exemple, des chercheurs d'EDF R&D ont une expérience significative de simulation de leurs réseaux électriques et connaissent les FMU qui ont des poids similaires : ils peuvent facilement les regrouper en listes homogènes. Donc, avec l'aide d'EDF R&D, nous avons pu identifier « k » listes de FMU (familles) dans nos simulations de réseaux électriques, dont chacune regroupe des FMU qui ont à peu près le même poids de calcul.

La mise en œuvre de cette stratégie s'appuie sur un fichier externe de « Métadonnées » qui contient les types de base des FMU associées à leurs familles. L'algorithme (FRR) permet de placer les FMU du système en essayant de mettre sur un même nœud multi-cœurs une FMU de chaque famille. Le but de cet algorithme est en fait d'équilibrer la charge entre nœuds de calcul en se basant sur la famille de la FMU mais sans pré-exécuter l'application sur cluster de PC.

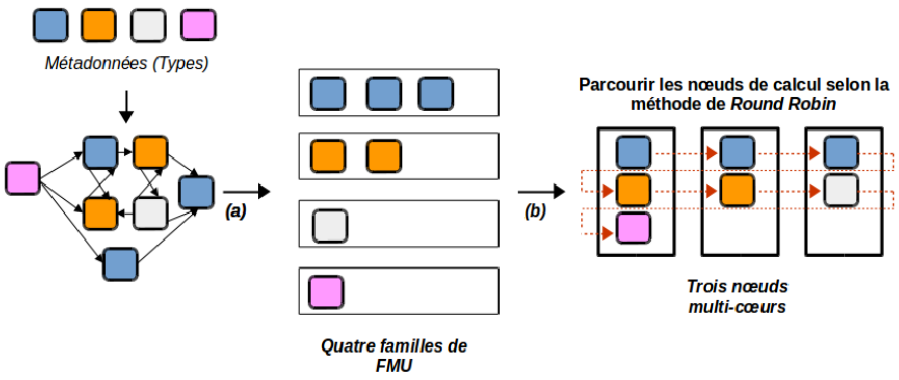


FIGURE 15 – Stratégie de *Family Round Robin*.

Dans l'exemple présenté dans la figure 15, nous classons les sept FMU sur trois nœuds physiques selon leur type de FMU. Ce processus commence par répartir les FMU de la première famille et il s'enchaîne par les FMU de la deuxième famille jusqu'à vider tous les vecteurs.

L'algorithme 8 *Family Round Robin* commence par trier les FMU dans des listes de FMU selon leur type. Une catégorie de FMU représente le poids de calcul des FMU du point de vue « métier ». Cette stratégie commence par attribuer les FMU de la première liste de FMU sur les ressources informatiques jusqu'à ce que la liste devienne vide. Ensuite, il enchaîne en affectant les autres listes de FMU l'une après l'autre sur les ressources informatiques. Cet algorithme réalise un équilibrage de charge approximatif entre les nœuds. Nous verrons dans le chapitre 6 qu'il s'est révélé très efficace sur nos grands *use-cases*.

Algorithm 6 ALGORITHME *LoadBalancingAndClusteringComms*(BOOLEAN *ImposedNode*, LIST<FMUTYPE> *ListFMU*, ENTIER [ ][ ] CARD) → *List<List<FMUType»* *ListNodeFMU*

Paramètre d'entrée *: ImposedNode qui indique si l'utilisateur a imposé le nombre de ressources (True) ou non (False)*
Paramètre d'entrée *: ListFMU qui représente la liste initiale des FMU.*
Paramètre d'entrée *: Card qui représente une matrice de communications des FMU*

Paramètre retourné *: ListNodeFMU représente une liste des nœuds dont chacune contient une liste des FMU*

Type *: FMUType est un couple <Entier rangFMU, Double ComputTime>*
Variables locales :
Entier $j \leftarrow 0$, $k \leftarrow 0$, $i \leftarrow 0$, $nbrFMUOneNode$, $nbrNode$
Couple <Entier *indice*, Entier *card*> Element
Instructions :
if $(ImposedNode = false)$ **then**
  $nbrFMUOneNode \leftarrow getNbrFMUOnOneFMU()$
  $nbrNode \leftarrow \lceil \frac{ListFMU.Size}{nbrFMUOneNode} \rceil$
else
  $nbrFMUOneNode \leftarrow \lceil \frac{ListFMU.Size}{nbrNode} \rceil$
end if
$ListFMU \leftarrow sortD(ListFMU)$
/* placer les *n* grosses FMU sur *n* nœud multi-cœurs */
while $((k < ListNodeFMU.size))$ **do**
  $ListNodeFMU[k].append(ListFMU[0])$
  $ListFMU[k] \leftarrow FMU_{Default}$
  $k++$
end while
while $(k \leqslant ListFMU.Size)$ **do**
  $ListFMUNode \leftarrow sortA(ListFMUNode)$
  if $(ListNodeFMU[k \mod nbrNode].size < nbrFMUOneNode)$ **then**
    $i \leftarrow 0$
    while $(i < ListNodeFMU[k \mod nbrNode].size))$ **do**
      $Element \leftarrow max(card[FMU[i].rangFMU][*])$
      if $((Card[FMU[i].rangFMU][*][Element.indice] \neq 0)$ && $(search(Element.indice, ListFMU) \neq FMU_{Default}))$ **then**
        $ListNodeFMU[k \mod nbrNode].append(ListFMU[indice])$
        $ListFMU[k] \leftarrow FMU_{Default}$
        $Card[k] \leftarrow 0$
        $i \leftarrow ListNodeFMU[k \mod nbrNode].size$
      end if
    end while
    $k++$
  end if
end while

Algorithm 7 ALGORITHME ITÉRATIF *Round Robin*(BOOLEAN *ImposedNode*, LIST<FMU <FMU> > *ListFMU* → *List<List<FMU>* > *ListNodeFMU*

Paramètre d'entrée *: ImposedNode qui indique si l'utilisateur a imposé le nombre de ressources (True) ou non (False)*
Paramètre d'entrée *: ListFMU qui représente la liste initiale des FMU*
Paramètre de sortie *: ListNodeFMU représente une liste des nœuds dont chacune contient une liste des FMU*

Type *: FMU est un String*
Variables locales :
Entier $nbrFMUOneNode$, $nbrNode$
Entier $k \leftarrow 0$

Instructions :
if $(ImposedNode = false)$ **then**
  $nbrFMUOneNode \leftarrow getNbrFMUOnOneFMU()$
  $nbrNode \leftarrow \lceil ListFMU.Size/nbrFMUOneNode \rceil$
else
  $nbrFMUOneNode \leftarrow \lceil ListFMU.Size/nbrNode \rceil$
end if
/* Tant que la liste initiale des FMU à placer n'est pas vide */
while $(ListFMU.Size \neq 0)$ **do**
  $ListNodeFMU[k \mod nbrNode].append(ListFMU[0])$
  $remove(ListFMU[0])$
  $k++$
end while

Algorithm 8 ALGORITHME ITÉRATIF *Family Round Robin*(BOOLEAN *ImposedNode*, LIST<LIST <FMU> > *ListNodeFMU*) → *List<List<FMU> > ListNodeFMU*

Paramètre d'entrée *: ImposedNode qui indique si l'utilisateur a imposé le nombre de ressources (True) ou non (False)*
Paramètre d'entrée *: ListFMU qui représente la liste initiale des FMU*
Paramètre de sortie *: ListNodeFMU représente une liste des nœuds dont chacune contient une liste des FMU*

Type *: FMU est unString*
Variables locales :
List<List <*FMU*> > *FamilyList*
Entier *nbrFMUOneNode*, *nbrNode* , $i \leftarrow 0$, $k \leftarrow 0$

Instructions :
if ($ImposedNode = false$) **then**
  $nbrFMUOneNode \leftarrow getNbrFMUOnOneFMU()$
  $nbrNode \leftarrow \lceil ListFMU.Size/nbrFMUOneNode \rceil$
else
  $nbrFMUOneNode \leftarrow \lceil ListFMU.Size/nbrNode \rceil$
end if
$FamilyList \leftarrow SortFamily(ListFMU)$
/* Placer les FMU selon leur type de FMU /*
while ($i < FamilyList.Size$) **do**
  while ($FamilyList[i].Size \neq 0$) **do**
    if ($ListNodeFMU[k \mod nbrNode].size < nbrFMUOneNode$) **then**
      $ListNodeFMU[k \mod nbrNode].append(FamilyList[i].getFirst())$
      k++
    end if
  end while
  i++
end while

5 Élaboration et utilisation d'un modèle de temps approximatif

Dans les sections précédentes de ce chapitre, nous avons vu que les tâches traitées sont des boites grises dont on ne connaît ni leurs poids ni leurs temps de communications. Pour extraire ces informations, nous étions obligé d'exécuter les FMU du système, dans une phase préliminaire, afin de mesurer à la fois les temps de calcul et les temps de communication de chacune d'elles. Cette phase était nécessaire pour déterminer les données d'entrée des algorithmes de placement (les heuristiques élémentaires et les heuristiques complexes). Toutefois, il n'est pas toujours pratique d'exécuter cette étape préliminaire à chaque fois qu'on veut faire des expérimentations. C'est pour cette raison que nous consacrons cette section à une étude plus fine permettant de proposer deux modèles qui *estiment* les temps de calcul et le temps de communications de chaque FMU, en utilisant le mode d'orchestration *Ordered* (voir le chapitre 3).

5.1 Fonctionnement avec une synchronisation relaxée

Comme l'indique l'équation 5.1 de la section 2.2, le temps d'exécution d'une FMU i est la somme d'un temps de calcul T_{Calcul}, d'un temps de contrôle $T_{Control}$ (temps passé entre les masters locaux et le master global), et d'un temps de communication T_{Comm} (temps de communication entre FMU en inter-nœuds). Sachant que le phase de contrôle englobe les traitements suivants : (a) le master global reçoit toutes les décisions des masters locaux, (b) il calcule le minimum de toutes ces propositions (traitements), et (c) il envoie la décision globale aux masters locaux. L'étape b est négligeable par rapport aux envois/réceptions des messages qui traversent le réseau d'interconnexion, c'est pour cela qu'on ne l'a pas modélisée.

Nous rappelons que le schéma d'exécution d'une co-simulation DACCOSIM s'appuie sur les étapes suivantes :

— Toutes les FMU exécutent leurs calculs de manière simultanée pendant un pas de temps.

— À la fin de ses calculs, chaque FMU vérifie la précision de ses résultats et informe de son souhait (continuer ou revenir en arrière) le « master local » qui se trouve au sein du même nœud. Ce « master local » agrège ensuite les décisions de toutes ses FMU et remonte sa décision finale au « master global ».

— Le « master global » agrège à son tour les décisions des « masters locaux » afin de retenir le minimum des pas de temps proposés (et revenir en arrière dés qu'un master local l'a demandé). Le master global communique ensuite sa décision globale à tous les masters locaux. Dès la réception de cette réponse, chaque « master local » autorise ses FMU à exécuter la décision du « master global » (poursuite des calculs ou retour en arrière, avec le pas de temps indiqué).

Pour la prise de décision globale, DACCOSIM s'appuie sur une synchronisation relaxée au niveau du master global pour garantir la prise en compte des demandes de toutes les FMU de la grappe. Néanmoins, cette synchronisation représente un goulot d'étranglement qui affecte les performances des exécutions distribuées de DACCOSIM, et qui complexifie la mesure des phases de calcul, de contrôle et de communication, car cette synchronisation engendre des temps d'attente pour chaque *wrapper* de FMU.

Pour poursuivre notre modélisation, nous proposons de découper le schéma d'exécution de DACCOSIM en deux parties, voir la figure 16 :

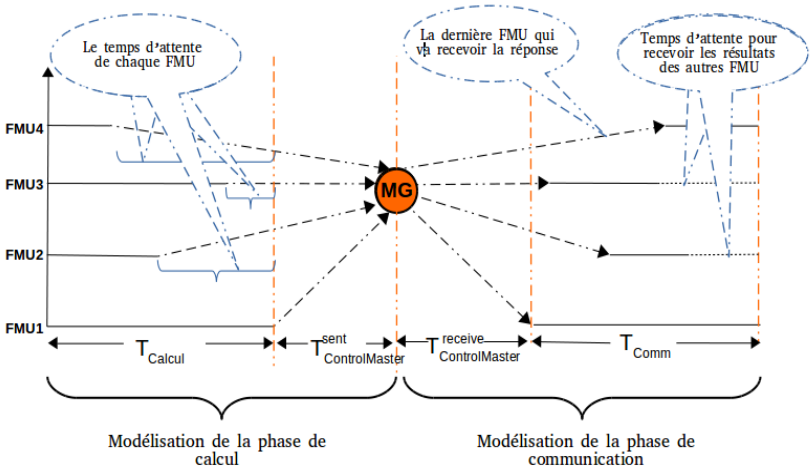


FIGURE 16 – Modélisation d'un temps de co-simulation pour une itération donnée

— Modélisation de la phase de calcul ($T_{ComputPhase}$) qui regroupe les calculs lancés par les solveurs numériques des FMU du système, et l'envoie des décisions au master global pour déterminer le prochain pas de temps (cette étape est le début le contrôle).

— Modélisation de la phase de communication ($T_{CommPhase}$) qui regroupe l'envoie de la réponse du master global aux différentes FMU du système (la fin du contrôle), et l'envoie/réception des résultats des FMU connectées entre elles.

Pour chaque modélisation, nous avons réalisé deux méthodes : expérimentale sur cluster de PC multi-cœurs, et théorique en appliquant les deux modèles approximatifs. Ils vont être expliqués dans la suite de ce chapitre.

Afin d'alimenter les modèles qui vont être présentés dans les sections suivantes, nous utilisons les paramètres ci-après :

Variable	**Définition**
h_i	le pas de temps *constant*[5] à l'itération i.
Δd	la durée de toute la simulation.
$iterations$	le nombre total d'itérations ($\lceil \frac{\Delta d}{h} \rceil$) .
$Size_{msg}$	la taille du message d'une seule donnée.
Max_{E-S}	le maximum entre les connexions d'entrée et celles de sortie d'une FMU.
Bw	la bande passante du réseau de communication.
N	le nombre de nœuds de calcul réservés.
K	le nombre total de FMU dans la co-simulation
$card$	la cardinalité d'un message envoyé, représente le nombre de données envoyées sur un même lien.
$latence$	$latence$[6] entre deux JVM[7] sur deux machines du cluster.

TABLE 5.1 – Les paramètres utilisés par le modèle de communication.

5.2 Modélisation de la phase de calcul

Comme déjà expliqué dans le chapitre 2, il est impossible dans le modèle de DAC-COSIM d'obtenir des informations concernant le nombre d'instructions des FMU du système : les codes sources des FMU ne sont pas accessibles. Le but de cette section est de modéliser la phase de calcul global $T_{ComputPhase}$ de la grappe physique de FMU, qui comprend les calculs de l'ensemble des FMU et leurs temps d'envoi de messages au master global pour une itération donnée. Il s'agit donc de modéliser une première partie du contrôle (pendant laquelle le master global reçoit les décisions des masters locaux). Puisque les FMU et leur master local se trouvent sur le même nœud physique, leurs temps de communications (intra-nœud) sont considérés nuls, comme l'illustre la figure 17.

5. $h_i = h$, on utilise cette notation par homogénéité avec les chapitres/sections précédents/précédentes.

6. Le temps nécessaire pour envoyer un octet qui se trouve dans un code Java sur une machine « A » vers la machine « B » en traversant le réseau d'interconnexion.

7. JVM est une instance de machine virtuelle Java qui est hébergée et gérée par un nœud de calcul physique.

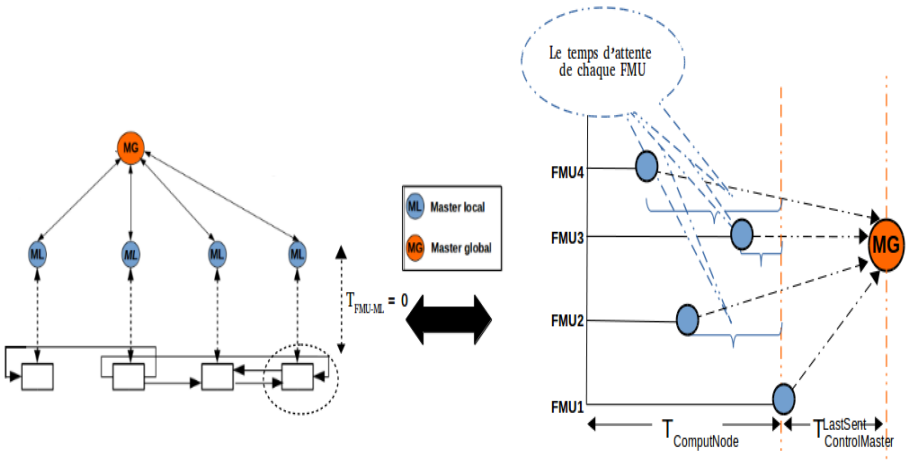


FIGURE 17 – Modélisation de la phase de calcul globale d'une grappe d'une FMU

Puisqu'on a des FMU hétérogènes, chacune d'elles lance ses calculs indépendamment des autres FMU. Si ses calculs sont légers, on doit ajouter un temps d'attente (T_{wait}) avant que le master global n'ait reçu les messages de tous les masters locaux du système, et ne commence à en déduire sa décision globale. Ce T_{wait} est différent pour chaque FMU, qui a fini ses calculs, et qui doit attendre la fin des calculs de la plus grosse FMU du système. Il est donc difficile de déterminer (calculer) la valeur exacte des temps d'attente de chaque FMU car ils dépendent des calculs des autres FMU. Pour contourner cette difficulté, nous modélisons cette phase de calcul du point de vue de la grappe de FMU, et on prendra le maximum des temps de calcul de tous les masters locaux qui prennent eux-mêmes le temps de calcul du FMU le plus long, ce qui absorbe (cache) les temps d'attente des FMU légers. On notera $T_{Calcul}^{TheLongest}$ le temps d'une phase de calcul du FMU le plus long de toute la grappe.

Si une des FMU est gourmande en calcul et nécessite plus de temps pour effectuer ses calculs, dés qu'elle les termine, elle envoie le dernier message au master de DACCOSIM. Nous pouvons donc formaliser le temps de la phase de calcul de l'itération i par :

$$T_{ComputPhase} = T_{Calcul}^{TheLongest} + T_{ControlMaster}^{LastSent} \tag{5.9}$$

$$= \max_{1 \leq k \leq K} T_{Calcul}(k) + latence \tag{5.10}$$

En revanche, si les FMU ont presque les mêmes temps de calcul, elles terminent toutes presque en même temps. À l'issue des calculs, les $(N-1)$ masters locaux envoient chacun un message qui traverse un réseau d'interconnexion au « master global » de la co-simulation. On a mis $(N-1)$ masters locaux car l'un d'eux est lui même le master global

de toute la co-simulation. Le master global reçoit donc $(N-1)$ messages séquentiellement, et le temps de la première partie de l'itération est formalisée par :

$$T_{ComputPhase} = T_{Calcul}^{TheLongest} + (N-1) \times (T_{ControlMaster}^{LastSent}) \quad (5.11)$$

$$= \max_{1 \leq k \leq K} T_{Calcul}(k) + (N-1) \times latence \quad (5.12)$$

Donc le pire cas des équations 5.10 et 5.12 correspond à l'équation 5.12, et on a :

$$T_{ComputPhase}^{WorstCase} = \max_{1 \leq k \leq K} T_{Calcul}(k) + ((N-1) \times latence) \quad (5.13)$$

Tel que le N est le nombre des masters locaux (nœuds de calcul réservés).
Afin de calculer la phase de calcul pour toute la co-simulation, on multiplie l'équation 5.13 par le nombre d'itérations comme suit :

$$T_{ComputPhaseCoSimul}^{WorstCase} = T_{ComputPhase}^{WorstCase} \times iterations \quad (5.14)$$

Enfin, étant donné que la seule information fournie par EDF était le nombre $d'équations$[8] de chaque FMU, nous l'exploitons en modélisant le temps de calculs des solveurs en fonction du nombre d'équations. Dès lors, nous avons tracé une courbe des temps de calculs en fonction du nombre d'équations $T_{Calcul} = f(nbr_{Eq})$ et nous avons trouvé une relation de la forme $T_{Calcul} = a \times Nb_{Eq} + b$. Cela nous a permis de modéliser les calculs de chaque FMU du système, c'est-à-dire de modéliser le premier terme de l'équation 5.9, en prenant le maximum des temps de calcul de tous les masters locaux (cette étape sera expérimentée dans le chapitre 6).

5.3 Modélisation de la phase de communication

Dans la plate-forme DACCOSIM, toutes les FMU du système font une transition d'une phase de calcul vers une phase de communication en passant par le master global de la co-simulation. Dans cette section, nous modélisons la phase de communication qui comporte l'envoi de la réponse du « master global » vers les masters locaux ($T_{ControlMaster}^{receive}$) et les temps d'interactions entre les FMU connectées entre elles. Il est à noter que les messages échangés entre les nœuds de calcul traversent le réseau d'interconnexion, c'est pour cela qu'on a rajouté des paramètres réseau pour modéliser cette phase de communication. Les FMU du système peuvent recevoir la réponse du « master

8. Le nombre d'équations représente assez bien le poids de chaque FMU fourni par EDF.

global » soit après « *latence* » temps si c'est la première FMU qui la reçoit, soit après $((N-1)\times latence)$ temps si c'est la dernière FMU qui reçoit le message. Dés la réception du message, les FMU peuvent commencer directement leurs communications avec celles connectées avec elles. Autrement dit, elles envoient leurs résultats de sortie, et attendent la réception des résultats des autres FMU connectées à elles. Comme le montre la figure 18. Chaque *wrapper* de FMU lance un *thread* de réception depuis le début jusqu'à la fin de toute la co-simulation, et un autre *thread* d'envoi qui s'active dès qu'on lance la phase de communication. Notons que le temps en intra-noeud est considéré nul.

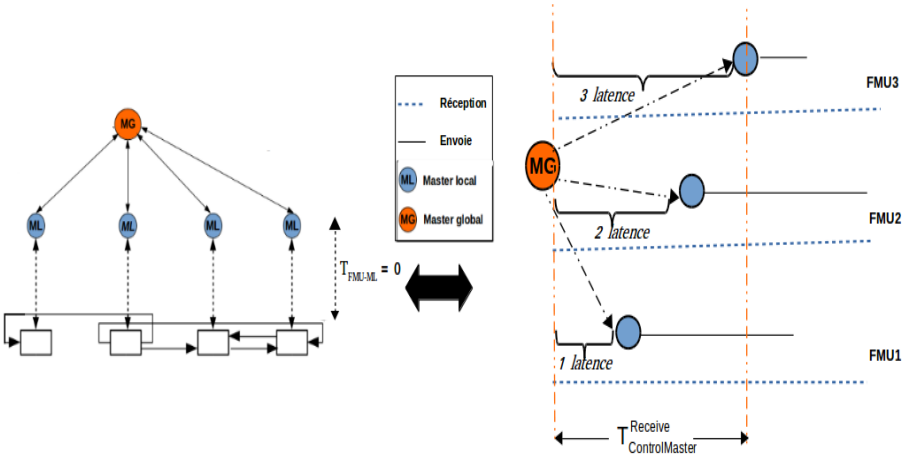


FIGURE 18 – Modélisation de la phase de communication d'une grappe de FMU

Si une des FMU du système est la première à recevoir la réponse du master global, dès qu'elle la reçoit, elle peut effectuer directement ses envois et attendre les données des autres FMU connectées avec elle (on a pris le maximum des temps de communication). Le temps de la phase de communication du premier nœud j recevant la réponse est comme suit :

$$T_{CommPhase}(j) = T^{Firstreceive}_{ControlMaster} + \max_{1\leq k\leq K_j} T_{Comm}(k) \tag{5.15}$$

$$= 1 \times latence + \max_{1\leq k\leq K_j} T_{Comm}(k) \tag{5.16}$$

Tel que K_j est le nombre de FMU se trouvant sur le même nœud j.
En revanche, le dernier nœud qui reçoit la réponse du master global devrait attendre $((N-1)\times latence)$ temps plus la réception des données des FMU connectées aux siennes.

$$T_{CommPhase}(j) = T^{Lastreceive}_{ControlMaster} + \max_{1\leq k\leq K_j} T_{Comm}(k) \tag{5.17}$$

$$= (N-1) \times latence + \max_{1\leq k\leq K_j} T_{Comm}(k) \tag{5.18}$$

Pour résumer, le formalisme de la phase de communication de toute la co-simulation, au pire cas, pourrait se définir ainsi :

$$T_{CommPhase}^{WorstCase} = ((N-1) \times latence) + \max_{1 \leq j \leq N} T_{CommPhase}(j) \tag{5.19}$$

Tel que N est le nombre total de nœuds de calcul (de masters locaux).

Dans DACCOSIM, une FMU peut envoyer des messages contenant plusieurs données à la fois (par exemple une FMU peut envoyer un message qui contient 10 données, le 10 représente la cardinalité de ce lien). Pour cela, nous avons adapté le modèle d'*Hockney* [39] pour estimer le temps d'une communication x mais en rajoutant un facteur *card* représentant la cardinalité des messages échangées entre FMU, comme l'illustre l'équation suivante :

$$T_{OneComm}(x) = latence + \left(\frac{Size_{msg} * card(x)}{Bw}\right) \tag{5.20}$$

Pour calculer toutes les communication d'une FMU k, on applique l'équation suivante :

$$T_{Comm}(k) = \sum_{x=1}^{Max_{E-S}(k)} T_{OneComm}(x) \tag{5.21}$$

$Max_{E-S}(k)$ est le maximum entre les entrées et sorties de la FMU k (en inter-nœuds).

Le temps global de la phase de communication comporte le temps d'envoi de la réponse aux masters locaux plus le temps des échanges des données en inter-nœuds. Selon le mode de communication en *sockets*, le temps d'envoi de la réponse depuis le global master correspond au $((N-1) \times latence)$ temps car l'envoi de la réponse aux masters locaux est séquentiel vu que le master global de la co-simulation se trouve sur un nœud de calcul et contient une seule carte réseau. L'équation 5.19 devient donc comme suit :

$$T_{CommPhase}^{WorstCase} = ((N-1) \times latence) + \max_{1 \leq j \leq N} \left(\sum_{k=1}^{K_j} \left(\sum_{x=1}^{Max_{E-S}(k)} T_{OneComm}(x) \right)\right) \tag{5.22}$$

Tel que K_j est le nombre de FMU qui se trouvent sur le même noeud j.

La reformulation exacte de la phase de communication pour toute la co-simulation est la suivante.

$$T_{CommPhaseCoSimul}^{WorstCase} = T_{CommPhase}^{WorstCase} \times iterations \tag{5.23}$$

Enfin, le temps de co-simulation globale d'une application parallèle de type DACCOSIM basée sur des FMU sera calculé comme suit :

$$T_{Co-Simualtion} = T_{ComputPhase} + T_{CommPhase} \tag{5.24}$$

Dans le chapitre suivant, nous évaluerons et validons ce modèle théorique par une série de *benchmarcking* avec des vraies mesures, afin de le comparer avec les performances obtenues par l'approche expérimentale, cette étape est illustrée dans le dernier chapitre.

5.4 Applications aux heuristiques complexes

Après plusieurs expérimentations, nous avons constaté que les performances d'une application basée sur des FMU dépendent étroitement de l'architecture utilisée et du type de problème traité. Vu que ces FMU ne fournissent pas d'informations sur leur charge de calcul, nous avons été obligé de réaliser une pré-phase qui définit les entrées de nos algorithmes de répartition (comme dans l'approche s'appuyant sur des heuristiques complexes). Il s'agit de réaliser des expérimentations sur des clusters afin de déterminer le temps de calcul de chaque FMU du système. Néanmoins, cette étape prend beaucoup de temps, surtout lorsqu'on exécute l'application sur plusieurs types de clusters d'expérimentation. Pour éviter cette perte de temps, nous avons proposé un modèle approximatif qu'on alimente avec les paramètres du cluster visé et du problème à traiter (présenté dans la section précédente). Ce modèle peut s'appliquer sur n'importe quel cluster sans passer par des *benchmarkings* complexes.

6 Conclusion

La problématique principale de ce chapitre est de savoir comment répartir les différentes FMU d'un système cyber-physique sur un cluster de PC multi-cœurs afin d'obtenir un temps de co-simulation minimal. Certaines solutions se basent uniquement sur des graphes DACCOSIM et d'autres exigent plus de connaissances sur l'architecture matérielle utilisée. Dans ce chapitre, nous avons listé les problèmes rencontrés en cherchant à exécuter le modèle DACCOSIM sur un cluster de PC multi-cœurs. Ensuite, nous avons détaillé nos approches de répartitions des FMU par des algorithmes, afin de décrire les différentes heuristiques et stratégies de placement des FMU que nous avons identifiées. Nous avons également décrit un modèle théorique approximatif qui permet d'estimer à la fois le temps de calcul et le temps de communication de chaque FMU quel que soit la

taille du problème.

Le prochain chapitre a pour objectif d'interpréter les résultats obtenus par les approches à base d'heuristiques, et par le modèle estimé présenté dans ce chapitre, en utilisant notre plate-forme de co-simulation DACCOSIM et le langage de programmation Java.

CHAPITRE 6

ÉVALUATION ET EXPÉRIMENTATION DES APPROCHES

Sommaire

1 Introduction

La co-simulation est une étape cruciale pour valider le fonctionnement des systèmes cyber-physiques. Elle permet, à travers de vrais cas métiers fournis par l'industrie, de critiquer et d'évaluer la pertinence des systèmes avant toute mise en œuvre sur le terrain. Cette co-simulation doit être effectuée en un temps acceptable, en utilisant un cluster de PC multi-cœurs qui offre à la fois un grand nombre de cœurs de calcul et une grande quantité de mémoire distribuée.
Le but du présent chapitre est d'évaluer et de valider la pertinence de nos différentes approches de placement des FMU sur un cluster de PC multi-coeurs. Ainsi, il résume tous nos efforts consacrés à distribuer efficacement et rapidement les applications basées sur des boîtes grises appelées FMU. De ce fait, nous avons effectué une série d'expérimentations en nous appuyant sur quatre différents *use-cases* métiers, de petites et de grandes tailles, conçus par EDF R&D et exécutés sur des clusters standards de PC multi-cœurs et sur un cluster HPC d'EDF possédant un réseau performant (*infiniBand*). La majorité des heuristiques proposées auparavant exigent d'avoir des méta-données qui vont être obtenues à l'aide d'une pré-phase de *benchmarks* (voir le chapitre 5). Néanmoins, la récupération de ces méta-données entraîne une perte de temps liée à des pré-exécutions. Pour éviter cela, nous avons proposé un modèle théorique qui permet d'avoir des estimations des temps de calcul et des temps de communications de chaque FMU du système (voir le chapitre 5). La deuxième partie de ce chapitre conduira donc à une évaluation de ce modèle théorique et paramétrable exécuté sur n'importe quel cluster de PC multi-cœurs. Cette évaluation sera faite en comparant les performances estimées (par le modèle) avec celles mesurées expérimentalement.

Dans ce chapitre, nous commençons par identifier les différentes métriques utilisées dans nos évaluations. Ensuite, nous présentons, à travers plusieurs séries de simulations, des résultats obtenus en appliquant nos algorithmes de placement mis en œuvre dans la plate-forme DACCOSIM. Enfin, nous évaluons notre modèle théorique à partir de mesures issues de l'expérimentation.

2 Métriques utilisées

Afin d'évaluer l'efficacité de nos approches de répartition appliquées aux co-simulations DACCOSIM à base de FMU, nous avons pris en compte les métriques suivantes :

- **Temps de calcul** (***Computation Time***)
  Puisque les FMU traitées lancent leurs calculs en parallèle, le temps de traitement global de cette co-simulation ($T_{Calcul-Cosim}$) sera calculée par le master hiérar-

chique à partir de l'ensemble des temps de calcul de ses FMU. Quel que soit le mode d'orchestration adopté (*overlapped* ou *ordered*), il représente le temps de calcul maximal de toutes les FMU, c'est-à-dire le temps de traitement de la FMU la plus longue. Notons que le temps de calcul (T_{Calcul}) d'une FMU est le temps mesuré lorsque le solveur de chaque FMU exécute ses calculs en utilisant la primitive *doStep(step*[1]*)* intégrée dans l'API JavaFMI et utilisée dans la plate-forme DACCOSIM.

- **Temps de co-simulation (*Co-simulation time*)**
  Le temps de co-simulation (T_{CoSim}) est défini comme étant le temps d'exécution de tout le système. Il représente en fait le maximum des temps de simulation obtenu sur les différents nœuds de calcul. Comme expliqué précédemment, le temps de simulation de chaque nœud regroupe à la fois le temps de calcul, le temps de contrôle et le temps de communication de toutes les FMU se trouvant sur le même nœud physique (voir le chapitre 5).

- **Accélération (*Speedup*)**
  Le *speedup* est une métrique qui évalue le parallélisme d'une application donnée, un facteur d'accélération et de rapidité permettant de qualifier les performances. Il s'obtient à partir de T(1) et T(p), ces valeurs représentent respectivement les temps de simulation obtenus avec une seule ressource informatique et avec « p » ressources de calcul. Par conséquent, l'accélération (S) d'une application parallèle est calculée par l'équation suivante :

$$S(p) = \frac{T(1)}{T(p)} \tag{6.1}$$

  En pratique, on peut trouver :

  — Une **accélération idéale** (théorie) lorsque $S(p) = p$: on peut trouver une accélération linéaire où on est, par exemple, « p » fois plus rapide en utilisant p nœuds de calcul plutôt qu'un seul nœud.

  — Une **accélération normale** si on trouve $(1 < S(p) < p)$.

  — Un **ralentissement** lorsque $S(p) < 1$: il s'agit d'une perte de performance, les tâches se parallélisent mal sur les « p » nœuds de calcul (par exemple à cause des synchronisations des threads), comme l'illustre la figure 1.

  — Une **hyper-accélération** lorsqu'on obtient un $S(p)$ supérieur à p. Cette accélération supra-linéaire est en fait obtenue suite aux cumuls de plusieurs

1. Le pas de temps adopté.

gain de temps : l'utilisation cumulée de plus de nœuds de calcul, de plus de mémoire RAM [2], et de plus de mémoire cache ...

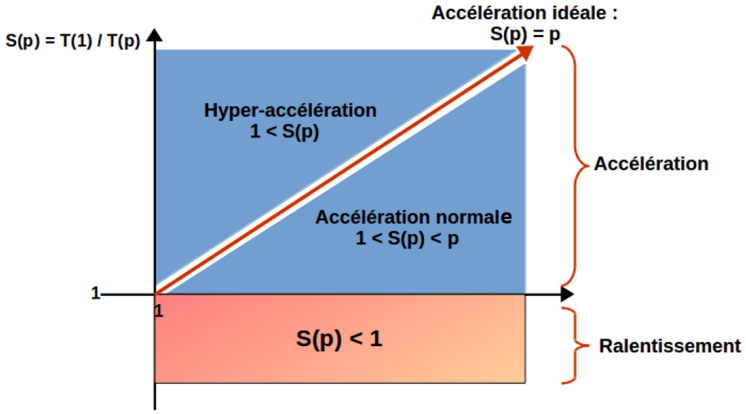


FIGURE 1 – Les différents cas de *Speedup* [67]

Dans nos expérimentations, nous appliquons deux placements de références *AllOnOneNode* et *OneFMUOneNode* pour calculer respectivement les temps de co-simulation sur un seul noeud et sur « Pone » noeuds de calcul, « *Pone* » étant égal au nombre total des FMU du système. Appelons également que le « T_{Pone} » est le temps de co-simulation avec le placement $OneFMUOnOneNode$. Dans notre cas, les FMU qui s'exécutent en parallèle ne sont pas homogènes en charge de calcul, ce qui veut dire que T_{Pone} est le temps d'exécution maximal de toutes ces FMU et on évaluera l'accélération avec :

$$S(Pone) = \frac{T(1)}{T_{Pone}} \tag{6.2}$$

On comparera aussi T_{Pone} à $T_{idéal}(Pone) = \frac{T(1)}{Pone}$. Notons qu'il est possible d'obtenir un temps de co-simulation idéal si on arrive à équilibrer la charge entre les différents nœuds réservés.

- **Efficacité (*Efficiency*)**
  Cette métrique évalue la bonne utilisation des nœuds de calcul, c'est-à-dire la façon dont les applications parallèles utilisent les p noeuds. Elle est généralement définie par :

$$e(p) = \frac{S(p)}{p} = \frac{T(1)}{p \times T(p)} \tag{6.3}$$

2. *Random Access Memory.*

En théorie, on devrait obtenir une valeur entre 0 et 1 ($0 \leqslant e(p) \leqslant 1$). Notons que si on obtient $e(p) > 1$ (100%), cela traduit une hyper-accélération.

En outre, l'efficacité est utile pour quantifier l'évolutivité de l'application sur des nœuds de calcul : si l'efficacité reste constante alors que la taille du problème et le nombre de nœuds de calcul changent, cela indique un passage à l'échelle linéaire qui va être présenté dans ce qui suit.

- **Capacité d'extensibilité (*Size up*)**

  L'extensibilité d'une application mesure sa capacité à traiter des problèmes de plus grandes tailles (plus de données d'entrée et de sortie, calcul plus fins, etc.) en utilisant plus de ressources informatiques, c'est-à-dire plus de noeuds de calcul offrant à la fois plus de CPU [3] et plus de RAM.

  Une première étape de nos expériences consiste à tester si un code distribué peut effectivement se déployer sur plus de machines pour traiter des problèmes plus gros, qui manqueraient de RAM sur un plus petit nombre de machines.

  Remarque :
  Des codes qui répartiraient leurs calculs, mais qui répliqueraient leurs données sur chaque nœud de calcul n'auraient aucune capacité d'extensibilité.

  Une seconde étape consiste à vérifier si on peut maintenir constant le temps de traitement quel que soit la taille du problème traité [69], c'est-à-dire de trouver des couples (q : taille du problème, p : nombre de noeuds de calcul) tels que :

$$T(q_1, p_1) = T(q_2, p_2) = T(q_3, p_3) = ... = T(q_k, p_k) = C^{ste} \tag{6.4}$$

avec $q_1 \leq q_2 \leq ... \leq q_k$ et $p_1 \leq p_2 \leq ... \leq p_k$.
Ce qui revient à vérifier les hypothèses de la lois de *Gustafson* [35].
Notons que dans les sections suivantes, nous utilisons souvent des problèmes de tailles multiples de celle d'un problème initial. L'équation 6.4 devient alors pour nous :

$$T(q \times 1, p_1) = T(q \times 2, p_2) = T(q \times 3, p_3) = ... = T(q \times k, p_k) = C^{ste} \tag{6.5}$$

Remarque
Certains codes ne permettent pas de garder constant le temps de traitement quel

3. *Central Processing Unit.*

que soit le nombre de noeuds utilisés (p), ce temps de traitement augmente en effet lorsque la taille du problème traité augmente (q). À l'opposé, les codes qui gardent constant le temps de traitement sont souvent très intéressants pour une utilisation intensive (en « production »).

- **Passage à l'échelle (*Scalability*)**

  En toute rigueur, pour garantir un passage à l'échelle, on doit assurer à la fois un *size up* en exécutant de plus gros problèmes sur plus de noeuds de calcul, et un *speedup* en accélérant chaque problème sur plusieurs nœuds physiques. En plus, on doit obtenir sur le graphique de passage à l'échelle des profils parallèles pour les différentes courbes d'accélérations. Ainsi, pour calculer le temps idéal d'une application parallèle passant à l'échelle, nous pouvons appliquer l'équation 6.6 suivante :

$$T_{ideal}(q,p) = \frac{T(q,1)}{p} \tag{6.6}$$

  Tel que le « q » représente la taille du problème traité, et « p » est le nombre de nœuds utilisés pour traiter le problème de taille q. Notons que le $T(q,1)$ est le temps de traitement d'un problème de taille q sur 1 nœud de calcul (exécution multi-cœurs sur un seul nœud de calcul).

  La figure 2 ci-dessous montre un cas idéal de passage à l'échelle.

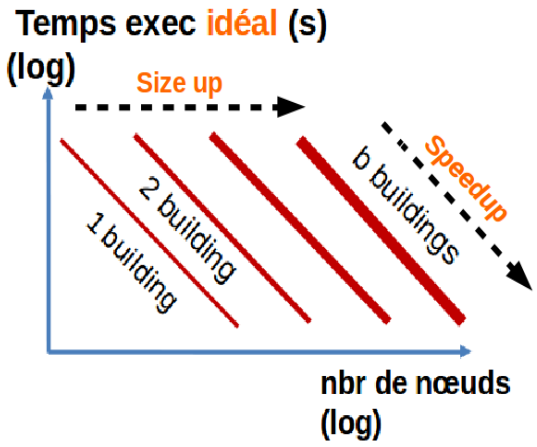


FIGURE 2 – Passage à l'échelle idéal

De plus, un graphique de passage à l'échelle peut devenir une abaque pour identifier des points de fonctionnement bien adaptés au degré d'urgence du problème traité. Ainsi, pour une taille de problème donnée, on peut chercher à le traiter rapidement en utilisant plus de nœuds de calcul, ou chercher à utiliser peu de ressources informatiques en acceptant un traitement plus long.

3 Mise en œuvre d'une co-simulation DACCOSIM sur cluster de PC multi-coeurs

Le déploiement d'une co-simulation DACCOSIM (des applications basées sur des FMU) s'appuie sur les étapes suivantes :

— ***Mapping* des FMU sur les nœuds virtuels « vnodes »**
C'est le placement des FMU sur l'ensemble de ressources définies dans l'interface DACCOSIM. Pour rappel, il comprend deux sous-étapes :

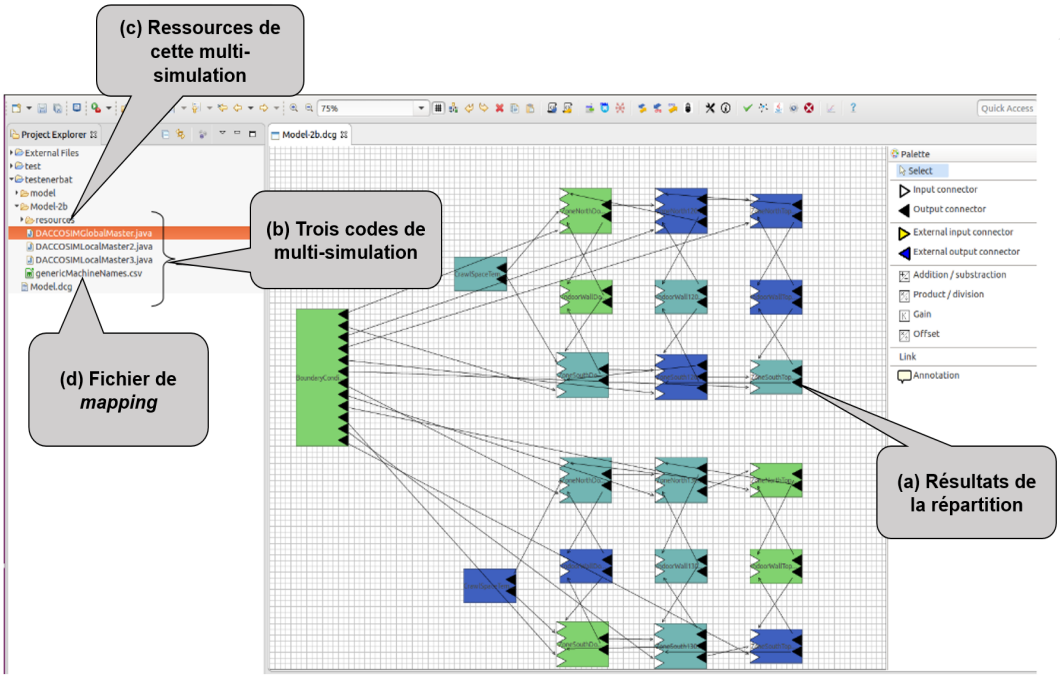


FIGURE 3 – *Mapping* de FMU sur des *vnodes*

1. Calcul de la répartition : Pour simuler rapidement ces diverses FMU, on a mis au point des approches de placement des tâches pour gérer efficacement les FMU sur un ensemble de nœuds multi-cœurs. Il s'agit de placer les diverses tâches de manière efficace sur ces ressources informatiques, en s'appuyant sur des approches qui tiennent compte soit du modèle à traiter, soit de la configuration matérielle des nœuds. Pour cela, on va tout d'abord raisonner sur des noeuds « virtuels » ayant les caractéristiques des noeuds physiques qui seront alloués ultérieurement. Le résultat de nos algorithmes de placement donne des graphes de FMU colorés, voir la partie « a » de la figure 3. Les FMU ayant la même couleur se trouvent sur le même nœud virtuel qui doit contenir un nombre de FMU inférieur ou égal au $nbrFmuOnOneNode$ (voir le chapitre 5).

2. Génération du code de co-simulation : DACCOSIM traduit ce graphe coloré en un ensemble de codes de co-simulation prêts à être exécutés sur les nœuds de calcul virtuels (figure 3-partie b), et dont l'un d'eux sera le master global de toute la co-simulation. Notons que le fichier de *mapping* (la figure 3-partie d) indique l'association des FMU sur les *vnodes* du cluster de ressources, et le dossier *ressources* contient les données des entrées/sorties des FMU, voir la partie « c » de la figure 3.

— ***Mapping* des nœuds virtuels sur les nœuds de calculs physiques :**
Cette étape permet d'associer chaque nœud virtuel (*vnode*) obtenu de l'étape précédente à un nœud physique (*pnode*), et d'exécuter les codes DACCOSIM des *vnodes* sur les *pnodes*.

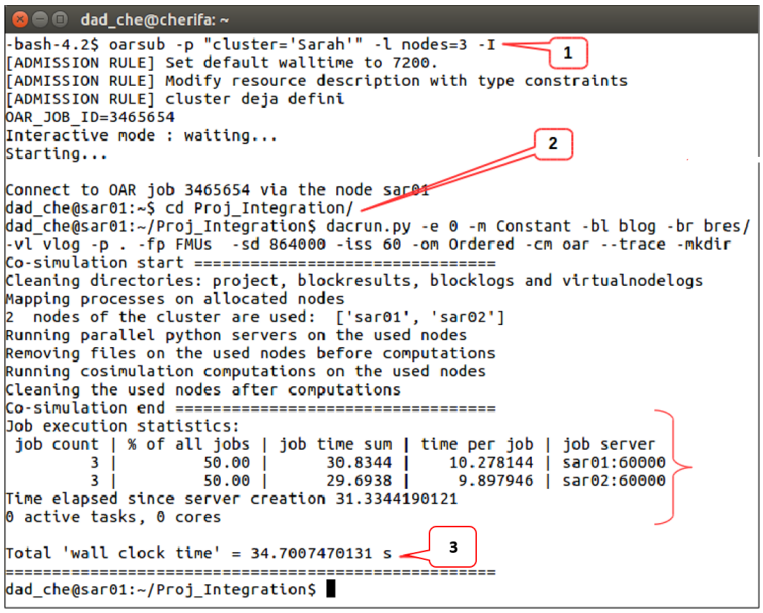


FIGURE 4 – *Mapping* et lancement des *vnodes* sur des *pnodes* du cluster

Finalement, le déploiement d'une co-simulation distribuée DACCOSIM suit les trois sous-étapes suivantes :

1. Tout d'abord, on alloue des nœuds de calcul (exemple : trois nœuds sur le cluster « *Sarah* »), en les demandant au gestionnaire du cluster visé (*Oar* ou *Slurm*).
2. On fait appel au « déployeur » *DacRun* qui réalise le *mapping* et le lancement des *vnodes* sur les *pnodes*, en précisant les options voulues (voir le chapitre 3), et en indiquant le répertoire qui contient les fichiers de co-simulation générés à l'étape de « mapping des FMU sur des noeuds virtuels ».
3. À la fin de l'exécution distribuée, *DacRun* affiche le temps total mesuré depuis

le début jusqu'à la fin de l'exécution (*wall clock time*) : il comprend le temps nécessaire pour la préparation du déploiement dans *DacRun* et celui de la co-simulation.

— **Résultats des résultats**
DACCOSIM sauvegarde les résultats numériques, logs et performances soit dans des répertoires montés et partagés, soit dans des répertoires temporaires locaux (*/tmp* de chaque nœud de calcul).

4 Observations préliminaires

Dans ce travail, nous nous sommes appuyés sur nos observations précédentes des co-simulations DACCOSIM exécutées sur des clusters de PC multi-coeurs. Les performances d'une application parallèle basée sur des FMU dépendent évidemment des ressources matérielles utilisées, mais aussi du bon déploiement des FMU sur ces ressources. Afin d'exploiter pleinement cette puissance fournie par les nœuds multi-cœurs, nous réalisons une série de simulations inductives afin d'évaluer le comportement des FMU sur un nœud multi-cœurs, puis sur une architecture distribuée.

Environnement expérimental de co-simulation

Pour réaliser les différentes expérimentations de nos algorithmes de répartition (implantés en Java) sur des architectures distribuées, nous avons employé quatre clusters de PC multi-cœurs assez différents :

Cluster	**Processeurs**	**Nbr de processeurs par noeud**	(**c.phy**,**c.log**)[*] par processeur	**Réseau d'interconnexion**
Skynet	*Intel Core i7-920 (Nehalem)*	1	(4, 8)	1 Gbit/s
Cameron	*Intel Xeon E5-1650 (Sandy-bridge)*	1	(6, 12)	10 Gbits/s
Sarah	*Intel Xeon E5-2637 (Sandy bridge)*	2	(8, 16)	10 Gbits/s
Porthos	*Intel Xeon E5-2697 (Ivy bridge)*	2	(28, 56)	*Infiniband*

* : (cœurs physiques, cœurs logiques).

4.1 Influence de la charge des FMU sur la fréquence des cœurs

Les processeurs d'aujourd'hui font varier leur fréquence en fonction de leur charge. Ils peuvent même adapter la fréquence de chacun de leur cœur en fonction de leur charge, presque indépendamment des autres cœurs. Malheureusement, ces mécanismes d'optimisation énergétique ont gêné notre étude.

Dans le cas de nos co-simulations, nos différentes FMU peuvent induire des charges de calcul trop variantes dans le temps. Ainsi, sur notre cluster Cameron, nous avons observé des cœurs qui pouvaient fonctionner à 1.33 Ghz ou à 3.1 Ghz selon la charge globale du noeud de calcul (mesure réalisée avec la commande « *cpupower monitor* »).

Lorsque la fréquence, et donc la vitesse, des processeurs variait, elle affectait les performances de nos co-simulations, c'est-à-dire qu'elle faussait les performances obtenues par nos algorithmes de répartition. Dès lors, il devenait très difficile pour nous d'étudier l'impact de nos algorithmes de répartitions des FMU : les performances observées lors de nos expérimentations fluctuaient beaucoup et masquaient nos efforts d'équilibrage de charge. Pour résoudre cela, nous avons donc forcé les noeuds de nos clusters en mode « performance » (fréquence élevée tout le temps imposée avec la commande « *cpuFreq-set* ») quand cela était possible pour évaluer nos solutions de placement des FMU, et identifier les plus efficaces sur nos co-simulations.

4.2 Variabilité des mesures

Lors de nos expérimentations, nous avons également observé que les performances obtenues ne sont pas stables. Cette instabilité s'explique par la variabilité liée à plusieurs facteurs et paramètres que ne nous pouvions pas contrôler (par exemple la disponibilité de la bande passante au moment de nos tests, la fréquence des processeurs). Pour remédier à ce problème, nous avons été forcés de refaire nos *benchmarks* plusieurs fois afin d'en prendre la moyenne, et d'en vérifier la dispersion.

Le tableau 4.2 montre la fluctuation des performances des deux *use-cases* qui modélisent des transferts de chaleurs dans un bâtiment. Nous avons évalué la dispersion des performances par rapport à leur moyenne. Dans ces deux simulations, on a un écart-type faible (1.23 s et $0,53$ s) comparé à la valeur moyenne (43.78 s et 91.35 s).
Les valeurs des performances fluctuent mais ne sont donc pas très dispersées.

Temps de co-simulation	test 1	test 2	test 3	test 4	test 5	Moyenne	Écart type
Application 1	43.38 s	43.21 s	45.93 s	42.85 s	43.51 s	43.78 s	1.23 s
Application 2	91.59 s	90.67 s	91.02 s	92,04 s	91.42 s	91.35 s	0.53 s

5 Accélération de co-simulation de thermique de bâtiment

Un bon *mapping* (association FMU/ressource physique) constitue une étape importante pour atteindre de bonnes performances. En effet, selon le placement utilisé, le temps global de la co-simulation augmente ou diminue. De ce fait, nous proposons, dans cette section, d'évaluer et de valider nos approches de répartition dédiées aux applications basées sur des FMU par le biais des *use-cases* industriels fournis par EDF R&D, en nous contentant tantôt de la description du graphe de FMU, tantôt des descriptions du graphe de FMU et du cluster utilisé. Nos *use-cases* incluent des modèles de FMU de petites et de grandes tailles, autonomes en exécution, mais induisant des dépendances entre tâches de calcul, dues aux liaisons inter-FMU. Ces liaisons sont nécessaires pour faire circuler les résultats des sorties des FMU vers les entrées d'autres FMU du système, à l'issue de chaque pas de temps.

5.1 *Use case* de petits calculs en « thermique de bâtiment »

Au début de la thèse, EDF R&D nous a fourni la première version implémentée d'un cas d'utilisation, appelé Enerbat1. Un cas d'utilisation de petite taille, qui simule le transfert de chaleur à travers une cloison (*wall*), et comporte deux zones thermiques (*ZoneA* et *ZoneB*) connectées en série et soumises à une température externe (*SineFMU*), comme l'illustre la figure 5.
Pour mémoire, dans la norme FMI, nous n'avons aucune information sur le comportement des composants (FMU), et on voyait les *use cases* métiers développés par EDF comme un graphe de boîtes grises intensément connectées.

Résultats numériques

Nous nous sommes appuyés sur l'outil de référence Dymola pour valider les résultats numériques de nos *use-cases* (voir le chapitre 3). La figure 6 illustre les résultats numériques obtenus de ce cas d'utilisation, d'un coté avec l'outil Dymola sur un seul nœud multi-cœurs, et d'un autre coté avec DACCOSIM mais sur un cluster de PC multi-cœurs. À travers cette figure, nous observons que les résultats obtenus sur cluster sont superposés et identiques à ceux de Dymola, ce qui montre la fiabilité des résultats. Pour réaliser

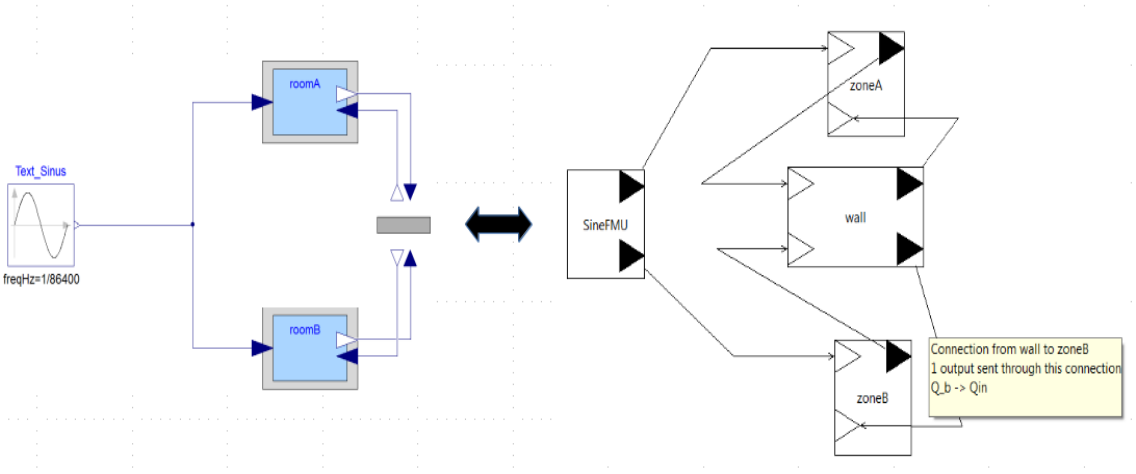


FIGURE 5 – Petit *use-case* en thermique de bâtiment (Enerbat 1)

cela, nous avons utilisé la même configuration de co-simulation pour les deux tests, en utilisant une méthode à pas constant et une autre à pas variable : une co-simulation d'une journée avec un pas de temps constant d'une minute, et avec un pas de temps variable d'une minute (pas de temps minimal = 0.01s et pas de temps maximal = 80s).

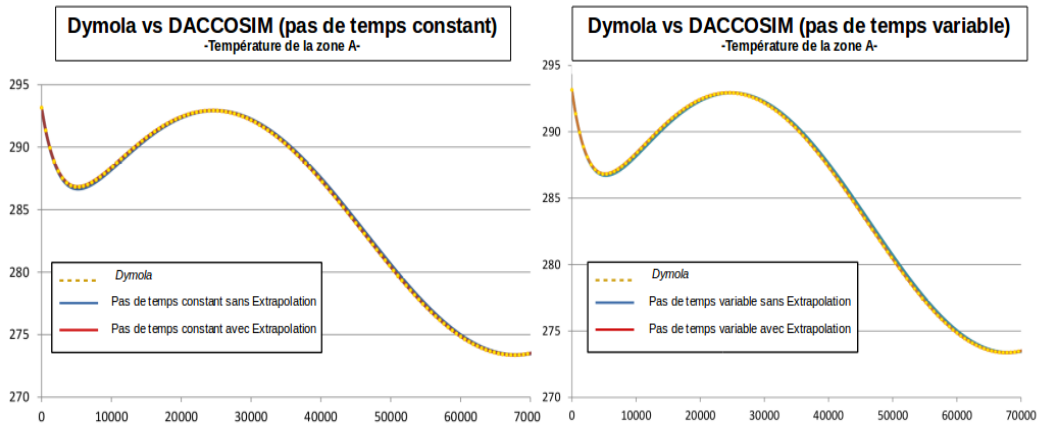


FIGURE 6 – Résultats numériques du cas d'utilisation Enerbat 1

Notons que parfois, dans certains *use-cases*, nous remarquons une petite différence localisée au début de la courbe, cela est due à la co-initialisation du système vu que DACCOSIM utilise son propre algorithme de co-initialisation.

Performances

Nous comparons, dans cette section, une méthode à pas de temps constant et une méthode à pas de temps auto-adaptatif (variable) en exécutant Enerbat1 sur un seul nœud multi-cœurs (en appliquant le placement *AllFMUOneNode*), et sur quatre nœuds multi-cœurs (en adoptant le placement *OneFmuOneNode*). Ces expérimentations sont

réalisées sur deux types de clusters *Cameron* et *Skynet* présentés auparavant.

	Pas de temps constant		**Pas de temps variable (Euler)**	
Stratégies/Cluster	Skynet	Cameron	Skynet	Cameron
AllFMUOneNode	23.49 s	13.55 s	69.64 s	55.55 s
OneFMUOneNode	35.83 s	44.48 s	206.89 s	177.52 s
Speedup (< 1)	0.65	0.31	0.33	0.31

TABLE 6.1 – Comparaison de performances entre une méthode à pas constant et une méthode à pas variable

Nous observons, à travers le tableau ci-dessus, que les temps de co-simulation sur *Cameron* sont plus courts que ceux trouvés sur *Skynet*. Cela est dû en partie au nombre de cœurs qui se trouvent sur un nœud *Cameron* : il contient six cœurs physiques hyper-threadés (douze cœurs logiques), contrairement à un nœud *Skynet* qui contient quatre cœurs physiques hyperthreadés (huit cœurs logiques). Nous observons également que l'augmentation du temps de co-simulation est importante lorsqu'on applique la méthode à pas variable (on trouve $\times 4$ sur Cameron et $\times 3$ sur Skynet). Cela est dû aux retours en arrière effectués par les FMU en cas de détection d'une imprécision dans les calculs. Généralement, les méthodes à pas variables sont efficaces et performantes comparées aux méthodes à pas constants surtout sur des co-simulations assez longues. Mais, comme l'illustre le tableau ci-dessus, notre méthode à pas variable a donné de mauvaises performances car elle était inadaptée à ce *use-case* et mal paramétrée.

En outre, sur ce petit *use-case*, le *speedup* calculé pour chaque méthode n'est pas intéressant car il est inférieur à 1, ce qui correspond à un ralentissement. Autrement dit, la parallélisation de ce *use-case* sur 4 nœuds (avec une FMU par noeud) n'est pas bonne.

Par ailleurs, dans la deuxième partie de cette section, nous avons évalué une partie de nos heuristiques élémentaires et complexes de placement de FMU, qui se basent sur la connaissance du graphe de calcul et du cluster utilisé : *AllFMUOneNode*, *LB*, *ClusteringComms*, *LBCComms* et *OneFMUOneNode*, voir le chapitre 5. Mais malheureusement, dans ce type de *use-case* où il y a très peu de FMU, et des FMU à petits calculs, une exécution sur un seul noeud multi-coeurs donne le meilleur temps de simulation comparé aux autres heuristiques sur plusieurs nœuds de calcul. Cela montre que l'utilisation d'un seul nœud multi-coeurs est nécessaire si le problème traité est léger en calcul et en communication, comme l'illustre la figure 7. Nous remarquons toutefois que *OneFMUOneNode* et *LB* donnent les mêmes performances, suivies par l'heuristique *ClusteringComms* qui regroupe les communications et par l'heuristique *LBCComms* qui essaie à la fois d'équilibrer au mieux la charge et de regrouper les communications.

Dans un deuxième temps, nous avons présenté une nouvelle heuristique appelée *nbIO*, qui représente une autre manière d'équilibrer la charge mais selon la somme d'entrée/sortie de chaque FMU (nombre total des connecteurs). Nous avons évalué l'heuristique *nbIO* et l'heuristique *LB* avec et sans extrapolation des entrées à l'intérieur d'un pas de temps

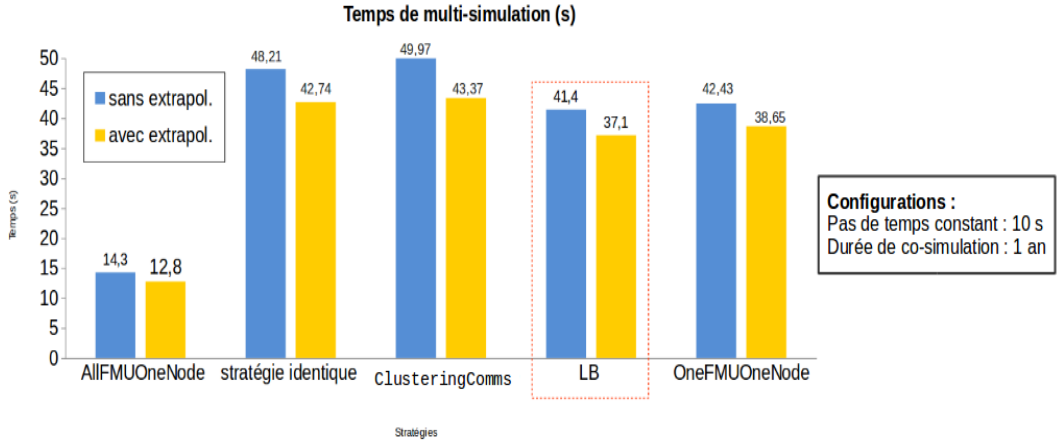


FIGURE 7 – Performances des différents placements de FMU

(une optimisation très efficace pour améliorer la précision des résultats de calcul). Sur ce *use-case*, les deux heuristiques *LB* et *nbIO* donnent presque les mêmes performances.

Extrapolation/heuristique	***LB***	***nbIO***
avec extrapol.	36.57 s	36.07 s
sans extrapol.	41.21 s	41.09 s

TABLE 6.2 – Performances obtenues : LB *vs* nbIO

5.2 *Use-case* de moyens calculs en « thermique du bâtiment »

Le deuxième cas d'utilisation est le même *use-case* que le premier mais avec des FMU effectuant des « calculs moyens » (plus gros que précédemment). Il s'agit d'un cas industriel fourni par EDF R&D comportant neuf FMU connectées entre elles, et qui représente encore un modèle de transfert de chaleur mais dans un bâtiment composé de quatre pièces (*Zone1*, *Zone2*, *Zone3bis*, *Zone4bis*). L'enveloppe du bâtiment est soumise à des conditions simples correspondant à la température de l'air extérieur « *OutDoors* » (pour les quatre zones) et à la diffusion de la chaleur interne « *Indoors* » (uniquement pour les zones 2 et 3), voir la figure 8 à droite.

Notons que les résultats numériques de ce *use-case* obtenus sur une machine ou sur cluster sont encore rigoureusement les mêmes que ceux obtenus avec l'outil de référence Dymola. Cela montre à nouveau que notre outil de co-simulation est aussi fiable en mono-machine qu'en distribué.

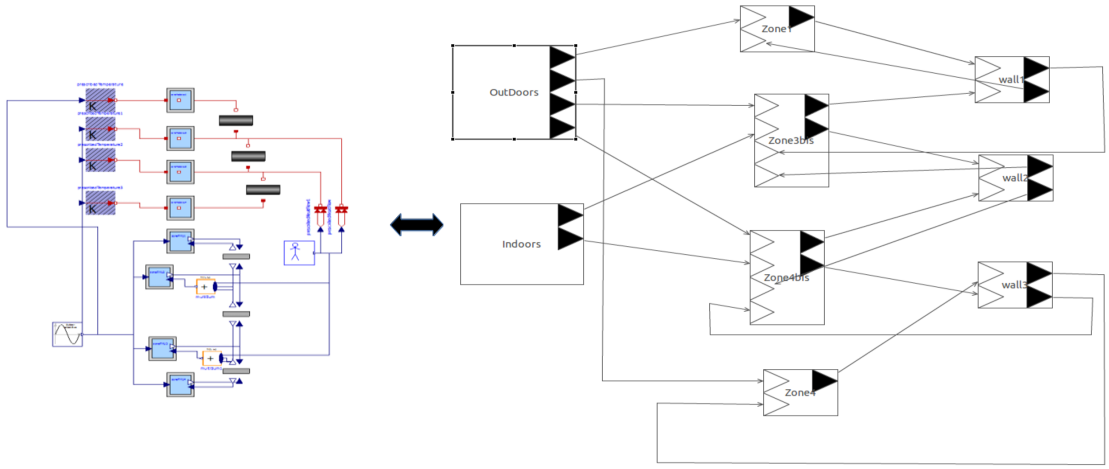


FIGURE 8 – Un *use-case* de taille moyenne en thermique de bâtiment (Enerbat 2)

Performances

Dans cette section, nous évaluons les stratégies de placement proposées auparavant. Comme le montre la figure 9, dans l'ensemble, c'est la stratégie *nbIO* qui donne les meilleurs résultats sur 7 et 8 nœuds. Cependant, nous avons obtenu un très léger *speedup* qui est supérieur à 1, comparé à une exécution sur un seul nœud de calcul. En revanche, dans certains cas, nous avons obtenu de moins bonnes performances, avec 35% (en moyenne) de temps en plus sur cluster de PC que sur une seule machine, voir la figure 9 (on a environ 100% de temps en plus sur deux machines).

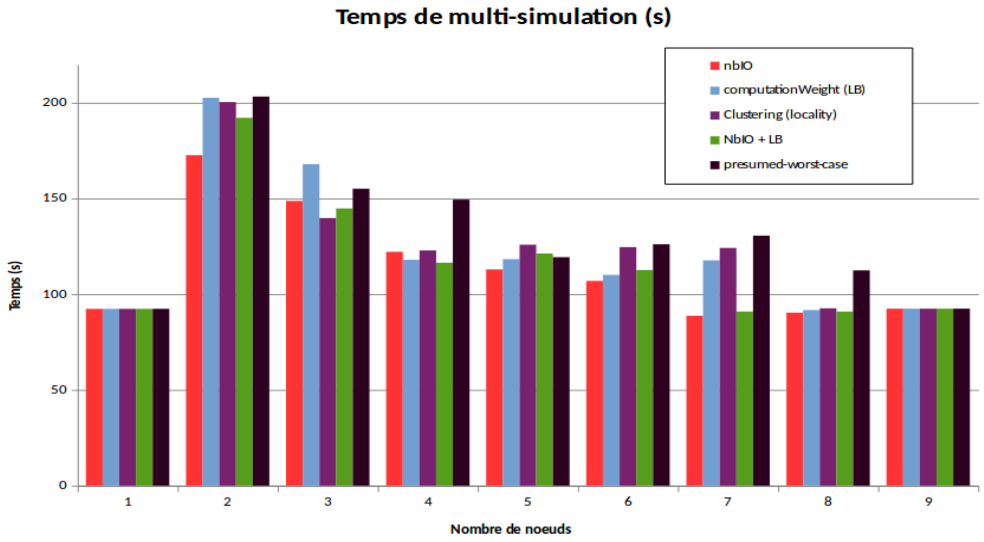


FIGURE 9 – Comparaison entre les différentes heuristiques

Remarque

Dans ces travaux de recherche, nous avons traité des problèmes avec chacun des caractéristiques propres en terme de calcul, communication et taille du système à traiter. C'est pour cela, nous avons proposé plusieurs heuristiques pouvant s'adapter au problème visé, et avons recherché la stratégie donnant des meilleures performances. Vu que nous étions responsable du déploiement des co-simulations de DACCOSIM sur des architectures distribuées, nous étions obligés de les déployer rapidement sur cluster afin de tester et évaluer nos heuristiques. Ces déploiements nous ont permis d'identifier quelques bugs et anomalies liées à l'installation de toute la suite logicielle DACCOSIM et JavaFMI sur une architecture distribuée. Nous nous sommes alors retrouvés face à plusieurs versions de la plate-forme DACCOSIM qui étaient incompatibles entre elles. De plus, les FMU des premiers *use-cases* étaient fabriquées avec une version de Dymola antérieure à celle des *use-cases* présentés dans la suite de ce chapitre. Ces deux premiers *use-cases* ont donc été difficiles à tester, car ils étaient associés à une période de debug de la suite logicielle et de sa procédure d'installation. C'est pour cette raison que nous n'avons pas pu tester « l'approche automatique » sur ces deux premiers *use-cases*.

5.3 *Use case* de gros calcul en « thermique du bâtiment »

Enerbat3 est un troisième cas d'utilisation qui permet le transfert de chaleur dans un bâtiment à trois étages. Il comporte onze FMU interagissant fortement entre elles. Chaque étage du bâtiment contient deux zones (ZoneNorthX, ZoneSouthX) séparées par un mur (*IndoorWall*). Un modèle nommé *Boundary Conditions (BC)* est en charge des conditions thermiques extérieures, et diffuse la même température aux différentes zones des étages, et un modèle *Crawspace (CS)* modélisant la température de l'espace sanitaire sous le bâtiment envoyée aux zones du premier étage 10. Les données échangées entre les FMU d'Enerbat3 sont des températures et des flux thermiques.

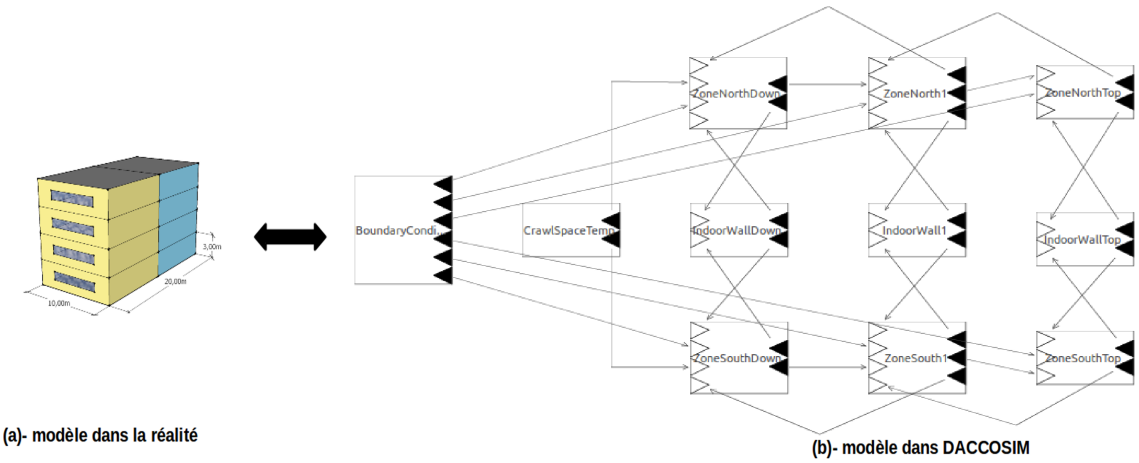


FIGURE 10 – Un gros *use-case* en thermique de bâtiment (Enerbat 3)

Nous avons demandé à EDF de nous fabriquer des *use-cases* métiers importants en calcul.

À cet effet, EDF nous a livré, pour le même *use-case*, cinq niveaux de complexité (appelés « livraisons »), comme l'illustre le tableau 6.3. Notons que la *Boundary Conditions* et le *Crawspace* sont les mêmes pour toutes les livraisons. Le passage d'un niveau à l'autre n'est rien d'autre qu'une augmentation de la précision et du poids des FMU du système, il renforce les calculs des FMU tout en gardant le même coût de communication. Cela nous a aidé à extraire les livraisons utiles qui répondaient à nos besoins, en terme de précision et de temps de co-simulation (performances). Le tableau 6.3 résume les poids de chaque FMU selon les livraisons.

Livraison (complexité et précision)	**Parois des zones**	**ZoneSouth et ZoneNorth**	**ZoneSouthTop et ZoneNorthTop**
	Nbr de mailles	**Nbr de variables**	**Nbr de variables**
cl1	5	410	482
cl2	10	485	582
cl3	25	710	882
cl4	50	1085	1382
cl5	500	7835	10382

TABLE 6.3 – Niveaux de complexité des *use-cases* fournis par EDF

Après la simulation et l'analyse de ces différentes livraisons, nous avons choisi deux complexités que nous avons trouvées utiles pour nos travaux, et qui sont conformes à la réalité des *Smart Grids* :

— La livraison *cl*5 qui comporte des FMU gourmandes en calcul, le renforcement des calcul rend alors les temps de communication négligeables, et on nomme cette livraison *Cal use-case*.

— La livraison *cl*3 qui comprend des FMU à moyen calcul, ce qui rend les communications proportionnellement plus importantes, i.e *cl*3 est un compromis entre les communications et les calculs. On a appelé cette livraison *CalComms use-case*.

Le premier but de cette étude est d'évaluer nos heuristiques sur ces deux types de livraisons. Et le deuxième but est de faire grossir ce type de *use-case* afin de vérifier si on peut réaliser un passage à l'échelle. Autrement dit, on va essayer de reproduire les même temps de co-simulation en grossissant de 1 à k fois la taille de ce type de *use-case* et en le traitant sur de plus en plus de noeuds. Le deuxième but se résume de ce fait à répondre à la question suivante : « Est-ce qu'on est capable de co-simuler des gros problèmes ? ».

5.3.1 Test de *speedup* et de *size up*

L'objectif final de cette étude expérimentale est de tester si on arrive à passer à l'échelle en appliquant nos approches de répartition sur des environnements distribués. Pour cela, nous commençons par étudier les performances obtenues en termes de *speedup* et de *size up* pour notre benchmark Enerbat3 avec les deux complexités présentées précédemment.

Optimisation expérimentale de distribution : recherche de *Speedup*

De manière générale, les applications parallèles de type DACCOSIM échangent de petits messages, dont les temps deviennent importants si les composants du système sont fortement connectés entre eux. Cela implique que, dans ce type de problème, le coût des communications est le premier critère qui influe sur les performances globales du système et qui mène à des mauvaises performances. Pour vérifier cela, nous avons utilisé l'heuristique *LBCComms* sur la complexité *cl*3 du CalComms *use-case* (courbes *s*2 sur la figure 11). Cette heuristique permet d'équilibrer la charge tout en minimisant les communications entre les nœuds physiques, et elle est adaptée quand les communications sont importantes vis-à-vis du poids de calcul. En revanche, l'heuristique *LB*, qui se focalise uniquement sur l'équilibrage de charge de la charge de calcul, donne de moins bonnes performances (courbes *s*1 sur la figure 11), car ici les communications ne sont pas négligeables. Notons que nous avons trouvé les mêmes placements de FMU (répartitions) sur les deux clusters utilisés : Cameron et Skynet. En conséquence, nous nous sommes contentés d'interpréter les résultats obtenus sur un seul cluster (Skynet).

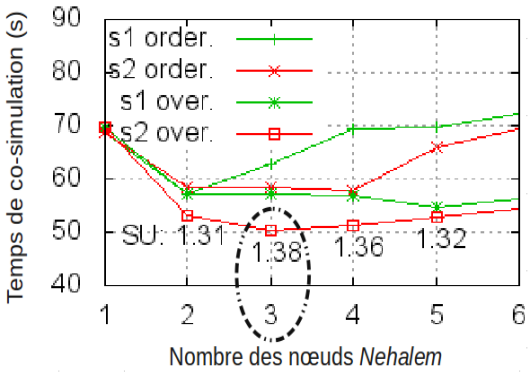


FIGURE 11 – Comparaison des heuristiques : *LB* et *LBCComms* sur le *CalComms use-case*

Dans un deuxième temps, nous avons évalué l'impact du mode d'orchestration sur le temps de co-simulation. À travers la figure 11, nous remarquons, dans l'ensemble, que le mode *overlapped* donne de meilleures performances comparées à celles obtenues avec le mode *ordered*. Ce gain est dû à la parallélisation des deux phases exécutées pendant un pas de temps, à savoir la phase de calcul et la phase de communication (voir le chapitre 3).

Finalement, nous observons, sur la figure 11, que le meilleur *speedup* est obtenu par l'heuristique *LBCComms* ($s2$), en répartissant ce *use-case* sur trois noeuds de calcul. Cela confirme la fiabilité de l'heuristique qui équilibre la charge et minimise les communications entre les nœuds de calcul sur un cas d'utilisation qui contient des FMU échangeant beaucoup de données entre elles.

En outre, la courbe s2 en mode *overlapped* montre que si on intensifie ou on allège la charge de calcul des coeurs physiques du noeud visé, le temps global de co-simulation augmente. Cette expérience montre que pour un problème donné, il faut bien identifier la bonne heuristique, et le bon nombre de nœuds à utiliser pour minimiser le temps d'exécution et avoir de bonnes performances.

En outre, nous avons observé, à travers nos expérimentations, que le *Cal use-case* donne de meilleures performances en appliquant l'heuristique *LB* qui permet d'équilibrer la charge entre les nœuds physiques (ses communications sont quasiment nulles par rapport aux calculs).

On a résumé dans le tableau suivant les *speedup* obtenus en adoptant l'heuristique *LBCComms* :

nbr de nœuds	**1**	**2**	**3**	**4**	**5**	**11**
speedup	1.0	1.31	**1.38**	1.36	1.32	1.10
***Efficiency* (%)**	100	65	**46**	34	26.4	10

En nous focalisons sur le *speedup*, comme l'illustre la figure 11, nous avons conclu que la meilleure distribution pour la livraison $cl3$ (*CalComms use-case*) se fait sur 3 nœuds alors que celle du *Cal use-case* est obtenue sur 4 nœuds (voir la figure 12). Toutefois, on note que l'efficacité (le taux de bonne utilisation des ressources informatiques allouées) chute rapidement sur ce petit *use case* de 11 FMU. Nous surveillerons donc l'efficacité de nos parallélisations dans nos plus gros *use-cases*.

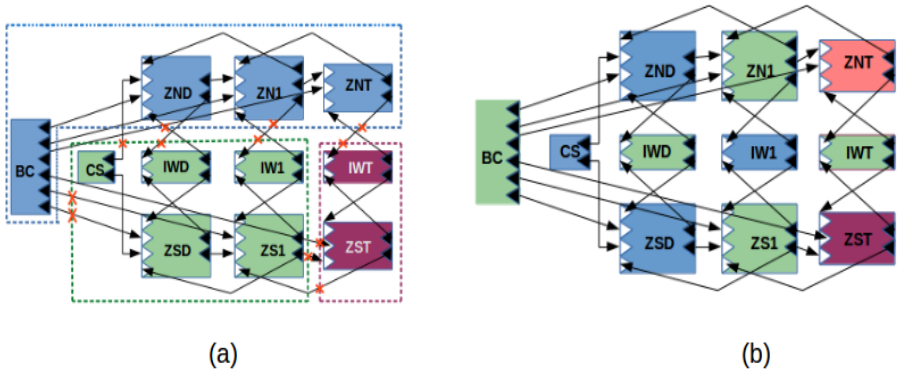


FIGURE 12 – (a) Meilleure répartition de *CalComms use-case* sur 3 nœuds. (b) Meilleure répartition de *Cal use-case* sur 4 noeuds. Les FMU de la même couleur sont situées sur le même nœud de calcul

Distribution de gros problèmes par réplications : recherche de *size up*

Dans cette section, nous voulons réaliser un *size up* de ces solutions, en simulant de gros bâtiments sur de plus en plus de ressources. Pour réaliser cela, il nous a fallu traiter de grands *use-cases*. Néanmoins, en 2016, nous n'avions pas ce type d'applications métiers contenant un plus grand nombre de FMU, et nécessitant plus de calculs. Pour pallier à ce problème, nous nous sommes appuyés sur la stratégie de réplication présentée dans le chapitre 5 pour agrandir la taille du problème. De ce fait, nous avons répliqué la bonne répartition obtenue pour un problème élémentaire d'un bâtiment (voir figure 12), afin de traiter un problème plus gros de « b » bâtiments.

Grâce à la réplication, nous avons pu co-simuler 1, 2, 3 et 4 bâtiments sur 1×4, 2×4, 3×4 et 4×4 noeuds de calcul, pour traiter la livraison ($cl5$), comme l'illustre les courbes de la figure 13 (gauche). Comme nous l'avons remarqué, nous avons obtenu un *size up* idéal car le *benchmark* traité comportait principalement de gros calculs (c'était un cas de référence). Nous avons obtenu un temps d'exécution constant de 53 s sur le cluster de processeurs *Nehalem* avec un réseau d'interconnexion d'un Gbit/s, et un temps d'exécution de 45 s sur notre cluster de processeurs *Sandy Bridge* avec un réseau de communication de 10 Gbits/s. En effet, ce *use-case* est gourmand en calculs et ses communications apparaissent négligeables vis-à-vis de ses calculs. Il garantit ainsi une parallélisation parfaite qui donne un *size up* parfait : $T(1b, 4 \times 1) = T(2b, 4 \times 2) = T(3b, 4 \times 3) = T(4b, 4 \times 4) = C^{ste}$.

À l'opposé, dans la partie droite de la figure 13, le *benchmark* de type $cl3$ inclut une quantité significative de communications par rapport à la quantité de calculs. Nous avons à nouveau répliqué la meilleure répartition (sur 3 noeuds), et avons simulé 1, 2, 3, 4 et 5 bâtiments sur 1×3, 2×3, 3×3, 4×3 et 5×3 nœuds de calcul. Cette fois, nous avons obtenu des performances vraiment imparfaites sur un cluster de PC d'un Gbit/s, mais intéressantes sur un cluster de 10 Gbits/s (le temps de co-simulation se maintient à peu près). Des communications importantes dans notre système (comme dans le $cl3$ *use-case*) peuvent être à peu près couvertes par un réseau de communication de 10 Gbits/s, et produisent un *size up* acceptable par rapport à celui d'un réseau d'interconnexion à 1 Gbit/s seulement.

Bilan

Cette partie d'expérimentation prouve que la qualité du réseau d'interconnexion joue un rôle important dans les performances de nos applications à base de FMU, mais qu'un *size up* de bonne qualité est possible même sur des clusters « bon marché ».

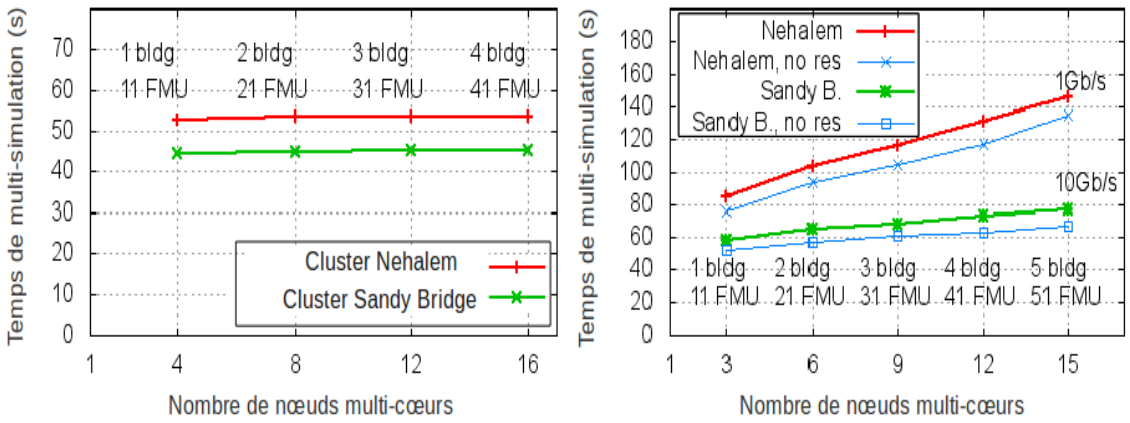


FIGURE 13 – *Size up* sur un cluster avec 10 Gbits/s et 1 Gbit/s pour un *cal use − case* (à gauche), et pour un *CalComms use − case* (à droite)

5.3.2 Premiers tests de passage à l'échelle par réplication (*scaling*)

L'idée principale de cette expérience est de garantir deux métriques à la fois : le *size up* et le *speedup*. On cherche à traiter de gros problèmes tout en essayant d'accélérer chacun d'eux sur plusieurs nœuds de calcul (voir la définition du passage à l'échelle dans la section 2). Le résultat attendu est donc d'obtenir des profils parallèles des temps d'exécution pour garantir un passage à l'échelle de nos co-simulations. Dans la figure 14

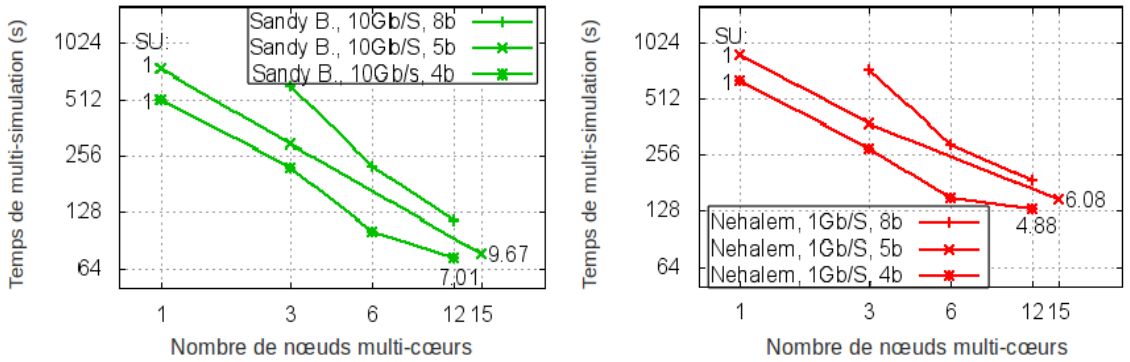


FIGURE 14 – *Scaling* du problème CalComms sur Cameron (à gauche) et *Skynet* (à droite)

ci-dessous, nous avons simulé 4, 5, 8 bâtiments avec 41, 51 et 81 FMU, sur deux clusters : des quadri-cœurs connectés en *Ethernet* à 1 Gbit/s (Skynet), et des héxa-cœurs connectés en *Ethernet* à 10 Gbits/s (Cameron). Nous observons, à travers cette figure, que nous avons pu accélérer chacun de ces problèmes sur de plus en plus de nœuds physiques, et que nous avons obtenu des profils presque parallèles. Ainsi, nous avons obtenu une bonne accélération de 9,67 sur Cameron et de 6.08 sur Skynet, en co-simulant 51 FMU sur 15 nœuds hexa-cœurs et quadri-cœurs. Nous remarquons aussi que si on exécute 51

FMU sur 6 nœuds héxa-cœurs, on obtient le même temps de co-simulation que celui du problème qui comporte 41 FMU sur 3 nœuds héxa-cœurs (mêmes observations sur les nœuds quadri-cœurs), cela montre que nous avons pu conserver le temps de co-simulation lors d'un *size up*.

En revanche, nous n'avons pas pu continuer cette expérience pour le problème contenant 81 FMU car nous n'avions pas assez de nœuds sur nos clusters : nous n'avions que 16 nœuds et il nous en aurait fallu $(8 \times 3 = 24)$ noeuds pour répartir les $(8 \times 10 + 1\ )$ FMU de nos 8 bâtiments. Pour rappel, ce *size up* est réalisé grâce à une simple réplication de la meilleure distribution du problème élémentaire d'un seul bâtiment. Notons également que nous n'avons pas pu simuler les 8 bâtiments sur un seul nœud de calcul vu qu'il ne contenait pas assez de mémoire, mais nous avons au moins pu les simuler sur 3, 6 et 12 nœuds par superpositions plus ou moins fortes des distributions élémentaires d'un bâtiment sur 3 nœuds de calcul.

Bilan

Étant donné que nous avons pu obtenir des profils à peu près parallèles et maintenir des temps de co-simulation presque constants, notre distribution du CalComms *use-case* (cl3) passe à l'échelle : on a pu accélérer chaque problème sur plusieurs noeuds de calcul, et traiter des problèmes de grandes tailles en temps constant sur des clusters bon marché. Par ailleurs, la figure 14 permet de choisir des points de fonctionnement, et donne aux utilisateurs de la co-simulation plus de souplesse pour traiter un problème donné en fonction de son degré d'urgence.

6 Passage à l'échelle de co-simulations de réseau électrique

Dans cette section, nous voulons vérifier si on peut réaliser un passage à l'échelle (*scaling*) d'un gros *use-case* métier basé sur des FMU, en appliquant notre approche « automatique ». Cette approche permet de répartir un grand nombre de FMU sur un grand nombre de cœurs et de nœuds, d'un cluster donné. Pour vérifier l'efficacité de notre approche, nous nous sommes appuyés à la fois sur le graphe DACCOSIM modélisant le problème, et sur des clusters soit standards de type « *Ethernet* », soit performants de type « *infiniband* ».

6.1 *Use-case* de grande taille en « réseau électrique et thermique »

EDF nous a fourni un *use-case* nommé IDEAS qui répond à un besoin de simulation de réseau électrique et de bâtiments associés (prémisse de futurs *Smart Grids*), et qui comporte un total de 1042 FMU exportées à partir de Dymola 2016 FD01. Le modèle

IDEAS est conforme à la norme FMI-CS 2 (voir le chapitre 3), il a été complètement implémenté en Modelica en utilisant la bibliothèque OpenIDEAS [7]. En complément, un plus petit modèle *IDEAS* avec moins de bâtiments et seulement 442 FMU, a également été conçu pour évaluer l'évolutivité et l'extensibilité de notre approche.

Comme l'illustre la figure 15, IDEAS se compose d'une ligne moyenne tension, des départs basses tensions et des bâtiments. On peut représenter un quartier comme étant une ligne basse tension (BT) avec les bâtiments (*dwellings*) qui lui sont connectés. De manière générale, IDEAS comporte 1000 bâtiments reliés à des lignes basses tension (*feeders*), et chaque bâtiment se compose d'une enveloppe thermique, d'un système de ventilation et d'un système de chauffage, et est modélisé par un *comportement d'occupation* stochastique. Ces bâtiments sont répartis sur les 20 *feeders* basses tensions (BT), modélisés chacun par une FMU. La fréquence du réseau électrique, propagée aux (*dwellings*) et (*feeders*), est fournie grâce aux 20 FMU supplémentaires. Tous les *feeders* du modèle sont connectées à une seule FMU qui simule le réseau moyenne tension (MT). Ce modèle met aussi en place une FMU qui permet de récupérer les mesures des lignes moyennes tensions.

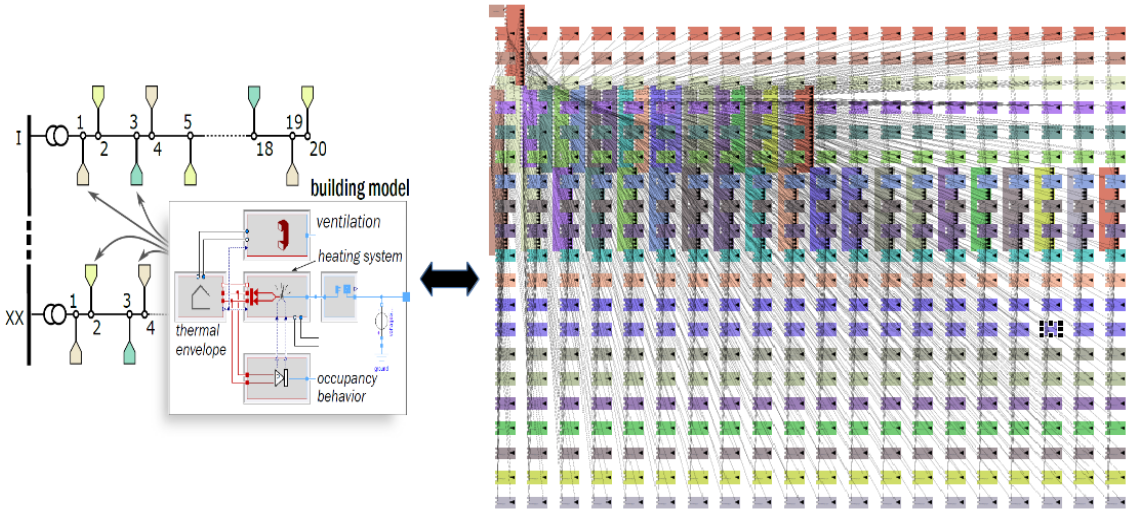


FIGURE 15 – Réseau électrique et thermique : modèle IDEAS à 442 FMU

Avant d'interpréter les performances obtenues sur cluster, nous avons tout d'abord vérifié la qualité et l'exactitude de nos résultats numériques de co-simulation. Pour cela, nous avons sélectionné au hasard un seul bâtiment de ce *use-case*, et l'avons simulé avec Dymola, en injectant des données de tensions obtenues à partir de la co-simulation sur cluster, et en fournissant le même environnement de co-simulation pour le bâtiment simulé : même tension d'entrée, mêmes données météo et même données d'occupation.

Après la simulation, la dynamique de la consommation d'énergie est bien reproduite, elle suit la même tendance que celle observée avec Dymola, limitée à un seul *feeder* de 20 bâtiments. De plus, cette co-simulation sur cluster a donné des résultats réalistes selon les jugements d'experts d'EDF, ce qui prouve la fiabilité de notre co-simulation sur cluster.

6.2 Analyse des performances obtenues

Dans cette partie, nous comparons trois algorithmes automatiques qui se résument comme suit :

— ***Round Robin* (RR)** est une méthode naïve qui place les FMU du système de façon aléatoire, en parcourant une liste de FMU non ordonnée (voir le chapitre 5).

— ***Familly Round Robin* (FRR)** est une méthode qui regroupe les FMU dans des familles ayant des charges de calculs similaires. Comme indiqué dans le chapitre 5, chaque famille formera une liste de FMU, et l'ensemble des listes obtenues seront distribuées sur les nœuds de calcul réservés selon un algorithme *round robin*. Cette approche nécessite une connaissance moyenne sur les FMU utilisées. Le but de cet algorithme est donc d'équilibrer la charge entre les nœuds de calcul réservés.

— **Business Experts (BE)** est un algorithme qui exploite des connaissances métier sur le système global simulé. Cet algorithme a été testé avec des collègues d'EDF, afin de répartir des quartiers entre les différents noeuds de calcul.

Évaluation de l'approche automatique et de ses variantes

Nous observons, à travers la figure 16, que la répartition FRR est systématiquement plus rapide que la répartition BE qui exploite pourtant plus de connaissances métier sur ce modèle. Cette courbe représente le temps de co-simulation en fonction du nombre de nœuds physiques. Nous remarquons, à travers cette figure, que l'algorithme métier BE donne des performances similaires à l'algorithme FRR sur 10 et 20 nœuds, surtout en mode *ordered* (le plus rapide sur ce *benchmark*). Lorsqu'on compare ces deux algorithmes, la méthode BE donne de mauvais temps d'exécution pour certains nombre de noeuds, car elle introduit un déséquilibre de charge entre les différents noeuds de calcul.

On remarque, sur le IDEAS *use-case*, que l'algorithme FRR en mode *ordered* est plus rapide qu'avec le mode *overlapped*. Ce ralentissement du mode *overlapped* peut s'expliquer par le fait que ce *use-case* contient beaucoup de FMU, et sature les nœuds avec des threads de calculs de FMU (plus de FMU que de coeurs physiques sur chaque noeud). Ces threads de calcul sont alors perturbés par l'exécution de threads de communication

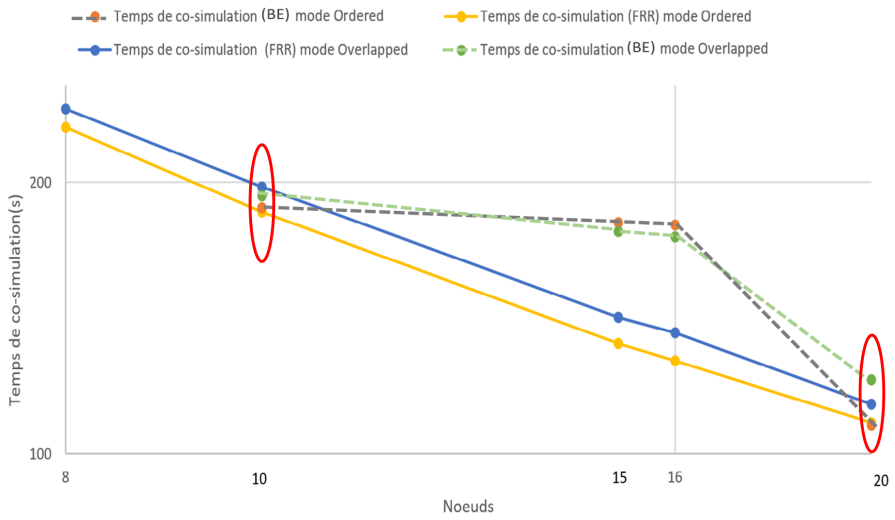


FIGURE 16 – Temps de co-simulation sur le cluster Porthos : FRR *vs* BE

en parallèle, et globalement les calculs ralentissent. Notons que nous avons utilisé les noeuds physiques en mode « exclusif », sans les partager avec d'autres applications, pour ne pas perturber nos calculs. De plus, l'heuristique FRR s'adapte bien à tout nombre de noeuds de calcul, alors que la méthode BE fonctionne mal quand le nombre de noeuds n'est pas un multiple du nombre de quartiers simulées. Grâce à ces expérimentations et ces observations, on peut conclure que la méthode « FRR » est significativement plus performante, souple et générique que la méthode « BE » pour co-simuler les applications à grande échelle sur n'importe quel nombre de noeuds imposés.

Dans un deuxième temps, nous comparons notre algorithme FRR avec la méthode classique RR. Pour concrétiser cela, il nous a fallu réaliser plusieurs répartitions « RR » avec des liste aléatoires pour pouvoir positionner et comparer ces deux algorithmes. Comme l'illustre la figure 17, nous avons ainsi tracé trois courbes différentes : celle de l'algorithme proposée *Family Round Robin*, puis le minimum et le maximum des valeurs trouvées en exécutant l'algorithme *Round Robin* sur différentes versions de la liste des FMU à placer.

Nous remarquons, à travers cette figure, que l'algorithme FRR (voir le chapitre 5 section 4.3) donne toujours des temps de co-simulation intéressants qui se situent entre les minimums et les maximums des temps obtenus par la méthode de répartition (RR), en fonction de l'état initiale de la liste des FMU. **Le gain obtenu n'est pas énorme sur cet exemple, mais l'heuristique FRR nous met à l'abri d'une répartition aléatoire défaillante et lente**.

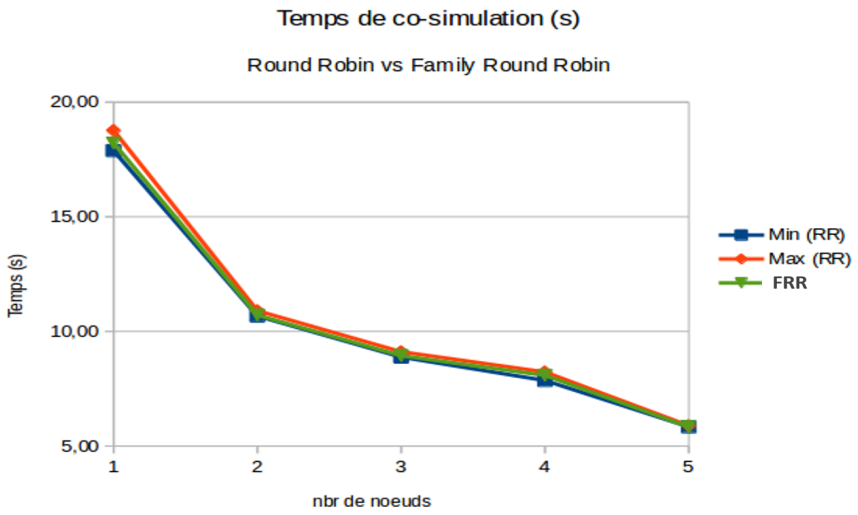


FIGURE 17 – Comparaison entre FRR *vs* RR

Passage à l'échelle d'un gros *use-case* métier

Comme l'illustre la section précédente, les expérimentations ont prouvé que la stratégie FRR, qui ne nécessite aucune pré-étude expérimentale (mais seulement une connaissance métier légère), donne toujours de bonnes performances sur notre problème à plus de 1000 FMU. Dans cette section, nous allons évaluer la capacité de passage à l'échelle de nos applications basées sur des FMU (sans passer par la stratégie de réplication), exécutées sur un cluster standard (*ethernet*) et un autre à haute performance (*infiniband*). Pour se faire, nous exécutons le cas d'utilisation IDEAS avec deux tailles différentes : 442 FMU et 1042 FMU sur les clusters de PC multi-coeurs de CentraleSupélec (en utilisant de 32 à 256 cœurs physiques) et ceux d'EDF (en utilisant de 112 à 1792 cœurs physiques). Nous appliquons aussi sur ces deux *use-cases* les deux modes d'orchestration présentés auparavant : *Overlapped* et *Ordered* (voir le chapitre 3).

Comme le montre la figure 18, dans l'ensemble, les courbes tracées sur des échelles logarithmiques apparaissent linéaires sur les deux clusters présentés précédemment. Il s'agit du cluster Sarah qui utilise un réseau d'interconnexion *ethernet* de 10 GBits/s, et du cluster *infiniband* Porthos qui utilise un réseau de haute performance (avec à la fois une faible latence et une bande passante importante). Les caractéristiques de ce réseau *infiniband* permettent de réduire les coûts de communications, ce qui augmente les performances globales de nos co-simulations, comme l'illustre la figure 18 : les pentes des courbes de temps du cluster *infiniband* sont plus proches de -1 (pente idéale sur une courbe en échelle logarithmique) que celles du cluster *Ethernet*. Nous remarquons également que les courbes des temps de co-simulation du modèle IDEAS 442 FMU sont plus ou moins

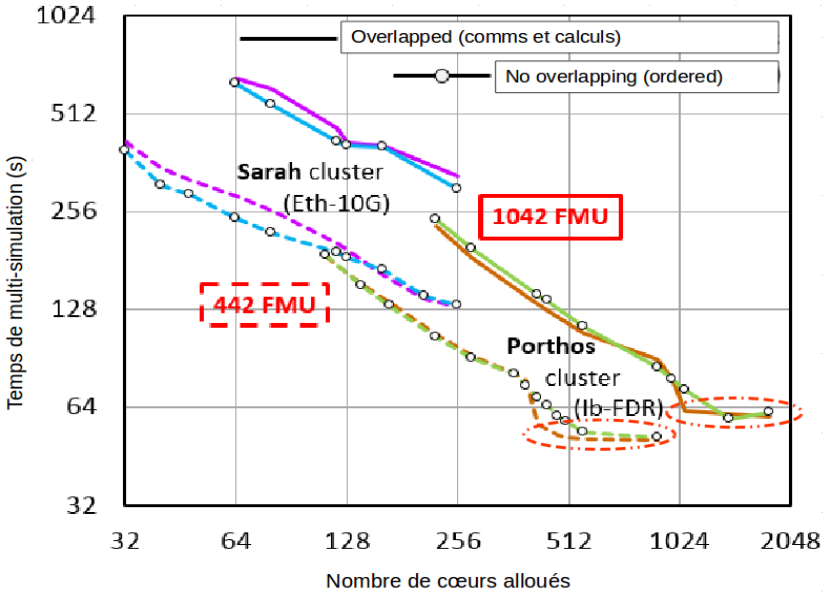


FIGURE 18 – Passage à l'échelle sur les clusters *Infiniband* « Porthos » et *Ethernet* « Sarah »

parallèles à celles du modèle à 1042 FMU, c'est-à-dire qu'on est arrivé à accélérer et à obtenir les mêmes profil d'accélération pour un petit et un gros problème sur nos clusters. De plus, nous arrivons à exécuter notre problème à 1042 FMU dans le même temps que notre problème à 442 FMU, en exploitant plus de coeurs physiques. Nous réalisons donc un *size up* satisfaisant en co-simulant des problèmes de différentes tailles sur différents types de clusters (*ethernet* et *infiniband*). Grâce aux *use-cases* IDEAS 1042 FMU et IDEAS 442 FMU, nous avons prouvé expérimentalement que notre stratégie FRR répond à la fois aux deux métriques importantes dans les applications parallèles : le *size up* et le *speedup*, et montre une réelle capacité de passage à l'échelle [75].

Par ailleurs, nous remarquons, sur la courbe de Porthos, que le temps de co-simulation devient stable entre 1042 et 2048 cœurs physiques, cela s'explique par le fait d'avoir autant ou plus de cœurs physiques que de FMU : même si une FMU est implémentée par plusieurs threads à la fois (calcul et communication), elle ne sature qu'un seul cœur.

De plus, comme l'illustre la figure 18, les deux modes d'orchestration donnent tout d'abord des performances similaires sur Porthos (*infiniband*), mais à partir d'un certains nombre de cœurs, le mode *overlapped* devient réellement plus rapide. En utilisant un grand nombre de cœurs physiques, chaque cœur n'exécute les calculs que d'une seule FMU. Dès lors, quand les FMU les plus rapides ont fini leurs calculs, des cœurs physiques deviennent totalement libres pour exécuter les threads de communications en parallèle des calculs plus long, ce qui accélère le mode *overlapped*. Ce mode (*overlapped*) permet donc d'atteindre le temps d'exécution optimal avec beaucoup moins de cœurs que le

mode *ordered*.

7 Évaluation de notre modèle approximatif de performance

Comme l'ont montré les expérimentations, nos approches peuvent influer significativement sur les performances globales de nos co-simulations. La plupart de nos solutions exigent d'avoir des informations sur les problèmes à traiter et sur le cluster utilisé afin de paramétrer les algorithmes. Comme l'indique les sections précédentes, nous avons proposé d'effectuer une série de *benchmarks* préliminaires pour obtenir ces informations, toutefois, cette pré-étape d'expérimentations prend un temps prohibitif. C'est pour cette raison que nous mettons à disposition deux modèles théoriques approximatifs du fonctionnement réel de nos applications basées sur des FMU. Un modèle qui permet d'estimer le temps de calcul et un autre pour prédire le temps de communication d'une co-simulation donnée, en utilisant un cluster de PC multi-coeurs. Cette section se concentre sur l'évaluation du modèle approximatif (global) regroupant ces deux modèles théoriques, en comparant ses prédictions avec les performances mesurées expérimentalement sur l'ensemble de la co-simulation.

7.1 Expérimentations pour calibration du modèle

Tout d'abord, nous commençons par déterminer les paramètres qui nous aident à calibrer nos deux modèles approximatifs, à l'aide d'une série d'expérimentations élémentaires. Dans ce chapitre, nous avons pu évaluer et valider nos deux modèles sur les deux cas d'utilisation IDEAS et Enerbat3 (voir la section 5.3 et la section 6.1 de ce chapitre). Mais dans les sections qui suivent, nous revenons sur le cas d'utilisation Enerbat3 (livraison cl3) comportant 3 bâtiments (31 FMU) afin d'évaluer notre modèle approximatif de performances.

7.1.1 Phase de calcul

Comme indiqué précédemment dans la section 5.2 du chapitre 5, on ne peut pas accéder au nombre d'instructions de nos FMU pour calculer leur temps de traitement (il est impossible d'accéder au code source de la FMU). Pour remédier à cette situation, nous nous appuyons sur une hypothèse et une observation de relation linéaire entre le nombre d'équations d'une FMU et son temps de calcul, en utilisant une régression linéaire. Cette régression linéaire nous fournit alors le moyen de prédire approximativement le temps de calcul d'une FMU (voir l'équation 5.23 qui se trouve dans le chapitre 5), afin d'en déduire le temps global de calcul de tout le système.

Cette étude est réalisée sur un problème élémentaire Enerbat3 livraison 3 contenant 11 FMU (le nombre d'équations est fourni par les développeurs des FMU). On a observé, à travers nos expérimentations, que les temps de calculs des FMU suivent à peu près une loi linéaire en fonction du nombre d'équations. Pour cette raison, nous avons matérialisé et tracé cette loi, sur la figure 19, par une simple régression linéaire. Et nous avons obtenu $T_{Calcul} = a \times NB_{Eq} + b$, avec $a = 1.32E^{-4}$ et $b = 6.11E^{-07}$. Cette modélisation nous fournit ainsi une approximation du temps de calcul d'une FMU en fonction de son nombre d'équations pendant un pas de temps.

Nous avons ensuite réalisé un *benchmark* d'Enerbat3-cl3 avec la stratégie de placement *OneFMUOneNode* (en plaçant une FMU par noeud). Le temps de calcul de l'ensemble des FMU d'Enerbat3-cl3 est alors le maximum des temps de traitements de chacune de ses FMU, c'est-à-dire le temps de calcul de la FMU contenant le nombre d'équations le plus élevé. Associé à notre démarche de modélisation, cela revient à prendre le maximum des temps estimés de toutes nos FMU, ce qui correspond au temps estimé T_1 sur la figure 19 (temps d'une FMU de NB_{Eq1} équations). Ce résultat reste une approximation acceptable du temps T_0 mesuré de façon entièrement expérimentale, dû à une FMU possédant NB_{Eq1} équations mais un peu à l'écart de notre régression linéaire.
Notons que dans le cas du modèle IDEAS, on a déterminé aussi une fonction qui estime les temps de calcul de chaque FMU en fonction de son nombre d'équations, afin de prédire finalement le temps de calcul de toute la grappe de FMU.

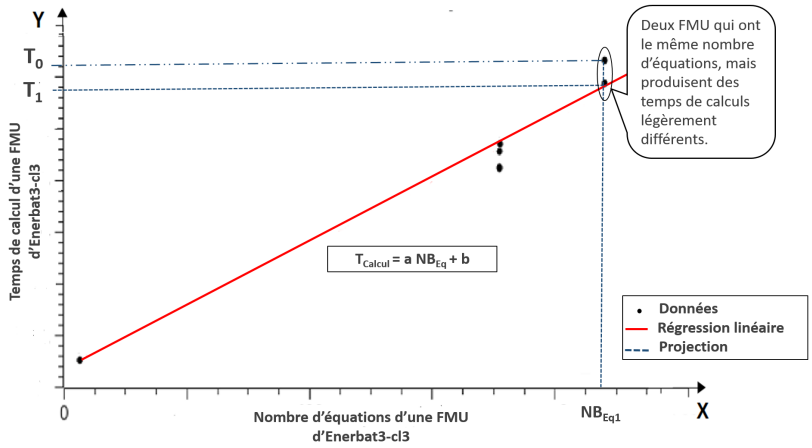


FIGURE 19 – Régression linéaire pour le modèle Enerbat3 livraison cl3 ($T_{Calcul} = a \times Nb_{Eq} + b$)

7.1.2 Phase de communication

Dans cette section, nous mesurons les paramètres qui permettent d'alimenter notre modèle de communication en s'appuyant sur l'équation 5.23 qui permet d'estimer la phase de communication (voir le chapitre 5).

Latence

Dans un premier temps, nous mesurons le paramètre qui influence le plus les performances d'une application distribuée. Ce paramètre de « temps de latence » représente le temps nécessaire pour acheminer le début d'un message d'une source vers une destination, en traversant un réseau d'interconnexion. Dans notre cas, il s'agit de mesurer une « latence applicative » qui représente le temps nécessaire pour envoyer des messages venant d'une application Java qui se trouvent sur un noeud vers une autre application Java se trouvant dans un autre noeud de calcul (temps pour traverser le réseau d'interconnexion plus le temps pour traverser les couches du modèle OSI [4] jusqu'aux applications).

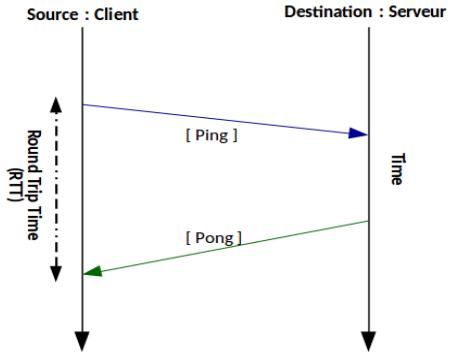


FIGURE 20 – Principe de *ping pong*

Comme l'illustre la figure 20, pour mesurer ce paramètre réseau, nous nous sommes appuyés sur le modèle de *Ping Pong*. De manière générale, le processus maître (source) sauvegarde le temps d'envoi sur le noeud de calcul de la source en utilisant sa propre horloge locale avant d'envoyer le message *« ping »* encapsulé dans un paquet au processus esclave (destination). Une fois que la destination reçoit ce paquet, le processus esclave renvoie à son tour un nouveau paquet contenant le message *« pong »* à la source du message. Dès que cette dernière reçoit le deuxième paquet, on peut calculer le temps d'un aller/retour d'un message (*ping/pong*), qui représente le temps d'un *Round Trip Time* (RTT). Le temps d'une latence est alors : ($latence = \frac{RTT}{2}$), voir la figure 20.

Dans nos expérimentations, nous avons trouvé les valeurs résumées dans le tableau ci-dessous :

Cluster	**latence moyenne**	**Variablité**
Sarah	5.230E-5 s	±20%
Cameron	4.608E-5 s	±8%

Notons que durant nos expériences nous avons remarqué que la latence est variable sur nos deux clusters, comme l'indique le tableau ci-dessus. Cette variabilité contribue à rendre nos modèles approximatifs et non pas exacts.

4. *Open Systems Interconnection*

Bande passante

Le deuxième paramètre est la bande passante qui détermine le débit maximum théorique entre deux noeuds de calcul, c'est un paramètre qui mesure la quantité de données traversant un canal de communication. Pour mesurer la bande passante, nous avons installé la commande « *Iperf* » sur l'ensemble des machines distribuées. Cet outil est lancé sur deux machines distantes, l'une des deux machines doit être configurée en tant que client et l'autre en tant que serveur. Comme l'illustre les figures 21 et 22, un des nœuds de calcul lance le serveur *Iperf* sur le port 5001 avec $-s$, et le second lance le client *Iperf* avec $-c$ en précisant l'adresse IP du serveur.

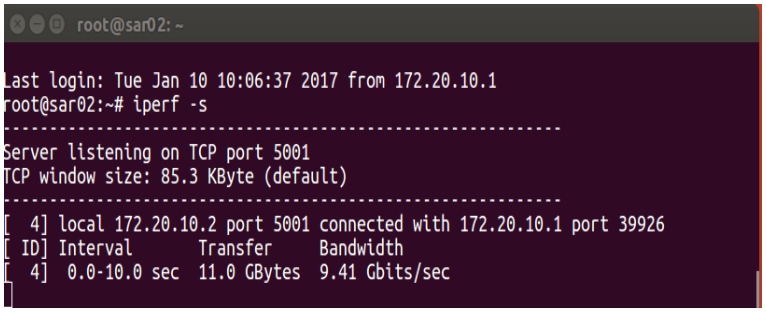


FIGURE 21 – Noeud de calcul « Serveur »

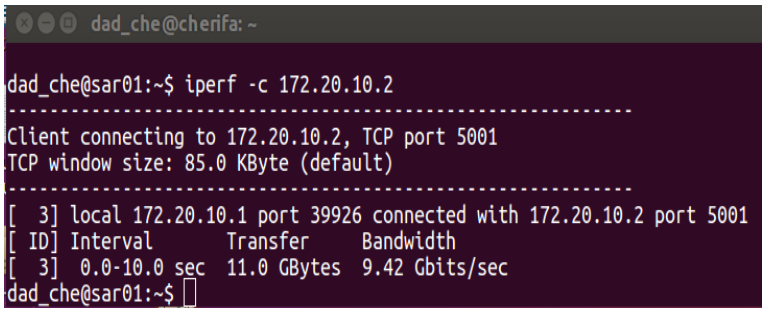


FIGURE 22 – Noeud de calcul « Client »

Comme le montre le tableau ci-dessous, nous avons obtenu des valeurs proches des capacités théoriques de nos réseaux d'interconnexion (9.4 Gbits/s au lieu de 10 Gbits/s).

Switch	**Mesure obtenue (bande passante)**	**Stabilité**
Sarah 10 Gbits/s	9.42 Gbit/s	stable
Cameron 10 Gbits/s	5.90 Gbit/s	± 15%

Nous avons estimé la taille approximative des messages par la moyenne des tailles de tous les messages envoyés, et nous avons trouvé 42 octets avec une variabilité de ±18%. Vu que dans le cas de DACCOSIM chaque lien entre deux FMU peut envoyer une ou

plusieurs données à la fois, nous avons extrait les cardinalités de chaque liaison dans le système, en nous appuyant sur la matrice de communication obtenue à partir du modèle traité. Grâce aux deux informations (taille de message et cardinalité des liaisons), nous avons pu calculer la quantité de communications par FMU (voir le chapitre 5 qui explique la modélisation).

7.2 Confrontation modélisation/expérimentation

Afin d'analyser les performances globales obtenues par notre modèle approximatif, nous allons les comparer avec les performances mesurées purement expérimentalement. Pour ce faire, nous avons effectué cette comparaison sur le placement de référence *OneFMUOneNode* appliqué au *use-case* Enerbat3 livraison cl3 comportant 3 bâtiments (CalComms *use-case*).

Durée de simulation	**Pas de temps**	**Nombre d'itérations**	**Théorie : prédictions de modèle**	**Mesure : DACCOSIM**
864000 s	10 s	86400	117,09 s	118,84 s
864000 s	15 s	57600	078,06 s	079,53 s
864000 s	30 s	28800	039,03 s	040,54 s
864000 s	60 s	14400	019,51 s	021,25 s
259200 s	10 s	25920	035,11 s	036,22 s
259200 s	15 s	17280	023,41 s	024,25 s
259200 s	30 s	8640	011,70 s	012,48 s
259200 s	60 s	4320	005,85 s	006,61 s

Comme l'illustre le tableau ci-dessus, nous remarquons que la méthode expérimentale et la méthode théorique sont proches. La différence est relativement faible, dûe à la variabilité de certaines mesures élémentaires comme la latence qui joue un rôle important dans l'estimation des performances. Notre modèle donne des temps plus optimistes, dû au fait que les mesures qui alimentent notre modèle sont des données moyennes des mesures élémentaires.

Au final, les performances obtenues dans cette section montrent la bonne précision de notre modèle global en les comparant avec celles mesurées lors d'expérimentations. En outre, ce modèle prédictif est prometteur car il peut remplacer les *benchmarks*. Il est donc possible d'appliquer ce modèle global à n'importe quel problème à traiter pour répartir les différentes FMU sur des clusters multi-cœurs, mais en l'alimentant avec les paramètres liées à la fois au problème traité et au cluster utilisé.

8 Conclusion

Ce chapitre s'est concentré sur l'évaluation et la validation de nos approches de répartition des FMU avec des applications à petite et à grande échelle. Pour exécuter nos co-simulations, nous avons utilisé des architectures distribuées variées comportant des nœuds de calculs multi-coeurs. L'ensemble des FMU se répartit et s'exécute de façon transparente sur un cluster grâce à la fois à des mécanismes de répartition des FMU sur des noeuds virtuels, à la génération de codes de simulations parallèles pour chaque noeud virtuel, et au *mapping* des noeuds virtuels sur les noeuds physiques de clusters de multi-coeurs.

Dans la première partie des expérimentation, nous avons prouvé, grâce à des cas d'utilisation métiers fournis par EDF R&D, que selon le problème traité, il existe une ou des approches qui donnent des solutions efficaces avec de meilleures performances que des méthodes basées sur des connaissances métiers simples. La deuxième partie a prouvé que nos solutions passent à l'échelle soit en traitant des *use-cases* basés sur la stratégie de réplication d'un cas simple, soit en traitant de gros cas d'utilisation métiers. Nous avons ainsi pu valider la faisabilité d'un passage à l'échelle en co-simulant 81 FMU, 442 FMU et 1042 FMU, et en utilisant des nœuds de calcul multi-coeurs sur des réseaux d'interconnexion standards et de haute performance. Notre modèle théorique utilisant des mesures expérimentales élémentaires a été validé sur un *benchmark* dans la dernière partie de ce chapitre.

CONCLUSION ET PERSPECTIVES

Les réseaux de distribution actuels deviendront des réseaux intelligents (ou des *Smart Grids)* grâce à l'introduction des technologies de l'information et de la communication dans tous les composants de leur système depuis la téléconduite jusqu'aux compteurs des clients. Ces technologies ajouteront aux systèmes électriques une capacité d'adaptation visant à maintenir leur performance actuelle dans un nouveau contexte de production de plus en plus décentralisé. De manière générale, ces réseaux électriques intelligents (*Smart Grids)* vont être constitués d'un grand nombre de sous-systèmes hétérogènes en interaction permanente, de manière à garantir l'équilibre entre la production et la consommation d'électricité sans surdimensionner les ouvrages, c'est-à-dire en jouant sur différents leviers de flexibilité.

Le developpement de tels systèmes nécessite évidemment une étude approfondie de leur fonctionnement avant toute mise en œuvre sur le terrain. EDF souhaite pour cela s'appuyer sur la modélisation et la simulation de ces systèmes, en partageant certains travaux de recherche avec l'institut *RISEGrid*, notamment pour étudier et observer leur fonctionnement et leur fiabilité. Compte-tenu de l'hétérogénéité des composants d'un *Smart Grid*, l'intégration système ne peut se concrétiser sans la co-simulation qui reste la manière la plus prometteuse pour envisager une approche globale. Dès lors, nous avons travaillé sur un outil de co-simulation distribué, nommé DACCOSIM pour *Distributed Architecture for Controlled CO-SIMulation*, développé par EDF et CentraleSupélec, et qui a largement évolué durant les travaux de cette thèse [30].

1 Synthèse des travaux de la thèse

Le but global de cette thèse a été de répondre aux verrous induits par la co-simulation distribuée, appliquée à une classe de problèmes associés aux *Smart Grids*, en fournissant

des méthodologies et des algorithmes de distribution appliqués à large échelle. En s'appuyant sur DACCOSIM, cette thèse a visé en fait deux axes d'études différents :

— Le comportement global d'un co-simulateur de *Smart Grid* et ses interactions systèmes : il s'est agi de concevoir un modèle d'interaction et de synchronisation qui intègre des simulations en temps continu discrétisé (pas de temps) et des simulations événementielles en temps discret.

— La qualité des performances de grosses co-simulations distribuées : il s'est agi de répartir efficacement les FMU du système sur les ressources informatiques disponibles, afin de réaliser un passage à l'échelle.

Contrairement aux solutions usuelles de co-simulation traditionnelles, nous avons proposé un modèle qui interface une grappe de composants physiques (DACCOSIM) sur un bus d'événements. Ce dernier assure un routage des événements et gère entièrement la progression des calculs des deux types de composants : gestion des pas de temps variable, possibilité de *rollback*, émission d'événements, etc. En recourant à un modèle classique de co-simulation, nous avons pu l'enrichir et l'améliorer en rajoutant deux sous-couches différentes : une sous-couche qui explore le futur de la grappe physique, et une sous-couche qui permet de synchroniser les données échangées entre les deux types de composants du système. Cette solution optimiste permet de garantir un compromis entre précision des calcul (qualité des résultats) et performance.

Malgré le fait que notre solution assure un compromis entre précision et performance, elle peut influencer négativement les performances, car la partie physique explore son futur ce qui parfois double le temps de calcul. Pour cette raison, nous nous sommes ensuite orientés vers une solution plus générique qui propose des améliorations de la norme FMI. Ces ajouts rendaient la norme plus souple en permettant une détection précise et rapide des événements, ce qui a fait l'objet d'une publication au *IEEE International Conference on Emerging Technologies and Factory Automation* [71].

Le deuxième axe d'étude de cette thèse a été d'étudier les performances des applications de type co-simulation de *Smart Grids* basées sur des FMU, en utilisant des clusters de PC multi-cœurs. Vu que la simulation la plus gourmande en calcul dans un *Smart Grid* a trait aux composants physiques, nous avons écarté les composants événementiels dont les temps de simulation restent négligeables face aux simulateurs de systèmes physiques. Dans ce contexte, l'objectif a été de répartir ces boîtes grises (FMU) sur un ensemble de machine multi-coeurs afin d'aboutir à des co-simulations à la fois précises et performantes. Étant donné qu'on ne connait pas certaines informations des FMU (on ne connaît ni leurs poids de calcul ni leur intensité de communication), nous avons proposé

trois approches dirigées par les graphes de FMU, et/ou par l'architecture matérielle utilisée. Nous avons prouvé que nos approches sont efficaces sur des cas d'utilisation fournis par EDF R&D. De plus, nos solutions passent à l'échelle sur des cas d'utilisation métiers extensibles. Ces travaux ont fait l'objet d'une publications au 25th *IEEE International Conference on Enabling Technologies : Infrastructure for Collaborative Enterprises* [22], et d'une autre publication au *12th International Modelica Conference 2017* [75].

Enfin, nous avons conclu ce mémoire par une proposition d'un modèle théorique basé sur des mesures expérimentales élémentaires, un modèle qui permet d'estimer à la fois le temps de calcul et le temps de communication. Ce dernier doit être alimenté par des données du problème traité et par des paramètres du cluster utilisé. Nous avons pu évaluer et valider ce modèle par une comparaison de ses prévisions avec des performances entièrement mesurées expérimentalement. Il apparaît possible d'utiliser ce modèle pour trouver une bonne répartition des FMU de la co-simulation sans passer par de longues expérimentations sur cluster (gain de temps pour l'utilisateur).

2 Perspectives

À notre connaissance, les travaux de cette thèse fournissent pour la première fois une étude approfondie des performances sur des simulations massivement parallèles de type *Smart Grid* et basées sur la norme FMI. Ils ouvrent au final plusieurs champs d'investigation :

— **Intégration de propositions dans la norme FMI**
La norme FMI est de plus en plus utilisée pour co-simuler des systèmes cyber-physiques car elle peut représenter n'importe quel composant sous forme de FMU. Il est possible de l'améliorer en lui appliquant nos propositions issues de cette thèse, afin de mieux traiter les événements provenant d'alarmes ou d'unité de contrôle-commande. Une fois adoptées par le projet FMI, le codage logiciel (API JavaFMI) sera assuré par EDF et ses co-traitants.

— **Introduction du domaine « Télécom » dans la co-simulation**
Dans cette thèse, nous nous sommes focalisés sur la co-simulation de la partie purement physique du *Smart Grid*, en considérant que le temps de simulation de la partie événementielle est négligeable. Nous avons déjà pu réaliser plusieurs co-simulations de réseaux électriques en ne considérant que les composants physiques. Néanmoins, pour pouvoir co-simuler un *Smart Grid* encore plus réaliste, il faudrait certainement modéliser plus finement le réseau télécom pour pouvoir intégrer la latence ou la disponibilité des composants télécoms du système.

— **Implémentation d'un outil de découpage en FMU**
Afin de pouvoir distribuer la co-simulation, il est nécessaire d'avoir tout d'abord une solution ou un outil qui permette de découper automatiquement le système en un ensemble de FMU relativement équilibrées tant du point de vue de la charge de calcul que de l'intensité des échanges. De plus, il est souhaitable de rajouter des méta-données supplémentaires dans chacune des FMU afin d'accéder à des informations comme le nombre d'instructions, le nombre d'équations, les opérations utilisées... Cela aidera l'algorithme de répartition à estimer les vrais charges de calcul des FMU.

Étude des futurs systèmes énergétiques

Au fil des 3 ans de cette thèse, il apparaît de plus en plus clairement que la taille des systèmes complexes rentrant dans le champ d'étude d'EDF devrait fortement augmenter, en passant progressivement de réseaux purement électriques à des systèmes énergétiques incluant notamment différents réseaux (d'électricité, de chaleur), avec une numérisation de plus en plus poussée. Les composants du système se traduisent par des composants de type « contrôle-commande » plus intelligents et d'avantage répartis, ils deviennent de plus en plus communiquant. L'étude de ces nouveaux systèmes complexes nécessitera de nombreuses co-simulations hétérogènes et large échelle, et ne pourra se faire qu'avec des outils comme DACCOSIM.

LISTE DES SYMBOLES

A&A *Agents & Artefacts*

API *Application Programming Interface*

BASH *Bourne-Again SHell*

BLAS *Basic Linear Algebra Subroutines*

BLAS *Basic Linear Algebra Subroutines*

CMB *Chandy Misra Bryant*

CS *Co-Simulation*

DACCOSIM Distributed Architecture for Controlled CO-SIMulation

DAE *Differential Algebraic Equations*

DAGs *Directed Acyclic Graphs*

DESS *Differential Equation System Specification*

DEV&DESS *Discrete EVent and Differential Equation Specified Systems*

DEVS *Discrete EVent System specification*

DGEMM *Double-precision General Matrix Multiply*

DSL *DACCOSIM Script Langage*

DSL *DACCOSIM Script Language*

EDF Électricité De France

EPOCHS *Electric POwer and Communication Synchronizing Simulator*

FCFS *First Come First Served*

FMI *Fonctional Mock-up Interface*

FMU *Fonctional Mock-up Unit*

FOM *Federation Model Template*

FRR *Family Round Robin*

G-DEVS *Generalized Discrete EVent System Specification*

GECO *Global Event-Driven Co-Simulation Framework*

HIL *Hardware-In-the-Loop*

HLA *High Level Architecture*

HPC *High-Performance Computing*

IFSpec *Interface Specification*

InProc *Inside-Process*

INSPIRE *Integrated Co-Simulation of Power and ICT Systems for Real-Time Evaluation*

JRE *Java Runtime Environment*

JVM *Java Virtual Machine*

LBCComms *Load Balancing And Clustering Communications*

ME *Model Exchange*

MIL *Model-In-the-Loop*

MPI *Message-Passing Interface*

MTAR *Maximum Time Advance Request*

NS2 *the Network Simulator*

NUMA *Non Uniform Memory Architecture*

ODE *Ordinary Differential Equation*

OMT *Object Model Template*

OPNET *OPtimized Network Engineering Tools*

RISEGrid *Research Institute for Smarter Electric Grids*

RR *Round Robin*

RTI *Run Time Infrastructure*

RTT *Round Trip Time*

SIL *Software-In-the-Loop*

SMA *Systèmes Multi-Agents*

TIGs *Task Interaction Graphs*

TSO *TimeStamp Order*

UMA *Uniform memory Architecture*

VTB *Virtual Test Bed*

WG *Working Groups*

TABLE DES FIGURES

LISTE DES TABLEAUX

BIBLIOGRAPHIE

[1] Smart grids-cre. http ://www.smartgrids-cre.fr/.

[2] *Model-Based Integration Platform for FMI Co-Simulation and Heterogeneous Simulations of Cyber-Physical Systems*, volume Proceedings of the 10th International Modelica Conference, Lund University, Solvegatan 20A, SE-223 62 LUND, SWEDEN, 03/2014 2014. Modelica Association and Linkoping University Electronic Press.

[3] Enrique Alba. *Parallel Metaheuristics : A New Class of Algorithms.* Wiley-Interscience, 2005.

[4] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), 1967.

[5] M. U. Awais, W. Mueller, A. Elsheikh, P. Palensky, and E. Widl. Using the hla for distributed continuous simulations. In *2013 8th EUROSIM Congress on Modelling and Simulation*, pages 544–549, Sept 2013.

[6] M. U. Awais, P. Palensky, A. Elsheikh, E. Widi, and S. Matthias. The high level architecture RTI as a master to the functional mock-up interface components. In *International Conference on Computing, Networking and Communications (ICNC)*, San Diego, USA, Jan. 2013.

[7] Ruben Baetens, Roel De Coninck, Filip Jorissen, Damien Picard, Lieve Helsen, and Dirk Saelens. Openideas-an open framework for integrated district energy simulations. In *In Proceedings of Building Simulation 2015, Building simulation, India*, 2015.

[8] Jens Bastian, Christoph Clauß, Susann Wolf, and Peter Schneider. Master for co-simulation using fmi. 2011.

[9] T. Blochwitz, M. Otter, M. Arnold, C. Bausch, C. Clauß, H. Elmqvist, A. Junghanns, J. Mauss, M. Monteiro, T. Neidhold, D. Neumerkel, H. Olsson, J. v. Peetz, S. Wolf, Atego Systems Gmbh, Qtronic Berlin, Fraunhofer Scai, and St. Augustin. The functional mockup interface for tool independent exchange of simulation models. In *Proceedings of the 8th International Modelica Conference*, pages 105–114, 2011.

[10] Julien Bourgeois and François Spies. Performance prediction of an nas benchmark program with chronosmix environment. In *Euro-Par 2000, Parallel Processing, 6th International Euro-Par Conference, Munich, Germany, August 29 - September 1, 2000, Proceedings.*, pages 208–216, 2000.

[11] Ilhem Boussaid. *Improvement of metaheuristics for continuous optimization.* Theses, Université Paris-Est, June 2013.

[12] Tracy D Braun, Howard Jay Siegel, Noah Beck, Ladislau L Bölöni, Muthucumaru Maheswaran, Albert I Reuther, James P Robertson, Mitchell D Theys, Bin Yao, Debra Hensgen, and Richard F Freund. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 61(6) :810 – 837, 2001.

[13] Benoît Bréholée and Pierre Siron. Design and implementation of a hla inter-federation bridge. In *European Simulation Interoperability Workshop - EURO SIW 2003*, Stockholm, Sweden, 2003.

[14] D. Broman, C. Brooks, L. Greenberg, E. A. Lee, M. Masin, S. Tripakis, and M. Wetter. Determinate composition of fmus for co-simulation. In *2013 Proceedings of the International Conference on Embedded Software (EMSOFT)*, pages 1–12, Sept 2013.

[15] Benjamin Camus, Virginie Galtier, Mathieu Caujolle, Vincent Chevrier, Julien Vaubourg, Laurent Ciarletta, and Christine Bourjot. Hybrid co-simulation of fmus using dev&dess in MECSYCO. In *Proceedings of the Symposium on Theory of Modeling & Simulation, TMS/DEVS 2016, part of the 2016 Spring Simulation Multiconference, SpringSim '16, Pasadena, CA, USA, April 3-6, 2016*, page 8, 2016.

[16] Benjamin Camus, Thomas Paris, Julien Vaubourg, Yannick Presse, Christine Bourjot, Laurent Ciarletta, and Vincent Chevrier. MECSYCO : a Multi-agent DEVS Wrapping Platform for the Co-simulation of Complex Systems. Research report, LORIA, UMR 7503, Université de Lorraine, CNRS, Vandoeuvre-lès-Nancy ; Inria Nancy - Grand Est (Villers-lès-Nancy, France), Sep 2016.

[17] K. M. Chandy and J. Misra. Distributed simulation : A case study in design and

verification of distributed programs. *IEEE Transactions on Software Engineering*, SE-5(5) :440–452, Sept 1979.

[18] Gilbert Chen and Boleslaw K. Szymanski. Lookahead, rollback and lookback : Searching for parallelism in discrete event simulation, 2002.

[19] Wei Chen. Task partitioning and mapping algorithms for multi-core packet processing systems. Master's thesis, Amherst, Massachusettes, USA, Feb 2009.

[20] M. J. Clement, M. R. Steed, and P. E. Crandall. Network performance modeling for pvm clusters. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, pages 6–6, 1996.

[21] Cherifa Dad, Stéphane Vialle, Mathieu Caujolle, Jean-Philippe Tavella, and Michel Ianotto. Parallelization, Distribution and Scaling of Multi-Simulations on Multi-Core Clusters, with DACCOSIM Environment, Mar 2016. Scientific and Technical Report.

[22] Cherifa Dad, Stéphane Vialle, Mathieu Caujolle, Jean-Philippe Tavella, and Michel Ianotto. Scaling of distributed multi-simulations on multi-core clusters. In *25th IEEE International Conference on Enabling Technologies : Infrastructure for Collaborative Enterprises, WETICE 2016, Paris, France, June 13-15, 2016*, pages 142–147, 2016.

[23] Judith S. Dahmann, Richard M. Fujimoto, and Richard M. Weatherly. The department of defense high level architecture. In *Proceedings of the 29th Conference on Winter Simulation*, WSC '97, pages 142–149, Washington, DC, USA, 1997. IEEE Computer Society.

[24] Mohammad I. Daoud and Nawwaf Kharma. A high performance algorithm for static task scheduling in heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 68(4) :399 – 409, 2008.

[25] J. Dingel, D. Garlan, and C. Damon. Bridging the hla : problems and solutions. In *Proceedings. Sixth IEEE International Workshop on Distributed Simulation and Real-Time Applications*, pages 33–42, 2002.

[26] D. Efimov, A. Polyakov, A. Levant, and W. Perruquetti. Discretization of asymptotically stable homogeneous systems by explicit and implicit euler methods. In *2016 IEEE 55th Conference on Decision and Control (CDC)*, pages 5545–5550, Dec 2016.

[27] Abir Ben Khaled-El Feki. *Distributed real-time simulation of numerical models : application to power-train*. Theses, Université de Grenoble, May 2014.

[28] Richard M. Fujimoto. Time management in the high level architecture. *SIMULATION*, 71(6) :388–400, 1998.

[29] Richard M. Fujimoto. *Parallel and Distribution Simulation Systems*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1999.

[30] V. Galtier, S. Vialle, C. Dad, Jean-Philippe Tavella, Jean-Philippe. Lam-Yee-Mui, and G. Plessis. FMI-Based Distributed Multi-Simulation with DACCOSIM. In *Proceedings of the 2015 Spring Simulation Multiconference (TMS/DEVS'15)*, 2015.

[31] H. Georg, S. C. Müller, C. Rehtanz, and C. Wietfeld. Analyzing cyber-physical energy systems :the inspire cosimulation of power and ict systems using hla. *IEEE Transactions on Industrial Informatics*, 10(4) :2364–2373, Nov 2014.

[32] H. Georg, C. Wietfeld, S. C. Müller, and C. Rehtanz. A hla based simulator architecture for co-simulating ict based power system control and protection systems. In *2012 IEEE Third International Conference on Smart Grid Communications (SmartGridComm)*, pages 264–269, Nov 2012.

[33] The IGRIDA Computing Grid. http://igrida.gforge.inria.fr/tutorial.html.

[34] Paul Grun. Introduction to infiniband for end users. *White paper, InfiniBand Trade Association*, 2010.

[35] John L. Gustafson, Gary R. Montry, and Robert E. Benner. Development of parallel methods for a 1024-processor hypercube. *SIAM Journal on Scientific and Statistical Computing*, 9(4) :609–638, 1988.

[36] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I (2Nd Revised. Ed.) : Nonstiff Problems*. Springer-Verlag New York, Inc., New York, NY, USA, 1993.

[37] Georg Hanno, Sven Christian Muller, Dorsch Nils, Rehtanz Christian, and Christian Wietfeld. Inspire : Integrated co-simulation of power and ict systems for real-time evaluation. In *Smart Grid Communications (SmartGridComm), 2013 IEEE International Conference on*, Oct 2013.

[38] Jin-Kao Hao, Philippe Galinier, and Michel Habib. Métaheuristiques pour l'optimisation combinatoire et l'affectation sous contraintes. *Revue d'Intelligence Artificielle*, 13(2) :283–324, 1999.

[39] Roger W. Hockney. The communication challenge for mpp : Intel paragon and meiko cs-2. *Parallel Computing*, 20(3) :389–398, 1994.

[40] John H. Holland. *Adaptation in Natural and Artificial Systems.* MIT Press, Cambridge, MA, USA, 1992.

[41] K. Hopkinson, Xiaoru Wang, R. Giovanini, J. Thorp, K. Birman, and D. Coury. Epochs : a platform for agent-based electric power and communication simulation built from commercial off-the-shelf components. *IEEE Transactions on Power Systems*, 21(2), May 2006.

[42] K.M. Hopkinson, K.P. Birman, R. Giovanini, D.V. Coury, X. Wang, and J.S. Thorp. EPOCHS : Integrated commercial off-the-shelf software for agent-based electric power and communication simulation. In *Proceedings of the 2003 Winter Simulation Conference*, Dec 2003.

[43] Mathieu Jan. *JUXMEM : un service de partage transparent de données pour grilles de calcul fondé sur une approche pair-à-pair.* PhD thesis, Université Rennes 1, 2006.

[44] David R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7(3) :404–425, jul 1985.

[45] C. Kleijn. Introduction to hardware-in-the-loop simulation. 2014.

[46] Jean-Philippe LamYeeMui. Daccosim developer's guide. Technical report, EDF and CentraleSupelec, 2014.

[47] Jean-Philippe Lam-Yee-Mui and Mathieu Caujole. Daccosim developer's guide. Technical report, EDF and CentraleSupelec, 2015.

[48] A. H. Land and A. G. Doig. An automatic method for solving discrete programming problems. *ECONOMETRICA*, 28(3) :497–520, 1960.

[49] W. Li, M. Ferdowsi, M. Stevic, A. Monti, and F. Ponci. Cosimulation for smart grid communications. *IEEE Transactions on Industrial Informatics*, 10(4) :2374–2384, Nov 2014.

[50] W. Li, A. Monti, M. Luo, and R. A. Dougal. Vpnet : A co-simulation framework for analyzing communication channel effects on power systems. In *2011 IEEE Electric Ship Technologies Symposium*, pages 143–149, April 2011.

[51] V. Liberatore and A. Al-Hammouri. Smart grid communication and co-simulation. In *IEEE 2011 EnergyTech*, pages 1–5, May 2011.

[52] H. Lin, S. S. Veda, S. S. Shukla, L. Mili, and J. Thorp. Geco : Global event-driven co-simulation framework for interconnected power system and communication network. *IEEE Transactions on Smart Grid*, 3(3) :1444–1456, Sept 2012.

[53] D. L. Long and L. A. Clarke. Task interaction graphs for concurrency analysis. In *[1988] Proceedings. Second Workshop on Software Testing, Verification, and Analysis*, pages 134–135, Jul 1988.

[54] Muthucumaru Maheswaran, Shoukat Ali, Howard Jay Siegel, Debra Hensgen, and Richard F. Freund. Dynamic mapping of a class of independent tasks onto heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, 59(2) :107 – 131, 1999.

[55] Nakul Manchanda and Karan Anand. Non-uniform memory access (numa). *New York University*, 2010.

[56] Nedialko S. Nedialkov and John D. Pryce. Solving differential-algebraic equations by taylor series (i) : Computing taylor coefficients. *BIT Numerical Mathematics*, 45(3) :561–591, 2005.

[57] M. D. Petersheim and S. N. Brennan. Scaling of hybrid electric vehicle power-train components for hardware-in-the-loop simulation. In *2008 IEEE International Conference on Control Applications*, pages 720–726, Sept 2008.

[58] Claudius Ptolemaeus, editor. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.

[59] Sk Riazur Raheman, Amiya Kumar Rath, and M Hima Bindu. An overview of program slicing and its different approaches. *International Journal of Advanced Research in Computer Science and Software Engineering*, 3(11) :435–442, 2013.

[60] Y. Rangelov, N. Nikolaev, and M. Ivanova. The iec 61850 standard - communication networks and automation systems from an electrical engineering point of view. In *2016 19th International Symposium on Electrical Apparatus and Technologies (SIELA)*, pages 1–4, May 2016.

[61] Rafael H. Saavedra and Alan Jay Smith. Analysis of benchmark characteristics and benchmark performance prediction. Technical report, EECS Department, University of California, Berkeley, Dec 1992.

[62] P Sadayappan, F Ercal, and J Ramanujam. Cluster partitioning approaches to mapping parallel programs onto a hypercube. *Parallel Computing*, 13(1) :1–16, 1990.

[63] Tom Schierz, Martin Arnold, and Christoph Clauß. Co-simulation with communication step size control in an fmi compatible master algorithm. In *Proceedings of the 9th International MODELICA Conference ; September 3-5 ; 2012 ; Munich ; Germany*, number 76, pages 205–214. Linköping University Electronic Press ; Linköpings universitet, 2012.

[64] Nicolas Sébastien. *Distributed and Parallel Execution of Agent-Oriented Simulation.* Theses, Université de la Réunion, November 2010.

[65] L. F. Shampine. Error estimation and control for odes. *Journal of Scientific Computing*, 25(1) :3–16, Oct 2005.

[66] Julien Siebert. *Approche multi-agent pour la multi-modélisation et le couplage de simulations.* Theses, Université Henri Poincaré - Nancy I, Sep 2011.

[67] CentraleSupélec Stéphane Vialle. Big data : Informatique pour les données et calculs massifs. 2017.

[68] P Straffin Jr. Newton's method and fractal patterns. *Applications of Calculus*, 3 :68–84, 1991.

[69] Xian-He Sun and John L. Gustafson. Toward a better parallel performance metric. *Parallel Computing*, 17(10) :1093 – 1109, 1991.

[70] Jean-Philippe Tavella, Mathieu Caujolle, Charles Tan, Gilles Plessis, Mathieu Schumann, Stéphane Vialle, Cherifa Dad, Arnaud Cuccuru, and Sébastien Revol. Toward an Hybrid Co-simulation with the FMI-CS Standard, Apr 2016. Research Report.

[71] Jean-Philippe Tavella, Mathieu Caujolle, Stéphane Vialle, Cherifa Dad, Charles Tan, Gilles Plessis, Mathieu Schumann, Arnaud Cuccuru, and Sebastien Revol. Toward an accurate and fast hybrid multi-simulation with the FMI-CSstandard. In *21st IEEE International Conference on Emerging Technologies and Factory utomation, ETFA 2016, Berlin, Germany, September 6-9, 2016*, pages 1–5, 2016.

[72] Slurm Team. https://slurm.schedmd.com/bluegene.html.

[73] Okan Topçu, Umut Durak, Halit Oğuztüzün, and Levent Yilmaz. *High Level Architecture*, pages 39–66. Springer International Publishing, Cham, 2016.

[74] H. Topcuoglu, S. Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3) :260–274, Mar 2002.

[75] Stéphane Vialle, Jean-Philippe Tavella, Cherifa Dad, Remi Corniglion, Mathieu Caujolle, and Vincent Reinbold. Scaling FMI-CS Based Multi-Simulation Beyond Thousand FMUs on Infiniband Cluster. In Modelica Association, editor, *12th International Modelica Conference 2017*, Prague, Czech Republic, May 2017.

[76] B. Wissiler. Systèmes et modèles. introduction critique à l'analyse de systèmes. In *Seuil, Paris (fra)*, page 255 p, 1977.

[77] Gregory Zacharewicz, Claudia Frydman, and Norbert Giambiasi. Lookahead computation in g-devs/hla environment. *Simulation News Europe Journal (SNE)*, 16(2) :15–24, 2006.

[78] Gregory Zacharewicz, Claudia Frydman, and Norbert Giambiasi. Lookahead Computation in G-DEVS/HLA Environment. *Simulation News Europe Journal (SNE)*, 16(2) :15–24, Sep 2006.

[79] Bernard P. Zeigler. *Theory of Modeling and Simulation.* John Wiley, 1976.

[80] Bernard P Zeigler. Embedding dev&dess in devs. In *DEVS Integrative Modeling & Simulation Symposium*, pages 125–132, 2006.