Intégration de simulateurs existants à une plateforme de co-simulation basée sur DEVS

Julien Vaubourg
julien.vaubourg@inria.fr

Vincent Chevrier
vincent.chevrier@univ-lorraine.fr

Laurent Ciarletta
laurent.ciarletta@univ-lorraine.fr

Inria Grand Est, Villers-les-Nancy, F-54600, France

13 novembre 2014

Table des matières

1 Introduction

Nous nous intéresserons dans ce document à la co-simulation (co-operative simulation). L'objectif est de proposer une méthodologie pour intégrer un simulateur existant à une plateforme de co-simulation existante. Nous nous restreindrons aux plateformes de co-simulation basées sur le formalisme DEVS [6], qui donne l'assurance de pouvoir intégrer tous les formalismes [7][8]. Cette méthodologie devra donc donner les étapes clés pour rendre compatible un simulateur existant avec les concepts de DEVS, du niveau le plus formel jusqu'à l'implémentation qui permettra de l'instrumentaliser logiciellement. Nous identifierons deux principales catégories de simulateur auquel nous pouvons être confrontés lors d'une intégration, en proposant deux preuves de concept, qui serviront d'exemple d'application des étapes que nous aurons proposées.

L'étude des co-simulations hétérogènes est particulièrement présente dans le cadre des simulations pour les systèmes électriques intelligents (*smart grids*) [1][2][3][4], notamment pour l'interconnexion entre les modèles de systèmes électriques et les modèles de réseaux numériques qui utilisent des formalismes différents. Nous prendrons soin de choisir nos deux preuves de concept en fonction de ce cas d'étude, avec un standard pour la simulation de modèles de systèmes électriques, et un simulateur populaire de réseaux numérique. Ce travail s'apparente aux travaux de [5], qui se limitent à une intégration au niveau logiciel.

La section 2 de ce document proposera une problématique concrète, qui nécessitera de faire co-simuler un simulateur qui n'est pas été prévu à cet effet, et qui devra donc être intégré à une plateforme de co-simulation. La section 3 proposera un rappel des concepts fondamentaux de DEVS, et présentera la plateforme AA4MM [9] qui servira d'exemple de plateforme de co-simulation basée sur DEVS. Trois étapes pour l'intégration des simulateurs seront identifiées, à partir des pré-requis de DEVS. La section 4 proposera une première preuve de concept, avec l'intégration du standard industriel FMI, que nous qualifierons de « cas favorable » à l'intégration. La section 5 proposera l'intégration du simulateur de réseaux numériques NS-3, qui sera notre seconde preuve de concept, que nous qualifierons de « cas défavorable », et qui permettra de répondre à la problématique. Nous conclurons enfin en section 6, à la fois sur la pertinence des trois étapes identifiées vis-à-vis de nos deux exemples, et sur la pertinence de la réponse à la problématique. Nous conclurons également sur l'utilité de l'intégration de ces deux exemples à une même plateforme de co-simulation vis-à-vis du domaine de la simulation des systèmes électriques intelligents.

2 Problématique

La figure 1 représente un réseau numérique dans sa forme la plus minimale, avec deux machines et un lien qui les relie. Le modèle de réseau est composé de trois principaux modèles de composant : un modèle pour la machine N^1, un pour le lien L^1 et un pour la seconde machine N^2 (figure 2).

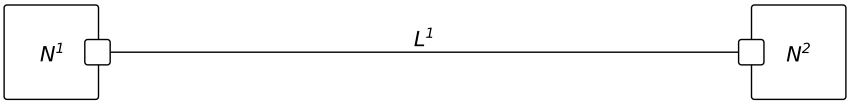

FIGURE 1 – Réseau simpliste à modéliser (système).

On ne souhaite pas modéliser nous-même ces trois composants, et nous préférerons réutiliser des modèles déjà existants, conçus par des experts du domaine. La qualité d'un modèle dépend de la pertinence de sa représentation du système vis-à-vis de la précision de ce qu'on cherche à mesurer ou observer. Mais la façon dont il peut être initialisé ou couplé avec d'autres modèles compte également. Trouver les trois modèles dont nous avons besoin pour composer notre modèle de réseau ne suffira pas, aussi prouvés et éprouvés puissent-ils être, s'ils n'ont pas été tous conçus pour fonctionner avec le même simulateur et s'il n'y pas d'interconnexions possibles entre les logiciels. Réimplémenter les modèles existants pour pouvoir tous les exécuter dans un même simulateur prend du temps, augmente le risque d'une erreur d'implémentation (alors que celle du modèle initial est éventuellement réputée fiable) ou n'est simplement pas possible si le modèle n'est pas open-source. La réimplémentation impose également d'assumer la responsabilité d'intégrer soi-même les évolutions ou les corrections du modèle initial.

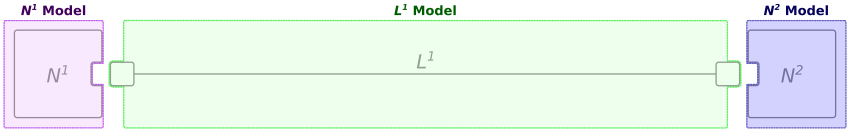

FIGURE 2 – Modélisation du système.

On souhaite donc pouvoir modéliser notre réseau numérique en couplant nos trois modèles de composant, éventuellement en passant par une co-simulation lorsque leurs implémentations ne sont pas prévues pour le même logiciel de simulation. Il faut donc trouver une solution pour faire co-simuler l'ensemble des ces logiciels. Une seconde problématique, qui formalise des objectifs similaires, est de réussir à distribuer facilement une simulation de réseaux numériques avec un logiciel de simulation qui ne le permet pas.

3 Co-simulations hétérogènes

3.1 Formalisme DEVS

Le formalisme *Discret EVent System* (DEVS) [6] est le formalisme le plus général pour les modèles à événements discrets. Il est particulièrement intéressant dans notre cas puisqu'il a été prouvé qu'il pouvait également intégrer tous les autres types de formalisme [7][8]. Puisqu'il est suffisamment générique pour décrire toutes les politiques d'ordonnancement du point de vue de l'exécution, il peut être utilisé comme modèle d'exécution pour faire évoluer une co-simulation hétérogène.

Les simulateurs intégrés à cette co-simulation doivent respecter le protocole des simulateurs DEVS. Il est défini comme utilisant quatre types de message (figure 3 inspirée de [6]) :

— messages d'initialisation `(i,t)`, reçus par tous les simulateurs avant le lancement de la co-simulation ;
— messages de transition d'état interne `(*,t)`, permettant de planifier les événements internes ;
— messages de sortie `(y,t)`, permettant de notifier les autres simulateurs d'événements externes ;
— messages d'entrée `(x,t)`, permettant de recevoir un événement externe.

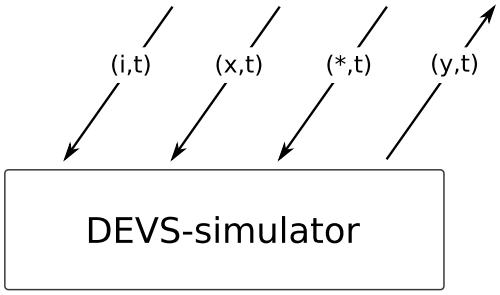

FIGURE 3 – DEVS simulator protocol.

Un simulateur DEVS doit également être capable de communiquer la date de son dernier événement exécuté, ainsi que celle du prochain qu'il traitera. La caractérisation des événements internes et externes dépendront du formalisme des simulateurs et de la sémantique des couplages.

La section suivante introduit un exemple de plateforme de co-simulation utilisant DEVS comme modèle d'exécution.

3.2 Plateforme AA4MM

Agents & Artefacts for Multi-Simulation (AA4MM) [9] est un méta-modèle pour la co-simulation qui s'appuie sur le paradigme *Agents & Artefacts* [10], une extension du paradigme des systèmes multi-agents.

Son formalisme repose sur celui de DEVS, qu'il utilise comme modèle d'exécution. La particularité du méta-modèle AA4MM est de proposer une approche totalement décentralisée de la co-simulation, en s'appuyant sur des algorithmes DEVS dont la preuve mathématique a été apportée qu'ils ne pouvaient pas engendrer d'inter-blocages ni enfreindre les contraintes de causalité. La plateforme AA4MM a été conçue pour faire co-simuler des modèles et des simulateurs hétérogènes. AA4MM correspond donc à l'une des solutions possibles pour résoudre notre problématique.

Le méta-modèle est décrit par trois niveaux ontologiques [1] :

1. le niveau paradigmatique décrivant les concepts ;
2. le niveau générique décrivant leurs spécialisations ;
3. et leur spécialisation au niveau domaine.

Les concepts et leurs spécialisations décrits dans la figure 4 ont leur propre représentation graphique (cf. figure 5 par Camus et al. [**?**]).

Outre les modèles, on y distingue trois éléments principaux :

— **m-agent :** agent en charge de la gestion des interactions entre son modèle associé et les autres qui composent la co-simulation ;
— **artefact de modèle :** propose le support permettant aux m-agents de communiquer avec les modèles (entrées/sorties, avancement du temps, etc.) ;
— **artefact de couplage :** propose le support permettant aux m-agents de communiquer entre eux, en intégrant des opérations pour conditionner les données (ou la représentation du temps) de façon à ce que les échanges soient cohérents.

Différents éléments sont instanciés de façon distincte pour gérer une co-simulation parallèle, permettant de collecter les résultats (et reliée à la co-simulation principale par le biais de simples artefacts de couplage) :

— **m-agent d'observation :** agent en charge de recevoir les données des autres m-agents à superviser ;
— **artefact d'observation :** propose le support permettant aux m-agents d'observation d'évacuer les données reçues vers un modèle d'observation (uniquement destiné à les formater et éventuellement les traiter) ;

1. Définis par Camus et al. [**?**].

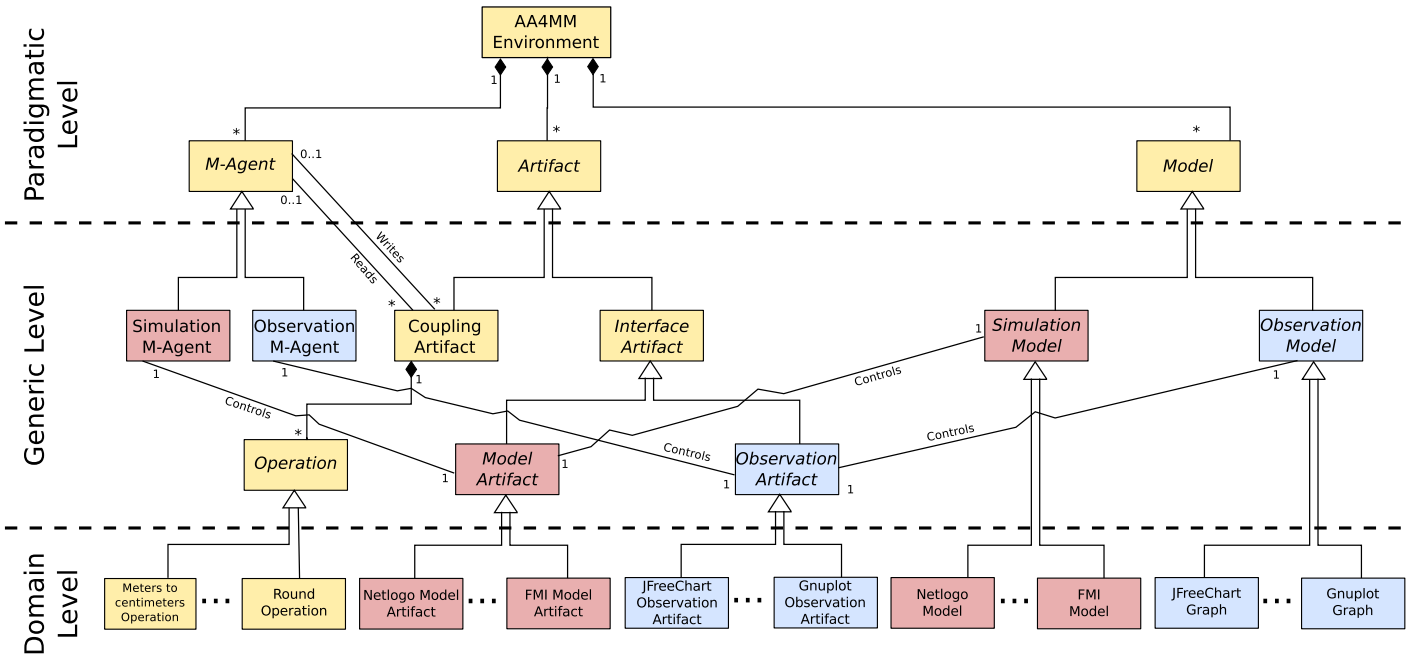

FIGURE 4 – Le méta-modèle AA4MM et ses 3 niveaux.

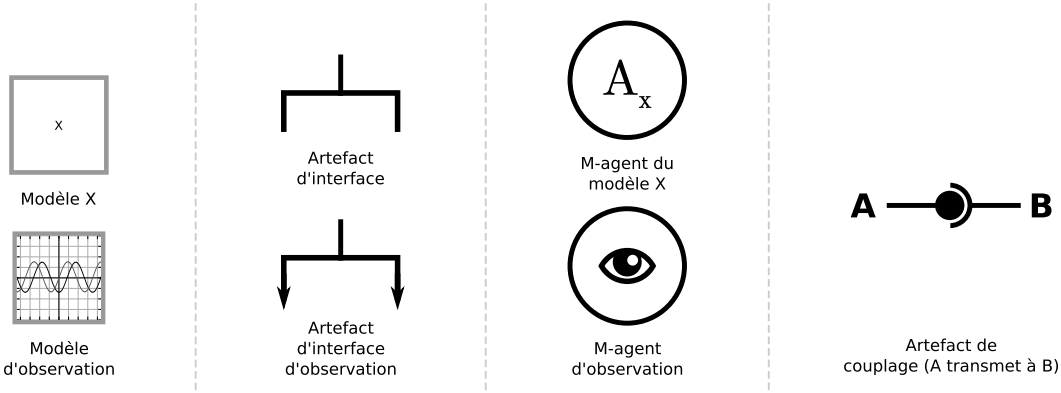

FIGURE 5 – Notations liées au formalisme de AA4MM.

Les spécialisations au niveau domaine dans la figure 4 permettent de faire le lien entre le monde AA4MM et le monde des simulateurs supportés.

L'intégration d'un nouveau simulateur à une co-simulation AA4MM requiert de suivre trois étapes systématiques, décrites dans la section suivante.

3.3 Intégration d'un nouveau simulateur

3.3.1 $1^{ère}$ étape : relier les concepts de DEVS avec les concepts du simulateur

Intégrer un nouveau simulateur à une co-simulation AA4MM consiste principalement à être capable de développer son artefact de modèle au niveau domaine (figure 4).

DEVS étant utilisé comme modèle d'exécution pour les co-simulations AA4MM, le simulateur à intégrer ainsi que ses modèles, doivent être compatibles DEVS. Il a été prouvé que ce dernier pouvait englober tous les autres formalismes [7][8]. Ainsi, dès lors que le formalisme du simulateur a été correctement décrit, il est possible de calquer sa sémantique sur celle

de DEVS.

Les concepts de DEVS comportent (figure 6) :
— une dynamique, décrivant la manière dont le système se comporte dans le temps ;
— un état, décrivant le système à un instant donné ;
— des ports, permettant de fournir des entrées et sorties aux modèles ;
— des événements internes, planifiés par les modèles ;
— des événements externes, indiquant qu'il s'est passé quelque chose d'intéressant lors de la dernière simulation de la dynamique.

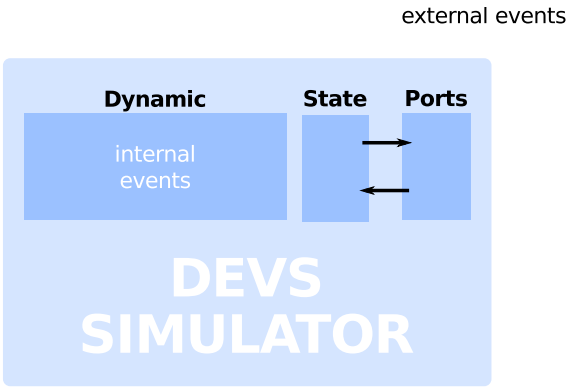

FIGURE 6 – Formalisme DEVS

Caractériser l'ensemble des concepts DEVS listés ci-dessus en fonction du domaine du simulateur à intégrer, permet d'obtenir des modèles compatibles DEVS et donc de passer à la seconde étape.

3.3.2 2^e étape : ajouter des contrôles externes DEVS aux modèles

Les algorithmes DEVS nécessitent le support de quatre principaux contrôles externes, pour pouvoir intégrer un simulateur (figure 7) :

1. *Traiter le prochain événement interne* (qu'on désignera ensuite par `executeModel`). Simuler la dynamique en avançant l'horloge du simulateur jusqu'au temps de l'événement interne le plus imminent, puis en modifiant l'état du modèle.
2. *Récupérer le temps du prochain événement interne* (`getNextEventTime`). Déterminer quel est le prochain événement interne (e.g. en consultant une pile d'événements) qui sera traité par la dynamique et retourner la date qui lui est associée.
3. *Traiter le prochain événement externe* (`setInputEvent`). Simuler la dynamique en avançant l'horloge du simulateur jusqu'au temps de l'événement externe entrant, puis en modifiant l'état du modèle en fonction du port et de la valeur associés à l'événement.
4. *Récupérer un événement externe* (`getOutputEvent`). Retourner la valeur d'un éventuel événement externe généré pour un port spécifié, lors de la dernière simulation de la dynamique, avec la valeur du temps qui lui est associée.

Ces contraintes permettent aux simulateurs impliqués dans la co-simulation d'être instrumentalisés depuis l'extérieur (par un *master* de co-simulation, un agent, etc). Cette instrumentalisation permet de synchroniser leurs horloges et de concrétiser leurs interactions en leur permettant d'échanger des données au cours de la co-simulation.

Caractériser le comportement à adopter pour chacun de ces quatre contrôles, en fonction du formalisme du simulateur à intégrer, permet de passer à la troisième et dernière étape pour l'intégration du simulateur à la co-simulation, au niveau implémentation.

3.3.3 3^e étape : implémenter les interfaces DEVS du simulateur

La dernière étape consiste à implémenter les contrôles externes, en s'assurant que les modèles sont compatibles. La caractérisation des comportements des contrôles externes lors de l'étape précédente permet de se restreindre désormais

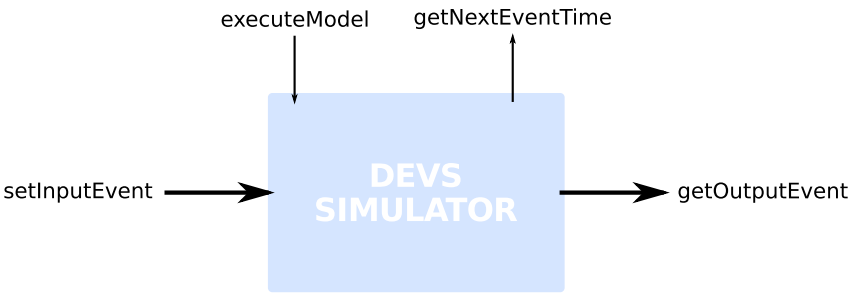

FIGURE 7 – Les quatre principaux contrôles d'un simulateur DEVS.

aux problèmes techniques liés à leur implémentation.

Il existe deux principaux types de simulateur, rendant cette étape finale plus ou moins aisée :

1. **Dans le cas le plus favorable**, le simulateur est conçu pour co-simuler. Il dispose donc déjà d'interfaces de contrôles externes. En étudiant leur comportement, il est possible de déterminer quelle fonction (ou quelle combinaison de fonctions) est équivalente à chacun des quatre contrôles étudiés lors de l'étape précédente. Si le simulateur repose sur un standard, celui-ci permettra de comprendre rapidement si le comportement des contrôles externes proposés est directement compatible avec nos besoins ou non. Dans le meilleur des cas, il n'y aura donc aucune modification à apporter au simulateur, comme à ses modèles, pour qu'il soit compatible DEVS.
2. **Dans le cas le plus défavorable**, le simulateur n'est pas conçu pour co-simuler. Il n'a donc pas été conçu pour avoir des interactions avec l'extérieur et ne propose par conséquent aucune interface pour des contrôles externes. Il faudra donc modifier l'implémentation du simulateur et/ou de ses modèles pour obtenir ces contrôles. Selon le logiciel de simulation, ces modifications seront plus ou moins lourdes et plus ou moins possibles. Il faudra notamment modifier la boucle événementielle pour qu'elle soit capable de s'interrompre, soit pour synchroniser l'horloge du simulateur avec celle des autres participants à la co-simulation, soit pour introduire un événement externe. Il faudra également être capable de modifier l'état des modèles de telle sorte que l'événement externe soit bien intégré, selon sa sémantique.

Une fois ces trois étapes achevées, le simulateur est prêt à être intégré dans une co-simulation AA4MM, avec son propre artefact de modèle.

3.4 Conclusion

La plateforme AA4MM permet de faire co-simuler des simulateurs hétérogènes et peut donc intégrer des simulateurs de réseaux numériques. Basée sur DEVS, son seul pré-requis pour l'ajout d'un simulateur est qu'il soit compatible avec le protocole des simulateurs DEVS, et que ses modèles soient compatibles avec le formalisme DEVS. Calquer les concepts du simulateur sur ceux de DEVS a fait l'objet des deux premières étapes pour l'intégration, qui ont été présentées. La troisième étape, qui concerne l'implémentation des contrôles, est la plus délicate puisqu'elle peut éventuellement conduire à une incompatibilité technique incontournable.

Il suffit donc de trois étapes systématiques à appliquer et de quatre principaux contrôles à implémenter pour pouvoir intégrer un simulateur à une simulation AA4MM.

La représentation de la solution à notre problématique en utilisant la plateforme AA4MM est disponible en figure 8. Elle implique donc deux m-agents AA4MM, qui contrôlent chacun une instance d'un simulateur, qui contiennent eux-même chacun une partie du modèle global à simuler. Chacun des simulateurs nécessite son propre artefact de modèle (quelque soit le modèle utilisé par le simulateur) pour être intégré.

L'exemple de simulateur de réseaux numériques qui sera pris dans la suite de ce document est NS-3. L'intégration de ce simulateur étant un exemple de cas défavorable (non conçu pour la co-simulation), nous étudierons au préalable un cas d'intégration plus favorable, avec un type de simulateur conçu pour co-simuler. C'est l'objet de la section suivante.

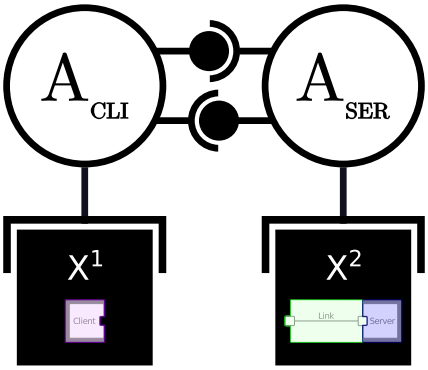

FIGURE 8 – Représentation de la solution à la problématique, avec AA4MM.

4 Intégration du standard FMI

4.1 Présentation du standard FMI

Présentation de FMI à faire

L'intégration des FMU aux co-simulations AA4MM nécessite de suivre les trois étapes systématiques qui ont été présentées lors des sections précédentes.

4.2 1ère étape : relier les concepts de DEVS avec les concepts de FMI

Puisque les FMU respectent les concepts du standard FMI, qui a été complétement formalisé, il est aisé d'identifier les concepts DEVS correspondants :

- **la dynamique :** elle réside dans son solveur, et donc dans notre cas, dans son équation ;
- **un événement interne :** un solveur équationnel étant à changement d'état continu [11], la dynamique sera simulée selon un pas de temps déterminé (éventuellement variable) et un événement interne correspondra donc à une résolution numérique pour un pas de temps ;
- **l'état :** les variables utilisées par l'équation de la dynamique constituent l'état du modèle ;
- **les ports :** le standard FMI prévoit la notion de ports, disponibles en lecture ou en écriture, et associés à des variables de l'état ;
- **un événement externe :** chaque fois que la dynamique sera simulée, l'état pourra avoir évolué, provoquant ainsi un événement externe (un événement entrant - resp. sortant - correspondra donc à la modification - resp. lecture - d'une valeur entre deux résolutions).

La figure 9 synthétise la correspondance entre le formalisme FMI et le formalisme DEVS. La seconde étape peut être abordée.

4.3 2^{e} étape : ajouter des contrôles externes DEVS aux modèles FMI

Les quatre principaux comportements à identifier découlent facilement du travail effectuée lors de la première étape :

1. `executeModel` : exécuter le solveur pour une résolution d'équation jusqu'au prochain pas de temps (cf. figure 10).
2. `getNextEventTime` : retourner la date correspond au prochain pas de temps.
3. `setInputEvent` : exécuter le solveur pour une résolution d'équation jusqu'au temps correspondant à l'événement externe entrant, puis modifier la variable concernée en fonction du port concerné.
4. `getOutputEvent` : retourner la valeur d'une variable entre deux résolutions de l'équation.

Une fois les concepts et les comportements des contrôles bien identifiés, nous pouvons passer à la dernière étape pour l'implémentation.

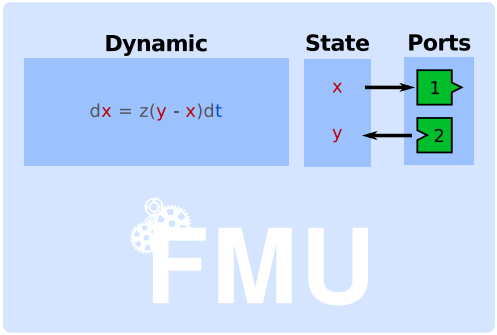

FIGURE 9 – Formalisme FMU (solveur équationnel) calqué sur le formalisme DEVS.

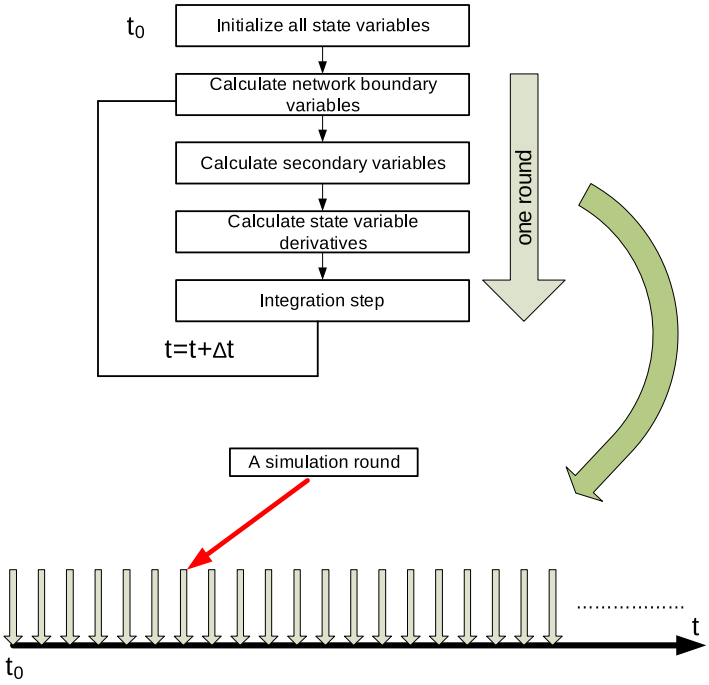

FIGURE 10 – Dynamique par pas de temps d'un solveur à équations (repris de [3]).

4.4 3^{e} étape : implémenter les interfaces DEVS des FMU

Le standard FMI étant conçu pour permettre aux FMU de faire de la co-simulation, l'intégration des simulateurs FMI est un cas favorable. L'utilisation d'un standard garantit également que les différentes implémentations proposeront toutes les mêmes fonctionnalités.

En considérant qu'on utilise l'implémentation JavaFMI de SIANI, il est possible de trouver directement des équivalents à nos quatre interfaces :

1. `executeModel` : la fonction `doStep` permet d'exécuter le solveur pour un pas de temps donné.
2. `getNextEventTime` : les changements d'états étant discrétisés, c'est l'entité qui contrôle la co-simulation qui décide du prochain pas de temps (`getSimulationTime` permettant de récupérer le temps courant de simulation).
3. `setInputEvent` : la fonction `writeVariable` permet de modifier une variable en fonction d'un nom de port, et la fonction `doStep` permet d'exécuter le solveur pour un pas de temps, correspondant dans ce cas au temps théoriquement écoulé entre la dernière exécution du solveur et l'événement externe entrant à traiter (`event.time−getSimulationTime()`).

4. `getOutputEvent` : la fonction `readVariable` permet de lire une variable en fonction d'un nom de port à n'importe quel instant.

Les trois étapes étant complétées, il devient aisée d'écrire l'artefact de modèle pour les simulateurs FMI.

4.5 Conclusion

La création d'un artefact de modèle pour les simulateurs FMI (à solveurs équationnels) n'a posé aucune difficulté majeure. Les trois étapes ont permis de s'interroger sur le formalisme pour aboutir naturellement à l'implémentation. Il s'agissait d'un cas favorable, puisque les simulateurs ont été conçus pour faire de la co-simulation. Le standard a également facilité l'intégration, puisque tous les concepts étaient correctement formalisés.

En complétant ainsi le domaine spécifique et en se contentant d'hériter des propriétés des artefacts de modèle du niveau générique, aucune des fonctionnalités du cœur du méta-modèle n'a été touchée, et toutes ses propriétés (preuves, décentralisation, etc.) restent valides. Le travail effectué est suffisamment générique pour permettre de désormais intégrer n'importe quel modèle FMU à solveur équationnel, et de le faire communiquer avec n'importe quel autre simulateur relié à la co-simulation AA4MM.

La section suivante s'intéresse à l'intégration des simulateurs de réseaux numériques, qui correspondent au type de simulateur impliqué dans notre problématique. Contrairement aux FMU, le simulateur NS-3 pris en exemple n'est pas conçu pour faire de la co-simulation, il s'agit donc d'un cas défavorable. Son intégration fait l'objet de la section suivante.

5 Intégration de NS-3

5.1 Présentation de NS-3

NS-3 est un simulateur de réseaux numériques développé à partir de 2008, sous l'impulsion de l'Université de Washington, la Georgia Institute of Technology, l'US National Science Foundation et le centre Inria de Sophia Antipolis. Il fait suite à son prédécesseur NS-2, qui a rencontré un énorme succès dans la communauté scientifique de la simulation. Pour différentes raisons, dont un changement de choix de langage et une réorganisation de sa structure, les modèles NS-2 sont désormais incompatibles avec NS-3. NS-2 n'est plus activement maintenu et laisse petit à petit la place à NS-3.

La figure 11 donne la tendance de l'intérêt de la communauté des chercheurs, vis-à-vis des principaux simulateurs de réseaux numérique, en fonction des quatre dernières années. Il s'agit de moyennes entre les moteurs de recherche de ACM et IEEE, pour le nombre de papiers qui citent les différents simulateurs pour chaque année de publication. Puisque les papiers qui ont choisi d'utiliser e.g. OMNeT++ pour leurs travaux citent probablement les autres simulateurs pour justifier leur choix, cette étude ne peut prétendre être un miroir exact de la popularité des simulateurs mais au plus une tendance.

On constate d'après la figure 11 que NS-3 est dans le trio de tête du classement, bien qu'il soit le plus jeune des simulateurs du comparatif. NS-2 est en déclin bien qu'il soit encore très populaire, et devrait petit à petit laisser de plus en plus de place à son successeur.

Les simulations NS-3 ont pour particularité de simuler les composants physiques, mais de quasiment émuler les couches logicielles, en reproduisant très finement leur fonctionnement (similaire à celui d'un système GNU/Linux). La section suivante fait le point sur l'organisation des modèles.

5.1.1 Organisation des modèles

Les modèles NS-3 sont composés d'un ensemble de sous-modèles, disponibles sous forme de bibliothèques C++. L'élément de base de NS-3 est le nœud, qui pourrait s'apparenter à un ordinateur dans sa forme la plus simple (sans cartes, sans périphériques et sans applications). Construire un modèle pour NS-3 consiste donc à commencer par équiper les modèles de ces nœuds basiques avec les fonctionnalités nécessaires à la représentation du système cible. Ainsi, selon si le nœud est destiné à modéliser un ordinateur de bureau ou un commutateur réseau, on ajoutera plus ou moins de cartes réseaux et on installera des applications plus ou moins haut niveau. Installer une carte ou une application consiste à instancier un modèle correspondant à cette fonctionnalité, et à utiliser les fonctions d'ajout du modèle de nœud pour l'associer.

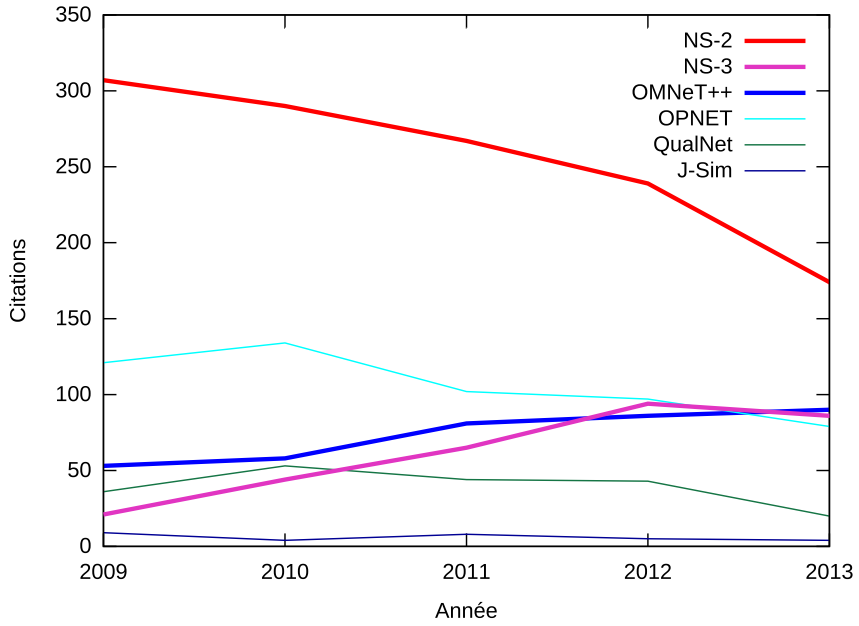

FIGURE 11 – Citation des simulateurs dans la littérature ACM & IEEE (statistiques effectuées à l'aide du nombre de papiers proposés par les moteurs de recherches officiels de ACM (`http://dl.acm.org`) et IEEE (`http://ieeexplore.ieee.org`), en fonction des filtres par année et des différentes orthographes possibles : NS2/NS-2, NS3/NS-3, OMNeT++, QualNet, JSIM/J-SIM), réalisé en mars 2014.

Les différentes bibliothèques C++ correspondant aux différents modèles disponibles constituent le cadriciel NS-3. Une simulation NS-3 correspond donc à l'exécution d'un modèle NS-3 compilé et lié à d'autres modèles.

Les développeurs du simulateur ont souhaité adopter un mode d'organisation qui se veut le plus proche possible de celui correspondant à un vrai système d'exploitation. Ils se sont en particulier directement inspirés du fonctionnement du noyau Linux pour reproduire les mêmes comportements. Ainsi, on retrouve naturellement une structure récursive des modèles, proches de la suite TCP/IP (cf. figure 12) ou plus généralement du modèle en couches OSI. Associer un modèle à un nœud signifie donc généralement associer une arborescence de modèles.

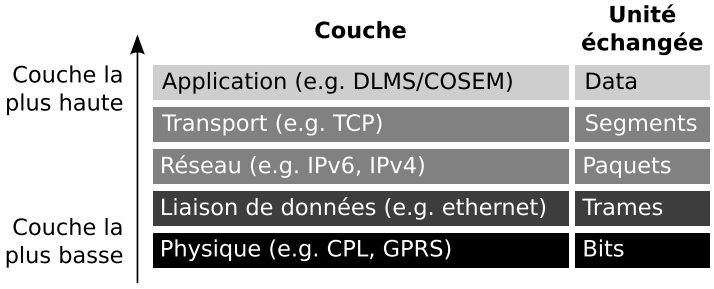

FIGURE 12 – Suite TCP/IP

Afin de ne pas avoir à systématiquement associer les modèles entre eux, de la couche la plus basse à la couche la plus haute, la notion de *helpers* vient compléter la puissance du cadriciel. Ainsi, des objets tels que `PointToPointHelper` permettent de modéliser facilement et rapidement deux cartes réseaux avec un lien entre les deux, le tout configuré pour faire transiter des paquets virtuels d'une carte à l'autre en utilisant le protocole PPP (RFC 1661). Les *helpers* proposent également des fonctions permettant de configurer le lien facilement, notamment en appliquant directement des propriétés dessus tels que *DataRate* (bande passante), *Delay* (délai) ou *InterframeGap* (temps minimum à attendre entre deux trames à transmettre). Enfin, ils disposent en général d'une fonction `Install` qui prend en argument une liste de nœuds (deux au maximum, dans le cas d'un lien PPP) et qui se charge automatiquement de leur associer les cartes réseaux. Tous les modèles NS-3 utilisent ces *helpers*, tant il serait laborieux sinon de les écrire. Ils aident à obtenir des modèles très lisibles,

mais ont l'inconvénient d'ajouter une couche d'astraction importante au fonctionnement du simulateur.

La section qui suit décrit la gestion des événements produits par ces modèles NS-3.

5.1.2 Gestion des événements

Comme la plupart des simulateurs de réseaux numériques, NS-3 utilise une pile d'événements.

Un événement NS-3 est dur à caractériser, et il ne semble y avoir aucune définition officielle de proposée. Dans les faits, au niveau implémentation, un événement se réduit à un appel programmé de fonction. Construire un événement consiste donc à associer un pointeur de fonction (lié à un objet instancié) à une date d'exécution, dans le temps de simulation.

Ainsi, lorsqu'une application reçoit un paquet virtuel et souhaite envoyer immédiatement un paquet en réponse, elle programme un événement associé à sa propre fonction `Send` et au temps courant de simulation. Si le modèle de l'application prévoit que le traitement du paquet reçu est coûteux en temps de simulation, le temps associé à l'événement correspondra au temps courant de simulation augmenté d'un temps de traitement calculé. Lorsque l'événement sera exécuté, la fonction `Send` sera appelée. Celle-ci programmera, à son tour, un événement dans la pile, pour simuler l'envoi du paquet. Le temps qui sera associé à ce nouvel événement correspondra au temps courant de simulation, augmenté du temps estimé par le modèle pour traverser le lien vers l'autre nœud. La fonction qui sera associée correspondra à la fonction `HandleRead` de l'objet modélisant le nœud distant, qui sera chargée de traiter le paquet reçu. Cet algorithme est décrit dans la figure 1 (on considère que les modèles des deux nœuds sont des objets de même type, possédant ces deux fonctions, et ayant pour caractéristique de s'échanger continuellement des paquets).

Algorithm 1 Exemple de gestion des événements dans NS-3, pour un ping pong entre deux nœuds.

```
function HandleRead
    t = CurrentTime()
    c = fixedEventHandlingDuration
    p = other.GetTransferPacket()
    HandleNewPacket(p)
    AddEvent(t + c, this.Send)
end function

function Send
    t = CurrentTime()
    c = fixedTransferDuration
    CreateTransferPacket()
    AddEvent(t + c, other.HandleRead)
end function
```

Les événements peuvent servir à demander l'envoi ou la réception d'un paquet entre nœuds virtuels, comme décrit dans [3] et illustré par la figure 13. Le même mécanisme est utilisé pour passer un paquet d'une couche à l'autre de la suite TCP/IP, pour un même nœud. La transmission d'un seul paquet entre deux nœuds ressemblera donc à l'enchaînement décrit en figure 14 (chacun des nœuds étant une instance de la classe implémentant ces deux méthodes).

La dernière partie de ce tour d'horizon de NS-3 concerne les solutions de distribution de NS-3 qui sont proposées nativement.

5.1.3 Distribution des simulations

Une solution de distribution des simulations est proposée par défaut dans NS-3. Il s'agit d'une distribution via MPI [**?**], un standard très utilisé dans l'industrie et la recherche. Il est particulièrement utilisé sur les clusters informatiques, pour déployer rapidement des calculs parallèles.

Distribuer une simulation NS-3 avec cette simulation consiste simplement à remplacer un lien PPP de son modèle par un lien PPP-MPI. Il faudra ensuite choisir une implémentation de déploiement MPI, pour déployer le binaire NS-3 sur deux machines différentes (ou sur une même machine pour se contenter de faire du multi-thread). Les deux binaires se différencient à l'exécution grâce à l'identifiant MPI qu'ils reçoivent en variable d'environnement. En fonction de ce dernier,

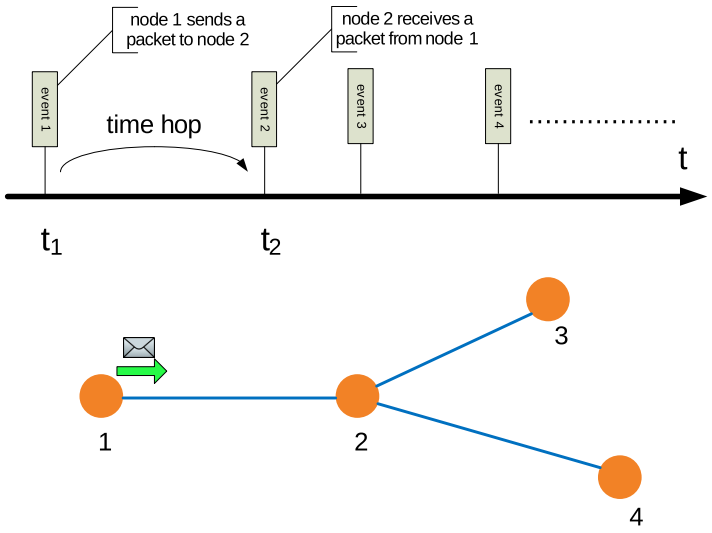

FIGURE 13 – Passage des paquets d'un nœud à l'autre via des événements internes (repris de [3]).

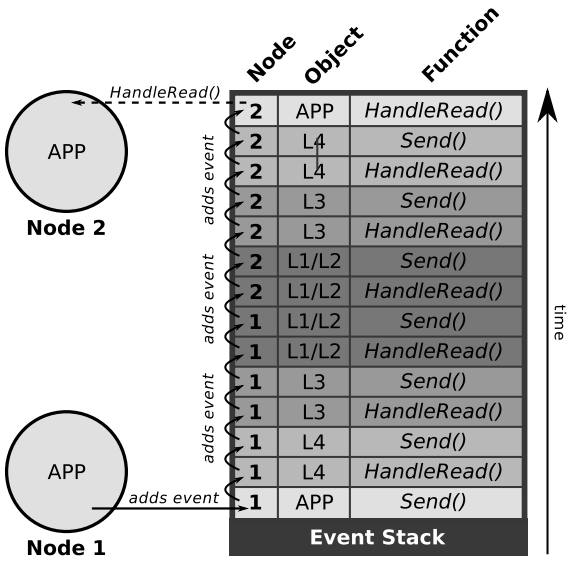

FIGURE 14 – Enchaînement des événements internes pour l'envoi d'un paquet entre 2 nœuds.

ils peuvent donc déterminer leur rôle et savent quelle partie du modèle ils doivent exécuter. Ainsi, chaque binaire n'en exécutera en réalité que la moitié, et récupérera les informations provenant de la seconde moitié via des messages MPI. Un exemple de simulation distribuée via MPI est proposé en figure 15.

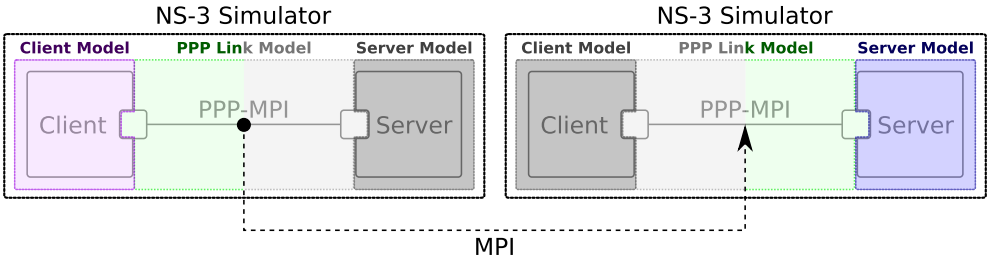

FIGURE 15 – Distribution d'une simulation NS-3 avec MPI

Cette solution permet d'augmenter les capacités de calcul pour une simulation, mais ne permet pas pour autant de faire de la co-simulation. Elle ne fonctionne qu'entre des instances NS-3, et nécessite d'avoir l'intégralité du modèle présent sur chacune d'entre elles. Actuellement, un modèle ne peut être séparé en deux à l'exécution qu'au niveau d'un lien de type PPP, qui ne sera pas systématiquement approprié. Les choix de modélisation ont donc un impact direct sur les possibilités de distribution à l'exécution. Cette solution native de distribution ne pourrait donc pas être une solution satisfaisante pour notre problématique.

Les sections suivantes décrivent l'intégration, étape par étape, de NS-3 à une co-simulation AA4MM.

5.2 Sémantique des couplages

La définition des concepts DEVS dans le cas de NS-3 impose de faire un choix au niveau de la sémantique des couplages souhaitée.

Nous considérerons, dans la suite de ce document, que nous souhaitons utiliser NS-3 pour faire des couplages spatiaux. Ce type de couplage implique que les sorties d'un simulateur correspondent aux entrées du simulateur avec lequel il est couplé, et que le méta-modèle de la multi-simulation représente la topologie du système. Il permet de répondre à notre problématique, avec des événements à échanger qui correspondent à des paquets IP qui transitent. Ce couplage est illustré par l'exemple de la figure 16 (le second simulateur pouvant aussi utiliser un formalisme différent).

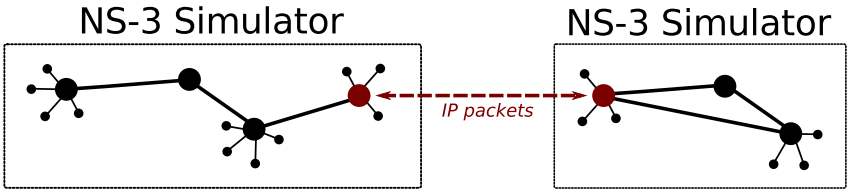

FIGURE 16 – Exemple de couplage spatial, avec deux simulateurs NS-3, qui ont chacun un modèle différent de topologie réseau.

5.3 1ère étape : relier les concepts de DEVS avec les concepts de NS-3

Le rapprochement du formalisme de NS-3 à celui de DEVS donne les concepts suivants, dans le cas d'un couplage spatial :

- **la dynamique :** la pile des événements du simulateur représente la dynamique ;
- **un événement interne :** les événements qui composent la pile d'événements du simulateur peuvent être qualifiés d'événements internes ;
- **l'état :** l'état correspond aux caractéristiques de l'ensemble des composants du réseau modélisé, avec une organisation des variables qui n'est pas standardisée ;
- **les ports :** un port correspond à une variable de l'état permettant de déclencher l'injection ou la récupération d'un paquet IP à un endroit précis de la topologie réseau modélisée.
- **un événement externe :** paquet IP simulé à transférer.

Les concepts DEVS étant reliés (et synthétisés en figure 17), la seconde étape pour l'intégration de NS-3 peut être abordée.

5.4 2^{e} étape : ajouter des contrôles externes DEVS aux modèles NS-3

Les contrôles externes principaux de DEVS peuvent se caractériser de la façon suivante pour NS-3 :

1. `executeModel` : avancer l'horloge jusqu'au temps du prochain événement interne et le dépiler en l'exécutant.
2. `getNextEventTime` : récupérer le temps associé à l'événement interne le plus imminent.
3. `setInputEvent` : avancer l'horloge jusqu'au temps associé à l'événement externe entrant et modifier la variable de l'état correspondant au port indiqué, ou injecter un paquet IP associé dans le modèle en fonction du port indiqué.

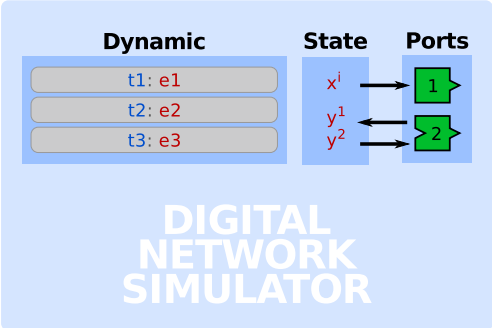

FIGURE 17 – Formalisme des simulateurs de réseaux numériques calqué sur le formalisme DEVS.

4. `getOutputEvent` : retourner la valeur de la variable de l'état correspondant au port indiqué, ou retourner un paquet IP éventuellement généré au niveau du port indiqué, depuis le traitement du dernier événement interne.

Le comportement des contrôles externes DEVS ayant été caractérisé, il reste encore à implémenter les interfaces correspondantes.

5.5 3^e étape : implémenter les interfaces DEVS de NS-3

5.5.1 Ports des modèles

5.5.2 Contraintes

La notion de port que nous avons identifiée lors de la première étape n'existe pas au niveau de l'implémentation de NS-3. N'étant pas conçu pour co-simuler, il ne propose aucune solution pour échanger avec l'extérieur.

Un port devant être capable d'échanger des paquets IP simulés, il doit avoir un équivalent au niveau de la topologie réseau du système modélisé, pour pouvoir le situer dans le modèle. Les paquets IP simulés doivent avoir un point dans leur réseau qui leur permettent de sortir vers un autre simulateur. Ce point doit être atteignable avec du routage simple, comme n'importe quel autres composants du réseau simulé. Selon la destination du paquet IP dans le modèle, il atteindra différents points de sortie vers d'autres simulateurs, qui correspondront aux ports. Ces mêmes ports doivent donc être également capables d'injecter des paquets IP dans le réseau.

Ces ports peuvent se caractériser par des puits, qui stockent tous les paquets IP qu'ils recoivent comme autant d'événements externes sortants à transmettre à l'artefact de modèle (faisant ainsi office de passerelle vers le monde AA4MM). La hiérarchie des modèles de NS-3 permet d'avoir à disposition un large panel de possibilités pour ajouter un puit dans la topologie réseau. Il peut s'agit d'un nœud spécial ou d'un nœud NS-3 classique sur lequel on installe une application spéciale jouant ce rôle, ou encore d'une carte réseau spéciale qu'on installe sur un simple nœud NS-3. La granularité peut aller jusqu'à nous permettre de remplacer la file de paquets d'un nœud pour créer une file spéciale, qui transmet à l'artefact de modèle les paquets qu'elle reçoit et qui se rempli avec les paquets qu'il lui transmet.

Pour décider de l'emplacement idéal pour créer des puits dans le réseau simulé, il faut prendre en compte trois contraintes :

1. la présence du puit ne doit avoir aucune influence sur les résultats de simulation ;
2. les paquets doivent être récupérés au niveau le plus bas (idéalement sans désencapsulation avant d'atteindre le puit) ;
3. les puits doivent pouvoir être reliés à un maximum de composants NS-3 existants, pour ne pas contraindre les modèles.

La première contrainte est nécessaire pour qu'un modèle simulé sur un seul simulateur puisse donner exactement les mêmes résultats de simulation que lorsqu'il est réparti sur plusieurs simulateurs. La seconde contrainte permet de s'assurer que le puit est potentiellement capable de récupérer n'importe quel niveau de paquet (y compris en couche liaison de données) et donc de récupérer l'intégralité des datagrammes qui arrivent jusqu'à lui, comme d'injecter n'importe quel

niveau de paquet qu'on pourrait lui transmettre depuis l'extérieur. Enfin, la dernière contrainte impose de trouver une solution suffisamment générique pour s'assurer qu'elle puisse s'ajouter à n'importe quel modèle NS-3 sans avoir à penser le modèle dans l'objectif de l'utiliser pour faire de la co-simulation (et donc pouvoir réutiliser un maximum de modèles existants).

À partir de ces contraintes, les ports DEVS peuvent être définis, en fonction des possibilités offertes par l'implémentation de NS-3 et la nature de ses modèles.

5.5.3 Définition des ports

Vis-à-vis de la troisième contrainte, le nœud qui accueille le puit doit absolument être un nœud générique NS-3, pour pouvoir être relié à un maximum de modèles existants possibles. Utiliser une simple application virtuelle sur le nœud pour récupérer les paquets et les injecter ne respecte pas la seconde contrainte, qui impose d'être à un niveau beaucoup plus proche de la couche physique. La carte réseau spéciale semble un compromis idéal, puisqu'elle permet d'accéder à l'ensemble des paquets reçus, en ayant en plus l'assurance qu'il n'y a eu aucun traitement avant (qui changerait les entêtes des paquets ou qui modifierait les résultats de simulation). Remplacer la file des paquets du nœd semble aussi efficace et encore plus transparent.

Au niveau purement logiciel, les objets C++ de NS-3 ne sont pas tous interchangeables dans un modèle, aussi facilement les uns que les autres. Ainsi, NS-3 considère qu'une carte réseau est forcément liée au type de lien (physique ou radio) auquel elle permet de se raccorder. Créer une carte réseau spéciale signifie donc en réalité créer une carte réseau spéciale par type de lien existant, ce qui serait fastidieux à déterminer et à réaliser. Concernant la file des paquets, l'implémentation de NS-3 ne permet pas de la remplacer sans réécrire plusieurs classes C++.

La solution qui a finalement été adoptée, est le *wrapper* de carte réseau. NS-3 permet de fournir à n'importe quelle carte réseau, un pointeur vers une fonction C++ à qui seront passés l'intégralité des paquets reçus. La fonction pour faire envoyer un paquet à la carte réseau est également facilement accessible. Ainsi, il suffit de créer un objet qui a connaissance d'une carte réseau, pour qu'il soit capable de récupérer et injecter du trafic et utilisant des variables qu'il ajoute à l'état.

Il suffit désormais d'ajouter des accès vers l'extérieur à n'importe quelle variable de l'état (dont celles correspondant à nos *wrappers*), et nos modèles disposent des ports qu'ils ont besoin. La figure 18 illustre la solution à notre problématique, du point de vue topologie réseau, avec les puits munis de *wrappers* qui servent de ports.

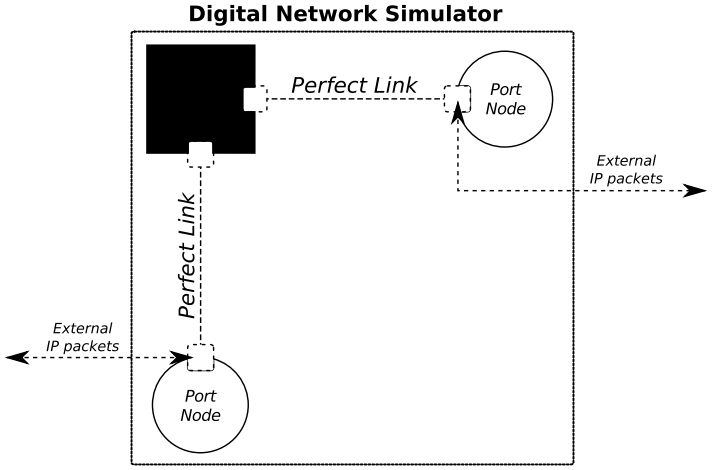

FIGURE 18 – Exemple de deux puits qui servent de ports (la partie noire représentant une topologie de réseau déjà modélisée).

Un avantage annexe, est de simplifier le travail des modélisateurs, pour créer de nouveaux modèles ou en modifier des existants. NS-3 peut se charger seul de l'adressage IP de l'ensemble des cartes réseau de son modèle. Ainsi, il suffit de lui indiquer une liste de nœuds qui sont censés être dans le même réseau (en choisissant IPv6 ou IPv4), toutes les relier, et le routage virtuel de tout le modèle est fonctionnel. Le peuplement automatique des tables de routage n'est possible que si NS-3 connaît l'ensemble des machines accessibles depuis le modèle. Il ne pourra donc pas créer les routes virtuelles dans la

première instance NS-3, qui permettront d'accéder au nœud qui est dans une instance de simulation tierce. Les nœuds qui sont ajoutés pour servir de puits peuvent servir à résoudre ce problème. Il suffit simplement de donner les mêmes adresses à leur carte réseau que celles que possède la carte réseau du nœud qui est sur l'instance de simulateur distante. Ainsi, les tables de routage peuvent être automatiquement complétées des deux côtés, grâce aux nœuds simples qui servent de support à nos puits. Si un réseau complexe se trouve de l'autre côté d'un puit, il faudra soit représenter l'ensemble du réseau derrière le puit, avec d'autres faux nœuds, soit simplement ajouter les routes manuellement.

L'utilisation de ces ports NS-3 impose de réussir à les situer dans la topologie réseau représentée par le modèle. L'emplacement choisi pour les ports conduira ensuite au découpage de la topologie réseau, permettant ainsi le découpage du modèle en plusieurs modèles distincts.

5.5.4 Modification de la topologie réseau du modèle pour ajouter les ports

L'exemple de la problématique pourrait correspondre à un cas d'utilisation basique, avec un modèle destiné à simuler un simple *ping* entre deux machines. Le système serait constitué d'un nœud qui sert de client (son modèle envoie des paquets UDP à intervalles réguliers), d'un lien quelconque et d'un second nœud qui sert de serveur (son modèle se contente de répondre en envoyant un paquet UDP, chaque fois qu'on lui en transmet un). Ce cas d'utilisation est illustré par la figure 19.

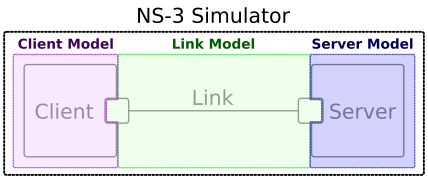

FIGURE 19 – Modèle NS-3 pour un simple ping UDP entre deux machines.

On considère qu'on a ce modèle fonctionnel à disposition, et qu'on souhaite séparer le modèle du client du reste du modèle réseau, pour l'isoler dans sa propre instance de simulation NS-3. Il s'agit donc de distribuer la simulation entre deux simulateurs (deux instances NS-3), en répartissant les modèles de composants, et en s'assurant qu'on pourra à terme remplacer une instance de NS-3 par n'importe quel autre simulateur. Pour intégrer nos deux modèles de réseau séparés à une co-simulation, nous devons donc les rendre compatibles avec les primitives DEVS, notamment en ajoutant les ports manquants. La solution, telle qu'elle a été conçue ci-avant, est illustrée en figure 20.

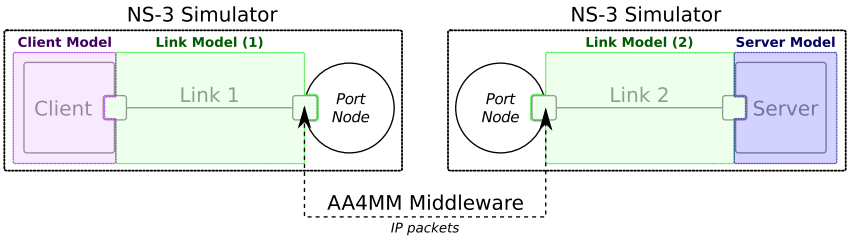

FIGURE 20 – Modèle NS-3 pour un simple ping UDP, réparti entre deux instances de simulateur.

Pour ajouter le nœud qui fait office de port dans chacun des deux modèles de réseau, nous avons dû utiliser un modèle de lien des deux côtés. Le modèle global n'est donc pas cohérent avec notre modèle de base, puisque les contraintes du lien s'appliqueront deux fois. Les résultats de simulation seront donc différents selon si la simulation est distribuée ou non, ce qui ne respecte pas la première contrainte que nous nous sommes fixés. Il n'est pas possible de se contenter de considérer qu'il suffit que chacun des liens ait des performances deux fois moins importantes que le lien initial pour retrouver les mêmes résultats. En effet, selon le type de lien, les caractéristiques ne seront pas obligatoirement linéaires et il pourra éventuellement y avoir un facteur stochastique qui entre en jeu (par exemple un bruit aléatoire sur la ligne, qui sera donc calculé deux fois).

La solution qui a été adoptée pour résoudre ce problème est de considérer que l'un des deux liens est parfait, c'est à dire qu'il a des caractéristiques permettant aux données qu'il transportent de toujours arriver intactes (séquence de

bits identiques), sans retransmission, et avec une vitesse telle que le simulateur ne pourra pas considérer que sa présence influence les temps de simulation. Pour ce faire, il suffit d'utiliser un modèle de lien PPP (le type de lien le plus simple existant dans NS-3) et de l'initialiser pour faire en sorte qu'il ait un débit et une latence permettant des transferts dépassant la résolution du temps du simulateur (limitée à la femtoseconde dans NS-3). La figure 21 illustre notre cas d'utilisation, avec le lien parfait, qui permet aux données de passer de la carte réseau du client à celle qui permet d'accéder au vrai modèle de lien, dans le second simulateur, de façon instantanée vis-à-vis du temps de simulation. Le modèle global deviendrait donc identique au modèle initial.

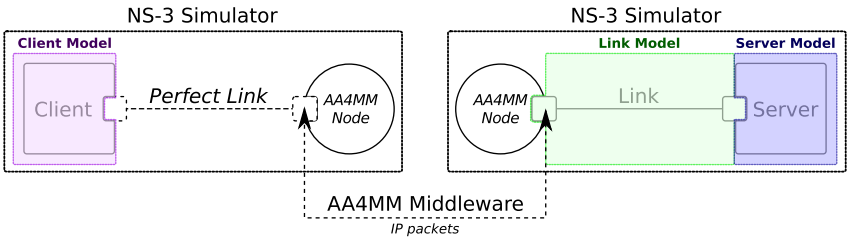

FIGURE 21 – Modèle NS-3 pour un simple ping UDP, réparti entre deux instances de simulateur, avec un lien parfait.

On constate pourtant toujours une différence dans les temps de simulation, entre une exécution mono-simulateur et multi-simulateur, dans certains cas. En effet, le lien parfait étant un lien PPP, il s'agit d'une connexion en pair-à-pair qui ne nécessite pas d'échanges de messages en couche liaison de données, de type NDP (pour IPv6) ou ARP (pour IPv4) pour associer une adresse MAC à une adresse IP. Si le paquet est transmis au port de la seconde instance de simulation avec l'adresse MAC du nœud à qui il doit le transmettre, il n'y aura jamais de découverte d'adresses MAC. Les échanges qui sont fait dans ce cadre peuvent pourtant consommer du temps de simulation si le type de lien utilisé les nécessite. On obtient alors des résultats de simulation avec des temps plus courts lorsque la simulation est distribuée avec un lien autre que pair-à-pair. Pour résoudre ce problème, il est essentiel de faire en sorte que le port qui reçoit le paquet IP utilise sa propre table de correspondance MAC-IP pour adresser le message, et déclenche éventuellement des échanges de niveau liaison de données pour découvrir les adresses manquantes, en utilisant le lien qui fait parti du modèle initial (et donc en prenant en compte ses performances vis-à-vis du temps de simulation). Les paquets doivent être injectés au niveau IP par les *wrappers*. Ces derniers doivent également s'interdire de transmettre les paquets de niveau liaison de données au port distant, pour que ces échanges restent locaux, sauf dans le cas où les puits représentent un équipement de ce niveau ou inférieur (e.g. switch ou hub).

Avec cette solution, il est désormais possible d'utiliser des ports pour implémenter l'artefact de modèle NS-3 et permettre l'échange d'événements externes dans le cadre d'une co-simulation. La section suivante évoque le second problème qui se pose généralement lorsque le simulateur à intégrer n'est pas conçu pour co-simuler, qui concerne l'intégration des événements externes dans la boucle événementielle du simulateur.

5.5.5 Dynamique et synchronisation

NS-3 n'étant pas conçu pour co-simuler, son objectif est de traiter le plus rapidement possible l'ensemble des événements de sa pile interne, pour exécuter ses simulations dans un temps le plus réduit possible. Puisqu'il n'a pas à avoir conscience de l'influence de simulateurs externes, il ne prévoit aucune solution pour intégrer un événement externe entre deux événements internes traités.

La boucle événementielle de NS-3 doit donc impérativement s'interrompre entre chaque événement interne traité, pour intégrer les événements externes au bon moment, et attendre les autres simulateurs de la muti-simulation, lorsque ceux-ci ne sont pas capables de lui garantir qu'ils ne généreront pas d'événements externes pour lui au-delà de son temps de simulation courant (e.g. avec l'algorithme DEVS de co-simulation de Chandy-Misra-Bryant[6]). Il faut donc de nouveau modifier l'implémentation des modèles, pour forcer un traitement des événements en pas-à-pas.

La solution adoptée au niveau logiciel est d'ajouter une attente de signal à la fin de chacune des itérations de sa boucle événementielle. L'envoi de ce signal pourra être utilisé par l'artefact de modèle pour demander au simulateur de traiter le prochain événement interne, en s'assurant au préalable que l'ensemble des événements externes a été traité.

Les deux limitations au niveau implémentation étant résolus, la troisième étape pour intégrer NS-3 à une co-simulation AA4MM est désormais possible.

5.5.6 Création des interfaces DEVS

Les sections précédentes ont permis de faire le point sur les limites de l'instrumentalisation de NS-3, qui n'est pas conçu pour co-simuler. Des solutions ont été trouvées pour ajouter la notion de ports, et pouvoir contrôler la dynamique du simulateur.

Les interfaces correspondant aux quatre principaux contrôles DEVS nécessaires pour implémenter notre artefact de modèle peuvent à présent être implémentés :

1. `executeModel` : une fonction `runMainThread` permet d'envoyer le signal permettant de débloquer la boucle événementielle de NS-3, juste le temps de traiter le prochain événement interne de la pile.
2. `getNextEventTime` : une fonction `peekNext` est accessible en ajoutant un simple *getter* public, et permet de récupérer l'événement le plus imminent de la pile, et donc le temps de simulation auquel il est programmé.
3. `setInputEvent` : des fonctions `setInputPacket` et `setInputVariable` publiques ajoutées en même temps que les ports, permettent d'injecter des paquets IP en fonction du nom du port associé.
4. `getOutputEvent` : des fonctions `getOutputPacket` et `getOutputVariable` publiques ajoutées en même temps que les ports, permettent de récupérer des paquets IP éventuellement généré depuis le traitement du dernier événement interne, en fonction du nom du port associé.

Le fonctionnement des interactions entre les différents threads créés par la bibliothèque qui permet toutes les modifications citées, et qui intègre l'implémentation de l'artefact de modèle, est expliqué dans l'annexe 26.

La section suivante détaille un cas de simulation limite avec NS-3, qui met en échec notre solution.

5.6 Limitation de notre solution

L'une des contraintes imposées par le formalisme de DEVS est qu'un événement externe ne peut pas en provoquer un autre de façon instantanée, vis-à-vis du temps de simulation.

Dans l'exemple du ping qui a été présenté dans la section précédente, le modèle du serveur (disponible dans NS-3 par défaut) renvoie un paquet dès qu'il en reçoit un du client. Le temps nécessaire pour recevoir le premier paquet, forger le second et l'envoyer au client est suffisamment court dans la réalité pour les modélisateurs aient estimés qu'il était inexistant. Du point du vue du temps de simulation, il s'agit donc d'une action instantanée. Pour autant, le paquet de réponse n'arrivera généralement pas immédiatement au client, puisque le modèle de lien entre les deux nœuds consommera probablement du temps de simulation pour modéliser le transport de l'information. Par contre, si le temps de transport modélisé par le lien est inférieur à la granularité du temps de NS-3 (lien parfait), il sera considéré comme instantané vis-à-vis du temps de simulation. Le client recevra donc instantanément un paquet, lorsqu'il en enverra un au serveur.

Pour comprendre pourquoi DEVS interdit ce type de comportement, il suffit de modifier le comportement du modèle du client. Plutôt que d'envoyer des paquets au serveur régulièrement, il en enverra désormais un seul au début de la simulation. Le reste du temps, il aura le même comportement que le modèle de serveur, en renvoyant un paquet chaque fois qu'il en reçoit un du serveur. Si les caractéristiques du modèle de lien permettent aux paquets transportés de consommer du temps, on obtient un échange perpétuel de paquets entre le client et le serveur (une sorte de ping-pong). Si les caractéristiques du modèle de lien font de celui-ci un lien parfait (les paquets sont considérés comme transportés instantanément vis-à-vis du temps de simulation), le ping-pong fonctionne toujours, mais le temps est figé. Tous les événements sont planifiés à la même date, c'est à dire celle correspondant à l'envoi du premier paquet. L'horloge du simulateur est figée, quel que soit la durée de la simulation et le nombre de paquets échangés.

Du point de vue du temps réel, l'observation des échances permet bien de visualiser l'ordre des échanges, avec une parfaite alternance entre les envois et les réponses. Cet ordre n'a pourtant aucun sens vis-à-vis du temps de simulation. Le résultat de la simulation est dépendant du temps réel.

Dans l'exemple du ping via AA4MM de la section précédente (figure 21), le modèle du lien parfait est couplé à celui du client, et le modèle de lien initial est couplé à celui du serveur. Vis-à-vis de notre solution, ce choix est arbitraire, et les deux modèles de lien auraient pu être inversés. Si ça avait été le cas, les événements externes reçus par le port AA4MM du serveur (correspondants aux paquets du client) auraient provoqués des événements externes de façon instantanée (cf. figure 22) et mis en péril l'algorithme DEVS chargé de la gestion de la co-simulation.

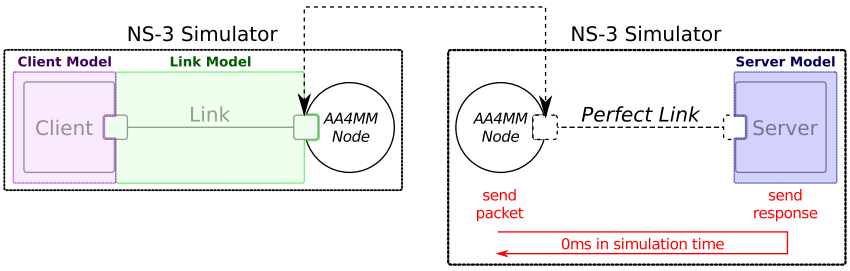

FIGURE 22 – Génération instantanée d'un événement externe (réponse du serveur), avec un modèle de serveur qui ne consomme pas de temps de simulation.

5.7 Généricité de l'intégration

Le travail d'intégration effectué ci-avant a permis de répondre partiellement à la problématique, en reliant deux instances de simulation NS-3 dans une même co-simulation. La connexion entre les deux instances n'ayant pas été faite de façon *ad-hoc*, cette intégration permet de se relier à n'importe quel simulateur d'une co-simulation AA4MM.

Par conséquent, il est par exemple désormais possible de faire co-simuler ensemble des simulateurs de réseaux numériques techniquement incompatibles entre eux. En admettant que le même processus ait été suivi pour intégrer un concurrent de NS-3, comme OMNeT++, il serait possible d'utiliser un modèle de serveur de ping de ce dernier, avec un modèle de client de ping de NS-3. Comme l'illustre la figure 23, de façon générale, le cœur de AA4MM permettra d'utiliser n'importe quel simulateur pour co-simuler avec NS-3, y compris avec un formalisme différent, avec une implémentation dans un langage différent ou s'exécutant sur un système d'exploitation différent.

Les propriétés de décentralisation de AA4MM permettent automatiquement de distribuer, de cette façon, n'importe quelle simulation, en répartissant les logiciels sur un nombre quelconque de machines.

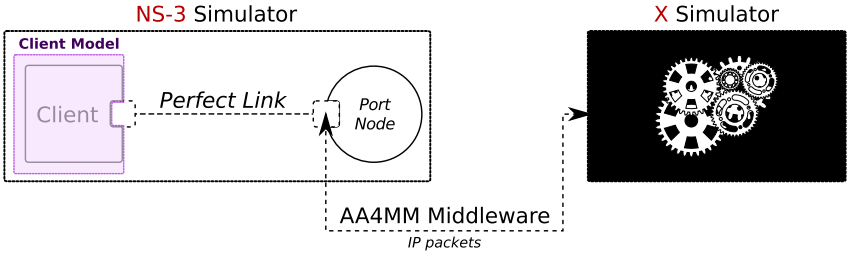

FIGURE 23 – Une simulation NS-3 peut désormais être reliée à n'importe quel simulateur d'une co-simulation AA4MM.

Le dernier obstacle pour l'interopérabilité se concentre sur la représentation des données, faisant l'objet d'événements externes qui sont échangés. Par exemple, connecter NS-3 avec OMNeT++ nécessite qu'ils utilisent la même représentation pour leur paquets IP. Dans ce cas précis, NS-3 a fait le choix de représenter ses paquets dans le format réel, tel qu'on les manipulerait dans le système. Il s'agit donc d'un standard de fait, qui permet d'augmenter significativement les chances pour que d'autres simulateurs de réseaux numériques comme OMNeT++ puissent être compatibles. Si ça n'est pas le cas, le cœur de AA4MM permet d'ajouter des opérations sur les artefacts de couplage, qui peuvent donc jouer le rôle de traducteur, entre les deux mondes. De nouveau, la représentation réelle des paquets NS-3 permet d'utiliser des outils comme la *libpcap*, qui permettent de facilement manipuler ce format, pour ensuite le traduire dans un autre. Ajouter une opération de ce type est une autre contribution au niveau domaine de AA4MM.

Récupérer, mettre en forme et interpréter des résultats est la finalité de toutes simulations. Dans le cadre de notre exemple simpliste de pings, un résultat de simulation pourrait être le temps que mettent les paquets de réponse du serveur à arriver, à partir du moment où le paquet correspondant du client a été envoyé (RTT [2]), en fonction des caractéristiques qu'on impose au modèle du lien qui sépare les deux machines.

Grâce à l'intégration de NS-3 dans la plateforme AA4MM, des agents d'observation peuvent être reliés à la simulation

2. RTT : *round-trip time*

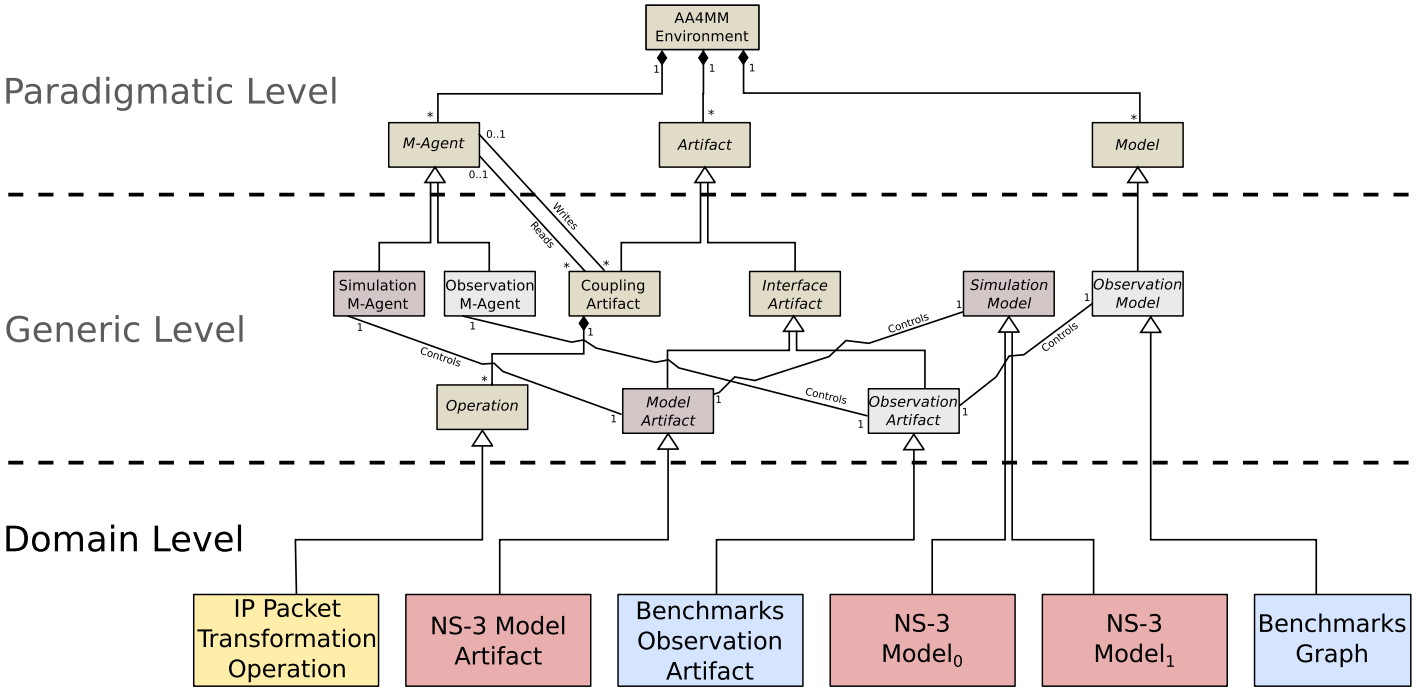

FIGURE 24 – Contributions au niveau domaine de AA4MM.

(figure 25 basée sur Camus et al. [**?**]). Ils permettent alors d'accéder à une collection de différents outils de représentation des données, permettant notamment de tracer en live les temps de réponses des pings.

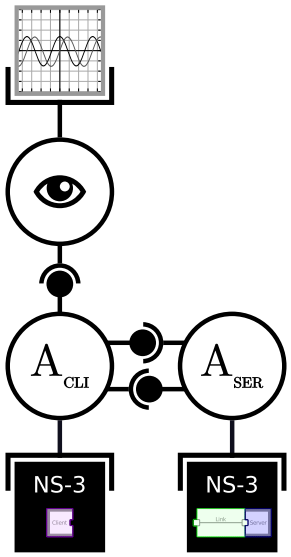

FIGURE 25 – Observation de la simulation.

5.8 Conclusion

Le travail de réflexion ci-dessus nous a permis de lever tous les verrous conceptuels et techniques pour rendre compatible un modèle NS-3 avec une co-simulation AA4MM. L'artefact de modèle AA4MM a pu ensuite être naturellement écrit, simplement en manipulant les interfaces DEVS à disposition. Il se concrétise par une bibliothèque C++, qui intègre également les différents outils nécessaires à la modification du comportement de NS-3 et à l'ajout des ports. Il suffit donc

désormais de compiler un modèle NS-3 en liant la bibliothèque *libms4sg-ns3* à l'édition des liens et de prévoir au préalable l'emplacement des ports dans la topologie réseau modélisée, pour l'intégrer dans une co-simulation.

L'algorithme 2 donne un exemple de la simple ligne de code à ajouter à un modèle NS-3, pour transformer une carte réseau NS-3 en port DEVS.

Algorithm 2 Instantiation d'un wrapper, transformant la première carte réseau du nœud numéro 1 en un port nommé arbitrairement *client*.

```
Ns3NetDeviceWrapper::create(nodes.Get(1)->GetDevice(0), "client");
```

La ligne systématique à ajouter aux modèles pour obtenir une exécution en pas-à-pas contrôlable correspond, quant à elle, à l'algorithme 3.

Algorithm 3 Modification de la boucle événementielle de NS-3 pour forcer une exécution en pas-à-pas.

```
GlobalValue::Bind("SimulatorImplementationType",
StringValue("ns3::Ms4sgNs3DefaultSimulatorImpl"));
```

Les deux premières étapes pour l'intégration de NS-3 peuvent être généralisables à la plupart des simulateurs de réseaux numériques.

6 Conclusion

Nous avons proposé trois étapes pour l'intégration d'un simulateur existant à une plateforme de co-simulation basée sur DEVS. Ces trois étapes ont été confrontées à deux preuves de concept, avec l'intégration du standard FMI et du simulateur de réseaux numériques NS-3. Les deux exemples ont permis de couvrir deux cas d'intégration, avec le standard FMU qui est formalisé et qui a été conçu pour faire de la co-simulation (cas favorable), et NS-3 qui n'est pas formalisé et qui n'est pas prévu pour co-simuler (cas défavorable).

La problématique proposée nous a permis de répondre à un cas concret pour l'intégration de NS-3, en réussissant à distribuer une simulation NS-3 sur deux instances du simulateur, en passant par la plateforme de co-simulation AA4MM, basée sur DEVS. Cette intégration étant générique (sans couplages logiciels *ad hoc*), nous avons vu que NS-3 pouvait désormais s'interconnecter avec d'autres simulateurs de réseaux numériques. De manière plus générale, il peut s'interconnecter avec n'importe quel autre simulateur intégré à la plateforme de co-simulation. En permettant de faire communiquer un simulateur du standard FMI avec NS-3, nous répondons à la problématique plus large de l'interconnexion de simulateurs de systèmes électriques (avec l'exemple de FMI) avec des simulateurs de réseaux numériques (avec l'exemple de NS-3), particulièrement présente dans l'étude de la simulation des systèmes électriques intelligents (*smart grids*).

Les propriétés de la plateforme AA4MM permet d'hériter de ses propriétés de preuves pour l'absence d'interblocages et le respect de la contrainte de la causalité, ainsi que de l'exécution entièrement décentralisée de la co-simulation. Elles nous permettent également d'ajouter des agents d'observation, pour expliciter aisément les résultats de simulation.

A Interactions entre les threads

Références

[1] K. Hopkinson, X. 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*, vol. 21, no. 2, pp. 548–558, 2006.

[2] K. Mets, T. Verschueren, C. Develder, T. Vandoorn, and L. Vandevelde, "Integrated simulation of power and communication networks for smart grid applications," in *2011 IEEE 16th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*, 2011, pp. 61–65.

[3] H. Lin, "Communication infrastructure for the smart grid : A co-simulation based study on techniques to improve the power transmission system functions with efficient data networks," Ph.D. dissertation, Oct. 2012. [Online]. Available : http://scholar.lib.vt.edu/theses/available/etd-10112012-170844/

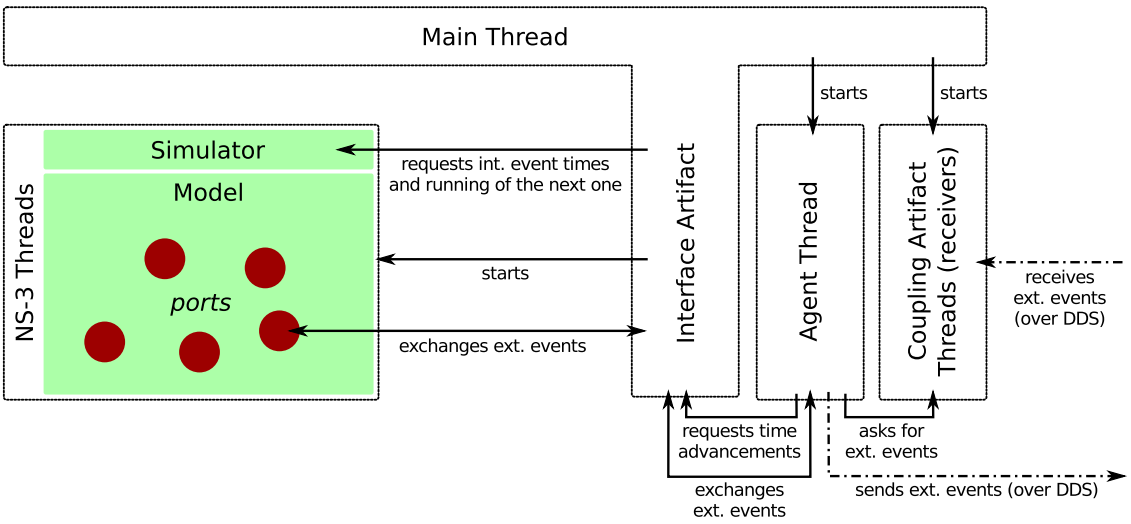

FIGURE 26 – Interactions entre les threads créés par la bibliothèque.

[4] M. Chenine and L. Nordstrom, "Modeling and simulation of wide-area communication for centralized PMU-based applications," *IEEE Transactions on Power Delivery*, vol. 26, no. 3, pp. 1372–1380, Jul. 2011.

[5] E. Galli, G. Cavarretta, and S. Tucci, "HLA-OMNET++ : An HLA compliant network simulator," in *12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications, 2008. DS-RT 2008*, Oct. 2008, pp. 319–321.

[6] B. P. Zeigler, T. G. Kim, and H. Praehofer, *Theory of Modeling and Simulation*, 2nd ed. Orlando, FL, USA : Academic Press, Inc., 2000.

[7] H. Vangheluwe, "DEVS as a common denominator for multi-formalism hybrid systems modelling," in *IEEE International Symposium on Computer-Aided Control System Design, 2000. CACSD 2000*, 2000, pp. 129–134.

[8] G. Quesnel, R. Duboz, and É. Ramat, "The virtual laboratory environment – an operational framework for multi-modelling, simulation and analysis of complex dynamical systems," *Simulation Modelling Practice and Theory*, vol. 17, no. 4, pp. 641–653, Apr. 2009. [Online]. Available : http://www.sciencedirect.com/science/article/pii/S1569190X08002165

[9] J. Siebert, "Approche multi-agent pour la multi-modélisation et le couplage de simulations. application à l'étude des influences entre le fonctionnement des réseaux ambiants et le comportement de leurs utilisateurs." Ph.D. dissertation, Université Henri Poincaré - Nancy I, Sep. 2011. [Online]. Available : http://tel.archives-ouvertes.fr/tel-00642034

[10] A. Ricci, M. Viroli, and A. Omicini, "Give agents their artifacts : The a&a approach for engineering working environments in MAS," in *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems*, ser. AAMAS '07. New York, NY, USA : ACM, 2007, pp. 150 :1–150 :3. [Online]. Available : http://doi.acm.org/10.1145/1329125.1329308

[11] Éric Ramat, *Modélisation et Simulation Multi-agents, Application pour les Sciences de l'Homme et de la Société*. Hermes Sciences, 2006, vol. Chapitre 2.