

Université Paris-XI | Ecole Supérieure d'Electricité | UFR Scientifique d'Orsay

N° d'ordre : 7772

UNIVERSITÉ PARIS XI

UFR SCIENTIFIQUE D'ORSAY

THÈSE

Présentée pour obtenir le grade de

DOCTEUR EN SCIENCES DE L'UNIVERSITÉ PARIS XI ORSAY

Spécialité : Informatique

par

Mokhoo MBOBI

SUJET :

Modélisation Hétérogène Non-Hiérarchique

Soutenue publiquement le 17 décembre 2004 devant la commission d'examen

M.	Olivier TEMAM	:	Président du jury
M.	Paul CASPI	:	Rapporteur
M.	Lionel MARCÉ	:	Rapporteur
M.	Emmanuel LEDINOT	:	Examinateur
M.	Frédéric BOULANGER	:	Examinateur
M.	Guy VIDAL-NAQUET	:	Directeur de thèse

À

Christine, Irène, Gaëlle, Erika, Capucine, Aimé,

Martin, Adèle, Bruno, Achilles, Serge, Hugues,
Patrick, Guyslain,

Joseph, Alphonse, Marie-M, Marie-N,

Betty, Chantal, Gisèle,

Shannon, Loïc, Ryan, Dyllan,
Loyd, Jotham,

Ramond, Félicianne et à
tous les Thimothée.

Une pensée profonde à mon grand-père parti à deux mois de la soutenance de cette thèse.

Remerciements

C'est avec une immense joie que je formule ces remerciements qui témoignent par écrit ma reconnaissance à toutes les personnes qui ont de près ou de loin manifesté leur gratification de soutien et de confiance en ma personne tout au long de ces difficiles années de thèse.

Je tiens tout d'abord à remercier Monsieur Oliver Friedel, ancien Chef du Département Informatique et actuellement Directeur des Etudes de l'Ecole Supérieure d'Electricité de m'avoir accordé l'opportunité de réaliser cette thèse dans d'excellentes conditions. J'exprime également ma reconnaissance à Madame Yolaine Bourda, son successeur de m'avoir permis d'assurer la continuité de ces recherches.

Je tiens à exprimer ma profonde reconnaissance à Monsieur Guy Vidal-Naquet, Professeur à l'université Paris-XI et à l'Ecole Supérieure d'Electricité d'avoir accepté la Direction de ma thèse. C'est grâce à sa direction éclairée et aux nombreuses discussions qu'il m'a accordées que ce travail a pu être réalisé et les objectifs ont été atteints.
Je voudrais aussi exprimer ma gratitude à Monsieur Olivier Temam, Directeur de Recherche à l'INRIA et Professeur à l'Ecole Polytechnique de m'avoir fait l'extrême honneur de présider le jury de ma soutenance.

Je formule ma profonde reconnaissance à Monsieur Paul Caspi, Directeur de Recherche au CNRS et Professeur à l'Université de Grenoble 1 ainsi qu'à Monsieur Lionel Marcé Professeur à l'Université de Bretagne Occidentale d'avoir tous les deux accepté la lourde tache de rapporter sur cette thèse.
Mes remerciements vont aussi à Monsieur Emmanuel Ledinot de Dassault Aviation d'avoir accepté de participer au jury de cette thèse.

C'est avec plaisir que je remercie Monsieur Frédéric Boulanger, Enseignant Chercheur à l'Ecole Supérieure d'Electricité pour la proposition de ce sujet, certes difficile et complexe, mais, ô ! combien passionnant. J'ai découvert en lui un agréable coéquipier de recherche. C'est du fond de mon coeur que j'ai apprécié ses fines suggestions et ses précieux conseils qui ont guidé ce travail.
Je tiens également à remercier Monsieur Dominique Marcadet, Enseignant Chercheur à l'Ecole Supérieure d'Electricité et à l'Université de Versailles Saint-Quentin en Yvelines de m'avoir éclairé de ses compétences sur la méthodologie et l'orientation objets.

J'exprime ma profonde gratitude aux Chercheurs du Département EECS de l'Université de Berkeley en Californie sur la qualité et la précision des réponses qu'ils m'ont données sur certaines questions précises concernant la plate-forme PTOLEMY II.

C'est avec joie que je reconnais la particulière contribution de tous mes collègues Chercheurs du Département Informatique. Qu'ils soient une fois de plus assurés de ma reconnaissance. Je présente mes encouragements à Messieurs A. Maran, M. Feredj, C. Jacquet, C. Jacquiot et à Mademoiselle L. Vescovo et à tous les autres pour la suite et l'accomplissement de leurs tâches.
Je salue la disponibilité de Gilbert Dahan, la gentillesse et l'efficacité de Madame Evelyne Faivre ainsi que le soutien de l'équipe du Centre de Ressource Informatique (CRI).

A mes anciens camarades de promotion : Ingénieurs de l'Enic-Telecom Lille 1 et ceux du Mastère Spécialisé de l'Ecole des Mines de Paris qui ont contribué de manière directe ou indirecte à la réussite de ce travail, mes amitiés leur sont acquises.
Je tiens à associer à ces remerciements, Monsieur Robert Mahl, Directeur du Centre de Recherche en Informatique et Professeur à l'Ecole des Mines de Paris, Monsieur Pierre Jouvelot, Professeur à l'Ecole des Mines de Paris, Monsieur Fabien Coelho, Enseignant Chercheur à l'Ecole des Mines de Paris et tous les autres membres du Centre de Recherche en Informatique de l'Ecole des Mines de Paris.

Une cordiale pensée à mes amis Henri Savina, Jean-Jacques Bokino, Félix Kpazaï, Wilson Omanga et tous les autres qui ont toujours su me témoigner de leur amicale attention, et, une chaleureuse pensée à tous mes anciens étudiants de l'Institut Supérieur des Techniques Appliquées (ISTA) et particulièrement à Jean Masumbuko.

Enfin, à mes enfants que j'adore, je crains que durant ces dernières années, je ne fus pas toujours un père disponible. Que Gaëlle, Erika, Capucine et Aimé me pardonnent infiniment.
Mes hommages à mon épouse Christine, qui a su dans sa patience naturelle assurer ma relève de manière parfaite. J'exprime ma gratitude à Irène et Yvette qui leur ont été des tantes remarquablement attentionnées ainsi qu'à Achilles, Hugues et Patrick qui leur ont été d'un merveilleux et inépuisable soutien. Que ma nièce Peggy s'y retrouve également.
Merci encore à vous tous pour votre soutien familial sans relâche, pour votre détermination et pour vos encouragements qui reposent sur une ferme croyance d'y arriver.
De Kinshasa à Paris, et, de Paris à Toronto en passant par la Guadeloupe, ce titre de « **Docteur en Sciences** » revient à nous tous.

Résumé

La conception des systèmes embarqués est une tâche complexe qui implique plusieurs sous-systèmes appartenant à différents domaines techniques. Chacun de ces domaines obéit à un ensemble de « lois physiques » ou d'« axiomes » appelé « modèle de calcul » (Model of Computation - MoC), qui gouvernent l'interaction de ses composants. Ainsi, un système embarqué est naturellement hétérogène. Actuellement, les outils employés pour la modélisation de ces systèmes utilisent une approche hiérarchique. Cette approche, bien qu'évitant l'explosion combinatoire du nombre d'interfaces entre différents MoCs, impose systématiquement un changement de niveau hiérarchique lorsque l'on passe d'un MoC à un autre. Or ce couplage entre la hiérarchie et les changements de domaine perturbe la structure du modèle, nuit à la réutilisabilité des composants, altère la modularité et réduit la maintenabilité des modèles.

Cette thèse propose une approche non-hiérarchique qui découple les changements des MoCs de la hiérarchie. Cette approche repose sur l'utilisation de « Composant à interface Hétérogène » (Heterogeneous Interface Component - HIC) et d'un Modèle d'Exécution Hétérogène Non-Hiérarchique. Le HIC dispose d'entrées et de sorties de natures différentes pour permettre la communication hétérogène dans le système. Il gère les flots de données (la communication et le comportement) et le contrôle. Quant au Modèle d'Exécution Hétérogène, il restructure le système en le partitionnant, i.e., en créant des sous-systèmes homogènes à la frontière des comportements hétérogènes, puis, ordonnance les activations de ces sous-systèmes en déléguant leur ordonnancement interne à leurs MoCs réguliers et exécute le système. Il gère donc le flot de contrôle entre les domaines.

Cette approche présente plusieurs avantages. En effet, sa capacité d'utiliser des composants disposant d'entrées et de sorties hétérogènes permet d'utiliser plusieurs composants hétérogènes au même niveau hiérarchique. Ensuite, la séparation entre le flot de contrôle et le flot de données augmente la réutilisabilité des composants. Enfin, sa capacité à spécifier explicitement le comportement hétérogène, permet d'indiquer ce qui se produit à la frontière des deux MoCs comme partie intégrante du système. Ceci contribue efficacement à la modularité et à la maintenabilité des modèles, et, donne au concepteur du système le contrôle entier du comportement de son système à la limite des différents MoCs.

Mots cles : Systèmes embarqués, ingénierie logicielle, modélisation hétérogène, conception, modèle de calcul, hétérogénéité, hiérarchie, composants, acteurs

Abstract

The design of embedded systems is a complex task that makes use of numerous subsystems which belong to different technical domains. Each of those domains obey a set of "physical laws" or "axioms" called " Model of Computation, (MoC)" that govern the interaction of its components. So, embedded system is naturally heterogeneous. Currently, modeling tools used for modeling embedded systems use a hierarchical approach. This approach although avoiding the combinatorial explosion of the number of interfaces between MoCs, forces the change of hierarchical level when passing from one MoC to another. But, this coupling between hierarchy and model changing perturbs the structure of the model. This coupling is harmful to the reuse of the components, affects the modularity and makes difficult the maintainability of the model.
This thesis proposes a new approach that dissociates the MoC from this hierarchy. This approach uses two components : a " Heterogeneous Interface Components (HIC) " and a " Non-Hierarchical Heterogeneous Execution Model ". A HIC have inputs and outputs of different nature to allow the heterogeneous communication in a system. So, it manages data flow (communication and behavior) and control flow. As for the Execution Model, it reorganizes the system by partitionning, i.e. by creating homogenous subsystems at the border of the heterogeneous behavior. It schedules the activation of those subsystems and delegates their internal scheduling to their regular MoC. Finally it executes the system. So, it manages the control flow between the domains. This approach presents several advantages : the use of the components that use heterogeneous inputs or outputs allows the use of several heterogeneous components at the same level of the hierarchy. Then, the separation of control flow and data flow increases the reuse of the components. And, the explicit specification of the heterogeneous behavior, allows the spécification of what happens at the boundary between different MoCs, then contributes efficiently to the modularity and the maintainability of the models. moreover, this gives to the designer of the system, the whole control of what happens at the boundary between different MoCs.

Key words : Embedded Systems, software engineering, heterogeneous modeling, design, model of computation, heterogeneity, hierarchy, components, actors

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Une récente étude du groupe d'experts dédié aux systèmes embarqués et temps réel [117] a mis en évidence l'explosion et la segmentation du marché des systèmes embarqués. Aujourd'hui, ce marché se structure sur une dichotomie marché « systèmes personnels » et marché « systèmes collectifs ». Le premier marché s'intéresse au type de systèmes comme l'agenda électronique évolué et connecté appelé aussi PDA, les appareillages domestiques télécommandés et le second s'intéresse au type de systèmes comme l'avion, le sous-marin, le satellite, le système de guidage de missiles, la centrale nucléaire.

En effet, actuellement, le marché de l'embarqué qui, jadis était essentiellement focalisé sur les applications industrielles, spatiales, militaires ou aéronautiques pénètre rapidement tous les secteurs économiques et le grand public, entraînant de nouvelles applications d'un ordre de grandeur plus complexes de celles d'il y a quelques années. Parmi ces applications, on peut citer le téléphone portable et le multimédia, le contrôle moteur sophistiqué en automobile, l'assistance à la conduite, le dictionnaire électronique, la robotique, les cartes à puce, la domotique, les pacemakers et prothèses automatisées etc. . .

Les vecteurs de cette expansion sont nombreux. Il en existe néanmoins trois qui en sont de véritables fers de lance à savoir :
Primo, la digitalisation, car, de nombreux appareils qui étaient dans le passé analogiques deviennent numériques, en conséquence, deviennent programmables, ce qui nécessite un savoir-faire et des outils spécifiques.
Secundo, la conjonction entre l'accroissement des performances et la baisse des prix des processeurs dictés par la loi de Moore [102]. Cette conjonction a rendu possibles de nouvelles utilisations. Ce qui explique par exemple qu'actuellement, dans une voiture ordinaire, un microprocesseur soit remplacé par plusieurs dizaines de processeurs sans modification substantielle du prix de la voiture.
Tertio, les appareils d'utilités diverses qui, autrefois étaient autonomes, sont aujourd'hui de plus en plus mis en réseau. De ce fait, de part la loi de Metcalfe[1], ils sont chacun devenus

[1]Loi de Metcalfe : l'utilité d'un réseau est proportionnelle au carré du nombre de ses utilisateurs.

un noeud d'un réseau. D'où leur mutuelle coopération, laquelle, a considérablement augmenté l'utilité des réseaux en engendrant une synergie qui a ouvert de nouvelles opportunités sur ces appareils. Un réfrigérateur, un lave vaisselle et un four à micro-ondes connectés à Internet en sont d'excellentes illustrations.

Ces raisons sont durables. En effet, sur le plan scientifique, les chiffres des dernières études [9] [117] viennent conforter ces tendances en révélant que plus de 95% des processeurs fabriqués ces dernières années sont destinés aux systèmes embarqués, et ce, avec une croissance annuelle de 30% dont la prévision de répartition par secteur pour 2004 est de 44% dans la communication, 28% dans l'électronique grand public et 28% du reste.
Sur le plan humain, ces études prédisent qu'un citoyen des pays développés utilisera environ 80 processeurs quotidiennement par le biais des systèmes embarqués qu'il utilisera dans sa vie courante.

Il s'ensuit qu'en effet, de nouveaux types de systèmes embarqués vont apparaître, des systèmes existants vont se modifier, les services autour de ces systèmes vont eux-mêmes se développer et le nombre d'objets familiers contenant un processeur embarqué va augmenter de manière continue dans les années à venir.

Cependant, pour répondre à ce marché sans cesse croissant et avec des temps de mise sur le marché « time to market » de plus en plus courts, les équipementiers et les chercheurs sont dans l'impératif de composer avec deux familles de contraintes. Le premier groupe de contraintes est d'ordre économique et intéresse les équipementiers et le deuxième groupe de contraintes est plutôt d'ordre technique et intéresse les chercheurs.

Pour les contraintes d'ordre économique, en considérant la loi de Moore, des choix économiques doivent être judicieusement faits afin de maintenir l'équilibre entre le prix moyen des composants embarqués qui est de plus en plus faible et leurs performances qui grimpent exponentiellement. Par exemple, aujourd'hui, un téléphone portable qui soit capable de modifier sa sonnerie par téléchargement d'une séquence musicale polyphonique, de consulter des programmes de train, de filmer des séquences vidéos et de les envoyer par messagerie électronique, etc... coûte moins d'un dixième du prix de son ancêtre de la première génération GSM dont le fonctionnement se résumait presque à la simple communication téléphonique. Donc, le grand challenge économique pour les équipementiers restera la maîtrise de l'ambivalence « augmentation de performances-baisse des prix ».

En effet, en regard de l'évolution du marché de l'embarqué et desdites contraintes économiques, le cycle de « conception - mise au point - production » d'un appareil doit se raccourcir. Cela impose une mutation profonde vers des modes de production répartis et industriels, favorisant des applications de plus en plus modulaires à base de composants réutilisables et facilement maintenables.
Ainsi, dans [122], l'auteur affirme que le développement des systèmes embarqués suit un processus partagé par plusieurs acteurs mettant en œuvre une coopération entre

équipementiers et constructeurs et que la notion de réutilisation de composants est un argument fort pour réduire les coûts d'étude et de fabrication. Apportant un argument économique, dans [63], l'auteur confirme que l'usage de la méthodologie modulaire entraîne un investissement qui n'est amorti que sur le long terme, après réutilisation.

Pour les contraintes d'ordre technique, elles sont intrinsèques au système embarqué et sont prises en compte dans ses différentes phases de conception. Elles se situent au niveau fonctionnel et opérationnel et sont intimement liées à l'essence même du système embarqué. Leur prise en compte permet au système embarqué de garantir ses caractéristiques de fonctionnement en réactivité continue avec son environnement immédiat de manière sûre et sécurisée. Tel est par exemple, le cas d'un stimulateur cardiaque dont les électrodes sont implantés à l'intérieur d'un coeur humain avec lequel il interagit pour réguler ses battements en réactivité continue et de manière sûre et sécurisée.
La conception de tels systèmes requiert une pluridisciplinarité de connaissances scientifiques, d'où le besoin de mettre en oeuvre plusieurs spécialistes des différents domaines. De plus, le fait que le cycle de vie des systèmes collectifs soit très long par rapport aux composants utilisés pour les fabriquer amène le besoin de limiter l'impact des dégradations et des obsolescences dans ces systèmes. En somme, ce double constat implique la prise en compte des solutions qui passent également par des techniques architecturales de modularité.

Ainsi, actuellement, les systèmes sont conçus de manière modulaire, ce sont des composants élémentaires ou composites rassemblées selon un schéma de communication bien défini. Dans [23], l'auteur soutient ce concept en précisant qu'actuellement, le caractère multi-partenaires des projets industriels requiert des capacités de développement modulaire, en particulier d'être capable de compiler séparément et indépendamment chaque composant de l'application sous forme de processus exécutables, logiciels ou matériels, puis de les intégrer au sein de l'architecture finale. Dans [12], l'auteur précise que la réutilisation est intrinsèquement un phénomène intervenant lors de la phase de construction. Elle implique la modification des structures de données et d'algorithmes présents dans les composants, et qui demeurent ensuite fixes durant l'exécution.

Relayant les idées ci-dessus, dans son développement présenté dans [55], l'auteur montre que les modèles ainsi que leurs briques de base doivent être réutilisables et les plus autonomes possible. Ensuite, il précise qu'une approche doit non seulement permettre d'exprimer les propriétés essentielles du système, mais aussi garantir une maintenabilité relativement facile des modèles. De plus, l'évolution des modèles doit suivre celle des systèmes embarqués sans remettre en question la totalité de ce qui existe déjà.

En conséquence, une approche de conception doit fournir des mécanismes de réutilisabilité efficaces qui permettent l'ajout, le retrait, la spécification des éléments du modèle sans remettre en cause toute la description du système.

Fort de ce qui précède, que les enjeux soient économiques ou scientifiques, nous pouvons présumer que la modularité dans les systèmes embarqués favorise donc la réutilisabilité et améliore la maintenabilité. Et, l'augmentation de la réutilisabilité des composants implique l'augmentation de la productivité des concepteurs de systèmes.

C'est pourquoi, la modularité et la réutilisabilité sont actuellement perçues comme des arguments forts dans la conception des composants. Ainsi, un système embarqué modulaire comportera l'interconnexion de plusieurs modules interagissant avec ou supervisés par d'autres modules. On voit d'ailleurs apparaître des systèmes embarqués qui contiennent d'autres systèmes embarqués. Ces différents modules du système peuvent cependant appartenir à des différents domaines techniques tels que l'électronique analogique, l'électronique numérique, la mécanique, la thermodynamique, l'optique, etc...
Cependant, chacun de ces modules possède ses outils de conception qui obéissent à des ensembles de *« lois physiques »* ou d'*« axiomes »* qui gouvernent l'interaction de ses composants, et, on les appelle *« modèle de calcul, (MoC), Model of Computation »*. Les Processus Séquentiels Communicants, les machines d'état, les flots de données synchrones, les événements discrets, les réseaux de processus de Kahn, etc... sont des exemples des modèles du calcul.

Aujourd'hui, il devient presque naturel que dans la conception des systèmes embarqués, un seul domaine technique implique plusieurs modèles de calcul. Dans un système de communication par exemple, le protocole peut être décrit avec des machines d'états et des événements, et la partie du traitement du signal peut être décrite à l'aide de la transformée en Z ou de la transformée de Laplace. Ces systèmes sont dits *« hétérogènes »* car ils mélangent plusieurs modèles de calcul.
Cette diversité de domaines d'application et leur complexité croissante accentuent donc le besoin de disposer de méthodes et d'outils de modélisation et de conception offrant de meilleurs mécanismes de structuration, d'abstraction et de réutilisabilité [55].

Concernant les outils actuels de modélisation hétérogènes, ils permettent en effet de prendre en compte et de structurer cette diversité de domaines d'application matérialisée par la coexistence de différents modèles de calcul, de la conception à la génération de matériel ou de logiciel.
Cependant, ils se concentrent généralement sur un petit ensemble de modèles de calcul interagissant qui sont souvent des signaux continus et discrets pour l'ingénierie électrique et les machines d'états et des équations différentielles pour les systèmes hybrides.
Toutefois, puisqu'ils n'utilisent qu'un jeu de domaines très peu limité, de tels environnements de modélisation peuvent facilement définir l'union des domaines qu'ils supportent, et, ainsi permettre aux composants d'obéir simultanément à plusieurs modèles de calcul. Tel est le cas d'un convertisseur de signal numérique-analogique qui peut être modélisé comme un composant simple avec les entrées qui obéissent à un modèle de calcul de signal continu et de sorties qui obéissent à un modèle de calcul de signal discret.

Quant aux environnements de modélisation qui supportent un large ou un nombre extensible des domaines, ils ne peuvent pas construire cette union des domaines. Ils exigent que chaque composant obéisse seulement à un modèle de calcul. Ainsi, puisque les composants interconnectés obéissent au même modèle de calcul, les changements de domaine ne peuvent seulement se produirent qu'à la frontière d'un composant. Or, le modèle de calcul utilisé à l'intérieur du composant peut ne pas être identique à celui utilisé en dehors de celui-ci. Ceci mène à la « modélisation hétérogène hiérarchique » utilisée par la plupart d'environnements de modélisation hétérogène tels que SpecC [124], SystemC [129] [41] [128], VHDL-AMS [34], ROSETTA [116], POLIS [113], el Greco [29], OMOLA [95], DYMOLA [43], MODELICA [44], PTOLEMY II [13] [14] [15] [42] [65], [82], etc. . . .

Nous pensons que cette hiérarchie obligatoire dans un modèle hétérogène, qui d'ailleurs n'est donc qu'une conséquence des limites des approches techniques utilisées par les outils actuels de modélisation ne devrait pas dépendre des outils de modélisation. Elle devrait plutôt représenter la structure compositionnelle d'un système suivant sa décomposabilité fonctionnelle par exemple.

En somme, quand bien même cette hiérarchie soit une manière efficace de contrôler la complexité en cachant les détail internes qui ne sont pas convenables à un niveau de modélisation, elle présente un certain nombre de désavantages qui président à la problématique qui motive cette thèse.

Ainsi, puisqu'en électronique embarquée, dans la plupart des systèmes et particulièrement dans les logiciels embarqués, un modèle qui commence comme une simulation évolue vers une mise en oeuvre logicielle du système [13], nous présumons de ce fait que, par essence, le matériel ou le logiciel issu d'une modélisation hérite directement de la problématique de ladite modélisation.
Ce postulat nous induit donc à considérer deux aspects dans la problématique de l'hétérogénéité hiérarchique à savoir : l'aspect lié à la modélisation et l'aspect lié à la génération matérielle ou logicielle qui en découle.

C'est pourquoi, nous avons décorrélé la problématique engendrée par cette approche hiérarchique suivant trois considérations, dont une problématique globale et deux problématiques dérivées.
La première problématique est située au niveau des outils de modélisation. La deuxième problématique est une des conséquences de la première, et, est au niveau des composants matériels ou logiciels générés. La troisième problématique est également une des conséquences de la première, et, est au niveau du concepteur de système.

– du point de vue des outils de modélisation, l'utilisation de l'approche hiérarchique pour la représentation d'un système hétérogène impose un changement de niveau hiérarchique à chaque changement de modèle de calcul. Elle génère parfois des constructions obscures dues aux imbrications induites par la coexistence des différents modèles de calcul.

En conséquence, la représentation d'un simple système peut donc perdre de sa clarté et de sa lisibilité à cause de cette explosion d'étages hiérarchiques. De plus, cette approche casse la modularité du système en obligeant à introduire un nouveau niveau hiérarchique dans le système à chaque changement de modèle de calcul. Ce qui peut limiter la réutilisabilité des composants et réduire la maintenabilité du système.

– du point de vue des composants matériels et logiciels générés, les composants fonctionnant à la frontière de plusieurs modèles de calcul, c'est-à-dire, possédant des entrées et des sorties obéissant à des sémantiques différentes sont interdits. En d'autres termes, les composants sont contraints de n'avoir que des interfaces qui n'apparaissent qu'à un seul niveau hiérarchique. Ainsi, l'ajout d'un composant utilisant un modèle de calcul différent de ceux existant remet en cause la structure du système.

– pour le concepteur du système, il ne peut pas explicitement intervenir sur la transformation des données qui se produisent à la frontière de deux domaines car elles sont cachées à l'intérieur des « composants passerelles ». Le concepteur peut simplement adapter ou interpréter les données dans le modèle de calcul d'arrivée. Il s'ensuit que le comportement du système à la frontière des différents modèles de calcul ne peut nullement être explicitement spécifié. Ce manque de flexibilité peut s'avérer pénalisant dans la mesure où l'adaptation ou l'interprétation des données peut parfois induire à des imprécisions qui risquent de mener le système dans un comportement inattendu. En conséquence, le concepteur du système n'a pas la nette compréhensibilité aux « coutures » des différents modèles.

Nous allons illustrer cette problématique par un exemple simple d'un détecteur d'une station de contrôle montré sur la figure 1.1 qui utilise trois différents modèles de calcul.

Exemple 1.0.1 Considérons le dispositif sur la figure 1.1 représentant un détecteur d'une station contrôlant un parc d'équipements mobiles opérant sur un territoire géographique donné. Chaque station mobile émet en permanence un signal radio en direction de la station de contrôle, qui, à son tour compare son amplitude à un seuil prédéfini. Et, chaque station mobile est équipée d'un terminal GPS, qui permet par voie satellitaire, la transmission de ses coordonnées géographiques à un dispositif informatique, lequel à son tour calcule son éloignement et transmet cette valeur numérique à la station de contrôle.
La station de contrôle considère qu'une station mobile est corrompue lorsque simultanément, son signal a atteint un seuil prédéfini en même temps que ses coordonnées géographiques sont en dehors du rayon du territoire sous son contrôle. Auquel cas, la station de contrôle génère un événement qui consiste par exemple à lui envoyer un signal brouilleur et à le déconnecter de la base de données des équipements sous son contrôle.

Ce composant représenté sur la figure 1.1 montre la problématique de cette étude. Sur sa première entrée, il reçoit un signal radio provenant de l'équipement mobile, il doit donc obéir à la sémantique de temps continu (CT) par exemple. Sur sa deuxième entrée, il reçoit sous forme d'un flot de données, les coordonnées géographiques de l'équipement

mobile, il doit donc obéir à la sémantique de flot de données (SDF) par exemple.

Lorsqu'une station mobile est corrompue, la station de contrôle doit générer deux événements : émettre un signal brouilleur et déconnecter l'équipement corrompu de la base de données. Pour cela, il doit obéir à la sémantique d'événement discret, (DE) par exemple. La présence de cet événement lui permet d'entrer d'abord dans la sémantique de temps continu (CT) par exemple, pour le brouillage ensuite dans la sémantique de flot de données, (SDF) par exemple pour le retrait dudit équipement de la base de données.
En somme, ce seul composant doit obéir à trois modèles de calcul différents.

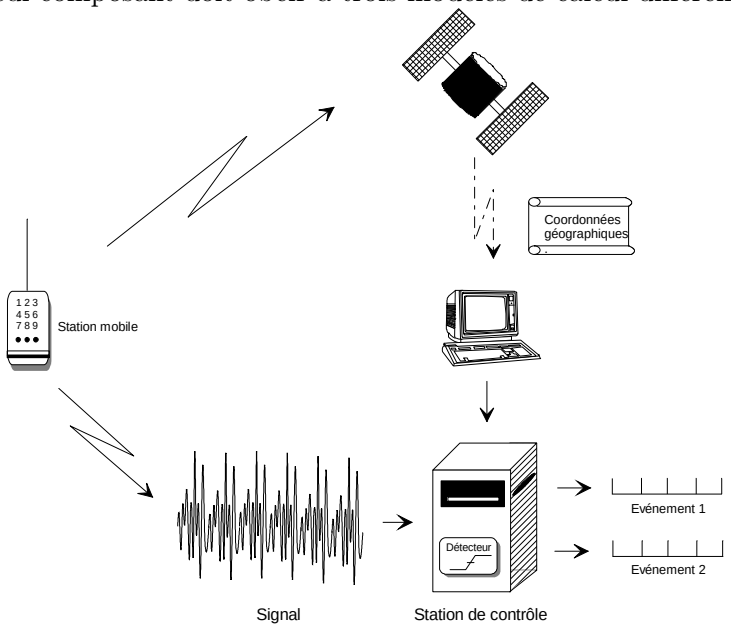


FIG. 1.1 – Exemple d'un dispositif obéissant à plusieurs sémantiques

Du point de vue de sa modélisation, avec des outils actuels de modélisation, il y a apparition artificielle des niveaux hiérarchiques pour gérer les différents changements des modèles de calcul. La figure 1.2 en donne une des représentations.

Du point de vue de la génération matérielle, actuellement, il est difficile de générer un pareil dispositif si l'on souhaite qu'il dispose d'une interface hétérogène pouvant prendre en compte à la fois la sémantique de l'événement discret, celle du temps continu et également celle du flot de données par exemple.

Du point de vue de son concepteur, le passage des données entre la génération de l'événement et le brouillage du signal par exemple ne sera pas explicitement défini par lui. Pour passer de l'événement discret vers du temps continu, certains outils de modélisation et de simulation utilisent des modules prédéfinis pour ces types de conversions en utilisant des bloqueurs d'ordre zéro et l'inverse en utilisant la technique de l'échantillonnage

prédictible ou non-prédictible selon que l'on échantillonne à une fréquence déterminée ou lorsque l'échantillonnage est lié à un événement [106]. Certains d'autres utilisent d'autres techniques pour cette conversion.

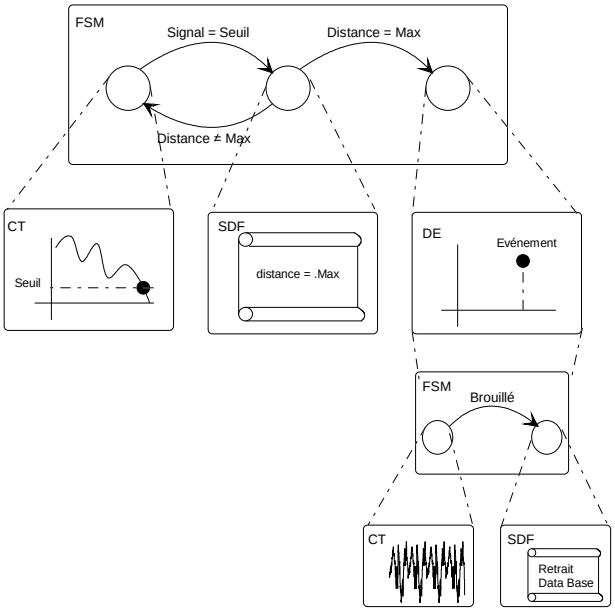


FIG. 1.2 – Niveaux hiérarchiques générés par les passages inter-MoC

Ainsi, le comportement de la station de contrôle à la frontière de ces deux modèles de calcul ne sera pas explicitement déterminé par le concepteur. Il sera plutôt déterminé par ce type de modules génériques prédéfinis. Le concepteur viendra en aval pour l'interprétation et l'adaptation de ces données dans le domaine de destination.

1.1 Objectifs

Il nous paraît fondamental de souligner que le but poursuivi par cette étude n'est, en effet, nullement de bannir l'approche hiérarchique.

Cependant, cette thèse s'inscrivant dans le cadre de la modélisation hétérogène d'une part, et, corollairement à la problématique présentée dans la section précédente d'autre part, elle entend proposer une approche modulaire de modélisation, utilisant des composants réutilisables et offrant une facilité de maintenabilité.
C'est pourquoi, nous proposons une alternative préconisant une approche non-hiérarchique.

Cette approche doit permettre :

– d'obtenir un modèle hétérogène « plat » qui permettra de changer de modèle sans changer de niveau hiérarchique et sans altération du comportement du système original.

Ceci donnera la possibilité de faire interagir deux ou plusieurs composants hétérogènes au même niveau hiérarchique,

– de modéliser, de concevoir et de générer des composants susceptibles de travailler à la frontière de plusieurs modèles de calcul, i.e., ayant des entrées et des sorties obéissant simultanément à différentes sémantiques,

– de donner au concepteur du système, la flexibilité lui permettant de concevoir explicitement le comportement du système à la frontière de plusieurs modèles de calcul comme partie intégrante de son modèle. Ainsi, donner au concepteur du système le contrôle entier du comportement de son système à la limite des différents modèles de calcul.

1.2 Contributions

Le domaine d'application des technologies sur lesquelles porte cette thèse est très vaste puisqu'il concerne la modélisation, la conception, la simulation et la validation de systèmes logiciels et matériels, notamment des systèmes dits « embarqués ».
Les concepts développés dans cette thèse ont été validés par simulation sur la plate-forme PTOLEMY II, développée par le « Department of Electrical Engineering and Computer Sciences » de l'université de Berkeley en Californie. Et, les résultats de simulation ont permis de se convaincre que l'approche est opérationnelle.

Ce travail introduit une nouvelle orientation de modélisation et de conception qui ne renie pas l'approche hiérarchique mais, pensons-nous, plutôt, qui lui est complémentaire surtout lorsque les imbrications des modèles ne sont ni utiles ni faciles ou lorsque le concepteur du système souhaite spécifier explicitement le comportement hétérogène d'un système à la frontière de plusieurs modèles de calcul.

La principale contribution de cette thèse est le développement d'une approche hétérogène non-hiérarchique qui prône la modularité en se basant sur l'utilisation de composant à interface hétérogène, et d'un modèle d'exécution hétérogène non-hiérarchique. Ces deux composants disposent chacun de possibilité de réutilisabilité.

En effet, puisque le composant à interface hétérogène peut être instancié à souhait, le code concernant son comportement hétérogène peut en conséquence être modifié par le concepteur conformément aux modèles de calcul qu'il utilise et également conformément à une spécification prédéfinie pour obtenir un comportement souhaité. Ce qui lui donne son caractère de composant flexible et réutilisable.

Quant au modèle d'exécution hétérogène non-hiérarchique, sa non-dépendance des modèles de calcul utilisés par les sous-systèmes, lui donne une possibilité d'évolutivité indépendante desdits modèles de calcul. Ainsi, un retrait ou une spécification d'un nouveau modèle de calcul, c'est-à-dire, une nouvelle extension du système à un nouveau domaine

supplémentaire ne remet nullement en cause sa description. Ce qui garanti sa réutilisabilité.

Au niveau du système, cette modularité est également valable dans la génération des sous-systèmes. Ces sous-systèmes qui regroupent des composants utilisant un même modèle de calcul sont dirigés par un modèle d'exécution régulier. A ce niveau, ces sous-systèmes sont perçus comme des blocs homogènes et modulaires ayant une structure interne verrouillée et communicant à travers une abstraction non-hiérarchique au même niveau hiérarchique.
Cette manière de spécifier le système apporte une facilité considérable dans la maintenance aussi bien des composants que dans la maintenance du système entier.

Ainsi, cette approche enlève donc la nécessité d'introduire artificiellement des niveaux hiérarchiques pour autoriser des changements de modèles de calcul. Elle introduit en conséquence, une nouvelle manière de générer des composants matériels et logiciels ayant des interfaces hétérogènes.

Sur le plan pratique, étant donné que les simulations sont faites dans PTOLEMY II, cette étude a aboutit à l'élaboration d'un nouveau domaine[2] « Hétérogène » que nous avons baptisé « Flat Heterogeneous Domain ».
Ce domaine introduit la prise en charge de l'hétérogénéité non-hiérarchique pouvant supporter les différents domaines au même niveau hiérarchique. Dans PTOLEMY II, il se présente sous forme d'une bibliothèque incluant le HIC et le modèle d'exécution hétérogène non-hiérarchique.

1.3 Organisation de la thèse

Cette thèse est organisée en quatre parties suivantes : Outils de modélisation et approche théorique de la non-hiérarchie, Composants d'appui à l'hétérogénéité non-hiérarchique, Intégration, Validation et Simulation dans PTOLEMY II et les Annexes.

La première partie contient les chapitres deux, trois et quatre.

– Dans le chapitre deux, nous présentons un état de l'art de la modélisation des systèmes hétérogènes en général et particulièrement celle des systèmes embarqués. Nous abordons également la notion de modèle de calcul, celle de modèle hétérogène et celle de hiérarchie. Nous présentons les origines et les différents types de hiérarchie que l'on peut rencontrer dans un système hétérogène. Nous y présentons et développons également la problématique liée à cette étude, celle des modèles hiérarchiques.

– Dans le troisième chapitre, nous commençons par une présentation de l'environnement de modélisation hétérogène en nous focalisant sur les différents langages et plates-formes

[2]Ici, domaine est pris dans le sens de PTOLEMY II

de modélisation hétérogène. De cette présentation, nous justifions le choix de Ptolemy II en nous basant sur son caractère unifié.
Ensuite, nous présentons les méthodologies de modélisation orientées composants ainsi que l'ensemble de primitives abstraites que nous utilisons dans cette étude. Nous justifions également le choix de la méthodologie orientée acteur que nous avons adoptée pour la réalisation de cette thèse.

– Dans le quatrième chapitre, nous présentons l'approche théorique de la modélisation non-hiérarchique en utilisant la méthodologie orientée acteur. C'est dans ce chapitre que nous introduisons la présentation des deux composants d'appui à la modélisation et à la conception : le Composant à Interface Hétérogène (Heterogeneous Interface Component, HIC) qui sera la pièce maîtresse dans la mise à plat de la topologie et dans la communication dans un système hétérogène et le Modèle d'Exécution Hétérogène Non-Hiérarchique dont la tâche sera de restructurer le système en sous-systèmes, d'activer des HICs, d'ordonnancer et d'exécuter les sous-systèmes.

Puis, la deuxième partie vient couvrir les chapitres cinq et six en détaillant les deux composants pivots de la modélisation hétérogène non-hiérarchique à savoir : le Modèle d'exécution hétérogène à qui revient la gestion de flot de contrôle du système et le Composant à Interface Hétérogène, HIC qui assume la tâche de gestion de flot de données

– Le chapitre cinq présente en deux étapes la modélisation du Composant à Interface Hétérogène du point de vue de sa composition structurelle et opérationnelle. Le HIC original est d'abord présenté. Ensuite, nous intégrons le concept de son double partitionnement structurel et opérationnel en répartissant les différentes variables et opérations dans les différentes entités issues de ce partitionnement. Enfin nous établissons les règles d'activation entre lesdites entités.

– Au chapitre six nous présentons la modélisation du Modèle d'exécution Hétérogène Non-Hiérarchique. Cette modélisation est présentée à travers les trois phases de son déroulement : le partitionnement, l'ordonnancement et l'exécution.

Ensuite, vient la troisième partie contenant les chapitres sept, huit et la conclusion de la thèse. Dans cette partie nous avons utilisé la plate-forme PTOLEMY II pour la validation de nos concepts présentés par simulation.

– Le chapitre sept offre une autre manière de présenter les composants d'appui à l'hétérogénéité non-hiérarchique modélisés dans les deux chapitres précédents. Dans ce chapitre, nous nous sommes servis de UML comme formalisme intermédiaire de modélisation et avons donc traduit cette modélisation de composants d'appui en représentation UML. Ce chapitre transitoire est donc un préalable au chapitre suivant qui sera consacré à l'implémentation.

– Au chapitre huit, nous intégrons notre concept de l'Hétérogénéité Non-Hiérarchique dans la plate-forme PTOLEMY II. Nous commençons par un aperçu de cette plate-forme en présentant les classes fondamentales. Dans cet aperçu, nous mettons également en évidence le mécanisme partagé de communication entre la classe Director et la classe Receiver et nous avons détaillé les différentes phases d'un modèle PTOLEMY II.
Ensuite, nous opérons à l'intégration de notre concept non-hiérarchique dans PTOLEMY II. Cette intégration est faite sur deux niveaux. Le premier niveau consiste à l'intégration des classes représentant le HIC et le modèle d'exécution hétérogène non-hiérarchique à la non-hiérarchie dans l'architecture de PTOLEMY II. Le deuxième niveau consiste à l'assignation des différentes phases de la non-hiérarchie dans les différentes méthodes du directeur hétérogène non-hiérarchique. Deux exemples viennent valider nos concepts.

– Enfin, nos conclusions et perspectives relatives aux travaux effectués dans le cadre de cette thèse viennent clôturer ce document. Nous y présentons nos conclusions qui découlent de ce travail de recherche et y présentons également les perspectives à moyen et à long termes relatives à cet axe de recherche.

La dernière partie contient les annexes relatives à ce travail.

Première partie

Outils de Modélisation et Approche Théorique de la Non-Hiérarchie

Chapitre 2

Modélisation des Systèmes Hétérogènes : cas des Systèmes Embarqués

Résumé -Dans ce chapitre nous présentons un état de l'art de la modélisation des systèmes hétérogènes en général et particulièrement celle des systèmes embarqués. Nous abordons également la notion de modèle de calcul, celle de modèle hétérogène et celle de hiérarchie. Nous présentons les origines et les différents types de hiérarchie que l'on peut rencontrer dans un système hétérogène. Nous y présentons et développons également la problématique liée à cette étude, celle des modèles hiérarchiques.

2.1 Introduction

Un système hétérogène est défini comme un ensemble de composants interagissant entre eux et dont la sémantique et la discipline d'interaction sont régies et imposées par un *« modèle d'exécution hétérogène »*.
D'un point de vue général, la modélisation de ce type de système implique sa spécification en différents modules hétérogènes, la spécification de la communication entre lesdits modules et la validation du modèle représentant le système.

La modélisation a pour but de réaliser le partitionnement et la représentation formelle du système à haut niveau en plusieurs sous-systèmes [13]. Généralement, ce découpage est réalisé selon la fonctionnalité de chacune de différentes parties du système. De cette décomposition modulaire, le système est vu comme une agrégation de composants interagissant entre eux et dont chacun peut aussi être décomposé aux niveaux plus bas, en

plus petits composants avec des modèles d'interaction probablement différents. Chacun de ces sous-systèmes sera modélisé par une équipe spécialisée dans la modélisation du type d'applications relatif à ce dernier.

Ainsi, pour atteindre ce but, le système entier doit donc être décomposé en petits morceaux, et plusieurs concepteurs généralement de spécialités différentes doivent travailler ensemble sur base de leur compréhensibilité modulaire.

En effet, il est actuellement difficile à un concepteur de contrôler un cycle complet de conception d'un système hétérogène à cause des fonctionnalités qui sont de plus en plus compliquées et de plus en plus complexes.
De plus, le fait que ces systèmes soient conçus de manière modulaire pour une exécution de plus en plus répartie orientent l'ensemble des communications et interactions de leurs modules vers une gestion via des infrastructures réseaux. Ceci implique des schémas de communication de plus en plus compliqués.

Cependant, la manière de décomposer un système et comment recomposer les différents composants pour réaliser les fonctionnalités définies dans le cahier des charges font partie de grands défis actuels pour des concepteurs de système.
Toutefois, une mauvaise compréhension du niveau système entre les experts du domaine et les équipes de développement risque d'augmenter considérablement le coût d'intégration de système à la fin d'un projet.

Pour la validation, selon que la simulation est faite soit en un seul langage ou soit est faite sur une plate-forme unifiée ou soit encore a nécessité un langage différent pour chaque module, elle se fera soit à travers le simulateur de ce langage ou soit à travers le simulateur de cette plate-forme unifiée ou encore soit à travers la cosimulation qui consiste à exécuter parallèlement et concurremment différents simulateurs à travers un bus de cosimulation. Dans le premier cas, l'optimisation est globale et dans le second cas, elle est faite par sous-système car elle est directement dépendante du langage utilisé pour chacun de ces sous-systèmes.

Ce concept de modélisation que nous avons introduite est presque général à touts les systèmes hétérogènes.
Cependant, puisque cette étude adresse particulièrement les systèmes embarqués qui d'ailleurs sont toujours de nature hétérogène, dans la section suivante, nous allons nous concentrer sur la modélisation de ces types de systèmes.

2.2 Modélisation des systèmes embarqués

En informatique, il existe plusieurs types de systèmes, nous citons entre autre, les systèmes transformationnels, les systèmes interactifs, les systèmes temps-réel, les systèmes réactifs, les systèmes embarqués et les systèmes sur puce.
Dans les paragraphes suivants, nous présentons les systèmes temps-réel et les systèmes embarqués qui sont au coeur de ce travail.

2.2.1 Systèmes temps-reél

Définitions

Au fil du temps, plusieurs définitions ont été énoncées pour décrire un système temps-réel. Il est intéressant de remarquer que l'élément déterminant dans toutes ces définitions reste la notion du temps. Nous donnons ci-dessous quelques définitions :

En 1965, dans [94], J. Martin a défini un système temps-réel comme un système qui contrôle son environnement en recevant des données, en les traitant et en produisant une action ou des résultats de façon suffisamment rapide pour intervenir sur le système à ce moment là.

En 1980, dans [50] Robert Glass définit un logiciel temps-réel comme un logiciel qui pilote un ordinateur qui interagit avec des dispositifs ou objets externes en fonctionnement, et, précise-t-il que le concept de temps réel et d'embarqué sont relativement interchangeables si ce n'est qu'un système embarqué est inclut dans le système qu'il contrôle.

En 1995, Gillie définit un système temps-réel comme un système dans lequel l'exactitude des applications ne dépend pas seulement de l'exactitude des résultats mais aussi du temps auquel ce résultat est produit.

Au demeurant, un système temps-réel peut être vu comme un système qui doit répondre à un ensemble de stimuli provenant de l'environnement dans un intervalle de temps dicté par ce même environnement, c'est-à-dire, des contraintes temporelles.

Caractérisation des systèmes temps-reél

Généralement, les systèmes temps-réel sont utilisés pour contrôler et agir sur un environnement qui leur sont extérieur.
Ils sont en conséquence constitués de deux sous-systèmes ; le système qui contrôle, qui est appelé *« Système Contrôleur »* et l'environnement qui est contrôlé qu'on appelle *« Système Contrôlé ou procédé »*. Ces deux sous-systèmes interagissent par des capteurs et des actionneurs comme montré sur la figure 2.1.

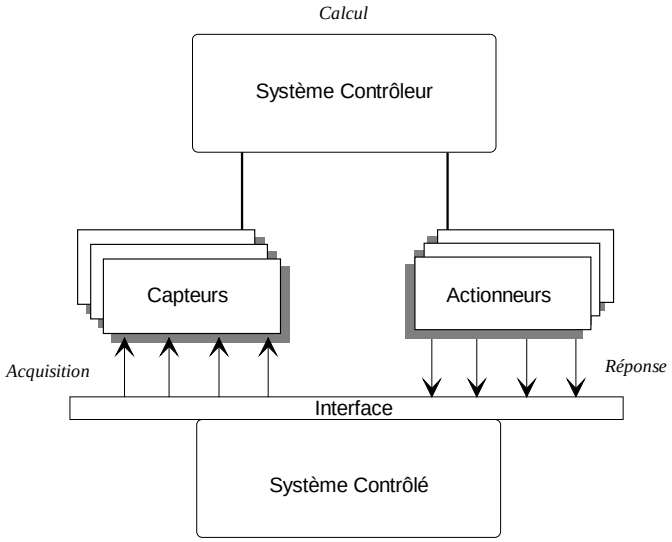


FIG. 2.1 – Interaction entre le contrôleur et le contrôlé

La relation entre les deux sous-systèmes est décrite par trois opérations fondamentales qui sont *l'acquisition de données, le calcul* et *la réponse*. Ces opérations doivent se réaliser à l'intérieur d'intervalle(s) d'un certain temps.
Cette contrainte de temps amène à une différenciation des systèmes temps-réel à savoir, les systèmes temps-reél *« doux »* dont le temps de réponse est important mais pas crucial et les systèmes temps-reél *« durs »* dont le temps de réponse est crucial.

En effet, une contrainte douce est moins contraignante, car elle permet une erreur raisonnable par rapport au moment exact où le processus aurait dû s'exécuter. Par contre, une contrainte dure ne permet aucune erreur sur le moment où le processus aurait dû s'exécuter.
De manière générale, dans un système temps-reél, un résultat de calcul mathématiquement exact mais arrivant au-delà d'une échéance prédéfinie est un résultat faux. Il est important de noter que particulièrement, en ce qui concerne les systèmes temps-reél durs, un résultat bien que correct mais arrivé en retard peut provoquer une défaillance, qui peut être fatale.

Considérons par exemple le cas d'un module temps-réel chargé de l'ouverture du parachute d'une sonde interplanétaire à une hauteur donnée de la surface d'une planète à explorer. Cette précision liée à plusieurs paramètres tels que la différence de pesanteur et de pression implique une précision de même nature sur le temps de réponse pour la prise de décision de l'ouverture dudit parachute. Puisque c'est une contrainte temps-réel dure, un dépassement de l'échéance temporelle peut entraîner l'échec des années d'études, du travail, de la navigation. En somme, l'échec total de la mission.

2.2.2 Systèmes embarqués

Définition des systèmes embarqués

Un système embarqué est un dispositif matériel comportant des parties logicielles et servant à résoudre des fonctions spécifiques. Il peut varier de simple contrôleur de lave vaisselle au système complexe de guidage de missiles par exemple. Il fonctionne par nature en temps-réel [38] et réside dans un autre dispositif plus vaste qui constitue son environnement, mais, qui n'est pas forcément un ordinateur [13].
Ainsi, de part sa nature, un système embarqué est utilisé pour contrôler et agir sur le système dans lequel il est logé et qui constitue son environnement.
Les systèmes embarqués sont classés selon leur utilisation. Il existe donc les systèmes embarqués personnels et les systèmes embarqués collectifs. Les systèmes embarqués personnels sont à vocation d'usage par un individu ou un petit groupe d'individus : agenda électronique, four à micro-ondes, téléphone, automobile. Les systèmes collectifs sont eux utiles à une communauté d'individus : satellite, avion, sous-marin, centrale nucléaire, poste électrique rural, poste de commande d'un réseau ferroviaire.

Caractérisation des systèmes embarqués

Puisqu'un système embarqué fonctionne par nature en temps-réel, il interagit avec son environnement en observant ses variations grâce à des capteurs et en agissant sur cet environnement qui l'englobe grâce à des actionneurs comme montré sur la figure 2.2. Les capteurs prennent en entrée des entités physiques et transforment les valeurs de cette grandeur en signaux électriques exploitables, réagissent et produisent des sorties en temps réel, car, nombre de ces systèmes interagissent directement avec le monde réel et également avec d'autres systèmes embarqués.
Ces interactions peuvent avoir lieu à des moments déterminés par une référence de temps interne au système ; on parle alors de système embarqué basé sur l'horloge. Elles peuvent également avoir lieu à des moments déterminés par l'environnement qui l'englobe ; on parle alors de système basé sur les événements.

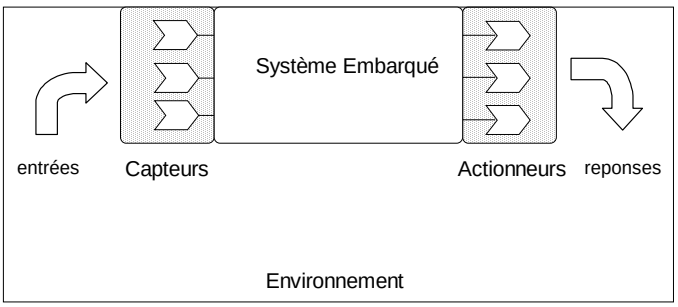


Fig. 2.2 – Système embarqué dans son environnement

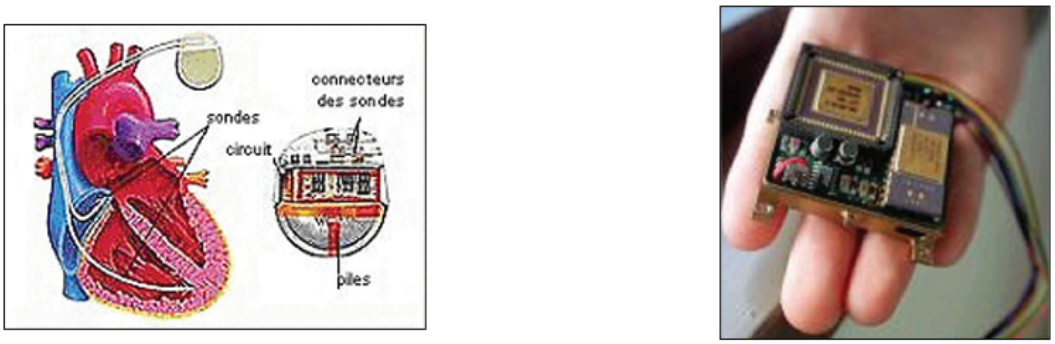
Les systèmes embarqués sont habituellement bien adaptés aux besoins du client, puisqu'ils ne sont prévus que pour effectuer seulement une gamme de tâches bien limitée. C'est pourquoi, la modélisation de tels systèmes impose un maximum de considération des contraintes sur les dispositifs d'environnement comme la puissance, l'énergie, le nombre d'instructions, etc. . ., qui typiquement ne sont pas bien caractérisés par l'informatique traditionnelle.

Quelques exemples de systèmes embarqués

Sur la figure 2.3, nous donnons deux exemples de systèmes embarqués dont l'un est dans le domaine médical et l'autre est dans le domaine spatial.

Un stimulateur cardiaque montré sur la figure 2.3(a) est un système embarqué personnel dont le principe est de contrôler son environnement à savoir, le coeur humain. Il délivre une impulsion électrique au niveau du muscle cardiaque de façon à entraîner son excitation et sa contraction. Il est composé d'électrodes et d'un boîtier. Ces électrodes sont ses capteurs et ses actionneurs. Elles sont introduites par une incision d'une veine et sont amenées jusqu'au coeur et précisément au muscle cardiaque. Dans le boîtier se trouve plusieurs composants dont une pile et un circuit électronique, plus ou moins complexe. Ce circuit électronique interagit avec le coeur en tenant le pacemaker informé en permanence de l'activité du coeur et en adaptant les impulsions électriques de quelques volts, qui seront appliquées à la cavité cardiaque qui doit être stimulée.

Le microphone montré sur la figure 2.3(b) est celui embarqué avec la sonde Mars Surveyor 98. C'est un système embarqué collectif. De part sa taille miniaturisée de cinq cm de côté, un cm d'épaisseur et cinquante grammes de masse, il fonctionne dans un environnement extrêmement particulier. En effet, son environnement est dans l'atmosphère de la planète Mars qui est dix fois moins dense que celle de la terre. Puisque physiquement le niveau sonore diminue avec la pression, la propagation d'un signal acoustique dans un domaine de fréquence audible devient difficile. En conséquence, ses capteurs sont donc reliés à des amplificateurs avec des gains importants pour lui permettre le traitement du son.



(a) (b)

FIG. 2.3 – Stimulateur cardiaque et Microphone embarqué avec la sonde Mars Surveyor 98

2.2.3 Modèles

La Modélisation [13] est la représentation formelle d'un concept, d'un système, d'un sous-ensemble etc...
Un modèle peut cependant être mathématique, dans ce cas, il peut être vu comme un ensemble d'assertions concernant les propriétés d'un système telles que sa fonctionnalité, sa dimension physique etc...
Un modèle peut également être constructif, dans ce cas, il définit un procédé informatique qui simule un ensemble de propriétés d'un système. Les modèles constructifs appelés également *« modèles exécutables »* sont souvent utilisés pour décrire le comportement d'un système répondant à une stimulation qui lui est extérieure.
Des modèles exécutables s'appellent parfois des simulations, un terme approprié quand le modèle exécutable est clairement distinct du système qu'il modélise. Cependant, dans beaucoup de systèmes électroniques et particulièrement pour les logiciels embarqués, un modèle qui commence comme une simulation évolue vers une implémentation logicielle du système. Dans ce cas, la distinction entre le modèle et le système devient floue.

La conception est par contre la définition d'une implémentation, c'est-à-dire d'un modèle exécutable sur une plate-forme cible qui impose des contraintes sur le fonctionnement du système. La conception implique généralement la réalisation de plusieurs modèles, chacun étant un raffinement du précédent. Le premier modèle peut être considéré comme la spécification formelle du système, et le dernier comme son implémentation.

En somme, le but de la modélisation est l'exploration des modèles pour une conception finale alors que le but de la conception demeure l'implémentation.

La conception et la modélisation sont évidemment étroitement liées. Dans certaines circonstances, les modèles peuvent être immuables, c'est-à-dire, qu'ils décrivent des sous-ensembles, des contraintes ou des comportements qui sont extérieurement imposés à une conception. Par exemple, ils peuvent décrire un système hydraulique qui n'est pas dans la conception, mais qui doit être commandé par un système électronique lequel par contre se trouve dans la conception.

Dans [55], l'auteur précise que quelle que soit la forme des modèles, ils partagent tous une même préoccupation qui est la capture du comportement d'un système et la description de ses propriétés pertinentes. Un modèle servira donc à valider le cahier de charges, à détecter les éventuelles erreurs de conception avant la réalisation du système, et, éventuellement contribuera à déterminer certains paramètres du système tels que les paramètres de contrôle/commande pour un système embarqué. L'analyse et la validation d'un modèle est donc une étape primordiale, notamment dans le cadre des systèmes embarqués. L'analyse et la validation doivent garantir que le système, une fois réalisé, se comporte correctement d'un point de vue fonctionnel et temporel.

La validation d'un modèle peut s'effectuer de deux manières à savoir, par preuve formelle ou par simulation. La validation par simulation consiste à exécuter un modèle et observer son comportement. La simulation fait appel à des environnements logiciels, appelés environnement de simulation, qui sont capables de reproduire un contexte d'exécution proche de l'environnement du système à valider, et d'exprimer les différentes conditions dans lequel peut se retrouver le système. MATLAB et PTOLEMY II par exemple, sont des environnements de simulation. Pour ce faire, un environnement de simulation adopte généralement un modèle de temps logique, appelé temps de simulation. Ce modèle de temps fournit une référence temporelle avec des unités pouvant être élastiques afin de simuler certaines situations ou conditions réelles du système.

2.2.4 Modèle de Calcul

A un niveau abstrait, un modèle d'un système peut être considéré comme un ensemble de composants qui ont des propriétés, et entre lesquels existent des relations. Les différents composants d'un modèle peuvent appartenir à des domaines techniques différents tels que l'électronique analogique, l'électronique numérique, la mécanique, la thermodynamique, l'optique etc... Ces domaines techniques ont des méthodes de modélisation et de conception différentes, et ne considèrent donc pas les composants et les relations entre composants de la même façon.
L'ensemble de *« lois physiques »* ou d'*« axiomes »* qui expriment des contraintes sur les propriétés des composants en fonction des relations entre composants est appelé *« modèle de calcul, (MoC), Model of Computation »*. L'ensemble des règles permettant de calculer les propriétés des composants est appelé *« sémantique du modèle de calcul »*.
Un modèle de calcul peut ainsi avoir plusieurs sémantiques, notamment quand ses lois autorisent plusieurs jeux de valeurs des propriétés des composants, chaque sémantique calculant un de ces jeux de valeurs. Dans la pratique, un modèle de calcul n'autorise généralement qu'un jeu de valeurs pour les propriétés des composants, et les différentes sémantiques correspondent à différents algorithmes de calcul des propriétés.
Du point de vue temporel, il existe deux types de modèles de calcul : les modèles de calcul temporisés qui implémentent le temps et les modèles de calcul non temporisés qui n'en implémentent pas.

Un modèle exécutable vu plus haut, est construit sous un modèle de calcul, qui fournit un cadre dans lequel un concepteur établit des modèles.
Ainsi, un modèle de calcul peut avoir plus d'une sémantique, du fait qu'il peut exister des ensembles distincts de règles qui imposent des contraintes identiques au comportement [13].

Il existe cependant plusieurs modèles de calcul utiles pour concevoir les systèmes embarqués [72]. Les interactions des composants dans des différents modèles, mènent ainsi les concepteurs à avoir des différentes manières de penser et utiliser plusieurs modèles de calcul différents.

Généralement, un modèle de calcul doit expliciter le comportement d'un système pendant son traitement qui peut s'effectuer de manière cyclique, réactive, concurrente ou séquentielle. Il doit également fournir le protocole de communication utilisé pour les interactions avec son environnement. Ce protocole de communication peut être le rendez-vous, le passage de messages, l'échange d'événements, etc... . De même un modèle de calcul doit spécifier le format de données à utiliser qui peut être des événements, des requêtes, des flux, etc...

Actuellement, plusieurs modèles de calcul sont spécifiés et d'autres encore sont en cours de spécification. La variété des domaines d'application des systèmes embarqués et de la croissance de leur complexité influencent la spécification des nouveaux modèles de calcul.

Dans les paragraphes suivants, nous présentons quelques modèles de calcul les plus utilisés à savoir : Flot de données (DF : Data Flow), Flot de données synchrones (SDF : Synchronous Data Flow), Temps continu (CT : Continous Time), événements Discrets (DE : Discrete Event, Synchrone/Réactif (SR : Synchronous/Reactive), Processus Séquentiel Communicants (CSP : Communicating Sequential Process) et Machines à Etats Finis (FSM : Finite State Machines).

Flot de données et Flot de données synchrones

Le Modèle de calcul Flot de données est utilisé dans des systèmes qui interagissent avec leur environnement en échangeant des flots de données [112] [51] [76] [78] [15]. L'entrée du système dispose de files d'attente qui reçoivent une suite de données, et, lorsque les toutes les entrées sont sollicitées, le système exécute un traitement suivi d'une production éventuelle des données de manière asynchrone. Ces données sont déposées sur la file d'attente de sortie.

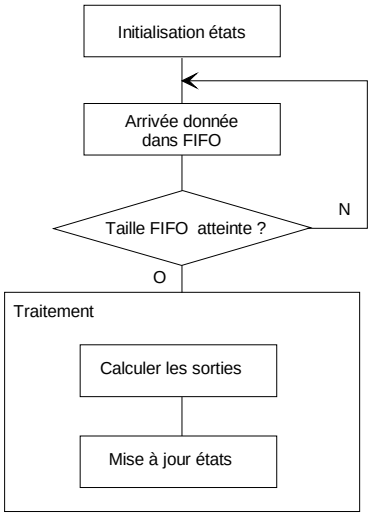


FIG. 2.4 – Comportement d'un système SDF

Le modèle de calcul Flot de données synchrones [77] [78] est dérivé du Modèle de calcul Flot de données. Ici le système consomme une suite de données de taille constante et produit une autre suite de données également de taille constante. Puisque les tailles des flots des données consommées et produites par chacun des composants du système sont connus d'avance, il est possible de spécifier les conditions qui éviteraient au système de congestionner ou d'entrer dans des interblocages. Il est également possible d'élaborer une programmation statique de l'exécution des composants.

Evénements Discrets

Le modèle de calcul des événements Discrets [103] [15] est utilisé dans des systèmes qui réagissent à des événements discrets ordonnés dans le temps. Un événement est vu comme un couple d'une donnée et d'une estampille de temps appelée date [72]. Cet événement peut être une variation de l'environnement observée à une date donnée dans le cas des systèmes pilotés par des événements. Il peut également être un top d'horloge dans le cas des systèmes pilotés par une horloge.

Lorsque cet événement arrive, le système est activé, il effectue son traitement en opérant son calcul et met à jour l'environnement par génération d'événements, instantanément ou dans le futur.

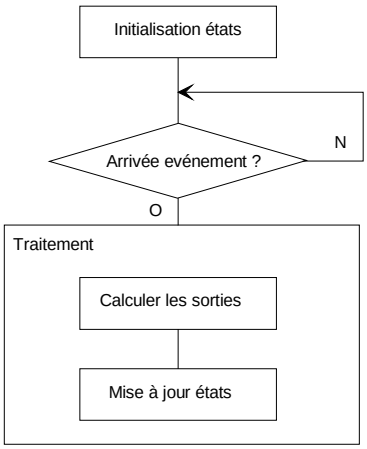


Fig. 2.5 – Comportement d'un système DE

Puisque tout événement doit être daté, ces types de systèmes disposent en conséquence d'une horloge globale. Cette horloge définit une référence commune à tous les sous-systèmes inclus. Ce qui permet au système global de définir un ensemble totalement ordonné d'événements du système y compris ceux survenus dans les sous-systèmes.

Les systèmes du type DE sont généralement utilisés dans la spécification matérielle et logicielle, dans la simulation des systèmes des télécommunications, dans les réseaux de

communication, dans les systèmes avec difficulté de maintien de la notion de cohérence globale, etc...

Temps Continu

Le modèle de calcul de Temps Continu [89] [15] est utilisé dans des systèmes qui utilisent une dynamique continue, c'est-à-dire, dans les systèmes disposant de composants interagissant par l'intermédiaire des signaux à temps continu. Ces composants indiquent des relations algébriques ou différentielles entre leurs entrées et leurs sorties.

Ces types de systèmes sont utilisés pour la spécification des circuits analogiques, mécaniques ou à micro-ondes et est également utilisés pour la spécification des systèmes de contrôle.
La simulation de ce type de système implique la résolution numérique des équations différentielles. Les techniques usuelles d'intégration déterminent un pas d'intégration (incrément du temps entre deux valeurs calculées) permettant d'obtenir une précision donnée sur la valeur des fonctions du temps. Lorsqu'on souhaite de plus avoir une certaine précision sur la date à laquelle une fonction passe par une valeur donnée (par exemple pour générer un événement), il est nécessaire de raffiner l'intégration jusqu'à ce que le pas indique suffisamment précisément la date de l'événement. L'état du modèle est alors obtenu par calcul d'un point fixe.

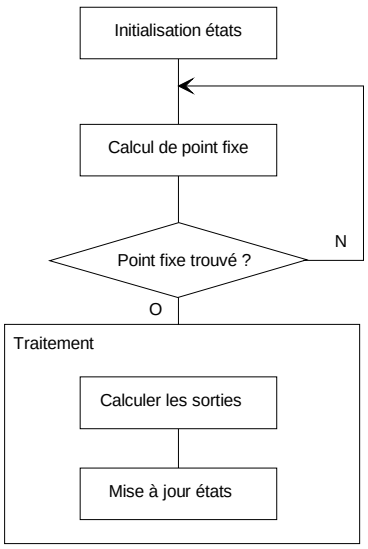


FIG. 2.6 – Comportement d'un système CT

Synchrone/Réactif

Le modèle de calcul Synchrone/Réactif [26] [139] [138] [11] [66] [15] [32] est utilisé dans les systèmes qui sont fortement synchronisés avec leurs environnements. Leur rythme de

fonctionnement est lié à celui de la variation de leur environnement pour être capable de répondre à toutes les occurrences de sollicitation émanant de son environnement.

Ces types de systèmes sont utilisés dans les applications avec logique de contrôle concourant et complexe, dans les applications critiques temps-réel et dans celles basées sur la sûreté, etc...

Un système réactif synchrone réagit à chaque événement qui se produit dans son environnement en produisant lui-même un événement et en modifiant son état interne. L'événement déclencheur et l'événement produit sont, selon l'hypothèse synchrone, simultanés. Dans la pratique, il suffit que la réaction du système à un événement soit terminée avant l'occurrence de l'événement suivant

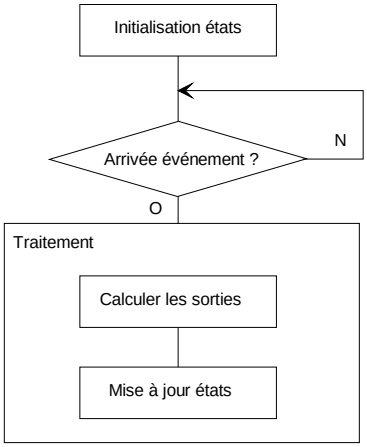


FIG. 2.7 – Comportement d'un système SR

Processus Séquentiel Communicants

Le modèle de calcul Processus Séquentiel Communicants [59] [123] [15] est utilisé dans des systèmes qui s'exécutent concurremment et qui peuvent avoir besoin de se synchroniser à des points précis dans leur traitement appelés « rendez-vous ».

Ces types de systèmes sont utilisés dans les applications avec partage ressources comme les bases de données Client/Serveur, dans les traitements multitâches, dans le multiplexage de ressources matérielles, etc...

Pendant son traitement, le système peut se retrouver bloqué à un point de rendez-vous en attente d'un message, d'une notification ou de la disponibilité d'une ressource partagée. Le système quitte son point de rendez-vous lorsque tous les systèmes concernés par ce rendez-vous sont présents.

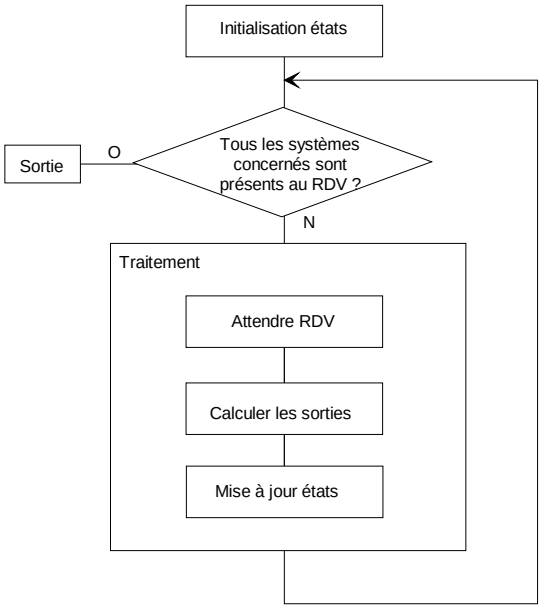


FIG. 2.8 – Comportement d'un système CSP

Machines à Etats Finis

Le modèle de calcul Machines à Etats Finis est utilisé dans les systèmes ayant des états discrets et finis [15].

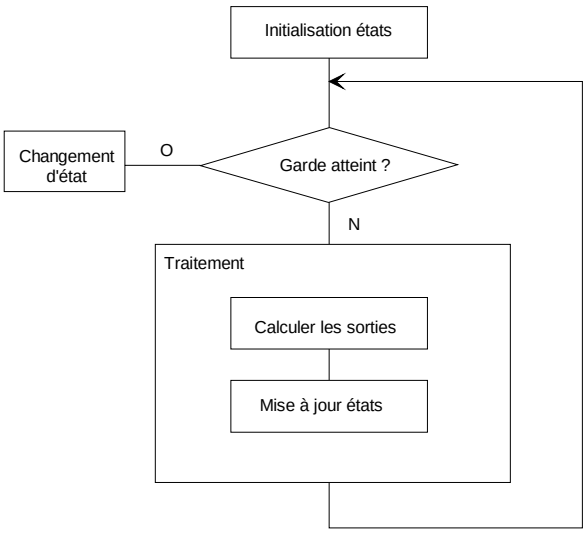


FIG. 2.9 – Comportement d'un système FSM

Ce modèle de calcul est utilisé dans les systèmes disposant d'applications ayant une logique de contrôle, en l'occurrence dans un contexte critique où le système doit assurer

un changement de mode tel que le passage en mode « déclenchement alarmes » en cas de grave anomalie dans le système. Ils sont également utilisés pour éviter des surprises de comportement. Ce sont des systèmes qui basculent d'un état vers un autre au cours de leur fonctionnement. Ces basculements successifs sont conditionnés par des expressions booléennes appelées « gardes ».

Actuellement, avec la complexification des systèmes embarqués et particulièrement dans les traitements concurrents, le modèle FSM ont été enrichi en prenant en compte ces contraintes de traitement. D'où le concept de Machines à Etats Finis concurrents [65] [67] [49]

Quelques modèles de calcul les plus utilisés

Le tableau ci-dessous résume quelques modèles de calcul les plus utilisés dans la modélisation des systèmes embarqués, leurs moyens de communication et leurs champs d'application.

		Moyen de Communication	Système ou Application
Modèles de calcul	CSP	RDV (passage de message synchrone)	Appli avec partage ressourçes (DB Client/Serveur), - Traitement multitâches, - Multiplexage de ressources matérielles
	CT	Signaux CT, Relations (algébrique ou différentielle in-out)	- Syst. Physique avec description d'équations linéaires ou non (algébriques ou différentielles), - Circuits analogiques ou à micro-ondes, -Syst. Mécaniques, - Syst de contrôle,
	DE	Envoi événements dans le temps suivant ligne RT	- Spé. Hard et Soft, - Simulation Syst des télécoms, réseau de communication, - Syst avec difficulté de maintien de la notion de cohérence globale, - VLSI (HF)
	FSM	Transition entre états	- Logique de contrôle, - Utilisé aussi pour éviter surprise de comportement
	SDF	Passage de message	- Calcul régulier sur flux, - Systèmes programmables à lancement parallèle ou séquentiel, calculable statiquement
	SR	Passage de message (Séquence de données)	- Applications avec logique de contrôle concourants et complexe, - Applications critiques TR, et celles basées sur la sûreté

FIG. 2.10 – Quelques modèles de calcul les plus utilisés

2.2.5 Orientation des systèmes embarqués

Classiquement, en modélisation comme en conception, on considère qu'un système peut être dominé par le contrôle ou par les données.
Le choix du modèle de calcul pour la modélisation se fait en fonction du type d'application qui caractérise le module qui lui-même dépend du système à modéliser.

Dans [8], l'auteur précise que généralement, lorsqu'un système est dominé par le contrôle, c'est-à-dire qu'il réagit aux événements[1] discrets, et si de plus il n'y a pas d'hypothèse sur l'occurrence des événements et tous les événements doivent être pris en compte, on utilise l'un des modèles de calcul suivants : événements discrets, réactifs synchrones, machine d'états finis, processus concurrents ou réseaux de Petri, etc...

Par contre lorsqu'un système est dominé par les données, c'est-à-dire les sorties sont fonctionnellement dépendantes des entrées et si de plus les occurrences des valeurs sur les entrées sont périodiques, on utilise l'un des modèles de calcul suivants : processus de Kahn, processus de flot de données, etc...

Le choix d'un modèle de calcul inadéquat peut compromettre la qualité de la conception en menant le concepteur dans une implémentation plus onéreuse ou moins fiable.
Pour les systèmes embarqués, les modèles de calcul les plus utiles manipulent à la fois la concurrence et le temps car ces systèmes sont des ensembles de composants qui fonctionnent concurremment et ont des multiples sources concurrentes de stimulations. En outre, ils fonctionnent dans un environnement synchronisé qui est le monde réel.

Nous avons vu que la modélisation d'un système est faite par sa représentation formelle. En ce qui concerne les systèmes embarqués, cette représentation est faite de manière abstraite sous forme d'un ensemble des « briques » emboîtées les unes dans les autres. Cette abstraction fait l'objet de la section suivante.

[1] Ici, un événement est pris dans le sens de [72], c'est un couple composé d'une valeur et d'un temps, où les temps peuvent venir d'un ensemble partiellement ou totalement ordonné. Dans les modèles temporisés, l'ensemble des temps est un ensemble totalement ordonné. Signalons toutefois que lorsque deux ou plusieurs événements partagent le même temps, on dit qu'ils sont synchrones.

2.3 Abstraction des systèmes embarqués

2.3.1 Concepts de base

Généralement, les schémas des blocks fonctionnels hiérarchiques supportent l'abstraction et le raffinement. L'abstraction permet de compresser un schéma fonctionnel dans un bloc simple et le raffinement permet de décompresser un bloc dans un schéma fonctionnel [5]. Cette abstraction est composable et dépend des modèles de calcul qui la composent. Dans [111] [131] [132] [7], les auteurs montrent que pour des systèmes hybrides par exemple, i.e., des systèmes qui combinent à la fois le modèle de calcul de temps continu et le modèle de calcul des machines à états finis, il existe deux abstractions : une continue et une discrète.

Actuellement, un système embarqué est modélisé comme un ensemble de modules hiérarchiques représentant une architecture abstraite. Cette représentation est indépendante du niveau d'abstraction. Pour chacun de ces modules, la communication et le comportement sont modélisés dans l'interface et dans le contenu. Et, ces différents modules sont finalement interconnectés par des canaux de communication qui respectent un schéma de communication bien précis.

L'interface de chaque module est ainsi vue comme un ensemble contenant deux sous-ensembles dont l'un contient des ports qui sont des points de communication du module avec l'extérieur et l'autre contient des opérations effectuées sur ces ports qui spécifient les interactions entre le module et le reste du système. Ces opérations peuvent être internes ou externes lorsqu'elles sont effectuées de l'intérieur du module contenant le port ou par un des modules externes. Dans PTOLEMY II par exemple, les opérations send, get et hasToken sont des opérations internes et l'opération hasRoom est une opération externe.

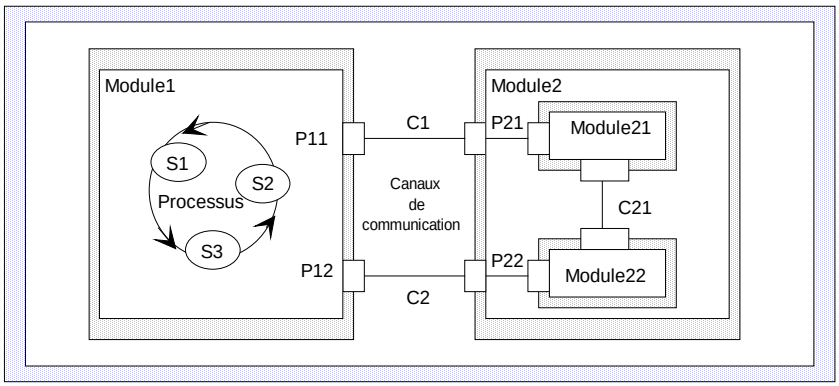


FIG. 2.11 – Système contenant deux modules qui communiquent

Illustration 2.3.1 Sur la figure 2.11, nous illustrons les concepts de base d'un système simple contenant deux modules qui communiquent entre eux. Ce système est modélisé comme un ensemble de deux modules dont chacun a un contenu et une interface.

Le module 1, a une interface comprenant un ensemble de ports P11 et P12 et un ensemble d'opérations internes et externes possibles sur ces ports. Il a également un contenu qui est une tâche ou processus qui suit un comportement composé d'opérations de calcul et d'opérations de communication.
Il communique à travers ses ports P11 et P12 et interagit avec le reste du système par les opérations contenues dans son interface. Le module 2 contient deux instances de modules, Module 21 et Module 22 qui communiquent entre eux par le biais du canal C21. Les modules 1 et 2 communiquent par l'intermédiaire de leurs ports via leurs interfaces, par des canaux de communication C1 et C2.

Les différents modules sont interconnectés par des canaux de communication qui assurent l'échange des données entre eux. Un canal de communication englobe tous les détails de communication de manière transparente pour les modules qu'il interconnecte, et garantit l'échange de données entre eux.

Un canal de communication est défini par le media qui véhicule l'information qui peut être des fils physiques ou abstraits, par un comportement par exemple la description des détails de communication et par le type de données transmises qui peuvent être des données génériques ou données en représentation déterminée.

2.3.2 Le canal de communication

Les sous-systèmes Module1 et Module2 présentés sur la figure 2.12 communiquent via deux canaux de communication C1 et C2 qui fournissent un ensemble de primitives de communication nommées *« services »*. Dans [37] [39], on présente l'appel de procédure à distance (RPC, Remote Procedure Call) comme un modèle d'unité de canal qui combine le concept de moniteurs et de passage de messages.

Dans la conception de systèmes, les aspects fonction et communication des modules doivent être clairement distinctes pour la réutilisation et la flexibilité des modèles [62]. La communication peut être modélisée à divers niveaux d'abstraction offrant, par conséquent, plus de flexibilité à la conception.

Généralement, dans le but de séparer les aspects de communication des aspects de comportement d'une spécification, on utilise le RPC pour invoquer les services de canaux. Ceci permet une représentation flexible, avec une sémantique claire et efficace, de modélisation des schémas de communication existants.

Le modèle du canal est composé d'un ensemble de services ou primitives de communication, d'une interface et d'un éventuel contrôleur qui assure la résolution des conflits d'accès au canal et l'adaptation des différents schémas de communication utilisés pour les sous-systèmes.
Ainsi, un sous-système peut communiquer au travers d'un canal en faisant appel à une primitive de communication du canal ou en faisant une lecture/écriture sur les ports d'interface comme montré sur la figure 2.12.

2.3.3 Le protocole de communication

La réalisation d'un ensemble de services est fournie par le *« protocole de communication »*. Il peut exister plusieurs réalisation de ce protocole parmi lesquelles le processus de synthèse choisit celle qui implémente les primitives de communication requises par un canal.
Un protocole contient une interface composée de ports et d'interconnexions nécessaires aux différents sous-systèmes utilisant ce protocole. Sur cette interface les primitives de communication lisent et écrivent comme montré sur la figure 2.12.

Le contrôleur est responsable de l'adaptation des protocoles de communication utilisés par les modules interconnectés par le même canal. Lorsqu'il n'y a pas de contrôleur, le protocole est complètement distribué entre les primitives de communications comme pour le cas du protocole Rendez-Vous. La complexité du contrôleur peut varier d'une simple file d'attente à un protocole complexe en couches. Il n'est pas exclu qu'un module existant fasse office de protocole de communication.

Dans la philosophie objet, cette notion de bibliothèque de protocoles est souvent remplacée par la puissance de la notion du *polymorphisme*, tel est le cas dans PTOLEMY II. En effet, la même méthode invoquée s'exécute en conformité avec la sémantique du domaine dans lequel elle est appelée, c'est-à-dire, en conformité avec la sémantique du modèle de calcul utilisé par l'environnement qui a évoqué cette méthode.

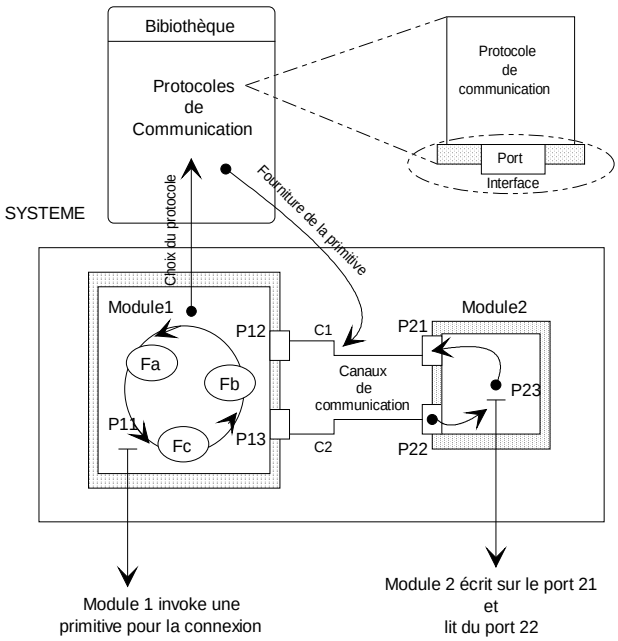


FIG. 2.12 – Modes d'accès au canal de communication

Illustration 2.3.2 Le module M1 fait le choix du protocole de communication dans la bibliothèque de protocoles. Puisque le protocole choisis peut avoir plusieurs réalisations dans cette bibliothèque, le processus de synthèse fait le choix du protocole approprié au modèle de calcul utilisé par les modules M1 et M2 et fournit la primitive qu'il faut sur le canal.

2.3.4 Communications et interfaces à travers les niveaux d'abstraction

Les systèmes électroniques embarqués sont modélisés, au travers de plusieurs niveaux d'abstraction. Au plus haut niveau d'abstraction, les détails peu essentiels pour une analyse préliminaire du système sont généralement ignorés. L'écart entre les concepts utilisés pour cette spécification et l'implémentation du système est réduit graduellement au travers des différents niveaux d'abstraction. En descendant dans l'abstraction, chaque niveau présente des aspects de plus en plus liés à la réalisation.

Généralement cette abstraction concerne *le comportement, la communication et le temps.* Les approches classiques définissent trois niveaux d'abstraction : le niveau physique, le niveau RTL, Register Tranfer Level et le niveau système[2]
Dans [37], le niveau système est décomposé en trois niveaux d'abstraction à savoir le niveau pilote des entrées-sorties, le niveau message et le niveau service tels que présentés sur la figure 2.13.

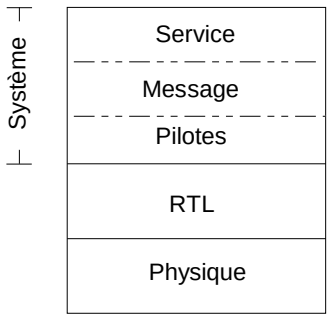


FIG. 2.13 – Décomposition du niveau système en trois sous-niveaux

Au niveau service, la communication est représentée comme une combinaison de requêtes de services. Ce réseau abstrait garantit le routage et la synchronisation des connexions établies dynamiquement. CORBA (Common Object Request Broker Architecture) [108] [109] et DCOM [121] sont des exemples de ce niveau d'abstraction avec des services comme la concurrence, le temps, l'évènement, etc. . .
Pour les interfaces des modules, elles sont composées d'un ensemble de ports d'accès à des

[2]Actuellement, pour l'abstraction des circuits électroniques, ces trois niveaux d'abstraction ne suffisent plus pour maîtriser la complexité de ces derniers. Un nouveau niveau d'abstraction appelé niveau de modélisation transactionnel, Transaction Level Modeling (TLM) a tété proposé pour modéliser les architectures SOC, System-On-Chip.

réseaux abstraits à travers lesquels communiquent les modules et d'un ensemble d'opérations sur ces ports, fournissant des services d'un certain type. Les tâches élémentaires sont des processus et Les opérations sur des ports sont des requêtes de services. Le temps de communication est non nul et peut ne pas être prévisible. Le temps global est abstrait et la relation entre les différents temps d'arrivée des requêtes de services est une relation d'ordre partiel, car, les requêtes de service peuvent arriver concurremment. A ce niveau les processus peuvent contenir des threads hiérarchiques similaires à ce qui est présenté dans StateCharts [56].

Au niveau message, les différents modules du système communiquent à travers un réseau explicite de canaux de communication, qui sont dits actifs. En plus de la synchronisation, ces canaux peuvent disposer d'un comportement complexe, par exemple la conversion des protocoles spécifiques aux différents modules communicants. Les détails de la communication sont englobés par des primitives de communication de haut niveau, par exemple send/receive et put/get et aucune hypothèse sur la réalisation de la communication n'est faite.

Des exemples de langages modélisant les concepts spécifiques de ce niveau sont entre autre : SDL (System Description Language) [119] [118] et UML (Unified Modeling Language) [133] [21] [110]. Certaines implémentations de StateCharts [56] [57] [140] peuvent être placées à ce niveau.
Pour les interfaces des modules, elles sont composées de ports d'accès aux canaux actifs. Cette fois-ci, les ports d'accès fournissent des appels de procédures de haut niveau. Le comportement des tâches élémentaires sont des processus qui communiquent avec l'extérieur par des envois ou des réceptions de messages. Le temps de communication est aussi non nul et n'est pas toujours prévisible. Le temps global est abstrait et la relation entre les différents temps d'envois de messages est une relation d'ordre partiel, car, les messages peuvent arriver concurremment. A ce niveau les processus peuvent contenir des threads hiérarchiques.

Le niveau Pilote est un niveau spécifique à la communication par des fils abstraits englobant des protocoles de niveau driver tel que le registre. Un modèle de ce niveau implique le choix d'un protocole de communication et la topologie du réseau de connexions. Un exemple de primitive de communication spécifique est l'écriture d'une valeur ou l'attente d'un événement sur un port. Quelques langages employés à ce niveau d'abstraction sont : CSP [59], SystemC 1.1 [129], et StateCharts [56] [140].
Les ports composant les interfaces sont des ports logiques, qui assurent l'interconnexion des différents modules par des fils abstraits. Les opérations effectuées sur ces ports sont des assignations de données de type fixé tel que l'entier, le réel etc... . Ces opérations peuvent cacher le décodage des adresses et la gestion des interruptions. Le temps de communication est non nul et peut ne pas être prévisible. Le comportement des modules de base est décrit par des processus qui réalisent un pas de calcul et des opérations de communication. La notion d'ordre global entre les événements du système apparaît avec

une hor1oge globale.

Au niveau RTL, la communication est réalisée par des fils ou bus physiques. L'unité de temps devient le cycle d'horloge, les primitives de communication sont du type set/reset sur des ports activés lors d'un nouveau cycle d'horloge. Les langages les plus utilisés pour la modélisation de systèmes à ce niveau sont SystemC 0.9-1.0 [129], Verilog [101] et VHDL [137] [60].
Les interfaces des modules, sont composées de ports physiques, qui permettent de connecter les différents modules par des fils physiques. Au travers de ces ports s'effectuent des opérations du type set/reset sur des signaux.
La communication étant réalisée par des fils physiques, les données sont transmises instantanément en représentation binaire. A ce niveau la gestion des interruptions et le décodage des adresses sont explicites. L'unité temporelle devient le cycle d'horloge et les processus correspondent à des machines d'états finis où chaque transition correspond à un cycle d'horloge.

				NIVEAU D'ABSTRACTION		
				SYSTEME		
CONCEPT				Service	Message	Driver	RTL
Module	Interface	Port		Port d'accès au réseau	Accès au canal actif	Port logique	Port Physique
		Opération		Demande d'un service	Send/Receive vers un processus identifié	Assignation de données sur un port	Set/Reset bits
	Contenu	Precessus/tâche		Threads hiérarchiques à la StateCharts	Threads hiérarchiques à la StateCharts	EFSM	Calcul au niveau cycle d'horloge
		Instance		Instance d'un module	Instance d'un module	Instance d'un module	Instance d'un module
		Canal de communication	media	Réseau abstrait	Canal actif	Fils abstraits	Fils physique
			comportement	Routage	Conversion de protocole	Protocole niveau Driver	Transmission
			donnée	Demande de service	Transmission de données génériques	Type fixé de données	Données en représentation fixe
PRIMITIVE DE COMMUNICATION TYPIQUE				Demande (Imprimer, fichier)	Send (fichier, disque)	Write (donnée, port)	Set (Valeur, port)

TAB. 2.1 – Synthèse de concepts de base à travers des niveaux d'abstraction

2.4 Modélisation hétérogène hiérarchique

Un système hétérogène typique [31] se présente comme montré sur la figure 2.14. Il est composé de plusieurs sous-ensembles qui peuvent utiliser différents modèles de calcul et peuvent avoir une partie matérielle ou une partie logicielle. Un modèle d'exécution prend soin de l'activation de ces sous-ensembles selon un ordonnancement approprié, et, la couche de communication permet à ces différents sous-ensembles d'échanger des données.

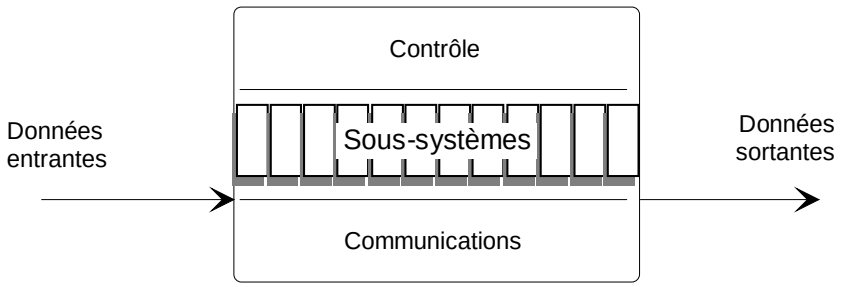
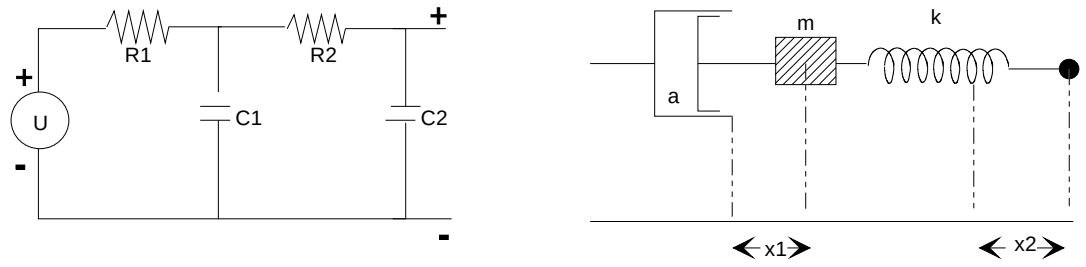


FIG. 2.14 – Structure type d'un système

Nous signalons par ailleurs que l'hétérogénéité dont nous faisons référence dans cette thèse est au niveau de la modélisation et non au niveau de la conception. Elle demeure relative aux règles qui régissent les interactions entre les différentes parties du modèle du système.

Une implémentation du système peut cependant utiliser un ensemble de règles simples sur lesquelles les différentes approches de modélisation seraient définies. Par contre, deux systèmes qui sont modélisés par le même modèle peuvent être implémentés avec différentes règles comme montré sur la figure 2.15, où, le modèle de temps continu est utilisé pour modéliser aussi bien un système mécanique qu'un système électrique.



$R_1R_2C_1\frac{d^2v}{dt^2} + (R_1C_1 + R_1C_2 + R_2C_2)\frac{dv}{dt} + v = U$ $\qquad$ $m\frac{d^2x_1}{dt^2} + a\frac{dx_1}{dt} + kx_1 = kx_2$

FIG. 2.15 – Deux différents systèmes modélisés par les équations de même type

2.4.1 Origine de l'hétérogénéité

Actuellement, la conception des systèmes embarqués fait appel à plusieurs modèles de calcul différents qui correspondent aux différentes techniques mises en œuvre. Par exemple, un système aéroporté de formation d'image contient les composants mécaniques et optiques, la cryogénie pour les capteurs à infrarouge, les algorithmes de traitement des signaux, les mécanismes de rétroaction etc... . Certains de ces composants peuvent être implémentés dans le matériel ou dans le logiciel. Chacun de ces composants exige le savoir-faire spécifique et les outils liés aux modèles de calcul spécifiques.
Ainsi, l'organisation d'un tel système et les interactions entre ces modules entraîne une mise en communication des modules n'ayant pas spécifiquement des caractéristiques fonctionnelles identiques, c'est-à-dire, n'ayant pas le même modèle de calcul ; d'où *« l'hétérogénéité du système»*.

En somme, la modélisation hétérogène est simplement la modélisation en utilisant plusieurs modèles de calcul. Puisque la plupart des systèmes sont hétérogènes de part leur nature, la modélisation hétérogène fournit des modèles plus naturels et plus complets.
Par exemple, le fait d'être capable d'utiliser à la fois les Machines d'état finis et des SDF permet de décrire explicitement le contrôle dans le modèle d'un système de traitement de signal numérique. Si nous n'étions limités qu'au modèle de calcul SDF, le contrôle et traitement de données devraient être codées ensemble et le modèle serait moins expressif et beaucoup plus difficile à maintenir.

Lorsqu'un des composants est contenu par un autre différent de lui, nous parlons de l'« hétérogénéité verticale ». Lorsque ces composants peuvent communiquer sans aucune notion de contenance, nous parlons de l'« hétérogénéité horizontale » [97] comme montré sur la figure 2.16.

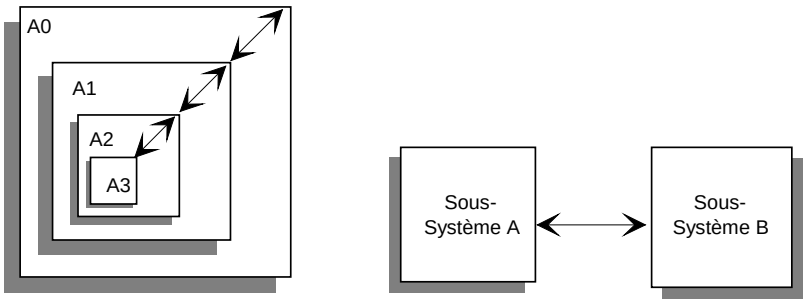


FIG. 2.16 – Représentation verticale et horizontale de l'hétérogénéité

Dans les deux cas, nous devons utiliser le fait que les composants puissent communiquer, mais en ayant différentes notions de communication du fait qu'ils utilisent différents modèles de calcul. Quand l'hétérogénéité est verticale, la sémantique des communications

est adaptée au croisement de niveau. Lorsque l'hétérogénéité est horizontale, nous devons adapter explicitement la sémantique des messages.

La conception hétérogène hiérarchique [8], ajoute la question fondamentale du choix du modèle de calcul supérieur qui régit le comportement global du système. Pour beaucoup d'applications, le contrôle dépend des données entrantes, et le traitement des données dépend du contrôle. Cependant, aucun modèle de calcul supérieur qu'il soit orienté contrôle ou orienté traitement de données ne peut entièrement satisfaire. L'utilisation de l'hétérogénéité horizontale permet de considérer les deux orientations du système au niveau supérieur.

2.4.2 Hiérarchie des sous-systèmes

Actuellement, en modélisation on utilise la représentation verticale de l'hétérogénéité, et, le passage des données entre deux domaines hétérogènes se fait grâce à la hiérarchie des sous-systèmes et à la synthèse de communication et d'interfaces.

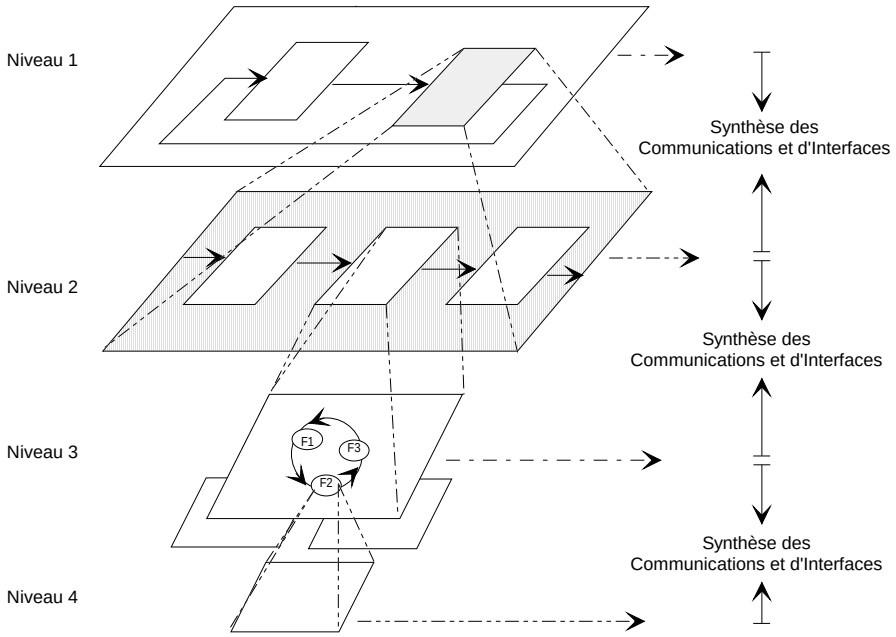


FIG. 2.17 – Représentation d'un modèle hiérarchique

Lorsque le système est hétérogène donc utilise plusieurs modèles de calcul, la communication d'un domaine vers un autre sans aménagements sur les concepts de communication devient impossible.

En effet, du fait que chacun des modules du système est spécifié dans un modèle de calcul différent et peut donc utiliser une sémantique différente des autres, il en découle des différences dans des schémas et protocoles de communication d'un domaine à un autre, et voire dans des topologies d'interconnexion.
Ainsi, la communication inter-modules impose une prise en compte du fait que lorsqu'un sous-ensemble donné utilise un modèle de calcul différent des autres, il doit être placé à un niveau différent dans la hiérarchie du système comme sur la figure 2.17.
De plus, la synthèse des communications et d'interfaces n'est généralement pas explicite. Ceci est rencontré dans la plupart des outils actuels de modélisation.

2.4.3 Conception hétérogène hiérarchique

La hiérarchie permet de considérer un système complexe comme un ensemble constitué des sous-ensembles élémentaires. C'est donc un outil qui permet de contrôler et de maîtriser la complexité d'un système [97] [98] [27] [46]. Cette complexité peut venir de la structure ou du comportement du système ; d'où la hiérarchie structurelle ou architecturale ou la hiérarchie comportementale [6].

Hiérarchie structurelle

La hiérarchie structurelle est utilisée pour identifier des sous-ensembles d'un bloc et de définir leurs interfaces par rapport [96] [97] aux autres sous-ensembles. Ces interfaces peuvent être composées d'un ensemble de signaux, ou même à un niveau d'abstraction plus élevé, elles peuvent être composées des canaux encapsulant des protocoles de communication plus complexes. La hiérarchie structurelle permet également de concevoir chaque partie d'un système de manière indépendante dès lors que les interfaces ont été spécifiées.

Hiérarchie comportementale

La hiérarchie comportementale est une manière de considérer un processus complexe comme résultat des processus élémentaires séquentiels ou concurrents. La composition des processus se sert des primitives de communication et de synchronisation telles que la détection d'arrêt, l'activation ou la suspension de processus ou le rendez-vous. Le traitement d'exception peut être considéré comme une hiérarchie comportementale puisqu'il peut être observé comme l'arrêt d'un processus et l'activation d'un processus de traitement d'exception [37]. Dans [13], l'équipe PTOLEMY II, du « Department of Electrical Engineering and Computer Sciences » de l'université de Berkeley en Californie a définit un autre type de hiérarchie appelée « hiérarchie de synchronisation » qui peut être considéré comme un cas particulier de hiérarchie comportementale.

Hiérarchie dans les plates-formes de modélisation et de conception

Un système hétérogène contient plusieurs modèles, chacun ayant son propre modèle de calcul et son propre modèle d'exécution, et les modèles doivent être structurés dans une hiérarchie, comme montré sur la figure 2.18.
Dans la plupart des plates-formes, le domaine principal définit la manière dont les composants sont activés et la manière dont ils communiquent.

Le système montré sur la figure 2.18 est hétérogène et hiérarchique. Il utilise le domaine A au niveau supérieur. Un de ses composants se comporte comme spécifié par un autre modèle qui utilise le domaine B. Un autre encore se comporte comme spécifié par une machine d'état fini en C. N'importe quel état de la machine d'état peut être raffiné comme une autre machine d'état ou comme un modèle utilisant le domaine A. Dans [65] [67] [49], les auteurs présentent ces types d'interactions avec des modèles concurrents.

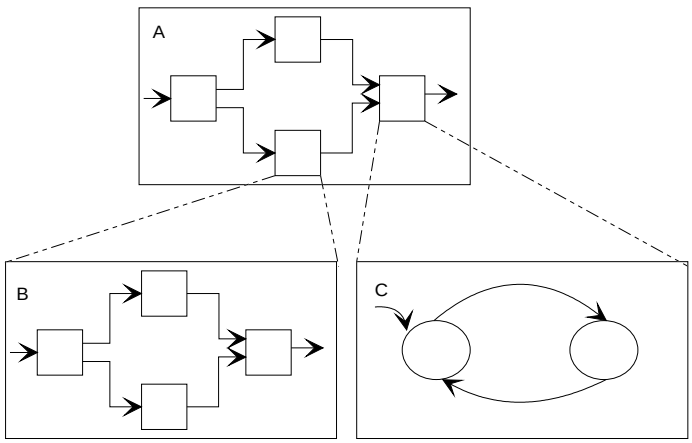


FIG. 2.18 – Modèles qui raffinent d'autres modèles

Nous avons vu au paragraphe 2.1 qu'un système embarqué est toujours de nature hétérogène, or, les environnements actuels de modélisation hétérogène se concentrent généralement sur un petit ensemble de modèles de calcul interragissants qui sont souvent des signaux continus et discrets pour l'ingénierie électrique et les machines d'états et des équations différentielles pour les systèmes hybrides.

Toutefois, puisqu'ils n'utilisent qu'un jeu de domaines très peu limité, de tels environnements de modélisation peuvent facilement définir l'union des domaines qu'ils supportent, et, ainsi permettre aux composants d'obéir simultanément à plusieurs modèles de calcul. Tel est le cas d'un convertisseur de signal numérique-analogique qui peut être modélisé comme un composant simple avec les entrées qui obéissent à un modèle de calcul de signal continu et de sorties qui obéissent à un modèle de calcul de signal discret.

Quant aux environnements de modélisation qui supportent un grand nombre ou un nombre extensible des domaines, ils ne peuvent pas construire cette union des domaines. Ils exigent que chaque composant obéisse seulement à un modèle de calcul. Ainsi, puisque les composants interconnectés obéissent au même modèle de calcul, les changements de domaine peuvent seulement se produirent à la frontière d'un composant. Or, le modèle de calcul utilisé à l'intérieur du composant peut ne pas être identique à celui utilisé en dehors de celui-ci. Ceci mène à la modélisation hétérogène hiérarchique utilisée par la plupart d'environnements de modélisation hétérogène tels que SpecC [124], SystemC [129], VDHL-AMS [34], ROSETTA [116], POLIS [113], el Greco [29], OMOLA [95], DYMOLA [43], MODELICA [44], PTOLEMY II [13] [14] [15] [42] [65], [82], etc. . . .
Ce constat nous amène à la problématique de la modélisation hétérogène hiérarchique présentée au paragraphe suivant.

2.5 Problématique de la modélisation hétérogène hiérarchique

2.5.1 Problématique

La hiérarchie est une manière efficace de contrôler la complexité d'un système en cachant les détails internes qui ne sont pas convenables à un niveau donné de modélisation.
Cependant, quand bien même cette hiérarchie faciliterait l'hétérogénéité dans les systèmes embarqués, cette approche présente plusieurs inconvénients.

En effet, lorsqu'on observe l'intérieur d'un composant, on obtient soit une description de bas niveau de son comportement s'il est atomique, soit un modèle de ce composant dans le même environnement de simulation s'il est composite. Dans les deux cas, l'interface du composant isole le niveau hiérarchique où il se trouve des détails internes de son comportement et de sa structure. Le modèle interne d'un composant peut donc utiliser un modèle de calcul différent de celui auquel il est soumis. Il faut toutefois définir pour cela comment les deux modèles de calcul vont interagir à la frontière du composant.

En effet, dans cette étude, nous considérons deux aspects dans la problématique de cette approche hiérarchique à savoir : l'aspect lié à la modélisation et l'aspect lié à la génération matérielle ou logicielle. C'est pourquoi, nous avons décorelé la problématique engendrée par cette approche hiérarchique suivant trois considérations, dont une problématique globale et deux problématiques dérivées.
La première problématique est située au niveau des outils de modélisation. La deuxième problématique est une des conséquences de la première, et, est au niveau des composants matériels ou logiciels générés. La troisième problématique est également une des conséquences de la première, et, est au niveau du concepteur de système.

Du point de vue des outils de modélisation, chaque changement de modèle de calcul, impose un changement de niveau hiérarchique. Ceci génère parfois des constructions

obscures dues aux imbrications induites par la coexistence des différents modèles de calcul. En conséquence, la représentation d'un simple système peut donc perdre de sa clarté et de sa lisibilité à cause de cette explosion d'étages hiérarchiques.
De plus, cette approche casse la modularité du système. En effet, par exemple, un simple ajout d'un composant dont les connexions vont sur deux autres composants situés à des niveaux hiérarchiques différents est quasi-impossible. Ceci réduit également la maintenabilité du système.

Du point de vue des composants matériels et logiciels générés, les composants fonctionnant à la frontière de plusieurs modèles de calcul, c'est-à-dire, possédant des entrées et des sorties obéissant à des sémantiques différentes sont interdits dans un modèle. En d'autres termes, les composants sont contraints de n'avoir que des interfaces qui n'apparaissent qu'à un seul niveau hiérarchique. Ceci altère aussi bien la modularité que la réutilisabilité des composants.
Par exemple, un convertisseur analogique/numérique pourrait être modélisé dans un domaine de temps continu, les sorties numériques étant considérées comme des signaux continus. Si un tel modèle reflète la réalité, il ne l'est pas au niveau d'abstraction lorsque l'on veut considérer les sorties comme séquence des valeurs discrètes. A l'inverse, le convertisseur analogique/numérique pourrait être modélisé en utilisant un domaine discret où les entrées continues seraient considérées comme des séquences d'échantillons discrets, transformant le dispositif en rééchantillonneur ou un no-op.

Pour le concepteur du système, il ne peut pas explicitement intervenir sur les transformations des données qui se produisent à la frontière de deux domaines. En effet, ce qui se produit quand les données croisent la frontière entre deux domaines dépend de l'environnement de modélisation, car, ce dernier cache les transformations qui se produisent à la frontière de deux domaines à l'intérieur des « composants passerelles ».
Puisque ces transformations dépendent de l'outil de modélisation, elles ne peuvent donc pas être explicitement énoncées dans le modèle d'un système. Le concepteur peut simplement adapter ou interpréter les données dans le modèle de calcul d'arrivée. Il s'ensuit que le comportement du système à la frontière des différents modèles de calcul ne peut nullement être explicitement spécifié. Ce manque de flexibilité peut s'avérer pénalisant dans la mesure où l'adaptation ou l'interprétation des données peut parfois induire à des imprécisions qui risquent de mener le système dans un comportement inattendu. En conséquence, le concepteur du système n'a pas la nette compréhensibilité aux « coutures » des différents modèles.

Pour remédier à ce problème, deux approches peuvent être employées.

– permettre au concepteur d'éditer le composant passerelle pour spécifier comment les données sont transformées quand elles passent par eux, c'est-à-dire, à la frontière des deux modèles de calcul,

– déplacez ces transformations des données, de composants passerelles vers l'intérieur du composant. Cependant, le déplacement des transformations à l'intérieur du composant crée une dépendance des spécifications internes du composant vis-à-vis du domaine dans lequel il est employé, ce qui altère la modularité et la réutilisabilité.

Ces éléments de problématique disparaissent lorsque nous allons permettre à plusieurs modèles de calcul de coexister au même niveau de la hiérarchie d'un modèle, ce que nous appelons « *Modélisation hétérogène non-hiérarchique* ».

2.5.2 Modèle hétérogène élémentaire

Nous savons qu'un système hétérogène met en communication plusieurs modèles de calcul. La représentation de cette coexistence est généralement peu lisible. C'est pourquoi, afin de construire un exemple sur cette approche hiérarchique, nous avons adopté une heuristique consistant à considérer un système élémentaire n'utilisant que deux modèles de calcul.

A la section 2.3.4, nous avons souligné qu'au plus haut niveau d'abstraction, les détails peu essentiels pour une analyse préliminaire du système sont généralement ignorés du point de vue de la modélisation. Ceci nous permet à ce niveau, de représenter ce système hétérogène comme sur la figure 2.19 où les deux ports TxOut et RxIn bien que n'utilisant pas le même modèle de calcul, sont directement connectés entre eux sans aménagement particulier sur le mécanisme de communication.

En effet, nous commençons par considérer un modèle élémentaire M qui contient trois variables internes dont deux acteurs hétérogènes et un canal de communication. L'acteur producteur Tx utilise le modèle de calcul tx et dispose d'un port de sortie TxOut. L'acteur consommateur Rx utilise le modèle de calcul rx et dispose d'un port d'entrée RxIn. Ces deux ports sont connectés via le canal de communication C1.

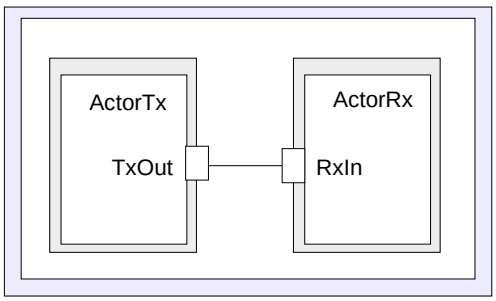


FIG. 2.19 – Modèle hétérogène élémentaire M

2.5.3 Exemple explicatif

Dans l'approche hiérarchique [49] [83] [84] [65] [85] [86] [87] [88] [30], lorsque deux ou plusieurs modèles de calcul différents coexistent, la question fondamentale est de la détermination du modèle de calcul supérieur. Si l'on applique l'approche hiérarchique au modèle élémentaire de la section 2.5.2, le modèle de calcul supérieur sera celui du domaine producteur tx, car, c'est ce dernier qui doit être activé en premier pour fournir des données au domaine consommateur rx.

Et, puisque l'acteur rx n'utilise pas le même modèle de calcul, il sera donc systématiquement encapsulé dans un composant composite utilisant son modèle de calcul comme montré sur la figure 2.20.
Cependant, il faut définir comment ce modèle de calcul supérieur interagira avec le modèle de calcul des composants composites et également comment les données seront transformées en passant de Tx vers Rx.

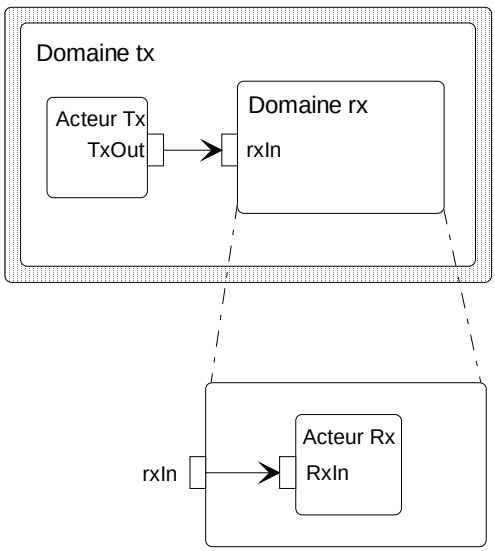


Fig. 2.20 – Modèle hétérogène élémentaire hiérarchique

C'est donc le modèle d'exécution supérieur, tx, qui s'occupe du transfert des données du port de sortie TxOut de l'acteur Tx au port d'entrée rxIn du composite rx. Ceci implique la fourniture par le modèle d'exécution supérieur de la sémantique de communication entre les ports TxOut et rxIn.

Pour la communication de l'acteur Rx avec l'extérieur du composite rx, elle est faite à travers l'abstraction hiérarchique du composite rx. En effet, le port rxIn du composite rx fait office de passerelle entre l'acteur Rx et le reste du système.

Dans certaines plate-formes par exemple, du point de vue de ses entrées et de ses sorties, l'acteur composite rx délègue le transfert de ses entrées à son modèle d'exécution interne, et le transfert de ses éventuelles sorties au modèle d'exécution tx. Ceci parce que, dans le mécanisme de transfert des données dans ces plates-formes, il revient au modèle d'exécution approprié au modèle de calcul du port de destination de manipuler le transfert des données. Parmi ces plates-formes on dénombre PTOLEMY II.

En effet, dans PTOLEMY II, le type de receiver utilisé dans un port dépend du protocole de communication, qui à son tour dépend du modèle de calcul. Puisque c'est le modèle d'exécution qui donne la sémantique du modèle de calcul, c'est donc lui qui détermine les receivers nécessaires sur les ports d'entrée. C'est ainsi que le modèle d'exécution contrôle les canaux de communication, car, ceux-ci sont implémentés par le receiver. C'est pourquoi, il revient au modèle d'exécution supérieur tx de fournir les receivers au port entrant rxIn du composite rx.

Lorsque les données sont dans le composant composite rx, c'est le modèle d'exécution rx qui les transfère du port d'entrée rxIn de l'acteur composite rx au port d'entrée RxIn de l'acteur Rx. La représentation de cette approche est montrée sur la figure 2.20.

Dans cette approche, trois observations sont à relever :

– Primo, sur le plan de la modélisation, lorsque le nombre de différents modèles de calcul utilisés par les composants croit, les étages hiérarchiques artificiels explosent. De ce fait, un système peut arriver à un nombre incalculable d'imbrication des composants.
  Pour gérer ce problème, du point de vue ergonomique et pour le confort de l'utilisateur, certaines plates-formes ont développé des interfaces utilisateur en conséquence. Tel est l'exemple de Vergil [13] [135][3] dans PTOLEMY II.

– Secundo, sur le plan de la communication, le passage des données de TxOut à RxIn n'est pas explicite. Les outils de modélisation effectuent cette transformation de manière implicite.
  Dans PTOLEMY II par exemple, le passage de données d'un domaine CT vers un domaine DE ou l'inverse est fait par des composants *« EventGenerator »* ou *« WaveformGenerator »* [82] [84] [88].
  Ces composants génériques ont un comportement prédéfini. Même si en tant qu'acteur, l'utilisateur peut le modifier à l'aide de ses paramètres, les cas limites ne trouvent pas toujours leurs satisfaction.

– Tertio, sur le plan du comportement hétérogène, il découle de la communication que, du fait que la transformation des données est implicite, le comportement hétérogène du système à la frontière des deux modèles de calcul l'est également.

[3] Vergil est une interface utilisateur graphique où l'on peut construire des modèles avec des imbrications « quasi-illimitées ».

2.5.4 Exemple réel : Redresseur de signal

Considérons l'exemple simple d'un redresseur de signal pour illustrer la problématique de l'hétérogénéité hiérarchique.
Ce système reçoit un signal d'entrée comme une séquence d'échantillons de données. Un multiplicateur multiplie chaque échantillon par 1 ou par -1 et bascule entre les deux comportements selon des événements produits par le détecteur de changement de signe.

Le système illustré sur la figure 2.21 montre un modèle hiérarchique de ce système. Le niveau supérieur utilise des flots d'échantillons de données, et le comportement du détecteur est modélisé en utilisant des événements discrets. Quand le flot d'échantillons entre dans le détecteur, il est converti en séquence d'événements ayant chacun une valeur. Quand un événement est produit à la sortie, sa valeur est utilisée pour construire un échantillon de données dans le domaine externe.
C'est seulement un exemple de ce qui peut se produire à la frontière d'un composant, et le point important est que ces transformations dépendent de l'outil de modélisation et n'est pas indiqué dans le modèle du système.

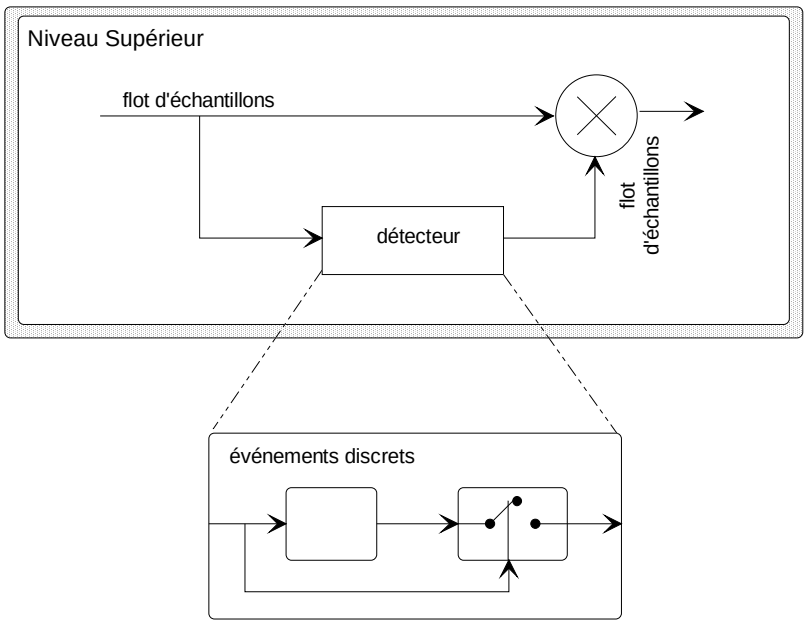


Fig. 2.21 – Exemple d'un modèle hétérogène hiérarchique

Si ce flot de données dans lequel le détecteur est utilisé est synchrone, c'est-à-dire, exige qu'un échantillon de données soit produit sur la sortie chaque fois qu'un échantillon est consommé sur l'entrée, le comportement d'événement discret du détecteur doit respecter cette condition. Ainsi même si le signal d'entrée ne change pas son signe et aucun événement ne doit être produit, le détecteur doit quand même produire quelque chose sur sa sortie pour obéir à la sémantique du domaine externe.

Ici, nous avons mis un échantillonneur qui utilise la valeur du dernier événement émis pour produire une sortie chaque fois qu'un échantillon d'entrée est consommé. Nous devons mettre cet échantillonneur dans le modèle interne du détecteur en raison de la sémantique du modèle de calcul externe, ainsi l'exécution du détecteur dépend du contexte dans lequel il est utilisé. Cette dépendance altère la modularité et la réutilisation.

2.6 Conclusion partielle

Dans ce chapitre nous avons présenté un état de l'art de la modélisation des systèmes hétérogènes en général et particulièrement celle des systèmes embarqués. Nous avons également abordé la notion de modèle de calcul et avons donné quelques-uns les plus utilisés. La notion de modèle hétérogène et celle de hiérarchie ont aussi été présentées, ainsi que les origines et les différents types de hiérarchie que l'on peut rencontrer dans un système hétérogène.
Un point important dans ce chapitre est la mise en évidence des désavantages qui président à la problématique qui motive cette thèse.
Pour la modélisation de ces systèmes hétérogènes, il faut disposer d'outils et d'une méthodologie adaptés. Le chapitre suivant présente ces différents outils, environnement et méthodologies de modélisation hétérogène.

Chapitre 3

Outils et méthodologies de modélisation hétérogènes

Résumé -Dans ce chapitre, nous commençons par une présentation de l'environnement de modélisation hétérogène en nous focalisant sur les différents langages et plates-formes de modélisation hétérogène. De cette présentation, nous justifions le choix de Ptolemy II en nous basant sur son caractère unifié. Ensuite, nous présentons les méthodologies de modélisation orientées composants ainsi que le primitives abstraites que nous utilisons dans cette étude. Nous justifions également le choix de la méthodologie orientée acteur que nous avons adoptée pour la réalisation de cette thèse.

3.1 Introduction

Dans le chapitre précédent, nous avons vu que le choix d'un modèle de calcul inadéquat peut compromettre la qualité de la conception en menant le concepteur dans une implémentation plus onéreuse ou moins fiable. En général, ce principe s'étend aussi bien sur les langages et plates-formes que sur les méthodologies de modélisation et de conception à utiliser.

Cependant, lorsque le système à modéliser est hétérogène, donc, implique plusieurs modèles de calcul, le choix de l'outil de modélisation est généralement orienté vers des plates-formes de modélisation unifiées, sachant que le choix d'un outil de modélisation est étroitement lié à celui de la méthodologie.

Cette étude, puisque dédiée aux systèmes hétérogènes n'échappe donc pas à cette règle. En conséquence, nous allons au préalable dégager clairement nos orientations en ce qui

concerne la détermination de notre choix aussi bien sur l'outil que sur la méthodologie de modélisation qui vont sous-tendre cette étude. En outre, nous choisirons une sémantique à utiliser en adéquation avec le fait que la modélisation au niveau système est abstraite. Elle est abstraite, parce qu'elle ne doit pas faire d'hypothèse sur les modèles de calcul à utiliser.

En somme, ce chapitre apporte quelques éléments qui ont concouru aux différents choix que nous avons élaborés en ce qui concerne l'outil, la méthodologie et la sémantique sur lesquels repose notre étude.

3.2 Outils de modélisation hétérogène

3.2.1 Tendance actuelle

Un langage idéal de description doit couvrir tout le flot de conception, tout en restant exécutable à tous les niveaux d'abstraction. Il doit également tenir compte des aspects fondamentaux du système tels que la concurrence et l'aspect temporel. Malheureusement, ce langage n'existe pas. Les langages existants accusent des limites. C'est pourquoi plusieurs méthodes de spécification fondamentalement différentes les unes des autres ont été développées.

En effet, actuellement, dans la recherche pour la modélisation d'un système, deux principales approches s'opposent : certaines initiatives tentent d'adapter les langages comme le C/C++ à la description hardware en tenant compte de ses aspects fondamentaux tels que la concurrence et l'aspect temporel. Cette solution implique le développement de bibliothèques ou de classes nécessaires pour représenter les composants matériels et leurs interactions. Quelques exemples de langages obtenus par extension du langage C++ sont SysternC, C-level et IP-modeling.

La deuxième approche cherche à étendre les langages de description matérielle, HDL[1] à la description système [93]. Cette extension de la syntaxe existante a été l'alternative adoptée pour la définition du langage SpecC [47].

Une troisième voie est la spécification de nouveaux langages qui peuvent suivre une nouvelle syntaxe, comme le langage Rosetta par exemple [116] [4].

[1]Hardware Description Langage

3.2.2 Concept

Il est important de pouvoir modéliser un système entier à plusieurs niveaux d'abstraction, allant de la spécification à l'implémentation [48]. Partant d'un niveau fonctionnel abstrait, le modèle est affiné en vue d'être partitionné suivant une spécification architecturale. Cet affinement se poursuit progressivement pour tenir compte des détails de l'implémentation et des contraintes de synthèse [120].

Nous avons vu à la section 2.2.5 que la modélisation d'un système consiste à sa représentation comme un ensemble composé de sous-ensembles communicants. Ces sous-ensembles, ayant des spécificités différentes les uns des autres, obéissent à des différentes règles abstraites et imposent d'une manière induite l'utilisation de la sémantique d'un modèle de calcul approprié. La conséquence de cette différentiation naturelle est l'utilisation de plusieurs modèles de calcul qui implique généralement plusieurs langages de modélisation pour différents composants.

L'adoption d'un langage système nécessite la définition d'une méthodologie de conception adaptée [127]. Il est envisageable de recourir à des outils formels en première phase de spécification tels que SDL, UML etc...
La méthodologie doit spécifier clairement le processus d'affinement progressif des modèles et la façon dont la transition se fait d'un niveau d'abstraction à un autre. Par ailleurs il est aussi essentiel de définir les procédures de simulation, de vérification à suivre ainsi que les techniques de partitionnement qui seront utilisées.

En effet, dans différentes étapes de la modélisation ou de la conception, un système peut être vu comme un ensemble de modules communicants dont chacun se comporte comme un modèle de calcul et est généralement spécifié dans un langage. Différentes stratégies de modélisation du système peuvent être utilisées. Chaque stratégie demande une organisation différente de l'environnement de modélisation. Des critères de choix d'un langage et une comparaison de différents langages de spécification sont présentés dans [61]. Le type d'application, la culture du concepteur et les outils de conception et de validation disponibles sont des éléments qui gouvernent le choix d'un concepteur dans l'utilisation d'un outil de conception.

Il existe des langages et des plate-forme de conception et de modélisation. Cependant, généralement, pour la modélisation des systèmes dans leur ensemble, on utilise des plates-formes de modélisation. Certains langages sont spécialisés sur la description matérielle, d'autres sur la spécification systèmes et d'autres encore sur la modélisation. Certaines plates-formes combinent à la fois la modélisation, la simulation et la conception.

3.2.3 Langage de modélisation des systèmes embarqués

Les langages de description sont classés en deux grandes familles à savoir,

– les langages orientés matériels et
– les langages orientés systèmes.

Langages de description matérielle

Les langages de description matérielle sont des langages qui supportent les concepts spécifiques aux systèmes matériels tels que le concept de temps, de parallélisme, de communication inter-processus, c'est-à-dire, signaux et protocoles, la réactivité et les types de données spécifiques, entiers signés par exemple. Parmi ces langages on peut citer Verilog [101], VHDL [137] [60], SystemC 0.9-1.0 [129].

En ce qui concerne SystemC 0.9-1.0 par exemple, il modélise un système comme un ensemble d'entités physiques interfacées par des ports physiques. Les opérations sur les ports sont du type set/reset bits ou assignations de données. La communication est représentée par des canaux physiques permettant le transfert des données dans une représentation fixe. C'est un langage, certes, très performant pour la modélisation des concepts au niveau matériel, mais, il accuse quelques faiblesses sur la modélisation des concepts interface et communication. Pour l'interface, le concept de port abstrait sur lequel peuvent s'effectuer des opérations indépendantes de protocoles n'est pas modélisé. Pour la communication également, le concept de canal abstrait qui transfère des données de type générique n'est pas non plus modélisé.

Langages de description des architectures (ADL)

Ces langages permettent une définition formelle de l'architecture de haut niveau d'un système complexe. Pour la spécification d'un système, les ADLs utilisent trois concepts de base [99] : le module, le connecteur et la configuration qui est une combinaison particulière des connecteurs et modules. Ces concepts sont interprétés différemment par les ADLs, le même système peut être décrit différemment en fonction du langage utilisé. Les langages suivants en sont des exemples.

SpecC [124] : langage de description et de spécification système basé sur le langage C, avec des extensions syntaxiques qui d'ailleurs le rendent incompatible à la norme ANSI. Ces extensions incluent, par exemple, des types de données adéquats, des commandes pour supporter la concurrence et la description de machines à états.

Superlog [126] : ce langage tend à étendre le langage Verilog, au langage de conception système en y incluant des possibilités de programmation en langage C.

Rapide [90] [91] [114] : Rapide est un langage exécutable de description d'architecture, EADL[2] pour la modélisation et la simulation du comportement de systèmes complexes. Il est prévu pour la conception d'architectures de systèmes distribués. Ces systèmes peuvent être composés de composants spécifiés dans différents langages par exemple VHDL, Ada, C++, C et Rapide. La communication entre les différentes parties du système est réalisée par des événements.

Langages orientés objet pour l'analyse et la conception des systèmes

Ces langages permettent une description des systèmes complexes à un très haut niveau d'abstraction. La principale idée est la décomposition des systèmes complexes en sous-systèmes plus faciles à maîtriser. Elles utilisent les avantages de l'approche objet pour la description, la visualisation et la documentation des systèmes. Le plus connu est UML [133] dont le principal inconvénient est l'absence de modèle exécutable pour la synthèse et la vérification.
StateCharts [56] [57] [140] et SDL [119] sont présentés comme des modèles exécutables d'UML. Il s'agit de langages existants auxquels sont associées des méthodes de décomposition de spécification conformément à UML.

Langages pour la modélisation des systèmes temps réel

Les systèmes temps-réel sont des systèmes dont le comportement doit respecter des contraintes temporelles : le temps de réponse est du même degré d'importance que la précision du résultat. Parmi ces langages temps réel, il y a ceux qui sont synchrones comme Lustre [53] [54], Esterel [16] [17] [18] [19] [20] [26] [22] [28] et ceux qui sont asynchrones tel que SDL [119].

Langages de programmation

Comme nous l'avons souligné plus haut, certaines approches ont étendu les langages utilisés en informatique pour la programmation, par l'ajout des concepts spécifiques au matériel. Le choix de ces langages est justifié par le fait qu'ils fournissent le contrôle et les types de données nécessaires, et, du fait aussi que la plupart des systèmes contiennent des parties matérielles et logicielles.

Hormis ces raisons d'ordre techniques, il y a une autre raison d'ordre culturel qui repose sur le fait que les concepteurs sont familiers avec ces langages et leurs outils relatifs. Les langages développés consistent en la définition des fonctions ou classes pour modéliser la concurrence, le temps et le comportement réactif. Les langages de ce type les plus utilisés sont N2C [136], SystemC [129], SpecC [124], [47], JavaTime [142] et OpenJ [143].

[2]Executable Architecture Descrption Language

3.2.4 Plates-formes de modélisation des systèmes hétérogènes

Certaines plates-formes de modélisation des systèmes hétérogènes proposent une représentation unifiée du système. D'autres plates-formes proposent la conception du système à partir de la spécification multilangage, sans réaliser une représentation unifiée du système entier. Quelques plates-formes les plus utilisées en modélisation sont :

SpecC [124] : au-delà du langage lui-même évoqué au paragraphe 3.2.3, les travaux menés par l'équipe de D. Gajski [47] introduisent des concepts intéressants pour la modélisation système et les techniques de raffinement. SpecC est développé dans le cadre du STOC, SpecC Technology Open Consortium.

Simulink [100] : Simulink est l'extension graphique du langage MATLAB permettant de travailler avec des diagrammes en blocs.

SystemC [129] : SystemC est une plate-forme de modélisation composée de librairies de classes C++ et d'un noyau de simulation permettant de faire de la conception au niveau système, au niveau comportemental et au niveau de l'implémentation. Certains concepts de SystemC comme les interfaces, les canaux et les événements sont inspirés de SpecC.

Space [33] : SystemC Partitioning of Architectures for Co-design of Embedded systems développé par l'Ecole Polytechnique de l'Université de Montréal consiste en l'analyse et la conception d'une plate-forme de codesign en SystemC. Il permet l'exploration architecturale à haut niveau d'abstraction. Space renforce le support logiciel dans SystemC 2.0 en encapsulant des fonctionnalités de système d'exploitation temps réel.

Rosetta [116] [4] : Rosetta vise à mettre ensemble, dans l'environnement du langage, des informations hétérogènes appartenant à des domaines différents. Chaque domaine utilise sa propre sémantique pour décrire ses modèles. Rosetta n'impose pas un modèle commun, mais permet de partager des informations entre des modèles hétérogènes.

Polis [113] : L'environnement Polis permet la spécification, la synthèse et la validation de systèmes temps réel orienté contrôle. La conception du système est réalisée à partir d'une représentation unifiée du comportement du système capable de spécifier à la fois les aspects matériels et logiciels. Le format unifié est utilisé dans toutes les étapes de conception afin de préserver les propriétés formelles de la conception.

Coware N2C : L'environnement Coware permet la conception de systèmes distribués à travers l'utilisation d'une représentation unifiée du système spécifiée dans le langage C/C++ ou dans des extensions de ces langages [129] [36]. L'objectif est d'utiliser le langage C/C++ pour la spécification du système au niveau système permettant la conception du matériel et le développement du logiciel en parallèle. La validation de la spécification du système est faite par l'exécution du modèle unifié.

Music : L'environnement Music est conçu pour la conception de systèmes hétérogènes multilangages spécifiés au niveau système [35]. Il permet la conception complète du système à partir du niveau système ainsi que la validation du système par cosimulation à plusieurs niveaux d'abstraction.

el Greco [29] : el Greco est focalisé sur la conception des outils au niveau système pour la génération des systèmes de communication numérique et multimédia. Il précise les conditions, les actions, et tout autre comportement dans un sous-ensemble de C++, qui est analysé et compris par l'outil.

PTOLEMY II [13] [14] [15] : Le projet de PTOLEMY vient du groupe CHESS, Center for Hybridand Embedded Software Systems du « Department of Electrical Engineering and Computer Sciences » de l'université de Berkeley en Californie. Ce projet des recherches fondamentales et appliquées dans des techniques logicielles pour les systèmes embarqués a donné lieu à la plate-forme PTOLEMY II qui utilise l'approche objet pour permettre la modélisation, la conception et de simulation de systèmes hétérogènes. Il utilise une approche unifiée et est programmé en JAVA et chacun de ses composants utilisables que l'on appelle « acteur » est codé par une classe JAVA.

Comme souligné plus haut, parmi les plates-formes de modélisation des systèmes hétérogènes, il en existe celles qui utilisent une approche unifiée, c'est-à-dire, capables de modéliser la quasi-totalité des modèles de calcul. Cette approche, permet l'unification des différents composants obéissant à différents MoCs au sein d'une même plate-forme ; ce qui élimine les problèmes d'intégration des différentes parties du système.

Dans cette thèse, nous ne nous concentrerons ni sur un langage ni sur une plate-forme particuliers. Cependant, le fait que nous utilisions plusieurs modèles de calcul, afin d'écarter d'éventuels problèmes d'intégration, nous orientons notre démarche vers une logique unifiée de modélisation. Ceci nous impose donc l'utilisation d'une plate-forme unifiée pour la modélisation et la validation par simulation de nos concepts. D'où le choix de la plate-forme PTOLEMY II.

Le choix de la plate-forme implique celui de la méthodologie à utiliser. C'est pourquoi, dans la section suivante, nous présentons quelques différentes méthodologie orientées composants et choisissons notre méthodologie en adéquation avec le choix fait sur la plate-forme.

3.3 Méthodologies de modélisation orientées composants

3.3.1 Introduction

Dans la modélisation et conception basées sur les composants, le problème fondamental est la décomposition d'un système en différents sous-ensembles qui soient individuellement à la fois facilement maniable et spécifique à un domaine.
Cette décomposition permet aux concepteurs de diviser et de dominer efficacement le problème de conception [80].
Les méthodologies de modélisation et de conception orientées composants préconisent des approches qui permettent de décomposer un système en composants avec des interfaces bien définies. Chacun de ces composants encapsule certaines fonctionnalités, telles que le calcul et la communication.
Parmi ces méthodologies de conception, il existe entre autre l'orientation objet, l'orientation middleware et l'orientation acteur [130].

3.3.2 Modélisation et conception orientée objet

La modélisation et conception orientée objet [80] contrôle la complexité dans un système par l'abstraction de l'objet, par les hiérarchies de classe, et par les interfaces d'appel de méthode. L'approche objet s'intéresse au comportement du système qu'elle modélise par l'interaction des comportements des objets qui le composent. Un objet ne peut interagir qu'avec un objet dont il a une référence, c'est-à-dire un identificateur unique. L'interaction entre objets se fait par invocation de méthode, ce qui entraîne le transfert du contrôle de l'objet demandeur à l'objet qui fournit le service demandé, comme montré sur la figure 3.1 où l'objet O2 est responsable de l'exécution de la requête portée par le message envoyé par l'objet O1.

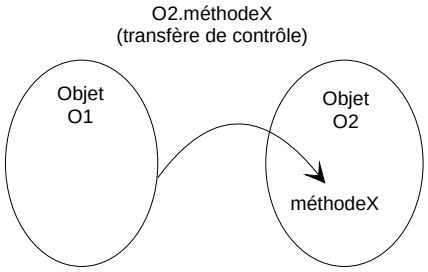


FIG. 3.1 – Transfert de contrôle dans l'appel de méthode

L'approche objet a été adaptée à la conception de systèmes embarqués et au logiciel temps-réel, tant au niveau de la modélisation, grâce à l'évolution du langage UML [25] et à la création de profils spécifiques, qu'au niveau de l'intégration des méthodes propres à l'embarqué dans l'approche objet, en l'occurrence les objets synchrones [26] et les extensions temps-réel de CORBA. De plus, des environnements de logiciel orientés objet, tels que RATIONAL ROSE [24] [21], GME [64], et DOME [40], ont aussi été appliqués dans des conceptions de système de contrôle.

3.3.3 Modélisation et conception middleware

Du fait de la coopération entre objets [80], pour atteindre un objectif commun, la conception orientée middleware préconise l'encapsulation d'un ou plusieurs objets dans des services conceptuels, et l'encapsulation des services composants dans un système. La puissance des services middleware est très significative dans les systèmes répartis, puisque la notion de communication peut y être beaucoup plus adaptée que des appels de procédures à distance utilisés en orientation objet. Ces techniques sont souvent rencontrées dans des applications à grande échelle telles que CORBA [109] et DCOM [121].

En dépit de leurs différences logiques, la structure de base des systèmes orientés objet et middleware sont des objets qui sont reliés entre eux par des références. Leur interface primaire d'interaction est un ensemble d'appels de méthode qui transfère directement le flot de contrôle d'un objet à un autre en cachant les caractéristiques importantes de système, telles que la concurrence et la communication.

Signalons par ailleurs que les deux méthodologies précédentes mettent certes en évidence la notion de décomposition d'un système en composants, mais la tâche de recomposition de ces composants est laissée aux concepteurs.

3.3.4 Modélisation et conception orientées acteurs

Historiquement, la notion de modélisation et de conception orientées acteur tire sa source des travaux de GUL AGHA et autres [1] [2] [3]. Quant au terme acteur, il a été présenté pour la première fois dans les années 1970 par CARL HEWITT de MIT pour décrire le concept des agents intelligents autonomes [58].

Actuellement, la modélisation par acteurs reprend les idées développées par AGHA et HEWITT en ce sens qu'elle décrit le comportement d'un système comme le comportement de composants autonomes dont chacun communique avec un ensemble connu d'autres composants qui lui sont connectés. Par contre, certains aspects comme le fait que chaque acteur dispose de son propre flot de contrôle, ou que les communications entre acteurs sont asynchrones, ne sont pas nécessairement conservés dans la modélisation par acteurs.

Cette méthodologie basée sur les composants est particulièrement efficace pour la modélisation et la conception au niveau système, car, elle décompose un système du point de vue de ses *« actions »* [80].
En effet, la méthodologie orientée acteur préconise la décomposition des systèmes en composants interagissant et la recomposition des composants avec des models de calcul bien définis. Dans ce type de modélisation, les acteurs s'exécutent et communiquent avec d'autres acteurs dans un modèle. Essentiellement, contrairement aux deux méthodologies précédentes, un acteur définit des activités locales sans se référencer implicitement à d'autres acteurs.

De part sa composition, un acteur a une composante interface bien définie. Cette interface différencie son état interne de son comportement, et, définit en plus l'interaction entre cet acteur et son environnement. L'interface d'un acteur inclut les ports qui représentent ses points de communication et les paramètres utilisés pour configurer ses opérations par une entité extérieure telle qu'une interface utilisateur par exemple.
Souvent, les valeurs de paramètre font partie de la configuration initiale d'un acteur et ne changent pas après l'exécution d'un modèle.

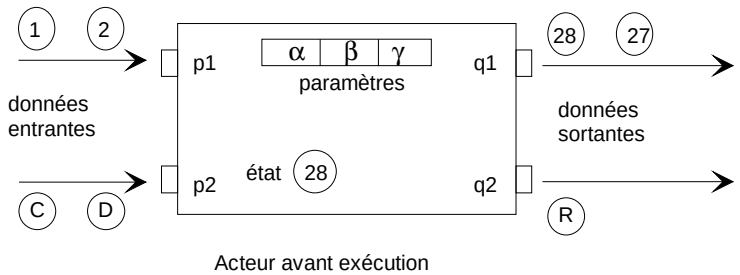


FIG. 3.2 – Paramètres, état et ports d'un acteur avant son exécution

Un acteur peut aussi être observé comme une encapsulation des actions paramétrables effectuées sur des données d'entrée et produisent des données de sortie après son exécution. Des données d'entrée et de sortie sont communiquées par des ports bien définis.

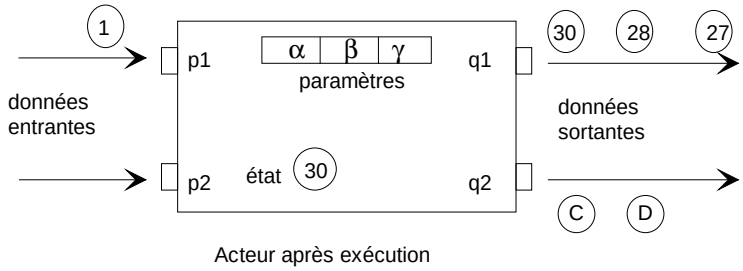


FIG. 3.3 – Paramètres, état et ports d'un acteur après son exécution

Concernant son contrôle, il est fait par le modèle qui l'englobe à travers un ensemble de méthodes généralement identiques à tous les composants. Typiquement, ces opérations exécutent des fonctions comme *« initialisation, » « activation »* et *« arrêt »* du composant.

En ce qui concerne sa communication, elle est définie par un ensemble d'actions, dont chacune traite des données d'entrée pour produire des données de sortie.

Un des principaux éléments dans la conception orientée acteur est le canal de communication où passent des données d'un port à un autre conformément à un schéma de messages. A la différence des méthodes dans la conception orientée objet, un port d'un acteur n'a pas

la sémantique de retour d'appel et son interface doit déclarer les propriétés dynamiques telles que des protocoles de communication et des propriétés temporelles. Le modèle de calcul doit inclure l'information partagée, telle que le temps sous forme de contrainte d'ordre total ou sous forme de contrainte d'ordre partiel [68].

En ce qui concerne le modèle, sa configuration contient des canaux de communication explicites qui véhiculent des données d'un port d'un acteur à celui d'un autre.
La question fondamentale pour les plates-formes orientées acteurs est la définition des modèles d'interaction entre ces acteurs.

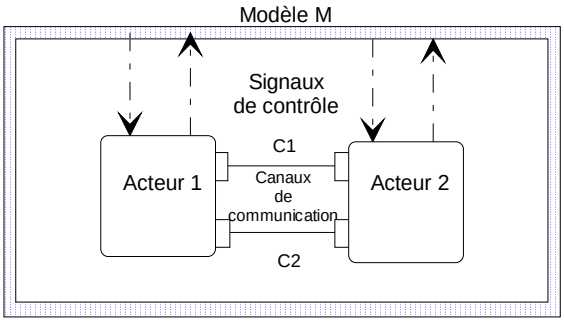


FIG. 3.4 – Acteurs en communication et contrôlés par un model

Pour son exécution, un acteur s'exécute dans le modèle dans lequel il réside. Ainsi, les sous-systèmes et les acteurs inclus dans un modèle définissent un *« système »*.

Comme pour les acteurs, un modèle hiérarchique peut aussi définir une interface externe appelée *« abstraction hiérarchique »* [69] [73] [107] [74] [75]. Elle est composée des ports et des paramètres externes, qui sont distincts des ports et des paramètres des différents acteurs dans le modèle. Les ports externes d'un modèle peuvent être reliés par des canaux à d'autres ports externes d'un autre modèle ou aux ports des acteurs inclus dans le modèle. Des paramètres externes d'un modèle peuvent être utilisés pour déterminer les valeurs des paramètres des acteurs à l'intérieur du modèle.

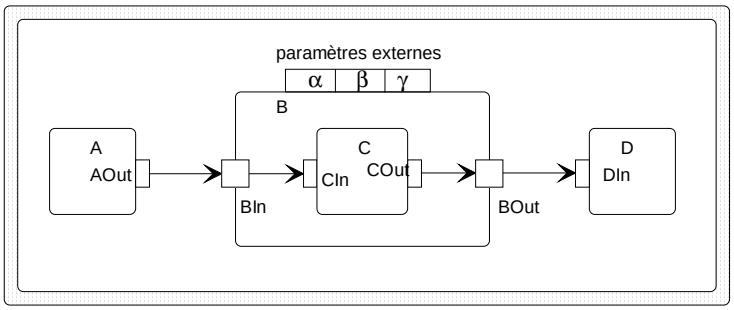


FIG. 3.5 – Abstraction hiérarchique

Sur la figure 3.5, l'abstraction hiérarchique de ce modèle est composé de port BIn et des paramètres externes α, β et γ.
Dans PTOLEMY II par exemple, de part ses ports, le composant B, doit être capable de stocker deux types distincts de receivers[3] dont l'un pour le port BIn et l'autre pour le port BOut.

Au point de vue de son utilisation, cette méthodologie orientée acteur est actuellement très répandue. Aussi, doit-elle sa croissance significative à la disponibilité d'outils qui l'utilisent, nous citons entre autre SIMULINK, COCENTRIC SYSTEM STUDIO, LABVIEW, SPW etc. ...

Par ailleurs, dans la communauté académique, certaines équipes de recherche manipulent les objets et les acteurs actifs. D'autres travaillent sur des objets basés sur des ports [125], et d'autres encore sur des automates d'entrées-sorties hybrides [92].

Quant aux plates-formes de modélisation utilisant l'orientation acteur, nous citons entre autre MOSES (Modeling, Simulation, and Evaluation of Systems) [45], POLIS [10], et PTOLEMY II [13] [14] [15].
Actuellement, parmi ces différentes plates-formes utilisant la modélisation orientée acteur, PTOLEMY II est l'une des rares plates-formes à utiliser une approche unifiée en offrant une large gamme des modèles de calcul.

3.3.5 Choix de la méthodologie

De ces trois méthodologies orientées composants exposées ci-dessus, dans le cadre de cette thèse, la méthodologie orientée acteur répond aux attentes de notre modélisations hétérogène non-hiérarchique pour des raisons de structuration, de séparation des préoccupations et de composabilité et décomposabilité modulaire que nous présentons ci-dessous :

- Sur le plan de la structuration du système, l'orientation acteur est une manière efficace de structurer et de conceptualiser un système. Elle considère un système comme une structure de sous-systèmes. Ceci met en évidence aussi bien la structure causale que les dépendances de communication entre les sous-systèmes. Ce qui nous permet d'établir efficacement la séparation entre le flot de contrôle et le flot des données.

- Sur le plan de la séparation des préoccupations, cette méthodologie complète les techniques orientées objet en y rajoutant une vision de découplage entre la transmission de données et le transfert de contrôle. Elle utilise donc la séparation des préoccupations [62] [42] qui est une condition importante dans la réutilisabilité des composants et dans la flexibilité du système. Elle est également fondamentale dans l'approche non-hiérarchie, car, nous verrons plus loin que chacune de ces préoccupations sera prise

[3]Objet encapsulé dans un port. Il implémente le protocole de communication d'un modèle de calcul. Il est explique en détail au chapitre 8.

en charge par un composant dédié

– Sur le plan de la décomposabilité et de la composabilité modulaire, comme énoncé plus haut, cette méthodologie préconise la décomposition des systèmes en composants interagissant et la recomposition des composants avec des models de calcul bien définis. Nous verrons plus loin que cet aspect est vital pour notre approche non-hiérarchique du fait que le système sera divisé aux frontières des modèles de calcul adjacents pour créer des sous-systèmes homogènes. Ce qui assurera la modularité des modèles et la réutilisabilité des composants.

Après ce choix de la méthodologie à utiliser, nous allons dans la sous-section suivante présenter la structure d'un acteur ainsi que celle d'un modèle hétérogène élémentaire selon l'approche présentée dans [81]. Toutefois, au lieu du terme conteneur, nous utiliserons les termes modèle ou sous-système selon le cas, afin de rester le plus proche possible des termes que nous allons utiliser plus loin dans cette étude.

3.3.6 Structures d'un acteur et d'un modèle

A la section 3.3.4, nous avons vu qu'un acteur pouvait être vu comme une entité contrôlée par un modèle, et qui, de ses ports d'entrée, consomme une séquence de données et produit une autre séquence de données à partir de ses ports de sortie après son exécution. Cette production de données à travers les ports de l'acteur est vue comme une fonction de l'état et des paramètres courrant de l'acteur et de ses séquences de données entrantes. Son état est donc variable et on parle plutôt du calcul des états successifs de l'acteur.

En considérant l'acteur montré sur la figure 3.6(a), celui-ci dispose d'un ensemble des variables noté A.X. Cet ensemble peut être subdivisé en deux sous-ensembles à savoir : le sous-ensemble des variables d'interface appelées *ports* et le sous-ensemble des variables internes noté S et contenant ses paramètres et ses états.

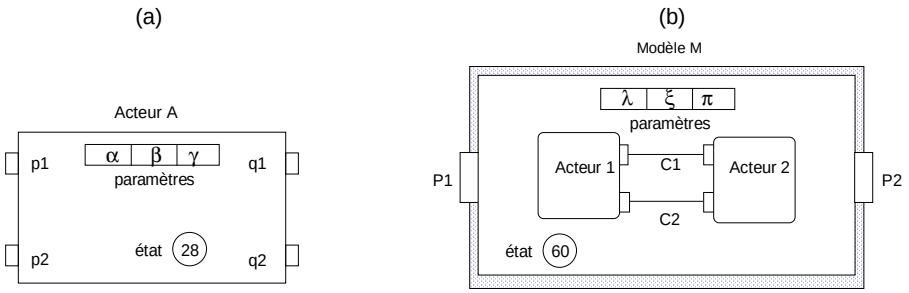


FIG. 3.6 – Acteur A et modèle M et leurs variables

Les variables d'interface sont de deux types : les ports d'entrée rassemblés dans le sous-ensemble P et ceux de sortie rassemblés dans le sous-ensemble Q. Dans la section 3.3.4 nous avons vu qu'un modèle hiérarchique peut aussi définir une interface. De la même manière,

sur la figure 3.6(b), le modèle M dispose d'un ensemble de variables que l'on note M.X. Cet ensemble contient trois sous-ensembles à savoir, le sous-ensemble noté M.S contenant les variables internes (acteurs et canaux de communication) et les deux sous-ensembles P et Q contenant les variables d'interfaces (ports d'entrée et ports de sortie) du modèle M.

Définitions 3.3.1 *Nous référant au [81], en considérant un acteur A, soit A.X l'ensemble de variables de A et soit V l'ensemble des valeurs que peuvent prendre ces variables,*
On appelle fonction d'évaluation des variables de A, une fonction f qui pour toute variable $x \in A.X$ *fait correspondre sa valeur dans V telle que* $x \rightarrow f(x) = v$.
On appelle état d'un acteur, le sous-ensemble des valeurs v telles qu'il existe x dans A.X pour lequel $v = f(x)$.

Variables du modèle

La description de ce modèle se basant sur les notations présentées dans [81], utilise un ensemble de variables M.X. Cet ensemble est composé du sous-ensemble des variables internes appelé M.S et des deux sous-ensembles des variables d'interfaces ; le premier nommé M.P contient les ports d'entrées et le second nommé M.Q contient les ports de sortie. Il s'ensuit que :

$$\text{M.X} = \{\{\text{M.P, M.Q}\} \bigcup \{\text{M.S}\}\}$$

Où

$$\text{M.S} = \{\{\bigcup \text{A}_i\} \bigcup \{\bigcup \text{c}_i\}\}$$

Avec $\bigcup \text{A}_i$ et $\bigcup \text{c}_i$ respectivement union des acteurs et union des canaux du modèle.

Opérations dans le modèle

On distingue deux types d'opérations :

- *Les opérations qui capturent le flot de données.*
  Elles traitent de la manière de calculer les nouvelles données à partir des anciennes et de la manière de les envoyer. Ces opérations ont la caractéristique de changer les évaluations des variables. Elles vont répondre à la question *« comment recevoir, comment calculer et comment envoyer les données »*

- *Les opérations qui capturent le flot de contrôle.* Celles-ci ne changent pas directement les valeur des variables, mais par contre s'occupent du contrôle des opérations entre elles. Elles vont plutôt déterminer à quel moment le calcul et la communication devront se produire, c'est-à-dire, elles vont répondre à la question *« quand le calcul et la communication devront se produire »*.

Dans les systèmes que nous présentons, un acteur n'interagira avec son modèles de calcul que par des flot de contrôle. En effet, le modèle M contrôle les activités des composants sous sa responsabilité directe par un ensemble d'opérations de contrôle défini comme union de leurs opérations de contrôle :

$$\mathrm{M.Ctrl} = \bigcup \mathrm{A}_i.\mathrm{Ctrl}$$

avec α.Ctrl défini comme opérations de contrôle pour l'acteur $\alpha \in$ M.

Parmi les opérations de contrôle d'un acteur, il en existe deux particulières, qui sont des *points de synchronisation* de l'ensemble de ses opérations. Ces opérations sont :

– *trigger()*, l'opération qui consiste à déclencher les activités d'un composant et,
– *finish_trigger()*, l'opération utilisée par un composant pour indiquer qu'il n'a plus d'opérations à exécuter pour un trigger.

En notant $\prec$, la relation de précédence entre les opérations, $\forall\ \xi$ appartenant à l'ensemble d'opérations d'un acteur, on aura toujours

$$trigger() \prec \xi \prec finish\_trigger()$$

Et, pendant l'exécution du modèle, il y aura toujours au moins un déclenchement parmi les opérations suivantes :

$$\mathrm{A}_i.trigger \subset \mathrm{A}_i.\mathrm{Ctrl}$$

Hormis les activités du déclenchement, pour le contrôle des acteurs A_i, le modèle M fournira également un ensemble d'opérations de rappel, appelées aussi opérations de call-back, défini comme union des opérations de rappel de ses composants que nous notons :

$$\mathrm{M.Clbk} = \bigcup \mathrm{A}_i.\mathrm{Clbk}$$

avec α.Clbk défini comme opérations de call-back pour l'acteur $\alpha \in$ M.

Ainsi, nous dressons le tableau suivant des différents flots et de leurs tâches associées :

Flot de données		Flot de contrôle
CALCUL	COMMUNICATION	Quand calculer et quand communiquer
Comment calculer les données (Comportement)	Comment réceptionner et comment émettre les données	

TAB. 3.1 – Différents flots et leurs tâches associées

A ce niveau, nous disposons de la méthodologie à utiliser et nous connaissons la structure des acteurs et celle des modèles. Puisque, notre modèle hétérogène doit être exécutable, il doit par conséquent être construit sous un modèle de calcul. C'est pourquoi, nous allons dans la section suivante présenter le modèle de calcul hétérogène et les primitives abstraites que nous avons choisies.

3.4 Primitives abstraites

Les interactions entre nos différents acteurs hétérogènes seront capturés par un modèle de calcul hétérogène. En effet, de manière générale, ce modèle de calcul définira la sémantique de flot de contrôle sur les acteurs et la sémantique de communication entre leurs ports.
En adoptant l'abstraction présentée dans [115], ce modèle de calcul sera donc un algorithme d'ordonnancement de flot de contrôle à travers l'ensemble de nos acteurs[4] et l'ensemble de canaux de communication.

Comme nous l'avons souligné à la section 2.2.4, la spécification de ce modèle de calcul nécessite une sémantique qui va régir les interactions entre les différents composants hétérogènes dans le modèle.
C'est ainsi que dans la suite, nous spécifions cette sémantique dans le sens de [115].
En effet, une réaction dans un acteur aura un temps fini, et, un modèle sera responsable, c'est-à-dire, un modèle ne lancera un acteur que lorsque ce dernier est prêt à l'être.

De plus, au paragraphe 3.3.4, nous avons précisé que dans l'approche orientée acteur, un acteur définissait des activités locales sans se référencer implicitement à d'autres acteurs. Ainsi, ce modèle opérationnel basé sur les primitives abstraites fait la distinction nette entre les activités des acteurs et celles du modèle d'exécution en ce qui concerne la communication, le comportement et le contrôle.

Cet ensemble de primitives ne peut pas être défini comme une union des sémantiques, mais plutôt comme leur intersection. Il est abstrait dans le sens qu'il représente les caractéristiques communes des modèles de calcul par opposition à l'union de leurs caractéristiques.

Cet ensemble de primitives est donc composé du flot de contrôle abstrait et de la communication abstraite. Et, l'acteur qui reçoit le contrôle et qui communique dans un modèle sera considéré comme un composant abstrait dans les sens de [105] :

- Il a une interface bien définie contenant les ports et les paramètres, par lesquels il peut interagir avec le reste du modèle. Les autres interactions ne lui sont pas permises.
- Il est généralisable à travers différentes utilisations du même composant. Par exemple, il peut être relié à différents ensembles de canaux de communication implémentant chacun

[4] A ce niveau, nous parlons des acteurs, mais, plus loin nous verrons que cette ordonnancement sera plutôt appliqué aux sous-systèmes

différents modèles de calcul.
– Il peut avoir différents niveaux du granularité allant du module à un processus.
– Des composants plus complexes peuvent être construis par le mécanisme primaire d'assemblage de composants élémentaires.
– Il est exécutable de par la spécification de son état interne et de son comportement.

3.4.1 Primitives abstraites de flot de contrôle

Dans cet ensemble de primitives proposé dans cette étude, les acteurs s'exécutent en trois phases ; *initialisation*, *itération*, et *finish*.
Une itération étant définie comme une séquence d'opérations qui lit des données d'entrée, produit des données de sortie, et met à jour l'état de l'acteur.

Les opérations d'une itération se composent exactement d'une invocation d'une preCondition, suivie de zéro ou plusieurs invocations démarrage ou trigger, et de zéro ou une invocations de la postCondition. La sémantique de flot de contrôle sera :

initialisation	:	initialise l'acteur avant sa première exécution,
preCondition	:	consulte les entrées d'un acteur et détermine si l'acteur doit être
	:	activé,
trigger	:	lit les entrées d'un acteur, si nécessaire et produit des sorties après
	:	avoir effectué le calcul du comportement,
postCondition	:	met à jour l'état de l'acteur,
finish	:	fin de son exécution.

Ces opérations sont abstraites, car aucune hypothèse n'est faite sur leur implémentation. Elles seront définies explicitement en association avec le modèle de calcul qui vont les utiliser.

3.4.2 Primitives abstraites de communication

Les primitives abstraites présentées dans [69] fournissent un ensemble d'opérations de base pour la communication. Elles permettent à un acteur d'interroger l'état de canaux de communication, et ensuite rechercher ou envoyer l'information à ces canaux. L'ensemble de primitives abstraites de communication sera :

Read	:	recherche de données par l'intermédiaire d'un port,
Write	:	production de données par l'intermédiaire d'un port,
existData	:	teste si l'opération Read peut être effectuée avec succès sur un port,
isFull	:	teste si l'opération Write peut être effectuée avec succès sur un port.

Ces opérations sont abstraites, dans le sens que les mécanismes de canaux de communication ne sont pas définis explicitement. Elles seront déterminées par le modèle de calcul qui les utilisent.

3.5 Conclusion partielle

Dans ce chapitre nous avons présenté les différents outils de modélisation à savoir les langages et plates-formes de modélisation. Chacun de ces outils étant développé et utilisé pour des besoins spécifiques. Nous avons montré que parmi ces plates-formes de modélisation, il en existe certains qui ont un caractère unifié. Et, nous avons écarté les éventuels problèmes d'intégration qui pourrait provenir de l'hétérogénéité en phase de validation de nos concepts en optant pour une plate-forme unifiée, en l'occurrence PTOLEMY II.

Par ailleurs, puisque nous avons choisi une plate-forme utilisant une méthodologie de modélisation orientée composant, il nous a semblé judicieux de faire une présentation de ces types de méthodologie et justifier le choix de la méthodologie orientée acteur que nous avons adoptée pour la réalisation de cette étude.
Lors de la modélisation, à un niveau d'abstraction élevé, nos composants utiliseront un ensemble de primitives abstraites. C'est pourquoi, une présentation de cette dernière a également été faite.

Maintenant que l'environnement de modélisation est présenté, la méthodologie et les primitives à utiliser dans cette étude sont connues, dans le chapitre suivant nous abordons notre approche non-hiérarchique préconisée comme une alternative à l'approche hiérarchique.

Chapitre 4

Approche théorique de l'Hétérogénéité Non-Hiérarchique

Résumé -Dans ce chapitre, nous présentons l'approche théorique de la modélisation hétérogène non-hiérarchique. C'est ici que nous introduisons les deux composants d'appui à la modélisation et à la conception utilisant notre approche non-hiérarchique : le Composant à Interface Hétérogène qui sera à la frontière du comportement hétérogène et le Modèle d'Exécution Hétérogène Non-Hiérarchique dont la tâche sera de restructurer le système en sous-systèmes, d'activer des HICs, d'ordonnancer et d'exécuter les sous-systèmes.

4.1 Introduction

Actuellement, les outils de modélisation utilisés pour la modélisation des systèmes hétérogènes utilisent une approche hiérarchique en imposant un changement de niveau hiérarchique lorsque l'on passe d'un modèle de calcul à un autre [124] [129] [34] [116] [113] [29] [13] [14] [15] [42] [65] [82].
Mais, la question fondamentale ne repose pas sur cette hiérarchie en soi, c'est plutôt l'impact du couplage de cette hiérarchie avec les changements de domaine sur la conception et la maintenance des applications.

Cependant, comme nous l'avons souligné dans l'introduction de ce document, nous considérons que cette hiérarchie obligatoire découle des limites accusées par les approches techniques utilisées par les outils actuels de modélisation. Par contre, sur le plan de la composabilité, même si quelquefois, cette hiérarchie peut suivre la structure d'un système donné, néanmoins, elle altère la modularité et la réutilisabilité des composants.
De ce fait, de notre point de vue, nous pensons que la hiérarchie dans un modèle

hétérogène ne devrait pas dépendre des outils de modélisation. Elle devrait plutôt représenter la structure compositionnelle d'un système suivant sa décomposabilité fonctionnelle.

Par ailleurs, à la section 2.4, nous avons souligné que l'hétérogénéité dont nous faisons référence dans cette thèse est au niveau de la modélisation et non au niveau de la conception. Ainsi, nous ne traiterons pas l'hétérogénéité issue du partitionnement dans le sens de [134] dont le but est de découper la fonctionnalité d'un système en un ensemble de partitions où chaque partition peut être exécutée, soit en logiciel, soit en matériel.
De plus, la modélisation hétérogène non-hiérarchique que nous présentons ne signifie nullement la modélisation hétérogène sans hiérarchie, elle signifie que la hiérarchie est utilisée pour refléter la structure du modèle sans être perturbée par des outils introduits par des changements de modèle de calcul.

Ainsi, dans ce chapitre, nous proposons une alternative préconisant une approche non-hiérarchique qui découple les changements des MoCs de la hiérarchie. Ce découplage est induit de l'utilisation des composants disposant d'entrées et de sorties de natures différentes et d'un modèle d'exécution hétérogène. Ces composants impliquent la création des sous-systèmes à la frontière des comportements hétérogènes du système et au même niveau hiérarchique. La communication entre ces sous-systèmes est assurée grâce à une *« abstraction non-hiérarchique »* reposant sur les canaux abstraits de communication.

En effet, cette approche permet l'utilisation des composants obéissant à différents MoCs au même niveau hiérarchique, impliquant qu'un modèle puisse contenir des acteurs qui communiquent suivant différentes sémantiques. Pour cela, nous proposons un composant qui peut avoir des entrées et des sorties de différentes natures que nous appelons *« Composant à interface l'hétérogène, Heterogeneous Interface Component, HIC ».*
Cependant, puisque la sémantique d'un modèle de calcul est imposée par un modèle d'exécution [80] [81], un modèle hétérogène non-hiérarchique contiendra plusieurs modèles d'exécution réguliers, chacun étant responsable des acteurs appartenant au sous-système qu'il gouverne. Ainsi, un composant à interface hétérogène sera manipulé par plusieurs modèles d'exécution réguliers car il a différents types d'entrées ou de sorties.
Or, d'une part, le système doit être restructuré en sous-systèmes et d'autre part ces sous-systèmes doivent être gouvernés, chacun par un modèle d'exécution régulier qui eux-même doivent être coordonnés. Pour assurer ces tâches, nous proposons un composant que nous nommons *« Modèle d'Exécution Hétérogène Non-Hiérarchique ».*
Il s'ensuit donc qu'en vertu du principe de la séparation entre le flot de contrôle et le flot de données, nous avons dédié au HIC les tâches liées au flot de données, c'est-à-dire, la communication et le comportement et avons dédié au modèle d'exécution hétérogène les tâches liées au flot de contrôle, c'est-à-dire, les activations des différents sous-systèmes.

Cette approche présente d'énormes avantages tels que, sa capacité à utiliser des composants disposant d'entrées et de sorties hétérogènes et à spécifier ce qui se produit à la frontière des deux modèles de calcul, i.e, lorsque les données passent d'un sous-système à un autre.

4.2 Spécificités de l'approche non-hiérarchique

4.2.1 Composants d'appui à l'hétérogénéité non-hiérarchique

Dans [13], les auteurs spécifient que la modélisation a pour but de réaliser le partitionnement et la représentation formelle d'un système ou d'un concept à haut niveau en plusieurs sous-systèmes. Ainsi, pour représenter et définir ce système dans l'approche hétérogène non-hiérarchique, un certain réaménagement s'impose.
Il va donc nous falloir une intelligence supplémentaire pour le partitionnement du système, la coordination et la gestion de la fluidité des données et toutes les autres tâches y relatives sans aucune dépendance aux outils de modélisation. Nous avons partagé cette intelligence supplémentaire entre deux composants d'appui à l'hétérogénéité non-hiérarchique.

En effet, nous avons tiré l'avantage de l'orientation acteur dans la séparation entre le contrôle et la communication et le comportement en dédiant la communication et le comportement hétérogènes à un premier composant qui assure la gestion de flot de données dont la communication et le calcul du comportement hétérogènes.
Le contrôle hétérogène du système est dédié à un deuxième composant qui assure la gestion de flot de contrôle, c'est-à-dire, les activations des différents composants et leur possibilité, au besoin, de réaliser un rappel [98].

Ainsi, le composant ayant en charge la communication et le calcul du comportement hétérogènes doit répondre à la spécification suivante :
– Sur le plan de la communication hétérogène, il doit disposer d'entrées et de sorties hétérogènes lui permettant de mettre en communication deux ou plusieurs composants ayant des ports utilisant des modèles de calcul différents.
– Sur le plan du comportement hétérogène, il doit donc être capable de fournir un comportement hétérogène aux frontières des différents modèles de calcul utilisés par le système.
Cet ensemble de critères a donné lieu à la spécification de notre premier composant que nous avons nommé *« Composant à Interface Hétérogène ou (Heterogeneous Interface Component, HIC) »*.

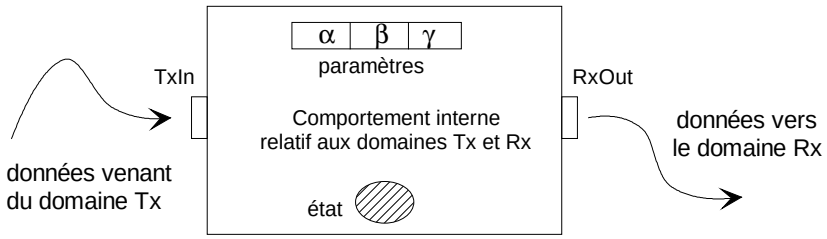


FIG. 4.1 – Composant à Interface Hétérogène

Illustration 4.2.1 Le Composant à Interface Hétérogène dispose de variables internes dont un état, des paramètres α, β et γ et des variables d'interface TxIn et RxOut qui utilisent les modèles de calcul Tx et Rx. Il a également un comportement hétérogène à la frontière des modèles de calcul Tx et Rx.

De plus, puisque le modèle doit être exécutable, il doit disposer d'un composant qui gouvernera cette exécution hétérogène. Ce dernier va gérer le flot de contrôle des interactions à travers les différents sous-systèmes. Comme pour le HIC, ce composant doit répondre à la spécification suivante :

– Sur le plan de la décomposabilité et de la recomposabilité modulaire du système, il doit être capable de diviser le système à la frontière des différents modèles de calcul utilisés par le HIC, créer ainsi des sous-systèmes et déléguer le flot de contrôle local et le calcul du comportement interne du sous-système au modèle d'exécution régulier.
– Sur le plan de l'exécution proprement dite, il doit donc être capable de générer un ordonnancement approprié des différents sous-systèmes crées et de les activer. Pendant cette activation, il doit aussi pouvoir manipuler le caractère hétérogène du HIC.

Comme pour le Composant à Interface Hétérogène, cet ensemble de critères a donné lieu à la spécification de notre deuxième composant que nous avons appelé *« Modèle d'Exécution Hétérogène Non-Hiérarchique »*.

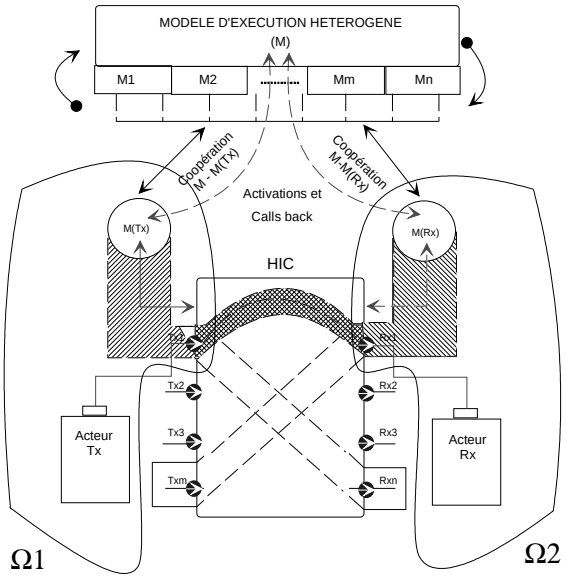


FIG. 4.2 – Modèle d'Exécution Hétérogène

Illustration 4.2.2 Le Modèle d'Exécution Hétérogène Non-Hiérarchique (M) divise le système en créant les sous-systèmes, Ω_1 et Ω_2 dans le cas du modèle hétérogène élémentaire. Il délègue le flot de contrôle local et le calcul du comportement interne de Ω_1 et Ω_2 à leurs modèles d'exécution réguliers [M(Tx)) et (M(Rx)] respectivement. Il maintient également la chaîne de communication entre les acteurs Tx et Rx en coopération avec leurs modèles d'exécution réguliers. Quant au HIC, il est à la fois géré par le Modèle d'Exécution Hétérogène Non-Hiérarchique et par les modèles d'exécution des différents sous-systèmes entre lesquels il est partagé

4.2.2 Composant à Interface Hétérogène (HIC)

Concept

Dans la modélisation hétérogène non-hiérarchique, un niveau donné dans la hiérarchie du modèle peut contenir des composants qui utilisent différents modèles de calcul. Pour cela, deux possibilités se présentent [27] :

Soit chaque composant n'obéit qu'à un seul modèle de calcul, mais, nous pouvons établir des connexions entre les ports des domaines différents comme sur la figure 4.3.
Nous ne pouvons utiliser cette première possibilité parce que le changement de la sémantique entre les domaines se produit le long des connexions entre les ports alors que les connexions ne sont pas généralement considérées comme entités actives dans les environnements de modélisation.
En effet cette option implique que les connexions soient dotées de mécanismes de conversion de protocoles de communication et de transformation de format de données à transmettre. Ceci implique par exemple que les connexions soient capables de convertir le protocole de passage de message au protocole rendez-vous ou qu'ils soient également capables de transformer les données d'un format à un autre.

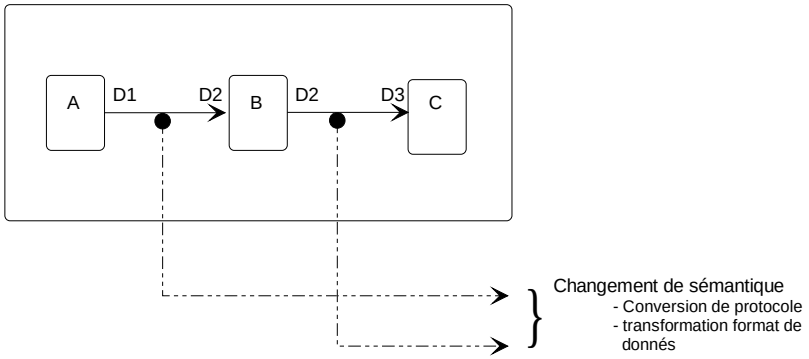


FIG. 4.3 – Exemple où les composants n'obéissent qu'à un seul MoC

Soit les connections ne sont autorisées que entre les ports appartenant au même domaine, mais, un composant peut obéir à plusieurs modèles de calcul comme sur la figure 4.4.

Quant à la deuxième possibilité, elle préserve la sémantique des données le long de la connexion entre les ports et a l'avantage de supporter les composants qui ont des ports qui obéissent à différents modèles de calcul. Le changement de la sémantique entre les domaines se produit à l'intérieur de ces composants comme un élément de leur comportement et est donc une partie explicite du modèle du système.

Nous appelons de tels composants les « composants à interface à hétérogène ». A l'instar de l'exemple du convertisseur analogique/numérique donné à l'introduction de ce document,

les HICs apparaissent naturellement dans les modèles. Toutefois, ils soulèvent la question de les faire obéir à plusieurs modèle de calcul de manière cohérente.

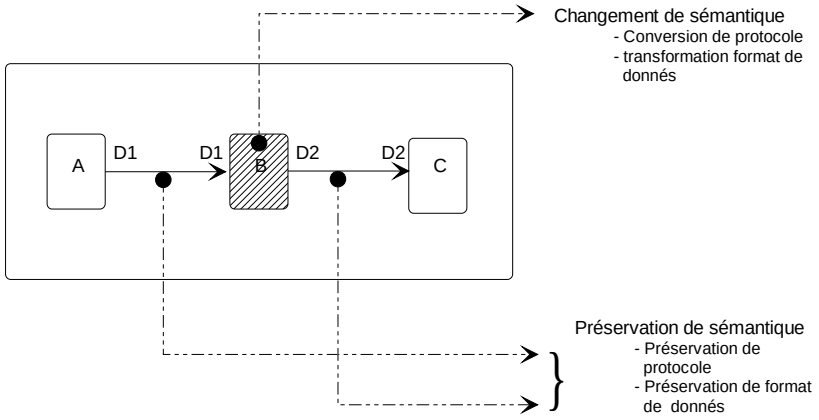


FIG. 4.4 – Exemple où les composants peuvent obéir à plusieurs MoCs

Nous considérons les HICs dans le contexte de la modélisation hétérogène générique. Quand les modèles de calcul disponibles sont connus et en nombre limité, il est plus efficace d'employer leur union pour soutenir les HICs. Cependant, le problème que nous adressons est plutôt l'utilisation des HICs sans connaître d'avance ni les modèles de calcul ni leur nombre.

Cependant, en modélisant un système complet dans un environnement à un haut niveau, le nombre de modèle de calcul impliqués augmentent. Même si chaque système n'emploie que très peu de modèles de calcul, le nombre de combinaisons possibles d'un nombre restreint de modèle de calcul choisis parmi tous les modèle de calcul possibles est trop grand pour utiliser l'approche d'union des modèles.

Comportement de HIC

Un HIC a donc les ports qui obéissent à différents modèle de calcul, donc, son comportement peut être décomposé en autant de comportements secondaire qu'il y a de modèle de calcul, et ces comportements secondaire sont couplés.
Il s'ensuit que le comportement du HIC selon un modèle de calcul puisse influencer son comportement suivant un autre modèle de calcul.

Chaque comportement secondaire d'un HIC est un « pont » entre un modèle de calcul et le comportement global du HIC.
En effet, lorsqu'un HIC interprète une entrée, il traduit la signification de cette entrée dans le modèle de calcul associé en signification interne pour le HIC. Quand un HIC produit une sortie, il traduit l'information recueillie de ses entrées dans la sémantique de cette sortie selon le modèle de calcul.

Par conséquent, la spécification du comportement d'un HIC mène à établir une représentation interne de la sémantique des entrées selon leurs modèles de calcul respectifs, et à traduire cette représentation interne dans des sorties.

Les HICs permettent de spécifier explicitement comment des données d'un domaine sont interprétées, et comment cette interprétation est employée pour produire des données pour un autre domaine. Ils ne sont pas seulement limités à la connexion des deux domaines . Ainsi, un HIC peut utiliser autant de domaines que nécessaires.

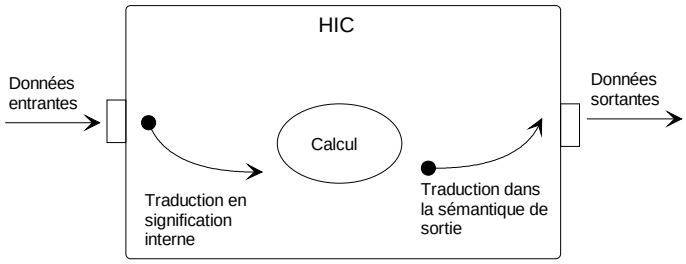


FIG. 4.5 – Synoptique du comportement de HIC

Exemple

Considérons l'exemple du redresseur de signal donné au paragraphe 2.5.1, pour illustrer la différence entre les modèles hétérogènes et hiérarchiques et non-hiérarchiques.
Le système, sur la figure 4.6, montre la représentation du même exemple de la hiérarchie sur la figure 2.21. Il reçoit le même signal d'entrée et a le même multiplicateur. Le détecteur a une entrée qui reçoit un flot d' échantillons de données et une sortie qui produit un événement discret chaque fois l'entrée change son signe. Le multiplicateur a une entrée et une sortie qui fournissent des flots d'échantillons de données et d'une entrée qui reçoit des événements discrets.
Le flot d'échantillons est interprété en spécification interne du détecteur et est traduit ensuite dans la sémantique de sortie.

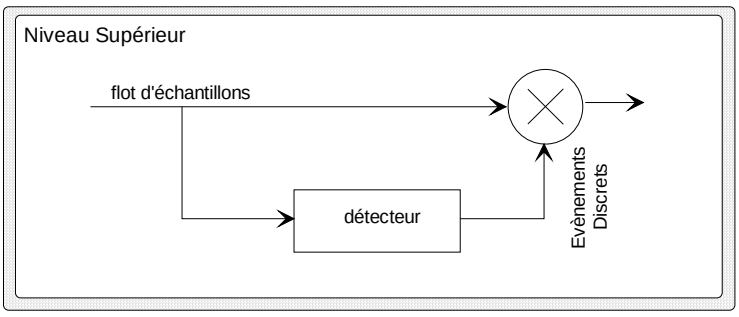


FIG. 4.6 – Exemple d'un modèle hétérogène non-hiérarchique

4.2.3 Modèle d'exécution hétérogène non-hiérarchique

Concernant les aspects pratiques, nous devons décrire comment le comportement hétérogène d'un HIC est spécifié, et comment il est interprété pour calculer le comportement du modèle d'un système.
En effet, notre but principal est de pouvoir utiliser les HICs avec les modèles de calcul existants. Cependant, ces modèles de calcul ne supportent pas les HICs, ou s'ils le peuvent les font d'une manière vraiment générique.

Notre approche est de transformer le modèle hétérogène plat original en ajoutant un niveau à sa hiérarchie et de construire des sous-ensembles homogènes au prochain niveau. La couche supérieure est contrôlée par un nouveau modèle de calcul « hétérogène » qui ordonnance les sous-ensembles homogènes et délègue le calcul de leurs comportements à leurs domaines respectifs.
Ce modèle d'exécution emploie la hiérarchie pendant la simulation pour isoler des modèles de calcul, mais ceci n'a aucun impact sur la structure du modèle du système et permet de concevoir des composants sans considérer le contexte dans lequel ils seront employés. Pour chaque domaine utilisé par un HIC, nous établissons un composant qui représente le HIC dans ce domaine. Nous appelons ce composant la *« projection »* du HIC sur le domaine.

Du point de vue du domaine, ce composant est un composant régulier, avec les ports qui obéissent à la sémantique du domaine, les autres ports étant masquées pendant le processus de projection. Cependant, la projection d'un HIC dans un domaine peut communiquer avec d'autres projections du même HIC dans d'autres domaines.
La projection des HICs sur leurs domaines, la communication entre les projections d'un HIC, et l'établissement de l'ordonnancement des sous-ensembles homogènes sont contrôlés par le domaine hétérogène qui implémente un modèle d'exécution pour les HICs.

La première étape dans l'exécution d'un modèle hétérogène plat est de projeter le HICs sur chacun des domaines qu'ils utilisent et le comportement de chaque projection est calculé par le domaine correspondant. Les composants projetés doivent se comporter exactement comme les autres composants réguliers appartenant au domaine dans lequel ils sont projetés, ainsi les ports des HIC qui obéissent à d'autres modèles de calcul sont cachés pendant la projection.

Une fois que chaque HIC a été projetée sur chacun des domaines qu'il utilise, les composants résultants sont groupés dans des sous-ensembles homogènes qui contiennent seulement les composants réguliers appartenant à un domaine. Le comportement de chaque sous-ensemble peut donc être calculé en appliquant les règles d'un domaine régulier.
Notre domaine hétérogène doit ordonnancer les sous-ensembles afin de calculer le comportement du système entier à partir du comportement des sous-ensembles homogènes. Ceci est illustré pour un système très simple sur la figure 4.7.

Le système contient un composant A qui a une sortie qui obéit au modèle de calcul D1, un autre composant B qui a une entrée qui obéit au modèle de calcul D2, et ces composants sont interconnectés à travers un HIC H.

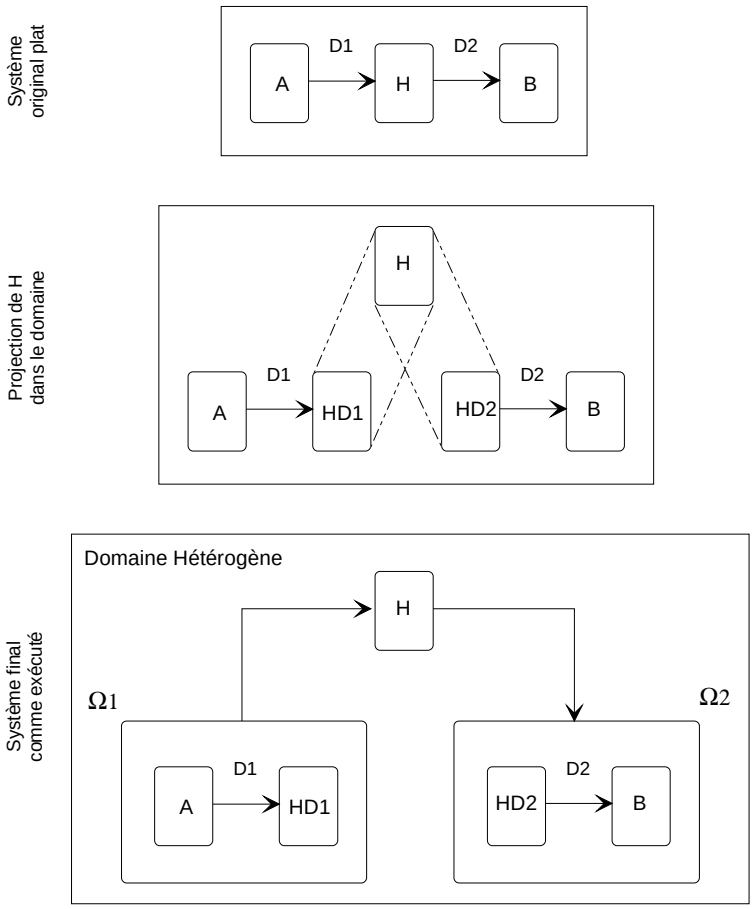


FIG. 4.7 – Transformations d'un modèle hétérogène non-hiérarchique

Puisque H utilise deux modèles de calcul, il a donc deux projections HD1 et HD2. HD1 a seulement une entrée parce que la projection sur D1 cache la sortie de H du fait qu'elle obéit à un autre modèle de calcul. De même, HD2 a seulement une sortie parce que la projection sur D2 cache l'entrée de H qui obéit à D1. Les composants sont groupés dans les sous-ensembles homogènes Ωi qui peuvent être ordonnancés par le domaine hétérogène.

Dans notre exemple, nous avons un sous-ensemble pour chaque domaine utilisé, et les connexions de $\Omega 1$ à H et de H à $\Omega 2$ codent la dépendance du comportement de HD2 sur le comportement de HD1. À l'intérieur de $\Omega 1$ et de $\Omega 2$ l'établissement de l'ordonnancement des composants est fait selon le domaine local, mais l'établissement de l'ordonnancement de $\Omega 1$ et de $\Omega 2$ est fait par le domaine hétérogène selon la topologie du système.

4.3 Mise à plat du modèle hétérogène par intégration du HIC

4.3.1 Mise à plat

Considérons l'abstraction du modèle élémentaire sur figure 2.19 représentant une communication hétérogène entre deux acteurs Tx et Rx à travers un canal C1. Puisque le modèle d'exécution ne peut pas manipuler un canal hétérogène, nous allons l'éclater par interfaçage d'un HIC entre les acteurs hétérogènes Tx et Rx.

Nous savons qu'un HIC interfacant deux acteurs hétérogènes éclate le canal de communication hétérogène les connectant en deux canaux de communication homogènes.
Ainsi, après éclatement, le système représenté par la figure 2.19 est transformé en celle représentée par la figure 4.8 où le canal hétérogène C1 est éclaté en deux canaux homogènes C1-1 et C1-2.

En effet, l'acteur Tx appartient au domaine tx et l'acteur Rx appartient au domaine rx. Tous les deux ont des ports qui ne communiquent qu'à travers la sémantique de leurs domaines respectifs à travers le HIC qui met les données produit par l'acteur Tx compatibles au domaine de l'acteur Rx.

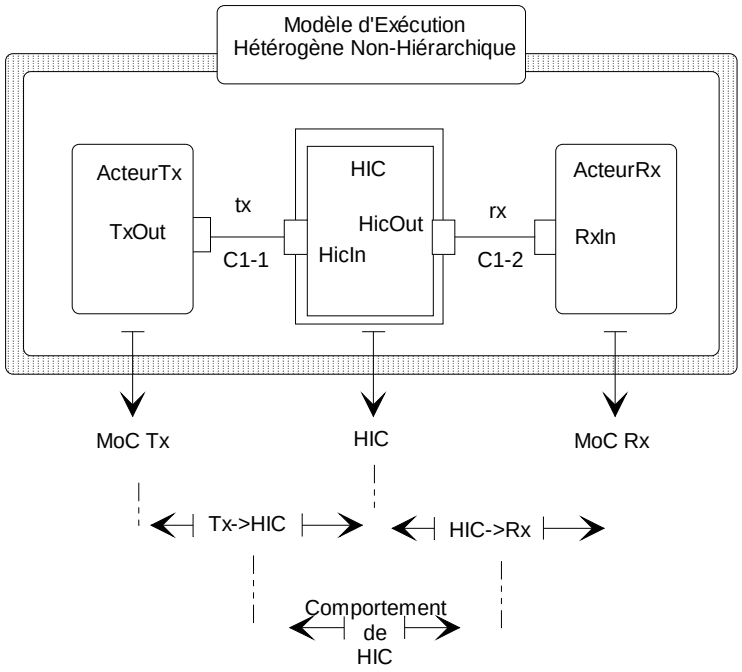


FIG. 4.8 – HIC interfacant deux acteurs hétérogènes

4.3.2 Division du système par le Modèle d'Exécution Hétérogène

L'acteur HIC ayant un comportement à la limite des deux domaines communicants se doit donc d'appartenir simultanément à chacun d'eux.
Pour cela, le modèle d'exécution hétérogène va diviser le système à la frontière des deux modèles de calcul et créer deux sous-systèmes que nous notons Ω_1 et Ω_2 gouvernés par leurs modèles de calcul respectifs. D'où la transformation du schéma de la figure 4.8 en celui de la figure 4.9.

En effet, la sémantique du domaine tx est donnée par le modèle d'exécution MTx qui le contrôle et celle de domaine rx est fournie par le modèle d'exécution MRx qui le contrôle également. Ceci permet de ramener l'intersection vide de la sémantique de ces deux domaines à celle du composant HIC.
Mais, malheureusement cette configuration est physiquement irréalisable dans la mesure où un composant ne peut nullement être simultanément présent dans deux différents domaines.

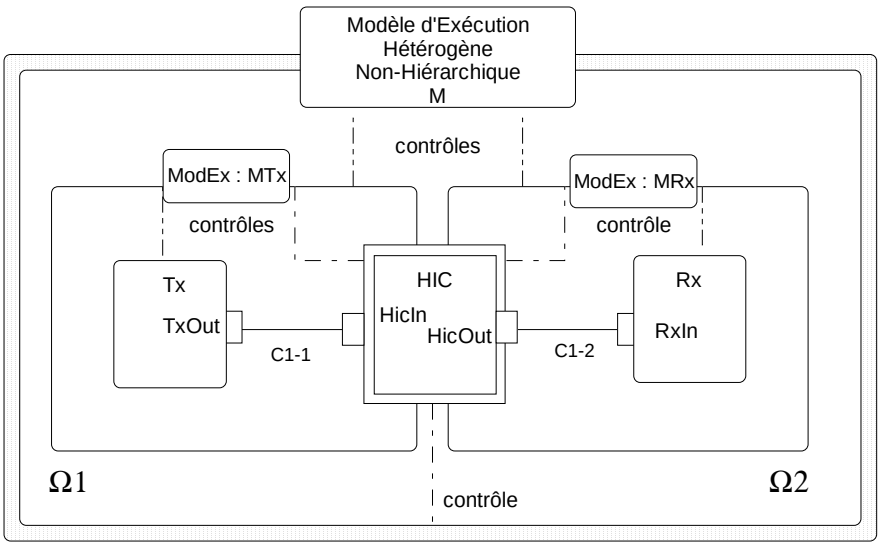


FIG. 4.9 – Hic à cheval entre deux sous-systèmes

Néanmoins, cette configuration bien que physiquement irréalisable nous permet déjà à un haut niveau d'abstraction d'avoir une première approche de la division du système en sous-systèmes.

Dans les deux sections suivantes, nous allons montrer que lorsqu'un système est restructuré en sous systèmes non-hiérarchiques, la communication peut être faite en utilisant soit l'abstraction hiérarchique ou soit l'abstraction non-hiérarchique. Nous montrerons également que dans un système restructuré non-hiérarchiquement, l'utilisation de l'abstraction non-hiérarchique présente plus d'avantages que l'utilisation de l'abstraction hiérarchique.

4.4 Approche non-hiérarchique par l'abstraction hiérarchique

A ce niveau, intuitivement, nous pouvons immédiatement penser à sortir le HIC des sous-systèmes Ω_1 et Ω_2 et utiliser l'abstraction hiérarchique. Ceci revient à doter les sous-systèmes Ω_1 et Ω_2 des ports respectivement Ω_1Out et Ω_2In. Ensuite connecter le port de sortie TxOut au port de sortie Ω_1Out qui lui-même sera connecté au port d'entrée HicIn de HIC. De même, connecter le port de sortie HicOut de HIC au port d'entrée Ω_2In qui également de l'intérieur est connecté au port RxIn de l'acteur Rx comme montré sur la figure 4.10.
Cette structuration compositionnelle du système est évidemment réalisable. Nous l'avons en effet réalisée, implémentée et exécutée. Malheureusement, cette manière de structurer, bien que réalisable, n'est guère réaliste car elle présente d'énormes inconvénients que nous présentons ci-dessous.

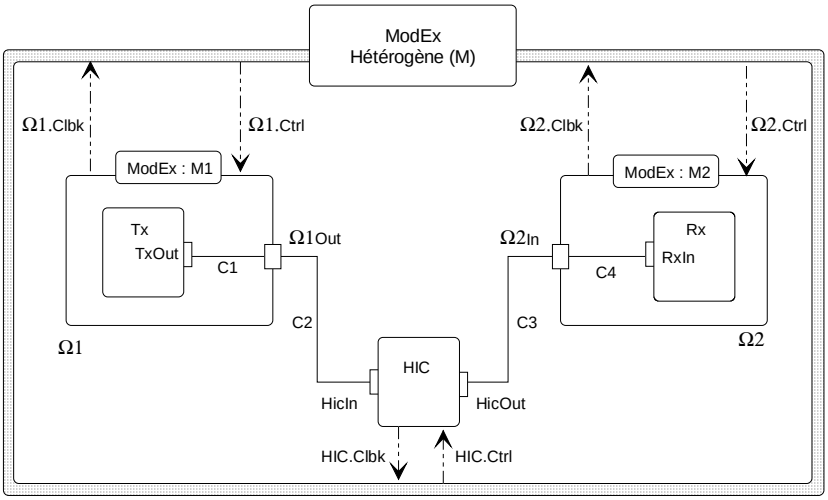


FIG. 4.10 – Utilisation de l'abstraction hiérarchique

4.4.1 Inconvénients

Cette configuration impose une intervention du modèle d'exécution hétérogène dans le flot de données. En effet, celui-ci doit assurer le transfert des données du port Ω_1Out à l'entrée du port HicIn et de la sortie HicOut au port Ω_2In. Or les deux ports de ces deux sous-systèmes sont supposés utiliser des modèles de calcul différents.
Ceci implique que le modèle d'exécution hétérogène doit avoir connaissance des modèles de calcul utilisés par les sous-systèmes pour assurer le mécanisme de communication imposé par lesdits MoCs. Ce qui transgresse la règle que nous avons prônée, celle de la non-connaissance par le modèle d'exécution hétérogène des modèles de calcul utilisés par les sous-systèmes. De plus, dans ces interactions hétérogènes, il pourrait arriver que les

données soient impliquées dans le contrôle ou que le contrôle interfère sur les données.

Dans certaines plates-formes, en l'occurrence PTOLEMY II, corollairement à ce que nous avons souligné à la section 2.5.3, le sous-système Ω_1 et le HIC délèguent le transfert de leurs sorties au modèle d'exécution hétérogène. Le fait de déléguer ce transfert au modèle d'exécution hétérogène, impose à ce dernier d'être capable de fournir la sémantique des deux modèles de calcul et contrôler les canaux de communication c2 et c3 en fournissant en conséquence les receivers hétérogènes spécifiques sur les ports HicIn et Ω_2In. Les conséquences qui en découlent sont :

– Sur le plan du code source : dans le modèle d'exécution hétérogène, il y a duplication du code relatif à la spécificité des communications pour chacun des modèles de calcul. Normalement, ces spécificités de communication sont assurées par les modèles d'exécution réguliers. Mais, du fait que ces derniers ne peuvent pas prendre en charge une communication en dehors des sous-systèmes qu'ils gouvernent, le modèle d'exécution hétérogène serait donc obligé d'assumer cette tâche. donc implémenter chaque mécanisme de communication ayant trait à chaque modèle de calcul.

– Sur le plan de la gestion du temps, nous savons que certains modèles de calcul sont très explicites et considèrent le temps comme une contrainte d'ordre total, c'est-à-dire, un nombre réel qui avance uniformément, et placent des événements sur cette ligne de temps ou font évoluer les signaux continus dans le temps. D'autres domaines non-temporisés sont plus abstraits et considèrent le temps comme simplement une contrainte d'ordre partiel imposée par la causalité. Cette configuration impose au modèle d'exécution hétérogène la gestion de ces différentes considérations du temps. Il doit par exemple aussi bien implémenter les files d'attente que des ordonnanceurs statiques. Lorsque les différents domaines en communication utilisent ces différents concepts du temps, la gestion et la signification simultanées du temps deviennent floues ou même impossible.

– Sur le plan de l'évolution, puisque le modèle d'exécution hétérogène doit avoir connaissance des MoCs utilisés par les sous-systèmes, une quelconque intégration d'un nouveau MoC imposerait une nouvelle implémentation d'un nouveau modèle d'exécution hétérogène qui prendrait en compte les spécificités introduites par le nouveau MoC.

Les inconvénients ci-dessus montrent l'inefficacité et militent en défaveur de cette manière de restructurer un système hétérogène.

Dans la section suivante, nous allons présenter un autre concept de restructuration de système. Ce concept consiste au retrait des ports des sous-systèmes et à l'établissement de HIC dans les différents sous-systèmes par ses projection . Ce concept permet au modèle d'exécution hétérogène de gouverner les sous-systèmes sans interférer dans les interactions locales des sous-systèmes en déléguant la gestion des activités des sous-systèmes à leurs modèles d'exécution réguliers respectifs.

4.5 Approche non-hiérarchique par l'abstraction non-hiérarchique

4.5.1 Abstraction non-hiérarchique par le canal hétérogène abstrait

Nous avons vu plus haut que pour sa composabilité hiérarchique, un modèle était composé d'une abstraction hiérarchique. Cette abstraction était composée de ses ports et de ses paramètres externes, qui sont distincts des ports et des paramètres des différents acteurs qu'il contient. Et, ses ports externes pouvaient être reliés par des canaux à d'autres ports externes d'un autre modèle ou aux ports des acteurs qu'il contient.

Nous allons remplacer ce concept d'abstraction hiérarchique par celui d'*« Abstraction non-hiérarchique »*. En effet, dans notre structuration non-hiérarchique, contrairement à l'abstraction hiérarchique, l'interface d'un sous-système sera dépourvue des ports de communication. Cette interface sera réduite à un simple ensemble de paramètres externes. Il n'existe donc pas de canal de communication reliant des sous-systèmes entre eux.

Or, la communication entre les composants se produisent le long des canaux de communication qui apparaissent comme des connexions entre les composants. Dans le système qui résulte de la transformation d'un système hétérogène, les canaux de communication entre les projections d'un HIC sur différents domaines ne peuvent pas avoir des connexions directes parce que la communication se produit à l'intérieur du HIC.

Cependant, ce canal de communication doit être pris en considération pour ordonnancer les sous-ensembles homogènes. Nous l'appelons « canal hétérogène abstrait ». La communication sur un tel canal obéit aux règles du domaine hétérogène. Ainsi, sur la figure 2.19, après la projection du HIC, le canal de communication original entre les acteurs Tx et Rx disparaît du fait qu'il ne peut pas être manipulé par les modèles d'exécution homogène. Il s'ensuit l'apparition d'un canal abstrait entre Tx et Rx.

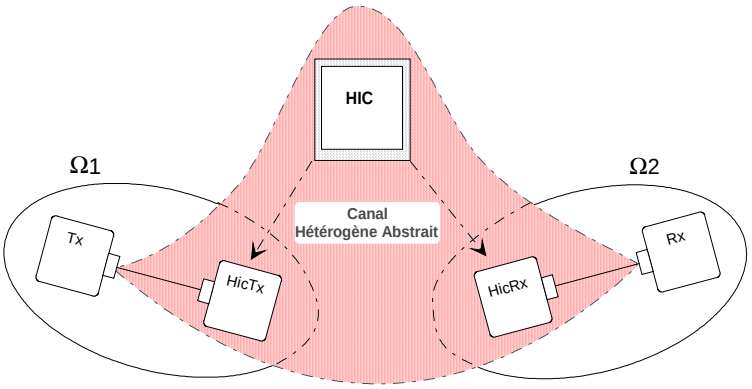


FIG. 4.11 – Canal hétérogène abstrait

Illustration 4.5.1 Les deux sous-systèmes Ω_1 et Ω_2 communiquent à travers ce canal hétérogène abstrait sous le contrôle du modèle d'exécution hétérogène.
Puisque Ω_1 et Ω_2 ne voient pas les connexions entre le HIC et l'acteur dans l'autre sous-système, le modèle d'exécution hétérogène doit à travers ce canal abstrait maintenir la chaîne de causalité entre la production des données HicTx et la consommation des données par HicRx pour assurer.

4.5.2 Segment de communication et Canal abstrait homogène

Segment de communication

Un chemin, pris dans le sens de la théorie des graphes [52] peut traverser plusieurs modèles de calcul en traversant des HICs. Nous allons de ce fait restreindre cette notion en imposant aux sommets d'un chemin d'être un couple bien précis qui de manière générale va délimiter un sous-système. Pour cela nous introduisons le concept de *«segment»* illustré par la figure 4.12.

Définition 4.5.1 *Nous définissons un segment d'un système comme un chemin entre deux HICs ou entre un sommet initial ou final du graphe et un HIC.*

Après le partitionnement, de manière générale, chaque segment du système donne lieu à la création d'un sous-système.

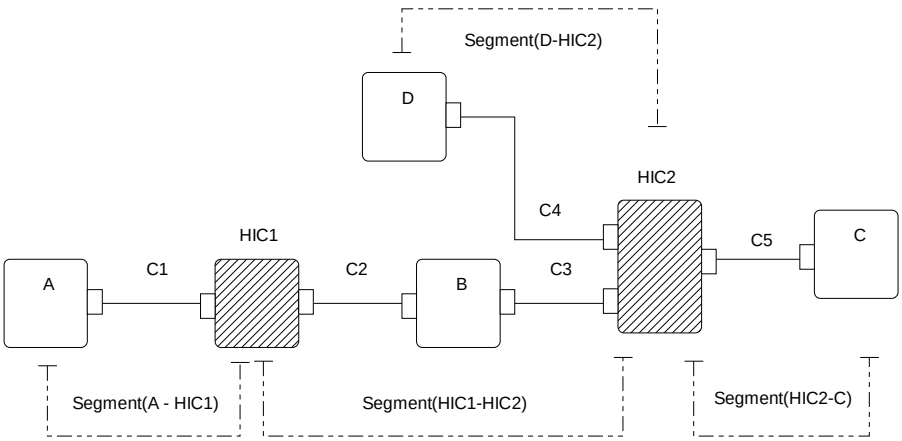


FIG. 4.12 – Quelques segments

Illustration 4.5.2 Ce système comporte quatre segments dont trois formés entre un sommet et un HIC : segment(A-HIC1), segment(D-HIC2), segment(HIC2-C), et un segment formé entre deux HICs : segment(HIC1-HIC2).

4.5.3 Projection du HIC dans les sous-systèmes

Proposition 4.5.1 *Puisqu'un acteur régulier ne peut appartenir qu'à un seul modèle de calcul, lorsque deux HICs forment un segment, la sortie du HIC producteur et l'entrée du HIC consommateur et tous les acteurs situés entre eux utilisent le même modèle de calcul, et, seront tous placés dans un même sous-système.*

En effet, les deux HICs aux extrémités du chemin seront projetés dans ce sous-système. Ces projections paraîtront comme des acteurs réguliers de ce sous-système du point de vue de son domaine, car, les ports d'entrée du HIC producteur et les ports de sortie du HIC consommateur seront masqués par le mécanisme de projection comme montré sur la figure 4.13.

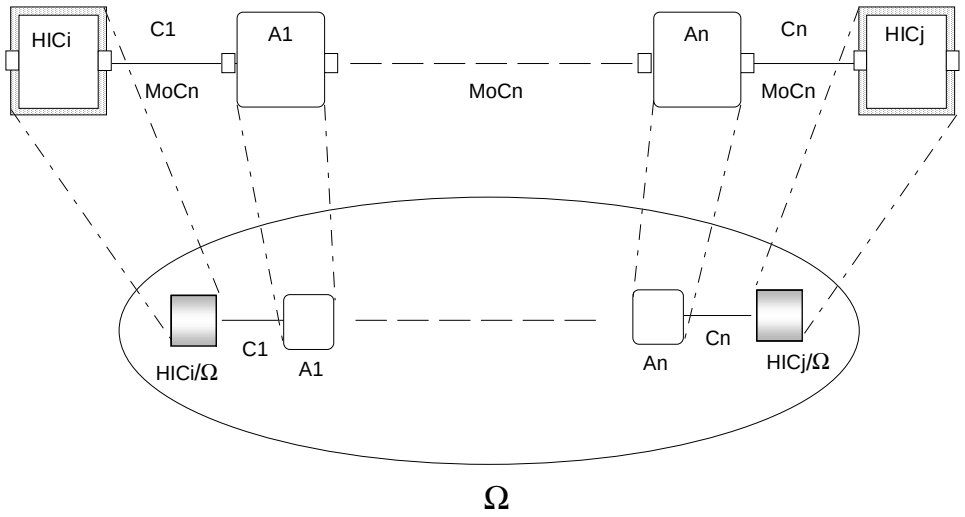


FIG. 4.13 – Placement des acteurs entre deux HICs dans le même sous-système

Illustration 4.5.3 Cette figure montre un segment dont les sommets sont HIC_i et HIC_j. Tous les canaux et tous les acteurs situés entre ces deux extrémités utilisent le même modèle de calcul MoCn. HIC_i et HIC_j sont projetés dans le même sous-système Ω.

A la section , nous avons vu qu'un composant à Interface Hétérogène est contrôlé aussi bien par plusieurs modèles d'exécution qui correspondent à différents modèles de calcul des différents sous-systèmes que par le modèle d'exécution hétérogène.

En effet, HIC sera projeté dans tous les sous-systèmes issus du partitionnement élaboré à la frontière de ses modèles de calcul hétérogènes qu'utilisent ses ports.
Ainsi, chaque projection obéit à la « sémantique locale », c'est-à-dire se conforme à la sémantique desdits sous-systèmes. En d'autres termes, ces projections apparaîtrons comme des composants homogènes dans ces sous-systèmes comme montré sur la figure 6.10.

Tout ce qui n'appartient pas au modèle de calcul de l'un des sous-systèmes sera masqué lors de la projection.

HicTx est une projection de HIC dans le sous-système Ω_1 et HicRx est celle de HIC dans le sous-système Ω_2. Les ports d'entrée correspondants, HicIn et HicTxIn sont mis en commun. Il en est de même pour ceux de sortie correspondants, HicOut et HicRxOut.
Le modèle d'exécution du sous-système Ω_1 verrait seulement la projection HicTx avec seulement son entrée HicTxIn . De même, le modèle d'exécution du sous-système Ω_2 verrait seulement la projection HicRx avec seulement sa sortie HicRxOut.
Les ports de HIC et leurs correspondants sur les projections seront mis en commun.

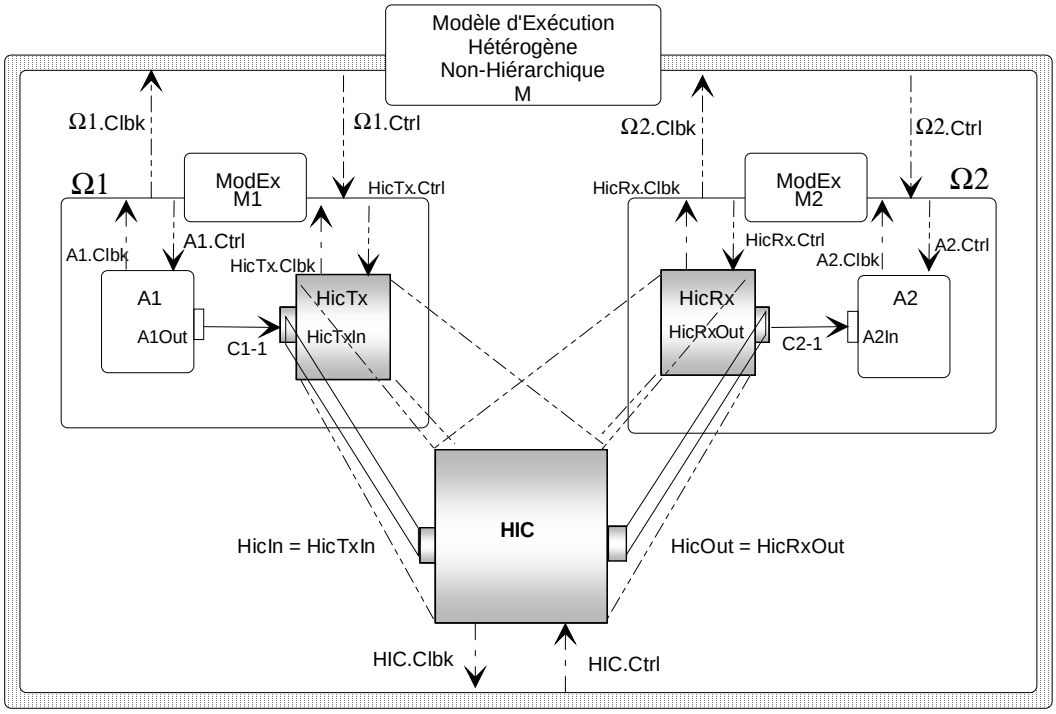


FIG. 4.14 – Projection de HIC et abstraction non-hiérarchique

Illustration 4.5.4

Le HIC original de la figure 4.8 est projeté dans les deux sous-systèmes Ω_1 et Ω_2 crées à la frontière de ses deux ports hétérogènes HicIn et HicOut.
HicTx est sa projection productrice dans Ω_1 HicRx est sa projection consommatrice dans Ω_2.
Le mécanisme de projection a masqué les ports HicOut et HicIn respectivement dans Ω_1 et Ω_2.
Les canaux de communications C1-1 et C1-2 sont refais comme sur la figure 4.8.
L'entité projection HicTx est contrôlée par Ω_1 et l'entité projection HicRx est contrôlée par Ω_2 et HIC est contrôlé par le modèle d'exécution hétérogène.

Canal abstrait homogène

Lorsque le système est partitionné suivant ses segments, il peut arriver qu'un canal de communication se perde comme montré sur la figure 4.15.

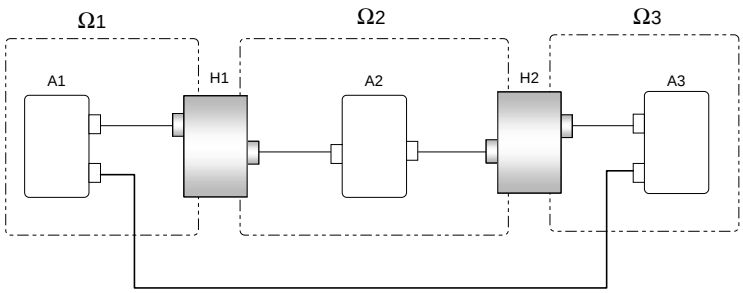


FIG. 4.15 – Canal de communication perdu

Les A_i sont des acteurs réguliers et les H_j sont des HICs.
A_1 appartient au segment Ω_1 avec le port d'entrée de H_1,
A_2 appartient au segment Ω_2 avec le port de sortie de H_1 et avec le port d'entrée de H_2,
A_3 appartient au segment Ω_3 avec le port de sortie de H_2.

Bien que A1 et A3 obéissent au même MoC, ils ne peuvent pas être mis dans le même sous-ensemble car cela mènerait à un système impossible à ordonnancer.

En effet, si nous mettons A3 dans Ω_1, Ω_1 dépend de Ω_2 parce qu'il contient A3 qui dépend de A2 par H2, et Ω_2 dépend de Ω_1 parce qu'il contient A2 qui dépend de A1 par H1.
Cependant, quand A1 et A3 sont dans différents sous-ensembles, nous ne pouvons pas les relier comme dans un système plat. Nous appelons un tel canal de communication servant à les relier, *« Canal abstrait homogène ».*

Ces canaux sont homogènes du fait que leurs extrémités utilisent le même modèle de calcul, mais ils ne peuvent pas être représentés par un canal régulier de communication dans un sous-système car les acteurs qu'ils connectent n'appartiennent pas au même sous-système.

Ils sont mis en application en utilisant un composant *« Relais »* source qui transmet des données à un composant cible correspondant. Sur l'exemple de la figure 4.16, les données qui sont disponibles sur l'entrée de Tx sont également disponibles sur la sortie de Rx, et l'ordonnanceur du domaine hétérogène s'assurera que Ω_1 est calculé avant Ω_3 de sorte que Tx puisse transmettre la valeur à Rx avant que la sortie de Rx soit utilisée.

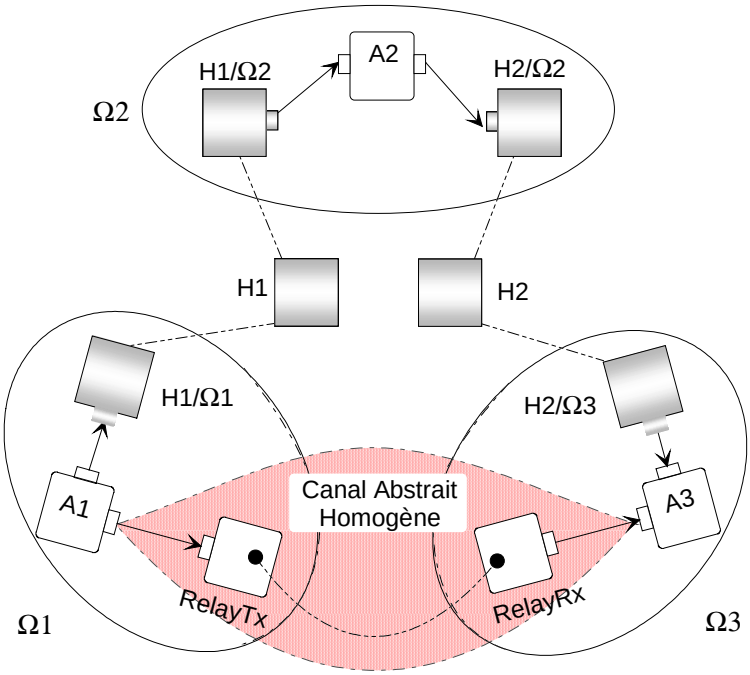


FIG. 4.16 – Canal abstrait homogène

4.5.4 Séparation entre le contrôle et la communication hétérogènes

Abstraction du contrôle et de la communication

Dans la section 4.4, nous avons vu que lorsque deux ou plusieurs modèles de calcul différents coexistent, la question fondamentale était la détermination du modèle de calcul supérieur.

Avec l'abstraction non-hiérarchique de notre approche hétérogène non-hiérarchique, ce souci est écarté, car, le modèle d'exécution hétérogène fournit un modèle de calcul « hétérogène » basé sur une sémantique unique de contrôle.
Ce qui abstrait le flot de contrôle, car, quels que soient les modèles de calcul utilisés par les différents sous-systèmes, leur contrôle par le modèle d'exécution hétérogène ne dépend pas de leurs modèles de calcul.

En effet, le modèle de calcul supérieur ne doit plus fournir la sémantique de communication comme dans l'approche hiérarchique, car les canaux hétérogènes abstraits ne disposant d'aucune hypothèse sur les protocoles de communication à utiliser, fournissent un mécanisme abstrait de communication dépendant plutôt du couple de protocoles de communication qui y sont effectivement utilisés.
Ainsi, nous disons, de manière abstraite, qu'un canal hétérogène abstrait implémente deux

différents protocoles de communication.
Ce qui abstrait également le caractère de protocole de communication car quel que soit le couple de protocoles de communication utilisés dans le canal abstrait hétérogène, le mécanisme de communication demeure le même.

Décharge du modèle d'exécution hétérogène du transfert des données

De ces deux caractéristiques relevée à la section 4.5.4, il en découle que la communication entre deux acteurs hétérogènes demeure transparente et ne nécessite aucune intervention de la part du modèle d'exécution hétérogène.

Dans PTOLEMY II par exemple, du point de vue de ses entrées et de ses sorties, l'acteur tx écrira directement sur le canal hétérogène abstrait et l'acteur rx lira directement du même canal. Il en résulte que le type de receiver utilisé dans un port lors d'une communication hétérogène dépend du protocole de communication d'extrémité, qui à son tour dépend du modèle de calcul utilisé dans le sous-système approprié.
Puisque c'est le modèle d'exécution régulier qui donne la sémantique du modèle de calcul dans le sous-système, c'est donc le modèle d'exécution régulier qui détermine les receivers nécessaires sur les ports d'entrée du HIC consommateur.

C'est ainsi que le modèle d'exécution hétérogène est complètement déchargé de la fourniture des receivers à chacun des ports hétérogènes. En conséquence, il est déchargé du transfert des données entre deux acteurs hétérogènes. Il n'y a donc plus délégation du transfert des sorties au modèle d'exécution hétérogène.

Avantages

Puisque nous venons de montrer que le modèle d'exécution hétérogène est complètement déchargé du transfert des données, et qu'il n'y a aucune raison pour ce dernier d'avoir connaissance des modèles de calcul utilisés par les sous-systèmes. Il en résulte donc :
– Sur le plan de la structuration compositionnelle du modèle, celui-ci est réaménagé et restructuré au même niveau hiérarchique. Les comportements à la frontière des différents modèles de calcul sont implémentés explicitement dans les composants à interface hétérogène.
  De plus, puisque les projections sont des variables internes des sous-systèmes, elles n'utilisent donc qu'un seul modèle de calcul. De ce fait, elles sont en parfaite harmonie avec les sémantiques de leurs modèles de calcul qu'elles utilisent.
  Par exemple, dans un sous-système SDF, la gestion des équations des balances entre un acteur quelconque et une projection est faite par le modèle d'exécution SDF qui gouverne ce sous-système. Dans un sous-système DE par exemple, chaque projection prendra soin de s'auto-poster un événement pur pendant son opération d'initialisation pour pouvoir assurer sa première exécution.

– Sur le plan de la gestion du temps, son implémentation et sa gestion sont faites par les modèles d'exécution réguliers. Soit sous forme de contrainte d'ordre total pour les modèles temporisés, soit sous forme de contrainte d'ordre partiel imposée par la causalité, pour les modèles non-temporisés. Cette configuration écarte le conflit qui pourrait germer sur l'interprétation du temps dans le modèle d'exécution hétérogène.
L'intégration ou le retrait du temps lorsqu'on passe d'un modèle temporisé vers un modèle non-temporisé ou l'inverse est explicitée dans le comportement du HIC.

– En ce qui concerne le code source, puisque le modèle d'exécution hétérogène est déchargé du transfert des données et de la gestion du temps, il n'y a pas de duplication du code relatif à la spécificité sémantique des communications et du temps, car ces derniers sont assurés et pris en charge par les modèles d'exécution réguliers.

– Sur le plan de l'évolution, puisque le modèle d'exécution hétérogène n'a pas connaissance de la variété des modèles de calcul utilisés par les différents sous-systèmes, une quelconque intégration d'un nouveau modèle de calcul n'imposerait nullement une nouvelle implémentation d'un nouveau modèle d'exécution hétérogène.

– Du point de vue des outils de modélisation, nous pouvons bien passer d'un modèle de calcul à un autre via le canal hétérogène abstrait sans être obligé de créer un nouveau niveau hiérarchique.
Dans PTOLEMY II par exemple, les sous-systèmes vont utiliser l'abstraction non-hiérarchique. Pratiquement, ils seront des acteurs composites sans ports. Cependant après la restructuration du système, les acteurs hétérogènes seront connectés via le canaux hétérogènes abstraits. Le modèle d'exécution hétérogène sera un Directeur Hétérogène qui créera, ordonnancera et activera les différents sous-systèmes, qui, à leur tour seront localement gouvernés par des Directeurs réguliers existants déjà dans PTOLEMY II.

4.5.5 Du point de vue de la communication et du comportement

Un composant à interface hétérogène est un composant qui a des entrées ou des sorties qui communiquent selon différents modèles de calcul comme montré sur la figure 6.10.
Un tel composant doit être activé par sa projection dans le contexte d'un sous-système quand il reçoit des données d'un acteur qui est contrôlé par ce sous-système. Quand la réaction à son activation par le sous-système déclenche la production des données vers un acteur qui est contrôlé par un autre sous-système, il doit être activé par sa projection dans ce deuxième sous-système afin de pouvoir transmettre ces données.

Ainsi, de façon élémentaire, le comportement limite de ce système peut être décomposé en trois phases : la phase Tx à HIC, le comportement de HIC et la phase HIC à Rx.
– Dans la première phase, le sous-système incite l'acteur Tx à réagir. Celui-ci produit les données sur sa sortie, et, puisque cette sortie est reliée à l'entrée Tx du HIC, la projection de HIC dans ce sous-système Tx fait réagir le HIC.

Du point de vue de ce sous-système, HIC est vu comme un acteur dans le domaine Tx. Ceci signifie que la sortie Rx du HIC n'est pas vue par le sous-système Tx.

– Puis, lorsque HIC est activé par le sous-système Tx, il voit les données sur son entrée Tx et les traite. Pendant ce traitement, il doit produire des données sur sa sortie Rx. Cependant, la production des données dans le sous-système Rx exige que celui-ci soit au courant de cette production de sorte que les données puissent être conduites à l'entrée de l'acteur Rx. Par conséquent, HIC doit être activé par le sous-système Rx.

– Enfin, du fait que le port de sortie du HIC est relié au port d'entrée de l'acteur Rx, le sous-système Rx active l'acteur Rx qui détecte et lit les données sur son entrée, et les traite.

4.6 Conclusion partielle

Dans ce chapitre, nous avons présenté une approche théorique de l'hétérogénéité non-hiérarchique que nous avons considérée comme une alternative à l'approche hiérarchique. Cette approche théorique, s'appuyant sur la méthodologie orientée acteur, utilise deux composants d'appui qui sont : le composant à interface hétérogène et le modèle d'exécution hétérogène non-hiérarchique.

Le composant à interface hétérogène qui, suivant sa spécification de communication, dispose d'entrées et de sorties hétérogènes lui permettant de mettre en communication des composants ayant des ports utilisant des modèles de calcul différents. Et, suivant sa spécification comportemental, il est capable de fournir un comportement hétérogène aux frontières des différents modèles de calcul.
Le modèle d'exécution hétérogène non-hiérarchique gouverne l'exécution hétérogène. De part sa spécification, il gère la décomposabilité modulaire du système en divisant le système à la frontière des différents modèles de calcul utilisés par le HIC, crée des sous-systèmes et délègue leur flot de contrôle et leur calcul du comportement locaux à leurs modèles d'exécution réguliers respectifs. En outre, il génère l'ordonnancement approprié de ces différents sous-systèmes avant de les activer.

Cependant, pour une restructuration non-hiérarchique d'un système hétérogène, nous avons proposé deux approches : l'approche non-hiérarchique par l'utilisation de l'abstraction hiérarchique et l'approche non-hiérarchique par l'utilisation de l'abstraction non-hiérarchique que nous avons préconisée de par ses avantages.

Toutefois, le premier point clé de cette étude est en effet, la subtilité introduite en remplaçant l'abstraction hiérarchique par l'abstraction non-hiérarchique. Ceci a engendré des « canaux abstraits hétérogènes » qui ont facilité la communication des données entre deux sous-systèmes non-connectés, et ce, sans aucune connaissance des modèles de calcul desdits sous-systèmes par le modèle d'exécution hétérogène.

Le second point clé de cette étude est la séparation entre le flot de contrôle, assuré par le modèle d'exécution hétérogène et le flot des données, assuré par le HIC. Dans le second flot, la communication est gérée par les projections de HIC et le calcul de comportement hétérogène est fait par HIC lui-même.
Cette configuration nous a permis de mettre à « plat » le système hétérogène et d'isoler le modèle d'exécution de la connaissance des modèles de calcul utilisés par les sous-systèmes.

Enfin, pour que cette théorie se rapproche de la réalisation, il faut la rendre réalisable. Tel est le rôle de la deuxième partie de cette étude dédiée à la modélisation de ces composants d'appui à l'hétérogénéité non-hiérarchique.

Deuxième partie

Modélisation des Composants d'appui à l'Hétérogénéité Non-Hiérarchique

Chapitre 5

Composant à Interface Hétérogène (HIC)

Résumé - Ce chapitre présente en deux étapes la modélisation du Composant à Interface Hétérogène. Cette modélisation est faite du point de vue de sa composition structurelle et opérationnelle. Le HIC original a d'abord été présenté. Ensuite, nous avons intégré le concept de son double partitionnement. Le partitionnement structurel et le partitionnement opérationnel en répartissant les différentes variables et opérations dans les différentes entités issues de ce partitionnement. Enfin nous avons établit les règles d'activation entre lesdites entités.

5.1 Introduction

Dans le chapitre précédent, nous avons vu que dans la méthodologie orientée acteur, un acteur était composé d'un ensemble de variables et d'un ensemble d'opérations. Puisque les composants à interface hétérogène sont des acteurs, ils ont par conséquent un ensemble de variables et un ensemble d'opérations.

Dans ce chapitre, à travers les primitives abstraites vues précédemment, nous modélisons le Composant à Interface Hétérogène du point de vue de sa composition structurelle et opérationnelle. Cette modélisation est faite en deux étapes.
En effet, nous avons d'abord modélisé un HIC original du point de vue de sa structure et de ses opérations.
Ensuite, du fait de la duplication de ses opérations et de la duplication de ses variables dues à sa projection dans différents sous-systèmes, nous avons donc intégré le concept de son double partitionnement en répartissant ses différentes variables et ses différentes opérations dans ses différentes entités. Ce partitionnement est fait suivant ses deux axes

compositionnels à savoir : un axe structurel et un axe opérationnel.
Enfin, pour une exécution cohérente, ces différentes entités issues de HIC doivent organiser une forme de coopération dans leur mécanisme de contrôle ainsi que dans les tâches de communication et de calcul de comportement qui leurs sont attribuées.
C'est pourquoi, dans le but de maintenir la chaîne de causalité entre les entités projection productrices et les entités projection consommatrices et le Composant à Interface Hétérogène lui-même, nous avons élaboré un mécanisme d'activation desdites entités reposant sur une logique booléenne.

5.2 Structure de HIC avant le partitionnement du système

Afin de simplifier la présentation, nous allons considérer un HIC n'ayant qu'une entrée et une sortie. Son ensemble de variables noté HIC.X est donc composé du sous-ensemble contenant ses variables d'interface, c'est-à-dire, ses ports et du sous-ensemble de ses variables internes tel que :

$$\text{HIC.X} = \{\text{HIC.P, HIC.Q}\} \bigcup \{\text{HIC.S}\}$$

avec

$$\text{HIC.P}=\{\text{HicIn}\},\ \text{HIC.Q}=\{\text{HicOut}\} \text{ et } \text{HIC.S}=\{hicParameter,\ hicState\}$$

Ainsi

$$\text{HIC.X} = \{\text{HicIn, HicOut}, hicParameter,\ hicState\}$$

où :
– HicIn est le port d'entrée à partir duquel le HIC lit les données venant d'un acteur producteur appartenant au domaine que nous notons Tx,
– HicOut est le port de sortie par lequel le HIC envoie les données vers un acteur consommateur appartenant au domaine que nous notons Rx.
– *hicParameter* et *hicState* symbolisent respectivement les paramètres et l'état courrant du HIC.

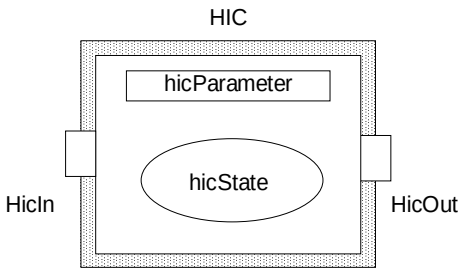


Fig. 5.1 – Structure d'un HIC

5.3 Opérations de HIC avant le partitionnement du système

De part sa spécification établie à la section 4.2.2, sur le plan de la communication hétérogène, le composant à interface hétérogène doit disposer d'entrées et de sorties hétérogènes lui permettant de mettre en communication deux ou plusieurs composants ayant des ports utilisant des modèles de calcul différents. Sur le plan de son comportement hétérogène, il doit donc être capable de fournir un comportement aux frontières des différents modèles de calcul.
Ainsi, le HIC dispose d'un ensemble d'opérations sur les données et d'un ensemble d'opérations sur le contrôle que nous exposons dans les sections qui suivent.

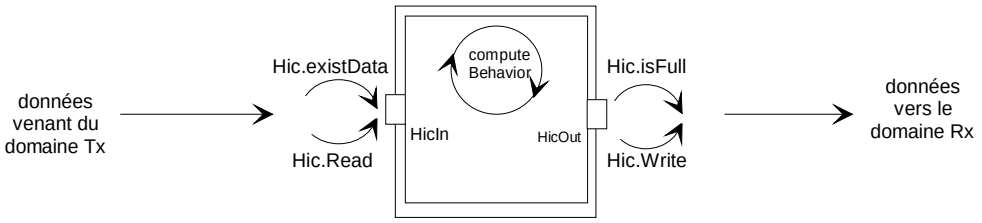


FIG. 5.2 – Opérations dans le HIC

5.3.1 Opérations de flot de données de HIC

Le comportement et la communication de HIC sont déterminés par l'ensemble de ses opérations de calcul noté Hic.Comp et par l'ensemble de ses opérations de communication noté Hic.Comm.
Le premier ensemble détermine la manière dont le HIC va calculer son comportement à la frontière des deux ou plusieurs modèles de calcul adjacents. Le second ensemble détermine la manière dont il enverra les données issues de ce comportement vers un acteur consommateur après transformation.

Opérations de communication de HIC

Les variables d'interface *HicIn* et *HicOut* de l'acteur HIC sont ses ports hétérogènes et appartiennent à son interface de communication. Il communique avec les acteurs hétérogènes via ces deux ports HicIn et HicOut.
Une valeur sur son port d'entrée HicIn ne peut être modifiée que par le modèle d'exécution qui le gouverne, car, c'est en effet ce dernier qui assure le transfert de ses données entrantes jusqu'à son port HicIn.
Conformément aux primitives abstraites présentées au paragraphe 3.4.2, l'ensemble d'opérations de communication de HIC noté Hic.Comm est composé des deux sous-ensembles

d'opérations à savoir :
– l'ensemble Hic.Read d'opérations de lecture effectuées par l'acteur HIC tel que : Hic.Read = $\{existData(),\ read()\}$,

– l'ensemble Hic.Write d'opérations d'écriture effectuées par l'acteur HIC tel que : Hic.Write = $\{isFull(),\ write()\}$.
Finalement,

$$\begin{aligned} \text{Hic.Comm} &= \text{Hic.Read} \bigcup \text{Hic.Write} \\ &= \{existData(),\ read(),\ isFull(),\ write()\} \end{aligned}$$

Comportement

Puisque le rôle fondamental du HIC consiste entre autre à générer un comportement hétérogène à la frontière des deux ou plusieurs modèles de calcul, il dispose en conséquence d'une opération unique dédiée à cette tâche. C'est en effet dans cette opération que nous nommons *computeBehavior*() que le concepteur du système définit selon sa spécification, le comportement hétérogène du système à la frontière de différents modèles de calcul. En d'autres termes, c'est donc ici que le concepteur du système définit de manière explicite l'interprétation des données lorsqu'elles passent d'un modèle de calcul vers un autre.
Au niveau de son exécution, cette opération utilise les données consommées, calcule le comportement et éventuellement fournit les données résultantes aux ports de sortie selon l'état du HIC telle que :

$$\text{Données sur } HicOut \leftarrow computeBehavior\text{de données sur } (HicIn)$$

Ainsi, l'ensemble d'opération de calcul de comportement de HIC est réduit à un singleton tel que :

$$\text{HicComp} = \{computeBehavior()\}$$

5.3.2 Opérations de flot de contrôle de HIC

Le HIC est contrôlé par les opérations de contrôle qui lui sont envoyées par le modèle d'exécution. Ces opérations sont rassemblées dans l'ensemble Hic.Ctrl.
Il est également contrôlé par les opérations de rappel, appelées aussi opération de Call-back qu'il utilise pour solliciter le modèle d'exécution. Ces opérations sont rassemblées dans l'ensemble Hic.Clbk.

Conformément aux primitives abstraites présentées au paragraphe 3.4.2, son ensemble d'opérations de contrôle est :

$$\text{Hic.Ctrl}=initialisation(),\ preCondition(),\ trigger(),\ postCondition()$$

De même, son ensemble d'opérations de rappel contient entre autre l'opération $finish()$ par laquelle il sollicite le modèle d'exécution pour lui signifier la fin de ses activités[1]

$$\text{Hic.Clbk} = \{finish()\}$$

5.3.3 Ensemble d'opération d'exécution de HIC

L'ensemble d'opérations d'exécution de HIC que nous notons Hic.Oper est donc l'union de toutes les opérations que peut exécuter HIC telles que :

$$\begin{aligned} \text{Hic.Oper} &= \text{Hic.Comm} \bigcup \text{Hic.Comp} \bigcup \text{Hic.Clbk} \\ &= \{existData(),\ read(),\ isFull(),\ write(), \\ &\quad computeBehavior(),\ finish()\} \end{aligned}$$

Nous résumons ces opérations dans le tableau suivant :

Flow de données		Flow de contrôle
CALCUL	COMMUNICATION	initialisation préCondition trigger postCondition finish
computeBehavior	read write isFull existData	

Tab. 5.1 – Récapitulatif des opérations de HIC avant le partitionnement

A la section 4.5.3, nous avons vu que pendant le partitionnement du système, le HIC était projeté dans différents sous-systèmes. Or, le partitionnement du système implique à la fois l'éclatement du HIC et celui des opérations.
Il est cependant intéressant à ce niveau de s'interroger sur la manière dont les différentes variables d'interface, les différentes variables internes et les différentes opérations de HIC seront reparties dans ces différentes entités issues de son partitionnement.

C'est pourquoi, dans les sections suivantes, nous présentons les partitionnements structurel et opérationnel de HIC.

[1] Dans la suite de ce document, nous utiliserons simplement la notation finish() en lieu et place de finish_trigger()

5.4 Partitionnement structurel de HIC

Pendant le partitionnement du système, la projection d'un HIC n'est autrement que ce HIC placé dans un sous-système. Cependant, cette projection ayant perdu ses ports qui n'appartiennent pas au sous-système sur lequel il est projeté, apparaît comme un acteur homogène dudit sous-système.

5.4.1 Variables internes

Les variables internes de HIC ne sont utilisées que pour le calcul du comportement hétérogène. Or, ce calcul étant à la frontière de plusieurs modèles de calcul, donc, à la frontière de plusieurs sous-système, il ne peut être effectué que par une entité issue de HIC, mais, qui n'est pas totalement incluse dans un quelconque sous-système.
C'est pourquoi, pour l'éclatement de la structure du HIC, les variables internes seront en conséquence utilisées par l'entité HIC elle-même, car, celle-ci n'appartient à aucun sous-système. Mais, elle est plutôt vue comme une variable interne du modèle global, au même niveau que ces sous-systèmes.

5.4.2 Variables d'interface

Après le partitionnement du système, une projection de HIC fait partie des variables internes d'un sous-système, et, elle est par conséquent connectée à une autre variable interne qu'est le canal de communication via une de ses variables d'interface qu'est son port comme montré sur la figure 5.3. C'est à partir de ce port que le HIC viendra lire ou écrire les données.
C'est pourquoi, toutes les variables d'interface des projections seront mises en commun avec les variables d'interface correspondantes du composant à interface hétérogène qui a généré cette projection. Les variables d'interfaces incompatibles étant masquées, ce qui donne au HIC son caractère d'omniprésent dans tous les sous-systèmes dans lesquels il est projeté, tout en respectant leurs sémantiques respectives.

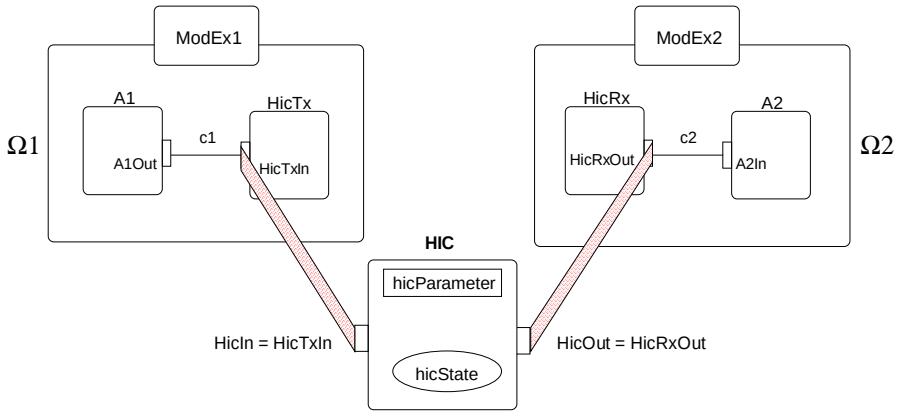


FIG. 5.3 – Structure éclatée du HIC

5.5 Partitionnement opérationnel de HIC

Le Composant à Interface Hétérogène est projeté dans tous les sous-systèmes et chaque projection y apparaît comme un composant homogène dans ces sous-systèmes. Ainsi, le HIC est contrôlé aussi bien par les modèles d'exécution des sous-systèmes que par le modèle d'exécution hétérogène.

Ce partitionnement est fait de telle manière que du point de vue de la communication et du comportement, certaines opérations soient exécutées par les projections du HIC, et d'autres par le HIC lui-même.
De même, du point de vue de contrôle, le partitionnement est élaboré de telle manière que certains contrôles soient effectués par le modèle d'exécution hétérogène et certains d'autres soient délégués aux modèles d'exécution des sous-systèmes par le modèle d'exécution hétérogène.
Pour les mêmes raisons qu'au partitionnement structurel, de manière générale, les projections ont donc le même ensemble d'opérations que leurs HICs originaux.

5.5.1 Opérations de communication

Selon la structure éclatée du composant à interface hétérogène présentée à la section 5.4, ce sont les entités projetions qui sont connectées aux différents canaux de communication des sous-systèmes. Ainsi, en ce qui concerne les opérations de communication, les opérations $existData(), read(), isFull()$ et $write()$ seront réparties de la manière suivante :

1 : Les opérations $existData()$ et $read()$ seront exécutées par les ports d'entrée de HIC mis en commun avec les ports correspondants de leurs projections.
2 : Les opérations $isFull()$ et $write()$ seront exécutées par les ports de sortie de HIC mis en commun avec les ports correspondants de leurs projections.

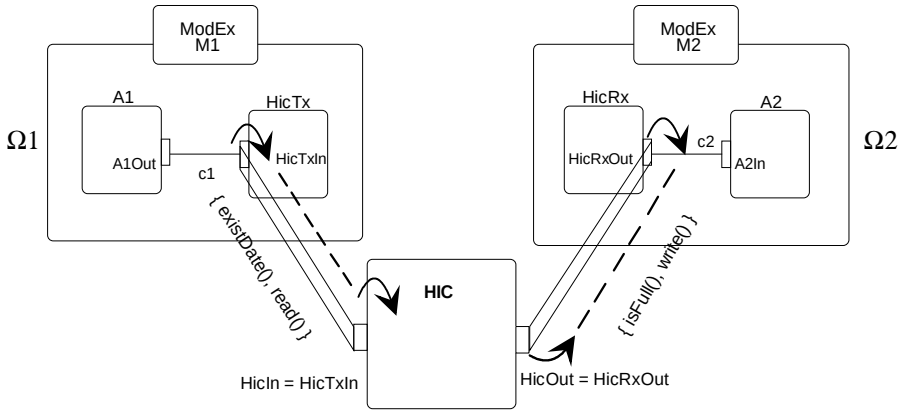


FIG. 5.4 – Eclatement des opérations de communication de HIC

5.5.2 Opération de calcul de comportement

Comme pour les variables internes, le calcul de comportement hétérogène est une opération à la frontière des différents modèles de calcul, elle est donc à la frontière des différents sous-systèmes. Ceci implique que l'opération de calcul de comportement soit affectée à une entité issue de HIC, mais, qui n'est pas totalement incluse dans un quelconque sous-système.
Du fait qu'une entité projection apparaît comme une composante homogène du sous-système dans lequel il est placé, cette opération ne peut donc pas être exécutée par elle.
C'est pourquoi, l'opération de calcul de comportement *computeBehavior*(), sera dédiée à l'entité HIC elle-même qui est une variable interne du modèle global, au même niveau que les sous-systèmes pour lesquels il calcule ce comportement hétérogène.

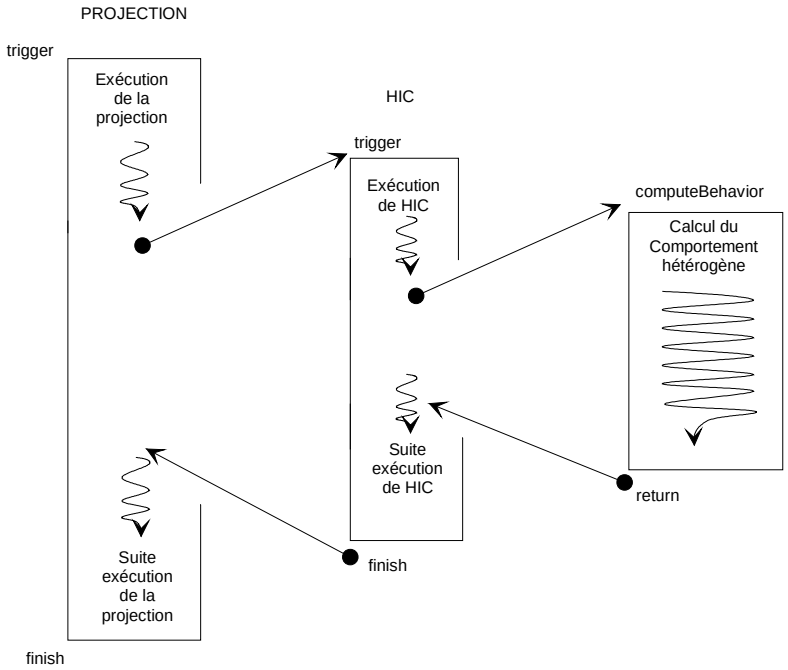


FIG. 5.5 – Opération de calcul de Comportement

Sur la figure 5.5, nous montrons que l'opération *computeBehavior*() est invoquée par l'opération *trigger*() de HIC qui elle-même est invoquée par l'opération *trigger*() de ses projections.
Ainsi, l'exécution de HIC est encadrée par les deux points de synchronisation d'une projection à savoir : projection.*trigger*() et projection.$finish$().
De même, l'opération *computeBehavior*() est encadrée par les deux points de synchronisation du HIC qui sont également : Hic.*trigger*() et Hic.$finish$().

5.5.3 Opérations de contrôle

Nous avons vu à la section 4.5.4 que le modèle d'exécution hétérogène était complètement déchargé du transfert des données entre deux acteurs hétérogènes, et donc, n'a pas connaissance des différents modèles de calcul utilisés par les différents sous-systèmes qu'il gouverne.

Puisque le modèle d'exécution a la méconnaissance totale des différents modèles de calcul utilisés par les différents sous-systèmes il est obligé de déléguer complètement le contrôle local desdits sous-systèmes à leurs modèles d'exécution locaux respectifs.
Par voie de conséquence, le contrôle de chacune des entités projections est délégué exclusivement au modèle d'exécution ayant la charge de son sous-système.
Or, une entité projection est une variable interne d'un sous-système, et, pour que son modèle d'exécution local qui le contrôle puisse d'une part assurer son déclenchement et d'autre part détecter la fin de ses activités, elle doit donc utiliser les deux opérations qui sont ses points de synchronisation : l'opération *trigger*() qui est son point de synchronisation initial et l'opération $finish()$ qui est son point de synchronisation final.

De ce fait, il est intéressant de noter que, comme nous l'avons vu à la section 5.5.2, les deux points de synchronisation d'une entité projection encadrent l'ensemble d'opérations de son HIC original. Et, les deux points de synchronisation de son HIC original encadrent son opération de calcul de comportement hétérogène.

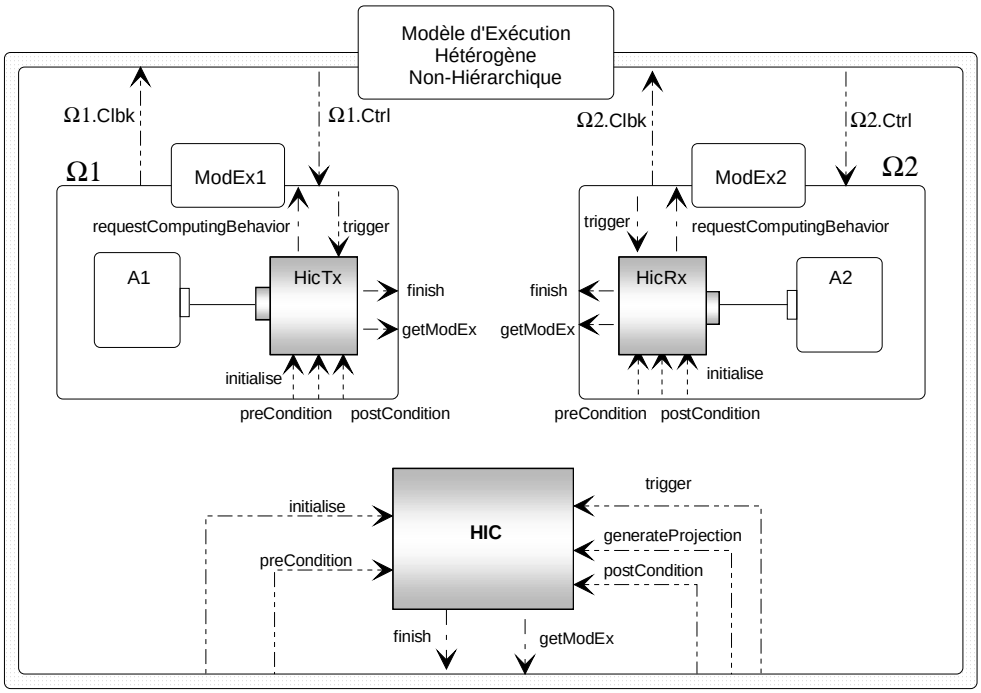


FIG. 5.6 – Signaux de contrôle de HIC

Opérations de contrôle d'une entité projection

1 : Une entité projection est contrôlée par le modèle d'exécution qui gouverne son sous-système. Ce contrôle passe par les opérations $initialise(), preCondition(), trigger(), postCondition()$.
2 : Une entité projection signifiera à son modèle d'exécution la fin de ses activités par son opération $finish()$.
3 : A travers son opération de call-back $getModEx()$, une entité projection sollicitera la nature de son modèle d'exécution, nature à partir de laquelle elle détermine la sienne.
4 : A travers son opération de call-back $requestBehaviorComputing()$, une entité projection sollicitera le modèle d'exécution hétérogène non-hiérarchique pour le lancement de son HIC original dans le but de calculer le comportement hétérogène.

Opérations de contrôle de HIC

1 : L'entité Composant à interface hétérogène est directement contrôlé par le modèle d'exécution hétérogène non-hiérarchique par ses opérations $initialise(), preCondition(), trigger(), postCondition()$.
2 : A travers son opération de call-back $getModEx()$, une entité HIC sollicitera la nature de son modèle d'exécution, nature à partir de laquelle elle détermine la sienne.
3 : Le HIC signifiera au modèle d'exécution hétérogène la fin de ses activités par son opération $finish()$.

5.5.4 Synthèse du partitionnement des opérations de HIC

	Tx	HIC	Rx
Communication	existData read	existData read isFull write	isFull write
Comportement		computeBehavior	
Contrôle	initialisation preCondition trigger getModEX requestBehaviorComputing postCondition finish	initialisation preCondition trigger getModEx postCondition finish	initialisation preCondition trigger getModEx requestBehaviorComputing postCondition finish

Tab. 5.2 – Récapitulatif des opérations de HIC

5.6 Mécanismes d'Activation de HIC

Puisqu'un HIC est à la frontière de plusieurs modèles de calcul, le nombre de ses projections générées est défini à la fois par les différents modèles de calcul utilisés par ses ports et par la topologie du système. Il en résulte qu'un HIC à la frontière de plusieurs modèles de calcul puisse générer plusieurs projections. De plus, ses projections sont des composants à part entière et sont dotées des mêmes opérations que leur HIC original. Elles utilisent par conséquent la même opération *trigger*().
Or, selon le type de l'entité, qu'elle soit une projection ou un HIC, l'exécution ne s'effectue pas de la même manière. Les entités sont donc obligées de s'auto-identifier avant de déterminer elles-mêmes la sous-routine appropriée à leur branchement.

– Pour décliner sa nature, par l'opération de call-back $getModEx()$, une entité interroge son modèle d'exécution pour connaître sa nature. Puisqu'un acteur n'est contrôlé que par un et un seul modèle d'exécution, la détermination de la nature de son modèle d'exécution induit la sienne.
C'est ainsi que les entités obtiennent la nature de leurs modèles d'exécution qui peut être soit un modèle d'exécution local ou soit le modèle d'exécution hétérogène ; ce qui implique pour une entité d'être soit une projection ou un HIC.

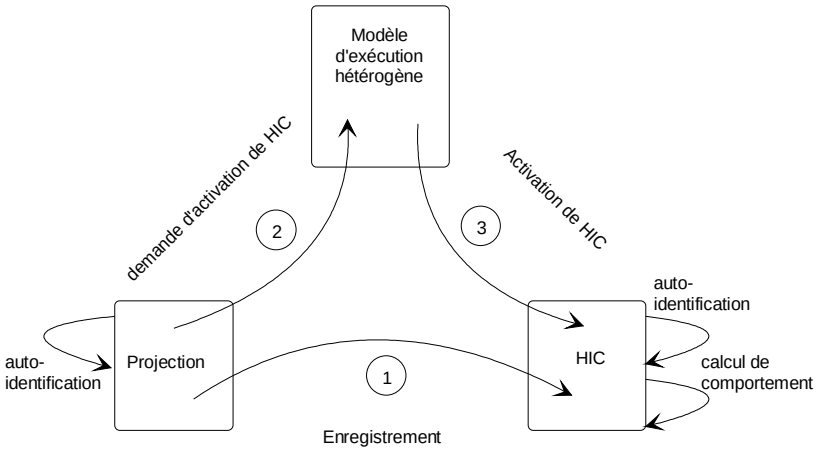


FIG. 5.7 – Mécanisme d'activation de HIC

– lorsqu'une entité est activée par un modèle d'exécution local, elle lance naturellement l'activation de son HIC original via le modèle d'exécution hétérogène. Du point de vue de la projection, l'exécution de son HIC original est vue comme sa sous-routine. Cette activation de HIC est effectuée à travers l'opération *requestBehaviorComputing*() et n'est simplement qu'une demande au HIC de calculer le comportement hétérogène.
Or, ce calcul de comportement étant fonction de l'entité qui l'a sollicité, impose au HIC la connaissance de cette dernière. Devant la pluralité des projections issues d'un HIC, avant de solliciter l'activation de son HIC, une projection doit au préalable s'enregistrer

auprès de ce dernier en utilisant son opération *recordRunningProjection*().

– lorsqu'une entité HIC sollicitée est activée par le modèle d'exécution hétérogène, après son auto-détection, elle lance naturellement l'exécution du calcul de son comportement hétérogène. Ce calcul est fait par l'opération *computeBehavior*().

5.6.1 Algorithme du mécanisme d'activation de HIC

Puisque l'opération d'activation du HIC est la même que celle de ses projections, nous donnons l'algorithme ci-dessous. Il est implémenté dans un HIC et dans ses projections et permet donc de tester si l'acteur (HIC ou projection)est activé en tant que HIC ou en tant que projection du HIC. Selon sa nature, l'acteur exécute les opérations qui lui sont appropriées

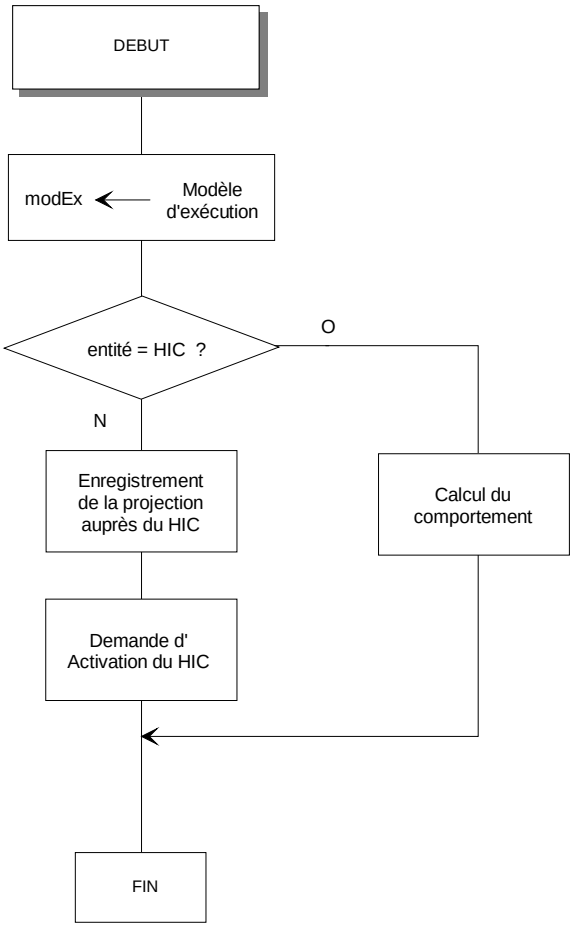


Fig. 5.8 – Algorithme d'activation de HIC

5.6.2 Postage du temps vers un sous-système DE

Pendant la phase du traitement du comportement de HIC, quand celui-ci doit produire les données dans un sous-système pour lequel il n'est pas activé, il doit remettre la production à plus tard jusqu'à son activation par sa projection productrice dans de ce sous-système.

Or, si sa projection utilise le modèle de calcul Discrete Event, pour son déclenchement, elle doit disposer d'une estampille de temps programmée dans la file d'attente de son modèle d'exécution. Il appartient donc à l'entité HIC de poster dans la file d'attente du modèle d'exécution qui gouverne sa projection DE, un événement pur pour garantir son déclenchement.
Nous avons résolu ce problème en dotant le HIC d'une opération supplémentaire que nous avons nommée $nextTimeToFire()$.

En effet, le HIC original connaissant le port de sa projection à activer, car, est mis en commun, peut obtenir la référence de celle-ci. De ce fait, dans son opération $nextTimeToFire()$, le HIC va activer l'opération $enqueueNextTimeToFire()$ de la projection concernée. Cette opération va obtenir, par le biais de l'opération $getModEx()$, le modèle d'exécution de ladite projection et mettre dans sa file d'attente, le temps spécifique pour le déclenchement de sa projection comme sur la figure 5.9.

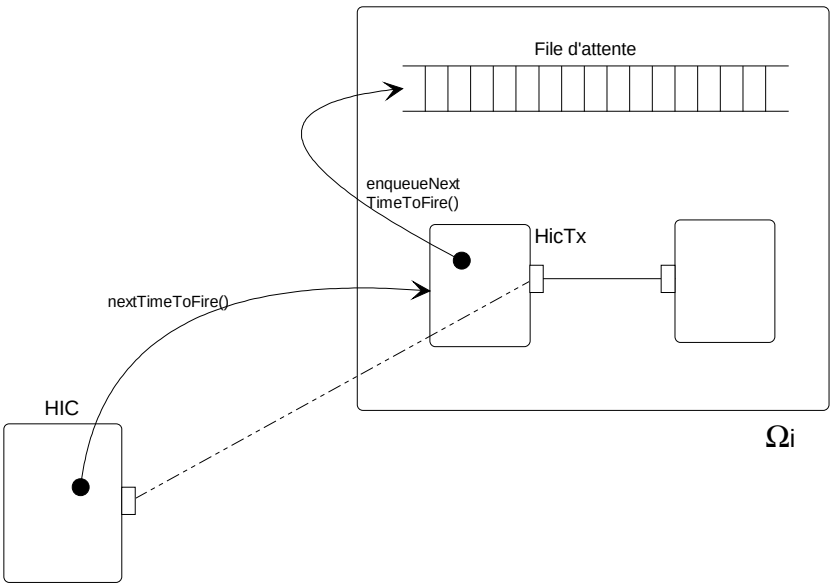


FIG. 5.9 – Postage du temps par un HIC DE

5.7 Spécification du comportement de HIC

Chaque fois qu'une projection d'un HIC sur un domaine est activée par le domaine local, le HIC doit, soit traiter des entrées, produire des sorties ou soit mettre à jour son état interne.

L'algorithme d'établissement du programme du domaine hétérogène s'assure que toutes les projections d'un HIC qui prennent des entrées sont activées avant que n'importe quelle projection qui doit produire des sorties soit activée. Cependant, quand nous concevons un HIC, nous ne savons pas dans quel ordre ses entrées seront disponibles parce que nous ne savons pas comment il sera projeté sur les domaines qu'il utilise [27].

Il n'est pas possible de spécifier le comportement du HIC globalement pour chacun de ses domaines, parce qu'il peut se produire que plusieurs ports qui utilise le même MoC soient projetées dans différents sous-ensembles.

Par conséquent, la seule solution pour spécifier le comportement d'un HIC est de spécifier comment son état est mis à jour pour chaque ensemble possible d'entrées connues, et de spécifier comment calculer ses sorties à partir de ses entrées connues et de l'état courrant. Ceci rend la programmation de HICs moins simple que la programmation des composants réguliers parce que le code doit vérifier quelles entrées sont connues avant de les traiter.

5.8 Conclusion partielle

Nous avons proposé le Composant à Interface Hétérogène qui a des entrées et des sorties qui utilisent différents modèles de calcul. Ce composant peut être utilisé comme pont entre les « îles » homogènes dans la modélisation des systèmes hétérogènes.
De plus, de part son opération *computeBehavior*(), il offre un cadre d'implémentation explicite du comportement hétérogène du système à la frontière des différents modèles de calcul hétérogènes qu'il utilise.

En effet, après le partitionnement du système, le composant à interface hétérogène est projeté dans différents sous-systèmes et y réside comme un acteur homogène. Ses entités projections l'activent pour calculer le comportement hétérogène à la frontière de différents modèles de calcul.
Or, puisque ces différentes entités utilisent les mêmes opérations, dans le but de maintenir la chaîne de causalité entre elle, nous avons élaboré un mécanisme d'activation approprié.

Le premier point important concernant ce Composant à Interface Hétérogène est sa faculté de se projeter dans des différents domaines. Ce qui, du point de vue structurel, permet de considérer un HIC comme une agrégation virtuelle d'une entité HIC originale et de ses entités projections. En d'autres mots ceci permet d'avoir plusieurs HICs en Un, ou encore d'avoir un HIC sous plusieurs formes.

Le deuxième point important concernant ce Composant à Interface Hétérogène est le partitionnement de ses opérations. Bénéficiant du partitionnement structurel, ses différentes opérations sont harmonieusement réparties dans ses différentes entités. Chacune des opérations issues de l'ensemble de ses opérations est donc dédiée à l'une ou l'autre de ses entités, conformément au principe de la séparation des préoccupations.

En somme, sa caractéristique d'omniprésence conjuguée avec sa faculté de s'homogénéiser sémantiquement dans un sous-système lui permet de se comporter comme un composant homogène dans tous les domaines où il est projeté.

Enfin, les activations des HICs ainsi que les activations des différents sous-systèmes doivent être coordonnées par un composant supérieur. Tel est l'objet du chapitre suivant.

Chapitre 6

Modèle d'Exécution Hétérogène Non-Hiérarchique

Résumé -Dans ce chapitre, nous présentons la modélisation du Modèle d'exécution Hétérogène Non-Hiérarchique. Cette modélisation est présentée à travers ses trois phases : le partitionnement, l'ordonnancement et l'exécution. Le Modèle d'exécution Hétérogène commence par le partitionnement où il divise le système à la frontière de son comportement hétérogène, c'est-à-dire, à la frontière de plusieurs modèles de calcul qu'il utilise et génère les sous-systèmes homogènes, déconnecte et reconnecte les ports et gère les ports virtuels. Ensuite, il ordonnance les sous-systèmes. Enfin il active les sous-systèmes selon un ordonnancement préalablement établi

6.1 Introduction

Dans le chapitre précédent, nous avons vu que notre approche hétérogène non-hiérarchique était basée sur deux principaux composants à savoir : *un composant à interface hétérogène et un modèle d'exécution hétérogène.*

Pour le composant à interface hétérogène présenté au chapitre 4, hormis sa capacité classique de communiquer de manière homogène car disposant d'entrées et de sorties hétérogènes, son rôle demeure aussi d'offrir un cadre d'implémentation explicite du comportement hétérogène du système à la frontière des différents modèles de calcul hétérogènes adjacents qu'il utilise.
Le modèle d'exécution hétérogène vient jouer un rôle plus structurel sur le plan de la décomposabilité et de la recomposabilité modulaire, et, également au niveau de l'exécution du système.

En effet, pendant la phase d'initialisation du système, le modèle d'exécution hétérogène commence par le partitionnement de celui-ci à la frontière des différents comportements hétérogènes implémentés par les différents composants à interface hétérogène, c'est-à-dire, à la frontière des différents modèles de calcul adjacents utilisés par les différents HICs.
Il met à « plat » le système en créant des sous-systèmes et en les faisant chacun doter d'un modèle d'exécution régulier approprié. Ensuite, puisqu'il est déchargé du contrôle interne des sous-systèmes, il délègue le flot de contrôle et le comportement des sous-systèmes à chacun de leurs modèle d'exécution. Il s'ensuit que, dans chacun de ces sous-système, un modèle de calcul unique s'applique.

Il est par ailleurs important de souligner que le partitionnement utilisé dans cette approche n'est dans le sens de [134] dont le but est de découper la fonctionnalité d'un système en un ensemble de partitions où chaque partition peut être exécutée soit en logiciel, soit en matériel.

Les composants à interface hétérogène appartenant à la limite de plusieurs modèles de calcul sont projetés dans chaque sous-système dans lesquels appartiennent leurs ports et les autres acteurs sont transférés à leurs sous-systèmes associés. Le modèle d'exécution hétérogène copie les connexions originales dans les différents sous-systèmes. Afin d'assurer l'ordonnancement interne dans chaque sous-système, il génère des dépendances virtuelles entre les différentes projections du composant à interface hétérogène dans chacun des sous-systèmes.

Ensuite, pour l'activation de ces différents sous-systèmes, le modèle d'exécution hétérogène ordonnance leur activation en déléguant leur ordonnancement interne à leur modèles d'exécution réguliers respectifs.

Enfin, il exécute le système conformément à l'ordonnancement opéré au préalable. Pendant cette exécution, du fait que tous les sous-systèmes utilisent une abstraction non-hiérarchique, la communication entre eux passent via les canaux de communication hétérogènes abstraits qui ont été générés automatiquement pendant la restructuration du système en sous-systèmes.

Ce chapitre est donc consacré à la modélisation de ce modèle d'exécution hétérogène non-hiérarchique. Cette modélisation se traduit par une représentation de ces différentes phases à savoir :

- Partitionnement du système hétérogène,
- Ordonnancement des sous-systèmes,
- Exécution d'une itération

6.2 Partitionnement d'un système hétérogène

Comme souligné dans l'introduction de ce chapitre, le partitionnement utilisé dans cette approche n'a pas comme but le découpage matériel-logiciel.
Dans notre approche, le modèle d'exécution partitionne le système par génération des composants composites représentant les différents sous-systèmes que nous notons Ω_i. Ce partitionnement est fait à la frontière des différents modèles de calcul utilisés par les composants à interfaces hétérogènes [98]. Ces composants composites vont englober les acteurs et les HICs générateurs de ce sous-système en respectant la dépendance causale des données. Et à l'intérieur de ces sous-systèmes, un modèle d'exécution régulier est appliqué. Ce mécanisme de partitionnement s'effectue en trois étapes à savoir :

– Génération des sous-systèmes et placement des acteurs,
– Déconnexion et Reconnection des ports
– Dépendances ad-hoc et ports virtuels

6.2.1 Précédence entre les projections

Au chapitre 4, nous avons montré que notre modèle d'exécution ignore la sémantique des modèles de calcul utilisés par les sous-systèmes. Il compte cependant sur les ordonnanceurs des domaines des sous-systèmes pour ordonnancer les acteurs dans les sous-systèmes.
Toutefois, il doit ordonnancer les activations des sous-systèmes qu'il a crées pendant le partitionnement du système. En ce qui concerne la précédence entre les sous-systèmes, elle est induite par les canaux abstraits hétérogènes entre les projections de HIC et par les canaux abstraits homogènes entre les acteurs relais.

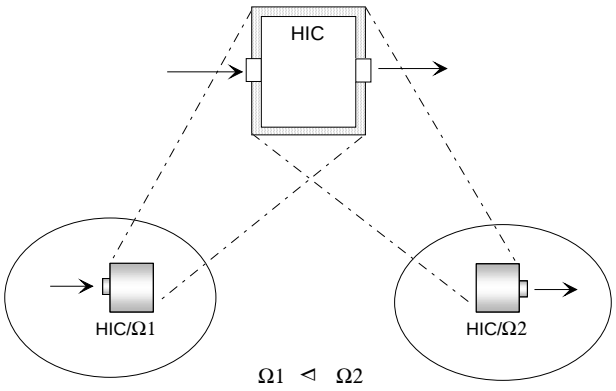


FIG. 6.1 – Dépendance induite par la causalité du HIC

Principe 6.2.1 *Nous disons qu'un sous-système Ω_1 précède un autre sous-système Ω_2 et nous notons $\Omega_1 \triangleleft \Omega_2$ si et seulement si :*
– *Ω_1 contient un relais de sortie et Ω_2 contient le relais d'entrée correspondant*
– *Ω_1 contient une projection de HIC qui contient un port input et Ω_2 contient une autre projection du même HIC contenant un port output.*

La figure 6.1 montre comment la relation de causalité entre les entrées et les sorties de HIC induit les précédences entre les sous-systèmes.
Pour être capable de produire des données correctes, la projection de HIC dans Ω_2 doit connaître le résultat de la réaction de HIC à ses entrées. Le HIC reçoit des données sur ses entrées dans le sous-système Ω_1, ainsi, ce sous-système doit être activé avant le sous-système Ω_2.

6.2.2 Génération des sous-systèmes et placement des acteurs

L'algorithme de partitionnement du système minimise l'utilisation des canaux homogènes abstraits en examinant chaque composant dans un ordre qui soit compatible avec le tri topologique du système [98] [27].

Ensuite, pour chaque acteur A_i lu à partir de ce trie, le modèle d'exécution hétérogène vérifie l'existence d'un éventuel sous-système utilisant le modèle de calcul de A_i. Si un tel sous-système existe, il place l'acteur dans ce sous-système si les contraintes liées aux dépendance de données le permettent. Sinon, il active la création d'un nouveau sous-système dans lequel il place l'acteur A_i.

Cependant, le fait qu'un acteur A_i par exemple utilise le même modèle de calcul qu'un sous-système S_j n'implique pas forcément son placement dans ce sous-système. Pour que A_i soit placé dans S_j, les conditions suivantes doivent être remplies :

– l'acteur A_i doit effectivement utiliser le même modèle de calcul que le sous-système S_j,
– il ne doit pas exister un chemin entre l'acteur A_i et un quelconque acteur du sous-système S_j passant par un HIC

Ces conditions doivent être respectées pour éviter le croisement de dépendances entre sous-systèmes issus du partitionnement. Ceci leur permet de pouvoir être ordonnancés.

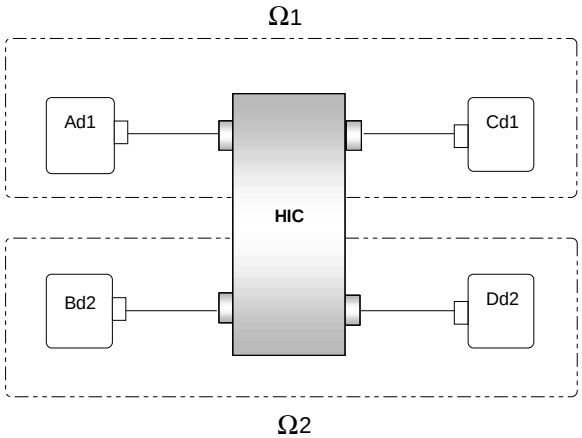


FIG. 6.2 – Ω_1 et Ω_2 ne peuvent pas être ordonnancés

Par exemple sur la figure 6.2, l'acteur A_{d1} et C_{d1} utilisent le même modèle de calcul d1 et les acteurs B_{d2} et D_{d2} utilisent le même modèle de calcul d2.
Si l'on met les acteurs A_{d1} et C_{d1} dans le même sous-système Ω_1 et les acteurs B_{d2} et D_{d2} dans le même sous-système Ω_2, il en résulte :

– la projection de HIC dans le sous-système Ω_1 doit être activé après l'activation de l'acteur B qui est dans le sous-système Ω_2 et
– la projection de HIC dans le sous-système Ω_2 doit être activé après l'activation de l'acteur A qui est dans le sous-système Ω_1.

D'où l'impossibilité d'ordonnancer les deux sous-systèmes Ω_1 et Ω_2. Dans ce cas, l'algorithme de partitionnement construira quatre sous-systèmes dont chacun contiendra un acteur et une projection de HIC. Toutefois, il est possible d'optimiser ce partitionnement à trois sous-systèmes en cassant la dépendance croisée comme sur la figure 6.3.

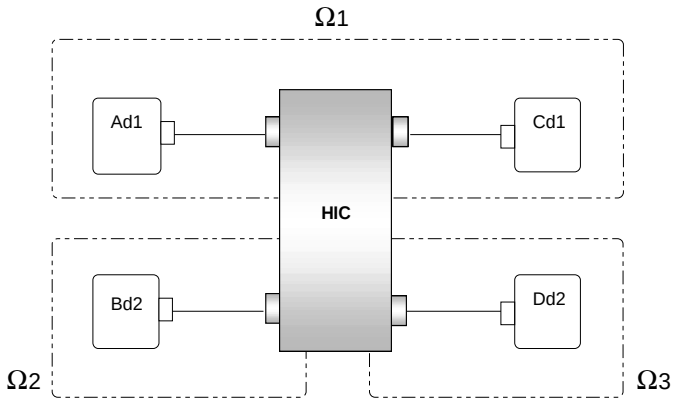


FIG. 6.3 – Un partitionnement optimum du système

Lorsque ces conditions sont remplies pour un acteur, du fait que plusieurs sous-systèmes peuvent les remplir, il se pose le problème du choix du sous-systèmes dans lequel l'acteur sera placé. Pour cela nous avons adopté le principe suivant :

Principe 6.2.2 *Lorsque plusieurs sous-systèmes satisfont aux conditions de placement d'un acteur, nous choisissons de placer l'acteur en priorité dans le sous-système qui contient déjà une projection d'un HIC appartenant au même segment que cet acteur. Sinon, s'il n'en existe pas un, cet acteur sera placé dans le dernier sous-système crée utilisant le même modèle de calcul.*

Algorithme de partitionnement

Une conséquence de cet algorithme est qu'il ne peut pas y avoir des boucles dans le graphe du système. C'est le prix à payer pour supporter n'importe quel modèle de calcul. L'ordonnancement d'un système qui contient des boucles dépend de la sémantique du modèle de calcul. Une manière de casser des boucles de dépendance est d'insérer le retard dans la boucle. Cependant la sémantique du retard est elle-même très dépendante de la sémantique du modèle de calcul. Par conséquent, si nous considérons les sous-ensembles et leurs modèles de calcul en tant que boîtes noires, nous ne pouvons pas permettre des boucles de dépendance entre les sous-ensembles. Toutefois, si une boucle est locale à un sous-ensemble et si le modèle de calcul correspondant soutient des boucles, la boucle est acceptée et sa sémantique sera donnée par le modèle de calcul du sous-ensemble.

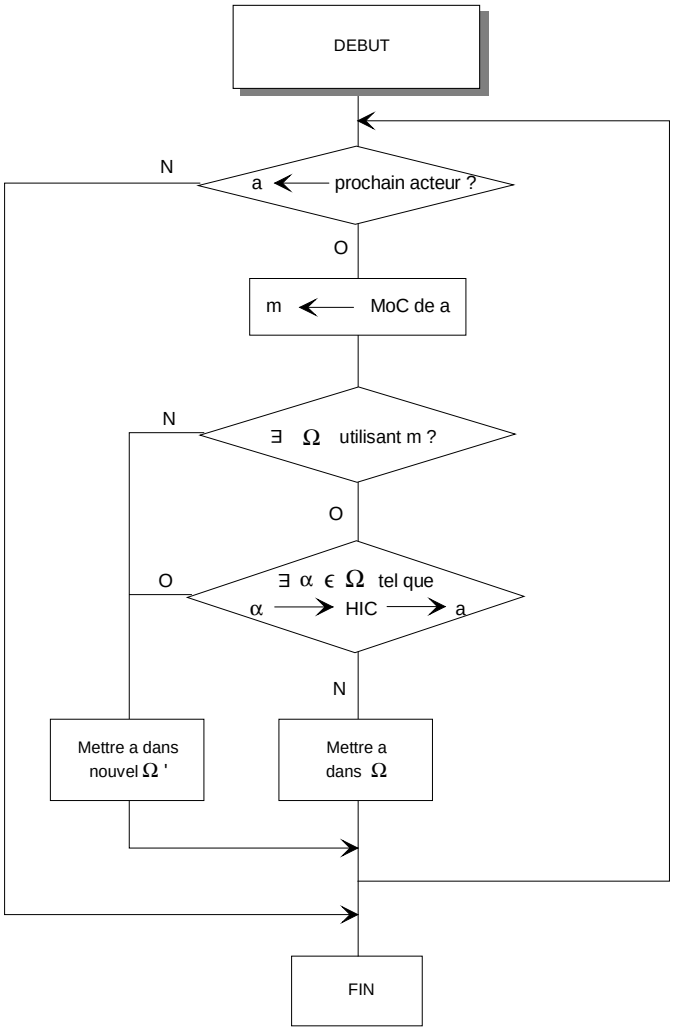
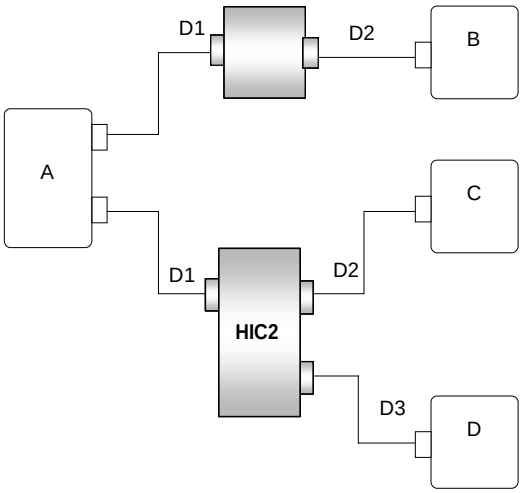


FIG. 6.4 – Algorithme de partitionnement

6.2.3 Dépendances ad-hoc et ports virtuels

A la section 4.5.1 nous avons vu que notre modélisation non-hiérarchique utilise une abstraction non-hiérarchique. Du fait de cette abstraction, les différents sous-systèmes, puisque dépourvus de ports, ne peuvent cependant pas être connectés entre eux. Ceci peut engendrer une situation de non-dépendance de projections entre elles dans un sous-système.



(b) : Système après le partitionnement
(Dans le sous-système Ω2, il ya création des ports virtuels et leur connexion)

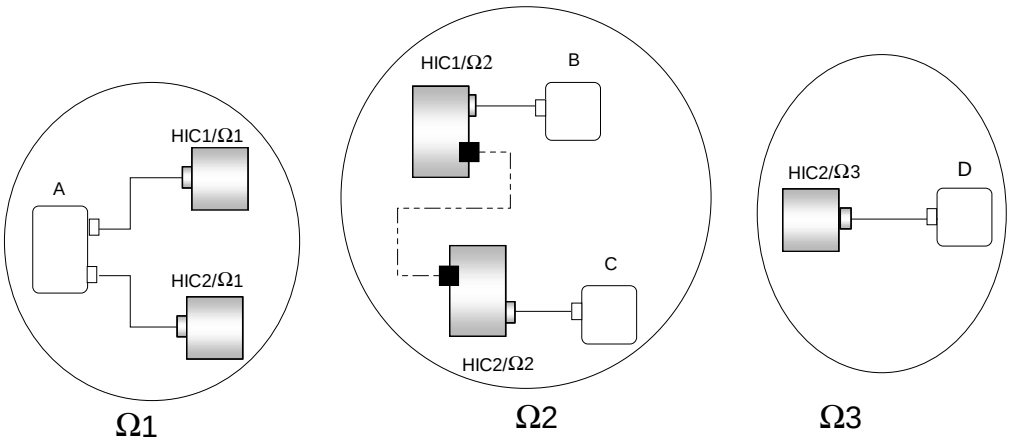


FIG. 6.5 – Exemple d'une connexion virtuelle

Par exemple le système montré sur la figure 6.5 est divisé en trois sous-systèmes. Le second sous-système n'est pas complètement connecté à l'intérieur parce que HIC1 et

HIC2 sont complètement indépendants l'un de l'autre.

Or, certains domaines ne supportent pas des systèmes non connectés. Nous allons connecter ce sous-système sans changer son comportement.

Pour cela, nous allons forcer une dépendance ad-hoc entre les projections des HICs en les connectant par le canal en pointillé à travers les ports qui seront ignorés par ces HICs. Du point de vue du domaine, ce sous-système sera vu comme un système réellement connecté. Nous appelons ces ports, *« Ports virtuels »*.

6.2.4 Déconnexion et reconnection des ports

Lorsqu'un acteur est placé dans un sous-système quelconque, le modèle d'exécution hétérogène n'a fait que le retirer du modèle original pour le placer dans ce sous-système du nouveau modèle qui le prend en charge. Mais, les connexions à ses ports ne sont nullement modifiées.

Pour que le nouveau modèle se comporte comme le modèle initial, avant son exécution, le modèle d'exécution déconnecte toutes les connections de tous les acteurs pour les reconnecter à nouveau dans chaque sous-système conformément aux canaux de communication du modèle initial.

6.3 Ordonnancement des sous-systèmes

6.3.1 Ordonnancement des sous-systèmes

L'ordonnancement des sous-systèmes devrait être fait conformément aux précédences induites par les HICs et conformément aux précédences induites par les relais utilisés pour préserver les canaux homogènes de communication à travers les sous-systèmes.

Cependant, à cause de la raison qui a conduit à la création des relais, la précédence induite par les relais sur les sous-systèmes est toujours induite par les HICs.
Ainsi, il est suffisant de ne considérer simplement que la précédence induite par les HICs pour l'ordonnancement des sous-systèmes.

Une fois que le système hétérogène est partitionné, le modèle d'exécution construit un squelette du système partitionné, exclusivement composé que des projections des HICs et de leurs dépendances.

Un tri topologique de ce squelette est alors utilisé pour déterminer la relation de précédence sur les sous-systèmes, et, n'importe quel ordre compatible à cette relation de précédence offre un ordonnancement possible des sous-systèmes.

6.3.2 Exemple de partitionnement et d'ordonnancement hétérogène

Considérons le système hétérogène de la figure 6.6 contenant dix acteurs et utilisant deux MoCs D1 et D2.

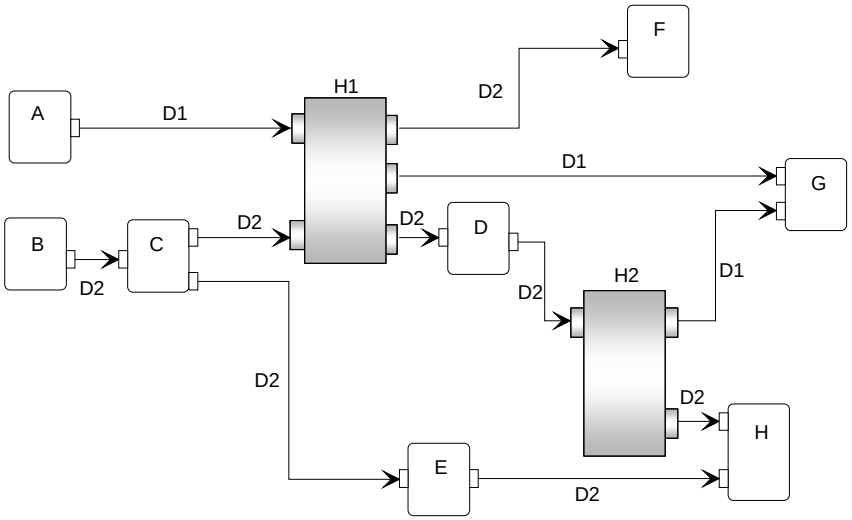


FIG. 6.6 – Exemple d'un système hétérogène

Selon les règles de partitionnement, les acteurs sont groupés comme sur la figure 6.7.

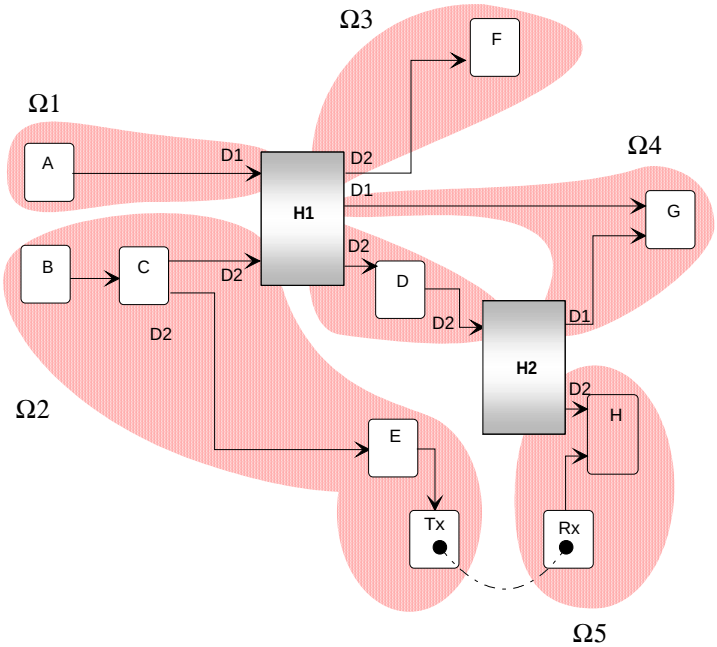


FIG. 6.7 – Vu du système selon les règles de partitionnement

– L'acteur A ne peut pas être mis dans le même sous-système qu'un autre acteur du système du fait que les acteurs qui sont du côté des entrées de $H1$ n'appartiennent pas au même domaine que A, et, les acteurs appartenant au même domaine que l'acteur A sont du côté des sorties de $H1$.
Ainsi, l'acteur A sera seul avec la projection de H_1 dans ce sous-système.

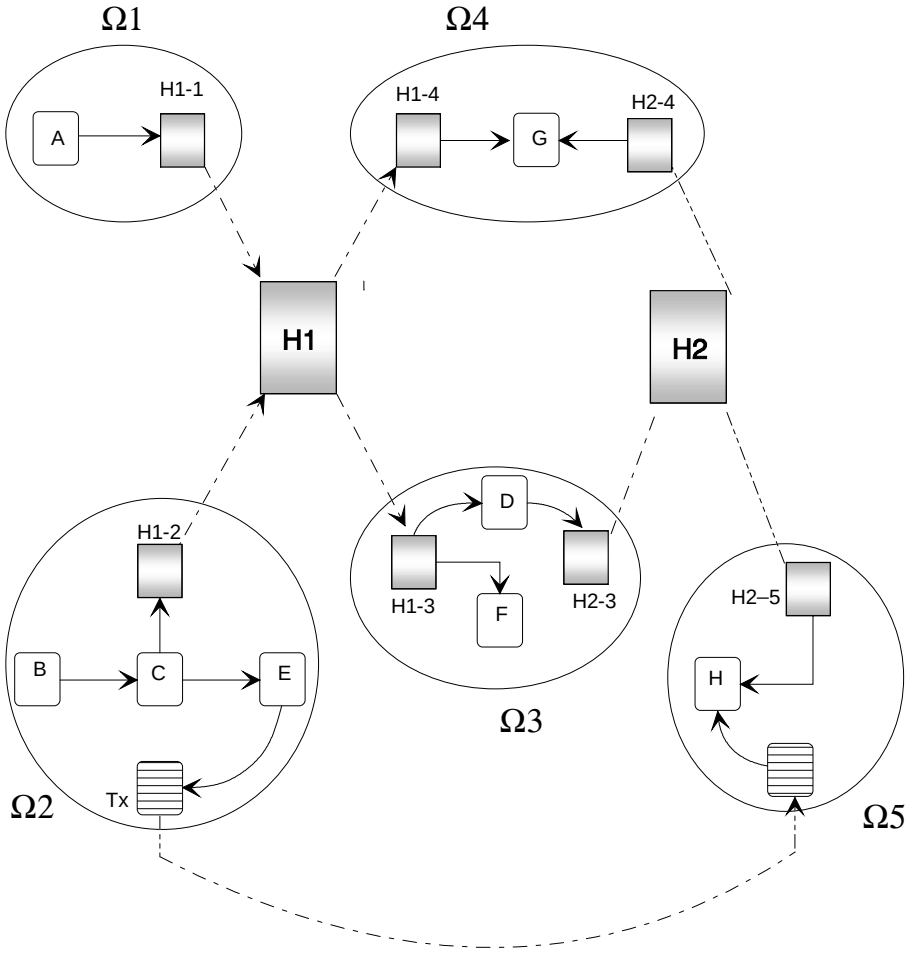


FIG. 6.8 – Système hétérogène partitionné

– Les acteurs B, C et E sont dans le domaine D_2 et seront mis dans le même sous-système. Cependant les acteurs D et H ne peuvent pas être mis dans le sous-système précédant car il y a un chemin entre les acteurs C et D passant par H_1, et il y a également un chemin entre les acteurs C et H passant par H_1 et H_2.

– L'acteur F peut être mis avec les acteurs D et E, car, ils appartiennent au même domaine et ne communiquent pas à travers un HIC.

– L'acteur G est le seul qui appartient au domaine D_1 du côté des sorties de H_1. Ainsi, il sera aussi seul dans son sous-système avec les projections de H_1 et de H_2.

– Puisque les acteurs E et H sont connectés mais ne sont pas placés dans le même sous-système, il y aura deux acteurs relais T_x et R_x qui prendront en charge leurs communications sur ce canal abstrait homogène.

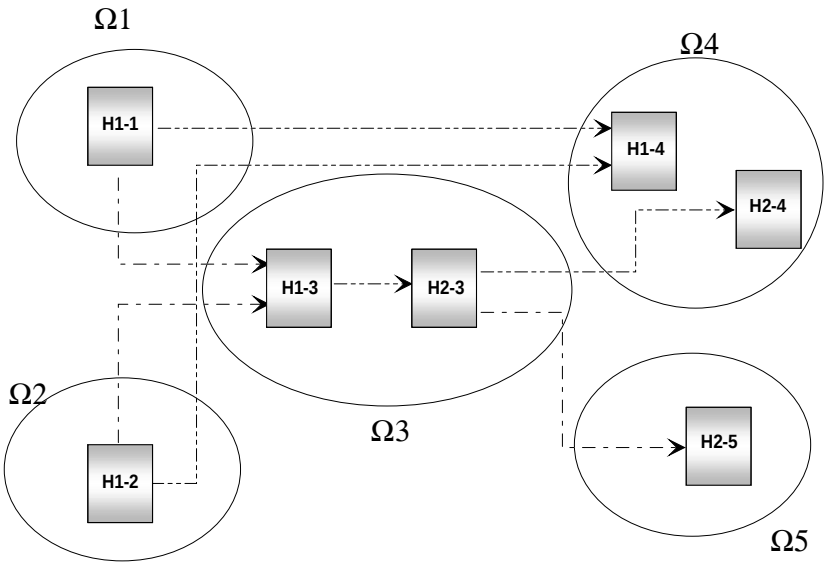


FIG. 6.9 – Squelette du système de l'exemple

D'où le partitionnement non-hiérarchique montré sur la figure 6.8 et le squelette de ce partitionnement est montré sur la figure 6.9 et donne les relations de précédence suivantes :

$$\Omega_1 \lhd \Omega_3 \qquad \Omega_2 \lhd \Omega_3 \qquad \Omega_3 \lhd \Omega_4$$

$$\Omega_1 \lhd \Omega_4 \qquad \Omega_2 \lhd \Omega_4 \qquad \Omega_3 \lhd \Omega_5$$

Ce qui permet les ordonnancements suivants :

$$\Omega_1 \quad \Omega_2 \quad \Omega_3 \quad \Omega_4 \quad \Omega_5$$

$$\Omega_1 \quad \Omega_2 \quad \Omega_3 \quad \Omega_5 \quad \Omega_4$$

$$\Omega_2 \quad \Omega_1 \quad \Omega_3 \quad \Omega_4 \quad \Omega_5$$

$$\Omega_2 \quad \Omega_1 \quad \Omega_3 \quad \Omega_5 \quad \Omega_4$$

6.4 Exécution d'une itération hétérogène

L'ordonnancement établi dans la précédente phase permet d'activer les sous-systèmes selon un séquencement bien défini. Lorsque ce séquencement à fait un cycle complet, nous parlons d'une *«itération»* du système.
Du fait que nous avons imposé au modèle d'exécution hétérogène de nullement avoir connaissance des modèles de calcul utilisés par ses sous-systèmes, le dialogue entre lui et lesdits sous-systèmes se résume aux activations et aux calls-back.

En effet, ces activations se traduisent par un premier type de signaux de contrôle allant du modèle d'exécution hétérogène vers les sous-systèmes et également vers les HICs.
Pour un sous-système Ω_i et pour un HIC quelconque, leurs ensembles des signaux de contrôle qu'utilise le modèle d'exécution pour les contrôler sont respectivement :

$$\Omega_i\text{Ctrl} = \{initialisation(),\ preCondition(),\ trigger(),\ postCondition()\}$$

$$\text{Hic.Ctrl} = \{initialisation(),\ preCondition(),\ trigger(),\ postCondition()\}$$

Dans l'autre sens, un deuxième type de signaux de calls-back allant des sous-systèmes et des HICs vers modèle d'exécution hétérogène. Par ces signaux, un sous-système ou une entité HIC signifie au modèle d'exécution hétérogène qu'il est soit prêt ou pas à être lancé, ou soit qu'il vient de terminer ses activités. Le HIC rajoute son opération d'obtention du modèle d'exécution.
Pour un sous-système Ω_i et pour un HIC quelconque, leurs ensembles des signaux de call-back sont respectivement :

$$\Omega_i.\text{Clbk} = \{finish()\}$$

$$\text{Hic.Clbk} = \{finish(),\ getModEx()\}$$

Pour représenter l'activation liée à une itération, nous allons utiliser les règles de déclenchement telles suivantes :

$$\begin{array}{rcl}
\text{Init} & \overset{true}{\rightarrow} & \Omega_1.trigger; \\
\Omega_1.finish & \overset{true}{\rightarrow} & \Omega_2.trigger; \\
\vdots & \vdots & \vdots \\
\Omega_j.finish & \overset{true}{\rightarrow} & \Omega_{j+1}.trigger; \\
\vdots & \vdots & \vdots \\
\Omega_{n-1}.finish & \overset{true}{\rightarrow} & \Omega_n.trigger;
\end{array}$$

Il est important de signaler que dans ces règles de déclenchement, nous n'avons pas représenté les activations des HICs par le modèle d'exécution hétérogène. Ceci parce que, l'activation d'un HIC est demandée par sa projection, et, cette demande n'intervient pas

dans le mécanisme d'ordonnancement des sous-systèmes. C'est plutôt les relations de précédence de cette projection qui sont prises en compte.

Afin de simplifier la présentation de l'exécution d'un système hétérogène «plat», nous allons représenter le même système élémentaire de la figure 4.8 ne contenant que deux sous-systèmes Ω_1 et Ω_2 gouvernés par des modèles d'exécution réguliers M1 et M2 comme sur la figure 6.10

Chacun de ces sous-systèmes contient une paire d'acteurs : A1 et l'entité projection HicTx appartenant à Ω_1 et A2 et l'entité projection HicRx appartenant à Ω_2. Mais, Ω_1 et Ω_2 n'ayant pas de port, il n'y a aucun canal de communication réel les reliant. Ils n'exigent donc pas de variable de sous-système autre que les paramètres extérieurs et les acteurs et les canaux qu'ils contiennent.

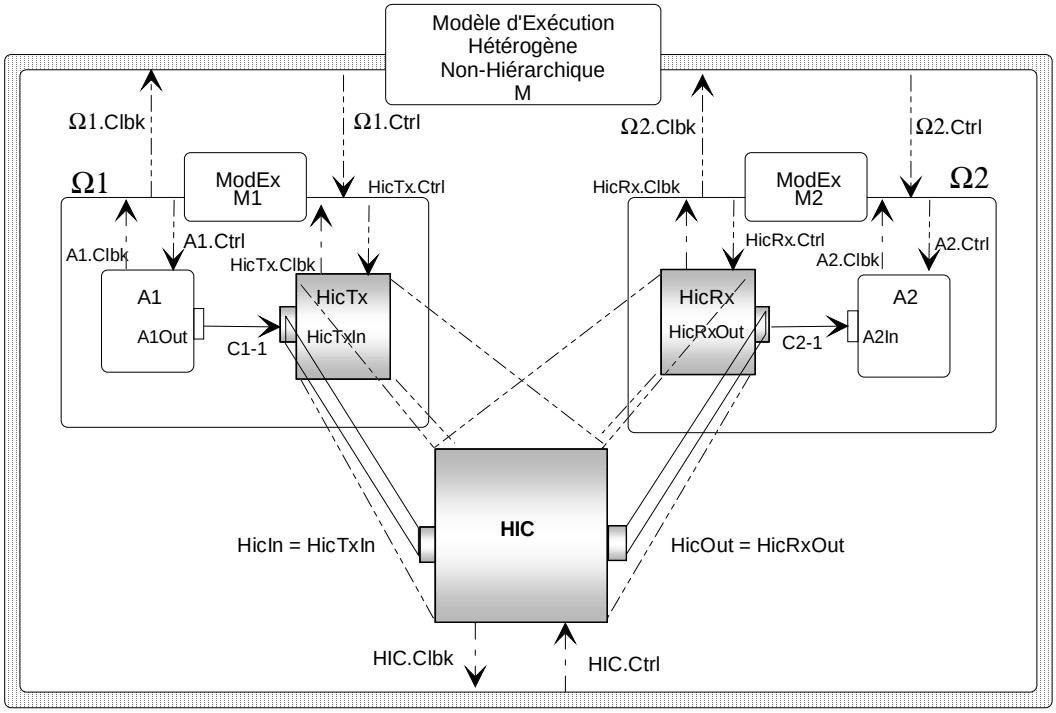


FIG. 6.10 – Variables Ω_1, Ω_2 et HIC de Ω

Comme souligné plus haut, puisque l'exécution du système se résume aux envois des signaux de contrôle du modèle d'exécution vers les sous-systèmes et les HICs et aux envois des calls-back dans l'autre sens, nous allons par la suite analyser ce qui se passe dans les sous-systèmes du point de vue de leur comportement et du point de vue de leur communication.

6.4.1 Exécution dans Ω_1

Variables de Ω_1

L'ensemble des variables de Ω_1 contient l'ensemble de ses variables d'interface et ceux des variables de A1 et de HicTx tel que :

$$\Omega_1.\text{X} = \{\Omega_1.\text{P}, \Omega_1.\text{Q}, \Omega_1.\text{S}\}$$

Or, on sait :
- d'une part, Ω_1 n'a pas de port d'entrée ni de sortie, donc, $\Omega_1.\text{P} = \Omega_1.\text{Q} = \emptyset$,

- et d'autre part, $\Omega_1.\text{S} = \text{A1.X} \cup \text{C1} \cup \text{HicTx.X}$

De plus, sachant que $\text{HicTx.X} = \{HicTxIn,\ stateTx\}$,
on a :

$$\Omega_1.\text{X} = \Omega_1.\text{S} = \{\text{A1.X} \cup \text{C1} \cup \{HicTxIn,\ stateTx\}\}$$

avec

– A1.X : ensemble de variables de A1
– HicTxIn : port d'entrée de HicTx mis en commun avec HicIn,
– C1 : le seul canal de communication de Ω_1.
– stateTx : symbolise l'état de HIC

Comportement et communication dans Ω_1

Le sous-système Ω_1 a un ensemble d'opérations de calcul parmi lesquelles on trouve les opérations du modèle d'exécution M1, celles des acteurs A1 et HicTx et les opérations de flot de contrôle entre M1 et les acteurs A1 et HicTx. Nous notons cet ensemble d'opérations par :

$$\Omega_1.\text{Comp} = \{\text{A1.Comp} \cup \text{HicTx.Comp} \cup \text{A1.Ctrl} \cup \text{HicTx.Ctrl} \cup \text{A1.Clbk} \cup \text{HicTx.Clbk}\}$$

Sachant que :

HicTx.Comp = $\emptyset$,
HicTx.Clbk = $\{finish(),\ getModEx(),\ requestBehaviorComputing()\}$ et
HicTx.Ctrl = $\{trigger(),\ initialisation(),\ preCondition(),\ postCondition()\}$

Ainsi, vu par le modèle d'exécution hétérogène, le sous-système Ω_1 peut être considéré comme un acteur simple ayant un comportement bien déterminé.

En ce qui concerne ses opérations de communication, en exécutant une opération d'écriture à partir de l'acteur A1, la valeur dans le canal C1, est transférée par le modèle d'exécution

local sur le port HicTxIn de l'acteur HicTx. Ainsi, conformément à la sémantique présentée au chapitre 4, nous notons :

HicTxIn.*isFull*	:	lorsque l'acteur HicTx procède à une vérification de la disponibilité de son port HicTxIn
A1Out.write	:	lorsque l'acteur A1 procède à une écriture de données sur le canal C1
HicTxIn.*existData*	:	lorsque l'acteur HicTx procède à une vérification de données transférées sur son port HicTxIn
HicTxIn.*read*	:	lorsque l'acteur HicTx procède à une lecture de données sur son port HicTxIn

Ceci nous permet de définir l'ensemble de communication de Ω_1 :

$$\Omega_1.\text{Comm} = \{\text{HicTxIn}.isFull,\ \text{A1Out}.write,\ \text{HicTxIn}.existData,\ \text{HicTxIn}.read\}$$

Mécanisme d'exécution de Ω_1

L'ensemble d'opérations d'exécution de Ω_1 que nous notons Ω_1.fire est l'union des ses deux ensembles d'opérations de calcul et d'opérations de communication telle que

$$\Omega_1.\text{fire} = \Omega_1.\text{Comp} \bigcup \Omega_1.\text{Comm}$$

Cependant, pour son déclenchement, Ω_1 doit obtenir son point de synchronisation initial et exécuter son second point de synchronisation final à la fin de son exécution.
Ainsi, sa réaction complète sera l'union de son ensemble d'opérations d'exécution Ω_1.fire et ses déclenchements $\Omega_1.trigger()$ et $\Omega_1.finish()$

Par ailleurs, concernant ses activations, Ω_1 sera déclenché par le modèle d'exécution hétérogène. Comme il n'a pas de données d'entrée à consommer, il invoquera simplement son modèle d'exécution local M1 à réagir et à activer les acteurs A1 et HicTx.

Supposons maintenant que le sous-système Ω_1 reçois un déclenchement initial venant du modèle d'exécution M, la règle de déclenchement sera la suivante :

$$\begin{aligned}
\text{Init} &\overset{true}{\rightarrow} \Omega_1.trigger;\\
\Omega_1.trigger &\overset{true}{\rightarrow} \text{A1}.trigger;\\
\text{A1}.finish &\overset{true}{\rightarrow} \text{HicTx}.trigger;\\
\text{HicTx}.trigger &\overset{true}{\rightarrow} \text{HIC}.trigger;\\
\text{HIC}.finish &\overset{true}{\rightarrow} \text{HicTx}.finish;\\
\text{HicTx}.finish &\overset{true}{\rightarrow} \Omega_1.finish;
\end{aligned}$$

Cette exécution est montrée sur la figure 6.11 et sa réaction complète est donnée par le diagramme de Hasse sur la figure 6.12

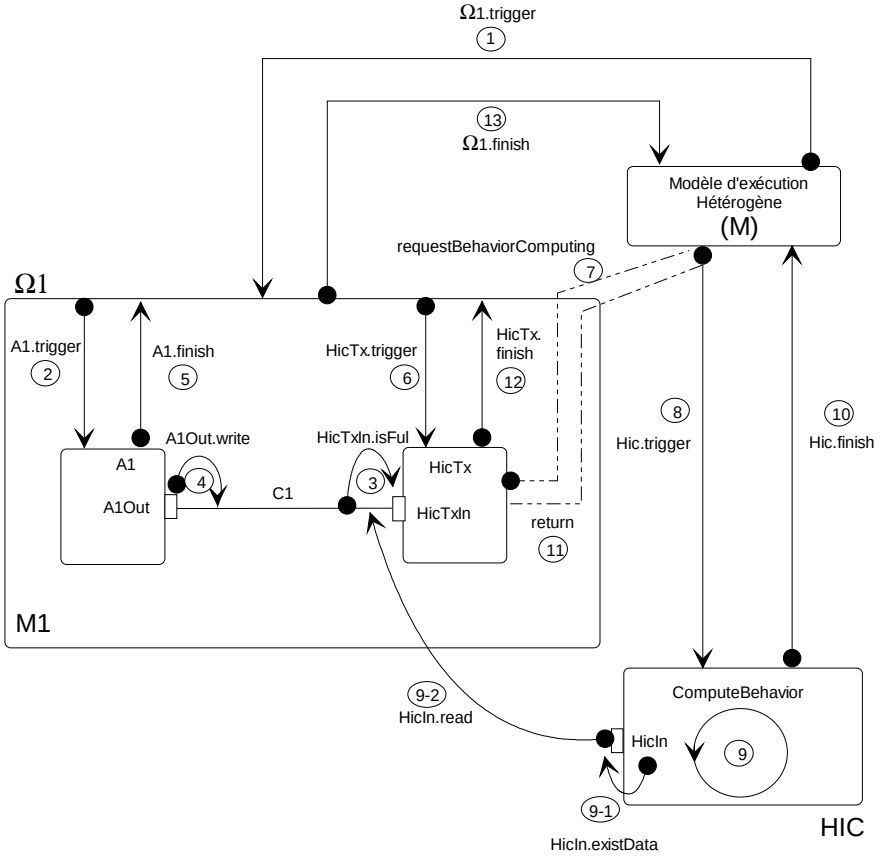


FIG. 6.11 – Exécution complète dans Ω_1

Illustration 6.4.1

1	:	M déclenche Ω_1
2	:	A son tour, Ω_1 lance A1
3	:	A1 teste la disponibilité de HicTx en réception de données
4	:	A1 écrit les données sur le canal C1
5	:	A1 signale à Ω_1 la fin de ses activités
6	:	Ω_1 lance HicTx
7	:	A travers son opération requestBehaviorComputing, HicTx sollicite le lancement de son
	:	HIC original par M
8	:	M lance le HIC
9	:	HIC débute le calcule de son comportement
9-1	:	test éventuel de l'existence de la donnée sur le canal C1 par HIC
9-2	:	lecture éventuelle des données à partir du canal C1 par HIC
10	:	HIC signale à M la fin de ses activités
11	:	M retourne
12	:	HicTx signale à Ω_1 la fin de ses activités
13	:	Ω_1 signale à M la fin de ses activités

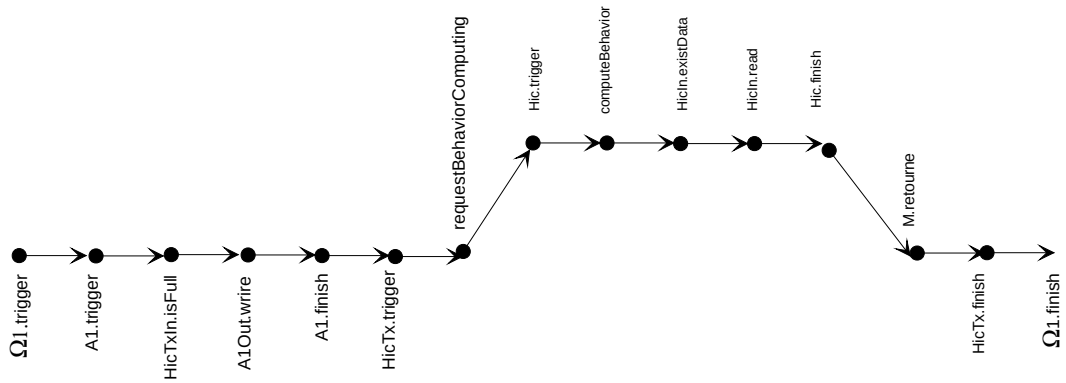


FIG. 6.12 – Réaction complète de Ω_1 représentée par un diagramme de Hasse

6.4.2 Exécution dans Ω_2

Variables de Ω_2

Comme pour Ω_1,

$$\Omega_2.\mathrm{X} = \{\Omega_2.\mathrm{P}, \Omega_2.\mathrm{Q}, \Omega_2.\mathrm{S}\}$$

De même, on sait :
- d'une part, Ω_2 n'a pas de port d'entrée ni de sortie, donc, $\Omega_2.\mathrm{P} = \Omega_2.\mathrm{Q} = \emptyset$,

- et d'autre part, $\Omega_2.\mathrm{S} = \mathrm{HicRx.X} \cup \mathrm{C2} \cup \mathrm{A2.X}$

De plus, sachant que $\mathrm{HicRx.X} = \{HicRxOut,\ stateRx\}$,
on a :

$$\Omega_2.\mathrm{X} = \Omega_2.\mathrm{S} = \{\{HicRxOut,\ stateRx\} \cup \mathrm{C2} \cup \mathrm{A2.X}\}$$

avec

– A2.X : ensemble de variables de A2
– HicRxOut : port de sortie de HicRx mis en commun avec HicOut,
– C2 : le seul canal de communication de Ω_2,
– stateRx : symbolise l'état de HIC

Comportement et Communication de Ω_2

L'ensemble d'opérations de calcul de Ω_2 est similaire à celui de Ω_1 :

Ω_2.Comp= {A2.Comp ∪ HicRx.Comp ∪ A2.Ctrl ∪ HicRx.Ctrl ∪ A2.Clbk ∪ HicRx.Clbk}

Sachant que :

HicRx.Comp = ∅,
HicRx.Clbk = {$finish()$, $getModEx()$, $requestBehaviorComputing()$} et
HicRx.Ctrl = {$trigger()$, $initialisation()$, $preCondition()$, $postCondition()$}

De même, vu par le modèle d'exécution hétérogène, le sous-système Ω_2 peut être également considéré comme un acteur simple ayant un comportement bien déterminé.

Son ensemble d'opérations de communication est :

$$\Omega_2\text{.Comm} = \{\text{A2In}.isFull,\ \text{HicRxOut}.write,\ \text{A2In}.existData,\ \text{A2In}.read\}$$

Mécanisme d'exécution de Ω_2

L'ensemble d'opérations d'exécution de Ω_2 que nous notons Ω_2.fire est également similaire à celui de Ω_1. Il est l'union des ses deux ensembles d'opérations de calcul et de communication telle que

$$\Omega_2\text{.fire} = \Omega_2\text{.Comp} \bigcup \Omega_2\text{.Comm}$$

De même, pour son déclenchement, Ω_2 doit obtenir son point de synchronisation initial et exécuter son second point de synchronisation final à la fin de son exécution.
Ainsi, une réaction complète de Ω_2 est l'union de l'ensemble Ω_2.fire et ses déclenchements $\Omega_2.trigger()$ et $\Omega_2.finish()$

Comme pour Ω_1, concernant ses activations, Ω_2 sera déclenché par le modèle d'exécution hétérogène. Comme il n'a pas non plus de données d'entrée à consommer, il invoquera simplement son modèle d'exécution local M2 à réagir et à activer les acteurs HicRx et A2.

Supposons maintenant que le sous-système Ω_2 reçois un déclenchement initial venant du modèle d'exécution M, la règle de déclenchement sera la suivante :

$$\begin{aligned}
\text{Init} &\overset{true}{\rightarrow} \Omega_2.trigger;\\
\Omega_2.trigger &\overset{true}{\rightarrow} \text{HicRx}.trigger;\\
\text{HicRx}.trigger &\overset{true}{\rightarrow} \text{HIC}.trigger;\\
\text{HIC}.finish &\overset{true}{\rightarrow} \text{HicRx}.finish;\\
\text{HicRx}.finish &\overset{true}{\rightarrow} \text{A2}.trigger;\\
\text{A2}.finish &\overset{true}{\rightarrow} \Omega_2.finish;
\end{aligned}$$

La figure 6.13 montre cette exécution et sa réaction complète est donnée par le diagramme de Hasse sur la figure 6.14

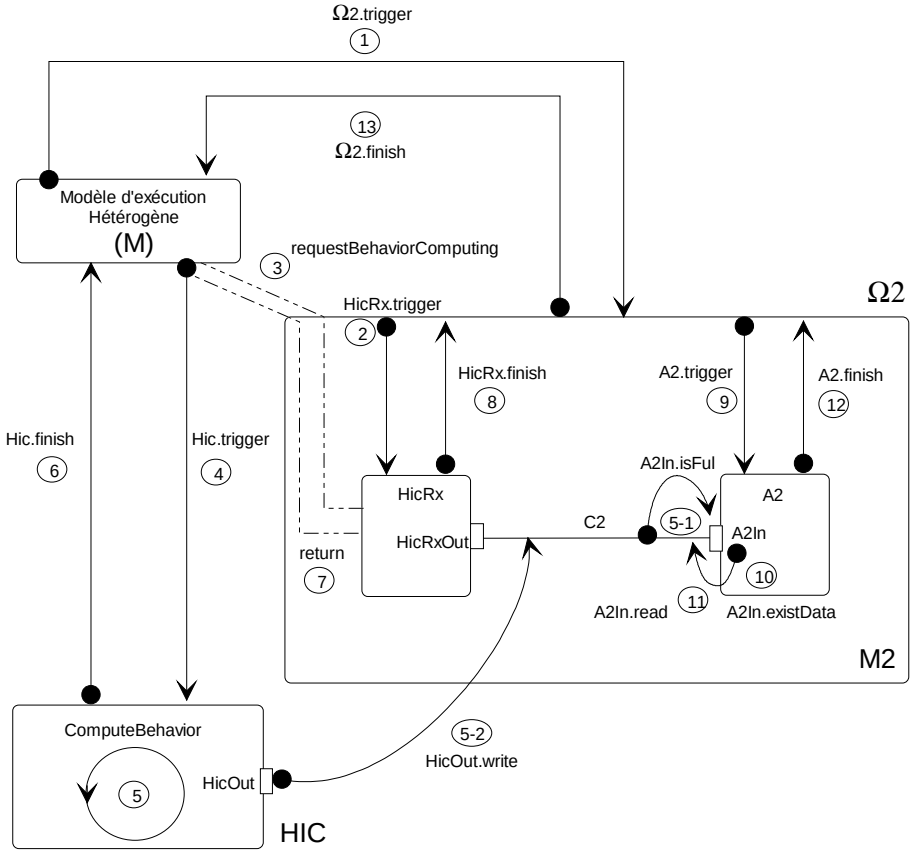


FIG. 6.13 – Exécution complète dans Ω_2

Illustration 6.4.2

1	:	M déclenche Ω_2
2	:	A son tour, Ω_2 lance HicRx
3	:	A travers son opération requestBehaviorComputing, HicRx sollicite le lancement de son
	:	HIC original par M
4	:	M lance le HIC
5	:	HIC débute le calcule de son comportement
5-1	:	test éventuel de la disponibilité réception des données de A2In
5-2	:	écriture éventuelle des données sur le canal C2 par HIC
6	:	HIC signale à M la fin de ses activités
7	:	M retourne
8	:	HicRx signale à Ω_2 la fin de ses activités
9	:	M lance le A2
10	:	A2 teste l'existence des données sur le canal C2
11	:	A2 lit les données du canal C2
12	:	A2 signale à Ω_2 la fin de ses activités
13	:	Ω_2 signale à M la fin de ses activités

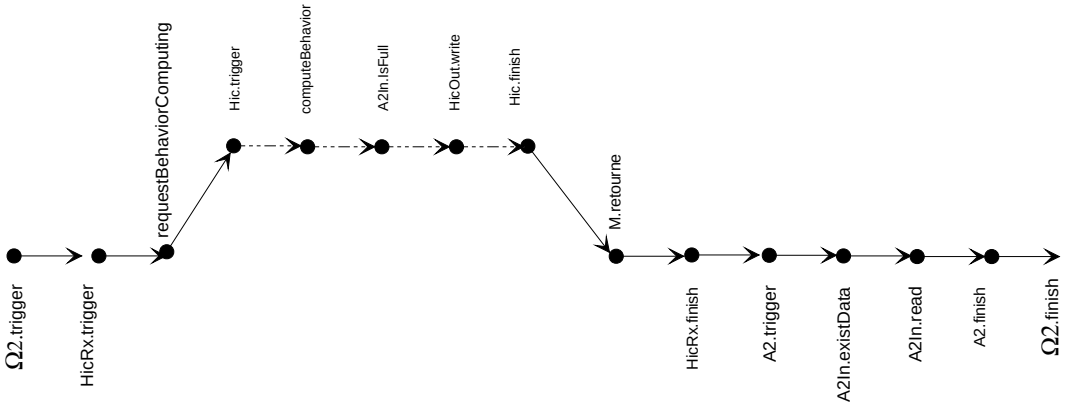


FIG. 6.14 – Réaction complète de Ω_2 par un diagramme de Hasse

6.5 Conclusion partielle

Nous avons conçu le modèle d'exécution hétérogène non-hiérarchique qui supporte les composants à interface hétérogène, HICs.

Ce modèle d'exécution hétérogène divise un système hétérogène à la frontière de ses modèles de calcul en plusieurs sous-systèmes homogènes. Dans ces sous-systèmes, il projette les différents HICs et place les différents acteurs en déléguant leur gestion à leurs modèles d'exécution réguliers. D'où la mise à « plat » complet d'un système hétérogène.
A ce niveau le système est vu comme un ensemble de modules homogènes communicants et situés au même et au seul niveau hiérarchique. Cette communication entre les différents sous-systèmes ne suit plus une profondeur hiérarchique « orthogonalisée », mais, elle passe plutôt de manière « horizontale » d'un sous-système à un autre à travers des canaux homogènes abstraits et des canaux hétérogènes abstraits.

Après cette étape, le modèle d'exécution hétérogène ordonnance ces sous-systèmes avant leur exécution sans tenir compte des sémantiques de leurs modèles de calcul respectifs.

Le premier point clé de la modélisation de ce modèle d'exécution est que d'une part, il est spécifié de manière à être complètement aveugle des modèles de calcul utilisés par ses sous-systèmes. D'autre part, il est également spécifié de manière à être totalement déchargé du mécanisme de transport des données d'un acteur vers un autre.
En effet, ces deux caractéristiques alliées à son habilité à manier les canaux abstraits hétérogènes sans intervenir sur les protocoles de communications utilisés, lui donne la capacité de supporter tous les modèles d'exécution homogènes.

Le second point clé de la modélisation de ce modèle d'exécution est en effet, le fait

qu'il soit spécifié à un niveau d'abstraction élevé, il ne fait aucune hypothèse sur les contraintes ni sur le fonctionnement d'une plate-forme particulière. De ce fait, ce modèle d'exécution peut être intégré avec facilité dans différentes plates-formes sans modification de l'architecture existante.

Néanmoins signalons que dans sa phase conceptuelle actuelle, les boucles ne sont pas encore supportées au niveau hétérogène.

Troisième partie

Intégration, Validation et Simulation dans PTOLEMY II

Chapitre 7

Modélisation intermédiaire : Utilisation du formalisme UML

Résumé -Dans ce chapitre nous proposons une manière proche de la réalisation pour présenter les composants d'appui à l'hétérogénéité non-hiérarchique. Nous utilisons le langage UML comme formalisme intermédiaire de modélisation et traduisons l'abstraction des composants d'appui en représentation UML. Ce chapitre transitoire que nous situons entre le haut niveau d'abstraction et la mise en oeuvre est donc un préalable au chapitre suivant qui sera consacré à l'implémentation et à la validation par simulation.

7.1 Introduction

Dans les chapitres cinq et six, nous avons modélisé le Composant à Interface Hétérogène et le Modèle d'Exécution Hétérogène à l'aide à un très haut niveau d'abstraction. Dans ce chapitre, nous allons nous rapprocher de la réalisation en utilisant un langage intermédiaire entre cette abstraction et le langage employé par la plate-forme que nous allons utiliser pour la simulation et la validation.

En effet, pour la validation des concepts présentés dans cette étude, nous allons employer la plate-forme PTOLEMY II qui utilise l'approche objet et dont les API sont programmées en JAVA.
La méthodologie utilisée pour cette plate-forme nous a induis vers l'utilisation du langage UML [104] pour une spécification intermédiaire.
Par ailleurs, cette spécification intermédiaire, en réduisant l'écart entre les concepts utilisés pour la spécification abstraite et l'implémentation dans PTOLEMY II, nous servira de données utiles dans les étapes d'intégration, de validation et celle de simulation.

Pour représenter respectivement la vue statique et les vues dynamiques du système hétérogène non-hiérarchique, nous utilisons donc les diagrammes structurels et les diagrammes comportementaux.

Pour la vue statique du système, nous utilisons les diagrammes de classes pour le composant à interface hétérogène et pour le modèle d'exécution hétérogène.

Pour la vue dynamique du système, nous utilisons les diagrammes d'états qui montre les différentes phases du modèle d'exécution hétérogène. Nous utilisons également le diagramme de séquence qui met en évidence le séquencement des activités des différents sous-systèmes ainsi que les lancements des entités issues des HICs.
Il est par ailleurs intéressant de signaler que dans cette traduction de sémantique, les opérations utilisées aux chapitres précédents seront transformées en méthodes.

7.2 Vue statique du système : Diagrammes de classes

De la modélisation abstraite présentée aux chapitres précédents, nous avons vu que la modélisation hétérogène non-hiérarchique reposait sur deux composants à savoir, le composant à interface hétérogène et le modèle d'exécution hétérogène non-hiérarchique.
Pour leur représentation en formalisme UML, nous caractérisons chacun de ces composants par les classes suivantes :

– la classe *HicComp* qui caractérise le composant à interface hétérogène,
– la classe *HModEx* qui caractérise le modèle d'exécution hétérogène non-hiérarchique,
– l'interface *HeterogeneousBehavior* qui doit être implémentée par les classes *HicComp*.

Chacune de ces classes englobe les attributs et les opérations concernant le composant qu'elle caractérise. Cependant, en ce qui concerne ces attributs et ces opérations, nous n'en donnons pas une liste exhaustive, car, nous limitons leur visualisation en nous focalisant simplement sur les éléments pertinents dans chaque diagramme de classe.

7.2.1 Classe HicComp

La classe HicComp implémente l'interface *HeterogeneousBehavior* pour imposer aux classes qui la spécialisent de faire l'engagement d'implémenter la méthode *computeBehavior* qui calcule le comportement de HIC à la frontière des différents modèles de calcul.
Dans la classe HicComp, cette méthode ne sera qu'une méthode vide. Il s'ensuit que toute classe spécialisant la classe HicComp doit redéfinir cette méthode afin de donner au HIC la capacité de pouvoir offrir un comportement hétérogène à la frontière des modèles de calcul qu'il utilise.
C'est donc dans cette implémentation que le concepteur définira selon sa spécification, le

comportement hétérogène entre deux ou plusieurs modèles de calcul en termes aussi bien d'un simple passage de données d'un domaine vers un autre qu'en termes de calcul d'un comportement impliquant une combinaison de données venant de plusieurs modèles de calcul.

Attributs

Les attributs de HicComp sont :

_avatar	:	projection crée
_container	:	entité composite représentant son sous-système
_hModEx	:	modèle d'exécution hétérogène
_modEx	:	modèle d'exécution de son conteneur
_original	:	le HIC qui a généré la projection qui utilise cet attribut
_runningProjection	:	projection qui a demandé l'activation de ce HIC
_typeProjection	:	le type d'une projection, consommatrice ou productrice

Méthodes

Les opérations de HicComp sont transformées en méthodes qui sont :

+computeBehavior()	:	utilisée par un HIC pour calculer un comportement
	:	hétérogène
+existData	:	utilisé par les projections pour le test en lecture
+getModEx()	:	retourne le modèle d'exécution de cette entité
+HicComp()	:	le constructeur de la classe HicComp
+initialize()	:	utilisée par toutes les entités en phase d'initialisation
+isFull	:	utilisé par les projections pour le test en écriture
+postCondition()	:	utilisée par toutes les entités, retourne un
	:	booléen pour accepter ou refuser d'être relancée
+preCondition()	:	utilisée par toutes les entités retourne un
	:	booléen pour accepter ou refuser d'être lancée
+read	:	utilisé par les projections pour lire des données
+trigger()	:	utilisée par toutes les entités pour leur lancement
+write	:	utilisé par les projections pour écrire des données
#enqueueNextTimeToFire()	:	utilisée par une projection productrice DE
	:	pour s'auto-poster un événement pur
#generateProjection()	:	utilisée par un HIC et sert à générer une projection
#getTypeProjection()	:	utilisée par un HiC et donne le type d'une projection,
	:	consommatrice ou productrice
#getOriginal()	:	utilisée par une projection et retourne son HIC
	:	original

_nextTimeToFire()	:	utilisée un HIC ayant une ou plusieurs sorties DE
	:	pour poster un événement pur
_recordRunningProjection()	:	utilisée par une projection pour s'enregistrer auprès
	:	de son HIC original, avant de demander son activation
_requestBehaviorComputing()	:	utilisée une projection pour demander
	:	l'activation de son HIC original

Diagramme

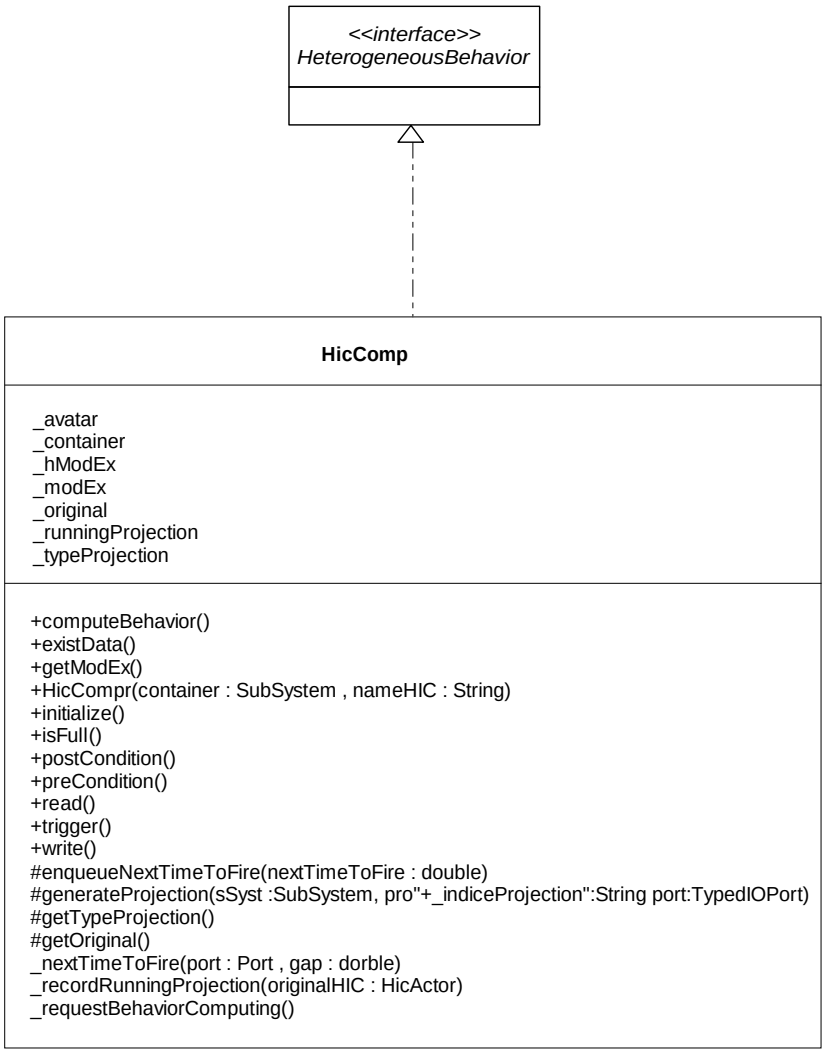


FIG. 7.1 – Diagramme de classe de HicComp

7.2.2 Classe HModEx

Attributs

Les attributs de HicComp sont :

_container	:	le modèle
_indiceOmega	:	indice d'un sous-système
_indiceOmegaTransfer	:	indice d'un sous-système qui reçoit un HIC
_indiceProjection	:	indice d'une entité projection
_indiceRelay	:	indice d'un relais

Méthodes

Les opérations de HicComp sont transformées en méthodes qui sont :

+HModEx()	:	le constructeur du modèle d'exécution hétérogène
+initialize()	:	utilisée par le ModEx en phase d'initialisation
+postCondition()	:	utilisée par le ModEx pour accepter d'être relancé
+preCondition()	:	utilisée par le ModEx pour accepter d'être lancé
+trigger()	:	exécution du modèle d'exécution hétérogène
#isCompatible()	:	retourne un booléen qui détermine la
	:	compatibilité entre un port et un modèle
	:	d'exécution
_computeDomainsSchedule()	:	calcule l'ordonnancement des sous-systèmes
_computeActorsTopologySort()	:	calcule un tri topologique sur les acteurs
_disconnectReconnectPorts()	:	déconnecte et reconnecte les ports
_divideSystem()	:	divise le système en sous-systèmes
_existPathBetweenActors()	:	teste l'existence d'un chemin entre deux acteurs
_existPathBetweenPorts()	:	teste l'existence d'un chemin entre deux ports
_getCompatibility()	:	retourne le port spécifique à un directeur donné
	:	sous-système
_insertRelay()	:	insère un relais dans un sous-système
_predecessorActor()	:	retourne les acteurs prédécesseurs d'un acteur
_predecessorHic()	:	retourne le HIC prédécesseur d'un acteur
_processVirtualPorts()	:	insère des ports virtuels dans les projections
_removeAllLinks()	:	déconnecte tous les ports
_runHic()	:	lance le HIC demandé par une projection
_setProjection()	:	projète un HIC dans un sous-système qu'il vient
	:	de générer
_setSingleProjection()	:	projète un HIC dans un sous-système
	:	existant et utilisant son MoC
_sharingPorts()	:	met en commun les ports d'un HIC

	:	et les ports correspondants de sa projection
_subSystemCreation	:	cette méthode crée un sous-système
_successorActor()	:	retourne les acteurs successeurs d'un acteur
_successorHic()	:	retourne le HIC successeur d'un acteur
_testViaHic	:	teste l'existence d'un chemin passant
	:	par un HIC entre un acteur et un sous-système
_topologicalSort	:	cette méthode établit un tri topologique

Diagramme

FIG. 7.2 – Diagramme de classe de HModEx

7.3 Vue dynamique du système

7.3.1 Diagrammes d'états

Dans la modélisation du modèle d'exécution hétérogène non-hiérarchique, nous avons spécifié son fonctionnement en trois phases à savoir : le partitionnement, l'ordonnancement et l'exécution. Nous représentons les comportements de ces différentes phases dans les diagrammes d'états suivants :

Partitionnement

La phase de partitionnement se déroule pendant l'étape d'initialisation du modèle d'exécution hétérogène non-hiérarchique.
En effet, pendant cette phase, le modèle d'exécution commence par la génération des sous-systèmes. A ce niveau, il crée les sous-systèmes, projète les HICs et place les différents acteurs dans les différents sous-systèmes. Ensuite, il passe au traitement des ports virtuels afin d'assurer l'ordonnancement à l'intérieur des différents sous-systèmes. Enfin, il déconnecte tous les ports avant de les reconnecter.

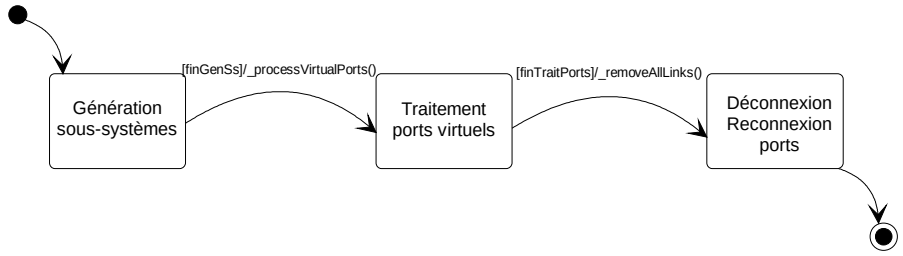


Fig. 7.3 – Diagramme d'états du Partitionnement

Ordonnancement

Pendant la phase d'ordonnancement, le modèle d'exécution hétérogène génère le squelette du système. Ce squelette est composé des différentes projections. Il effectue enfin un tri topologique qui lui donne l'ordre d'activation des différents sous-systèmes.

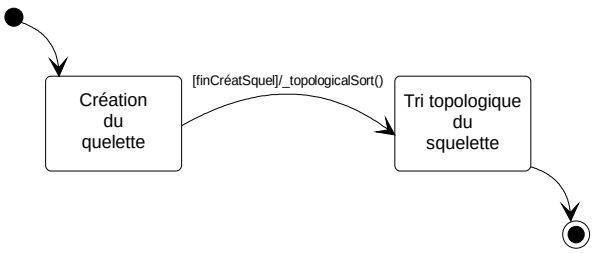


Fig. 7.4 – Diagramme d'états de l'ordonnancement

Exécution

Après les phases d'ordonnancement et de partitionnement, la phase d'exécution intervient pour lancer le modèle.
Pendant cette phase, les sous-systèmes sont lancés séquentiellement les uns après les autres selon l'ordre du séquencement établi au préalable dans l'ordonnancement.

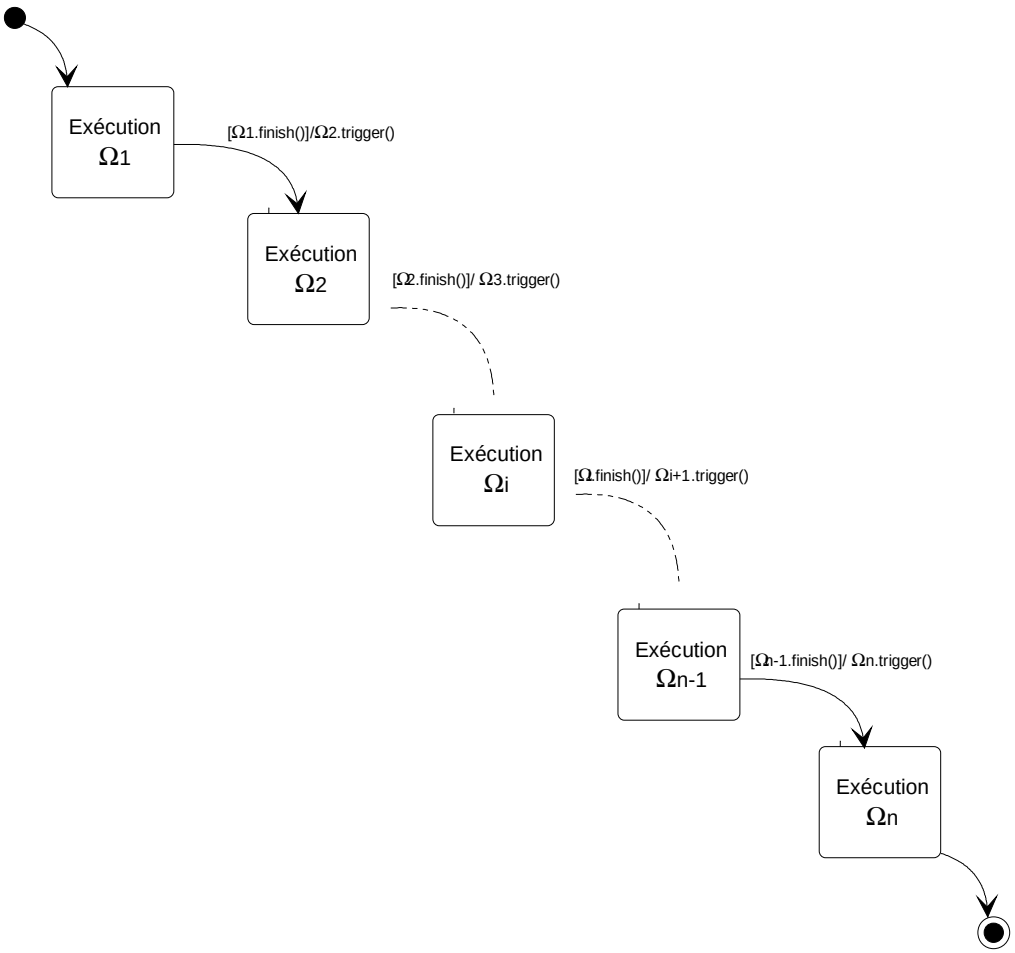


FIG. 7.5 – Diagramme d'états de la phase d'exécution

Le système effectue des itérations de manière cyclique soit jusqu'à l'arrêt de l'exécution du modèle par l'utilisateur ou soit jusqu'au décompte du nombre d'itérations fixé au préalable par l'utilisateur.

7.3.2 Diagrammes des séquences

Nous représentons ci-dessous le séquencement de l'exécution d'un système partitionné à la frontière des deux modèles de calcul. Chacun de ses sous-systèmes englobe entre autre une projection de HIC.
Cette représentation du séquencement de l'exécution du système met en évidence la dimension temporelle des déclenchements de chacune des entités. Le déclenchement d'une entité étant conditionné par la fin de l'autre.

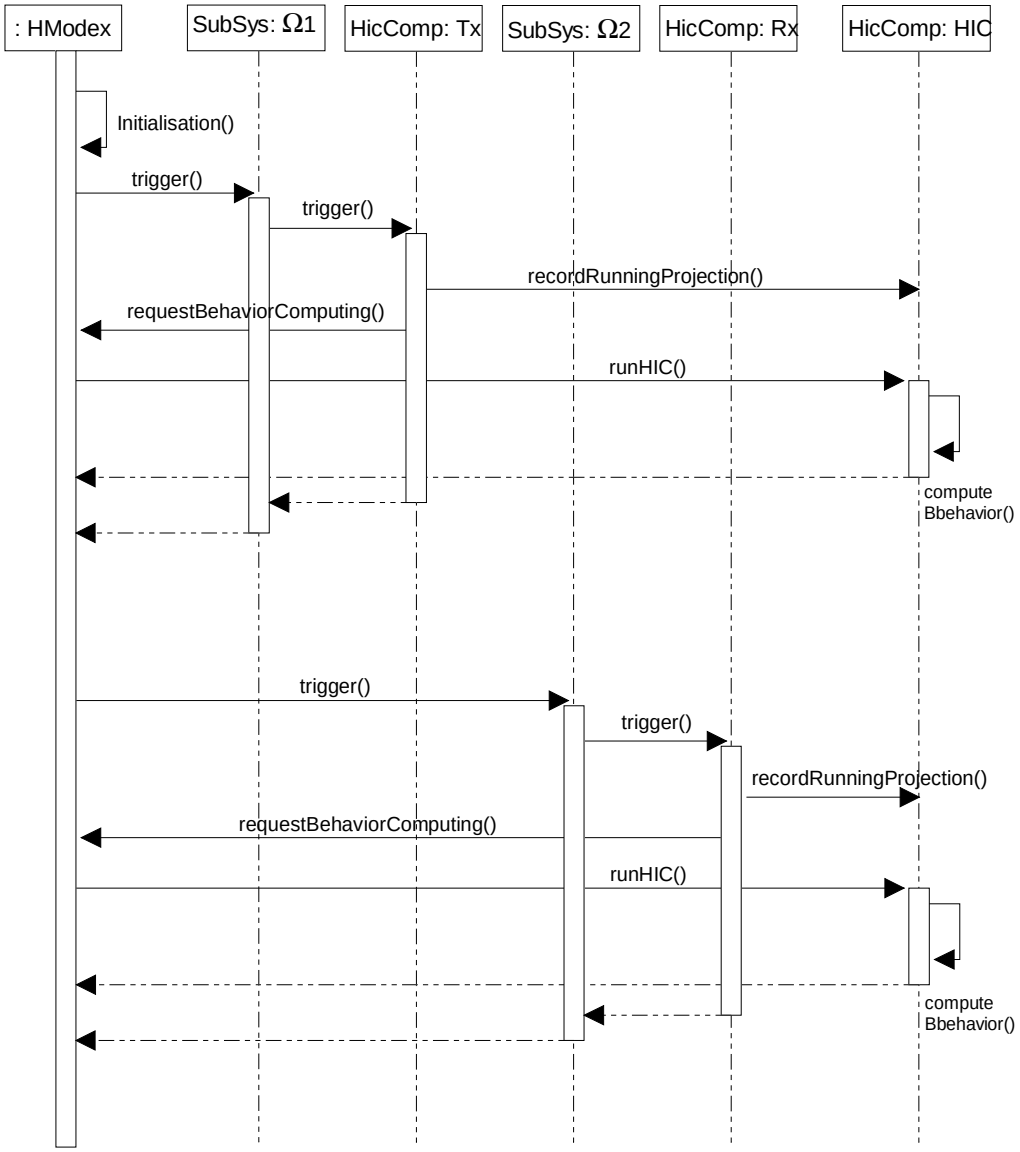


FIG. 7.6 – Diagramme de séquence de l'exécution d'un modèle

Il est intéressant de relever que les sous-systèmes Ω_1 et Ω_2 ont un double comportement : face au modèle d'exécution hétérogène, ils se comportent comme des acteurs simples et face aux acteurs qu'ils englobent le comportement de leur modèle d'exécution est celui d'un modèle d'exécution régulier.
En effet, lorsque le modèle d'exécution hétérogène active un sous-système, celui-ci active les acteurs qu'il englobe dont la projection de HIC. Cette projection demande l'activation de son HIC générateur auprès du modèle d'exécution hétérogène. Une fois activé, le HIC calcule le comportement hétérogène et termine ses activités. Ainsi, la projection et son sous-système associé terminent également les leurs respectivement.

7.4 Conclusion partielle

Nous avons représenté les composants d'appui à la modélisation hétérogène non-hiérarchique.
Cette représentation est faite de manière visuelle par une vue statique à l'aide des diagrammes des classes et par des vues dynamiques à l'aide des diagrammes d'état et des séquences.
Néanmoins, dans cette représentation, nous n'avons pas représenté tous les attributs et méthodes car, ces détails seront intégrés à l'étape de l'implémentation.
Toutefois, cette spécification intermédiaire se situant à un niveau d'abstraction élevée est faite sans tenir compte des spécificités des différentes plates-formes. Ainsi, lors de l'intégration des différentes classes présentées dans ce chapitre dans une quelconque plate-forme, il nous parait judicieux que le concepteur les adapte à des spécificités aussi bien terminologiques que structurelles de sa plate-forme.
C'est pourquoi, au chapitre suivant, lors de l'intégration des classes spécifiées dans ce chapitre dans la plate-forme PTOLEMY II, nous adapterons cette représentation intermédiaire à la structure et à la terminologie de ladite plate-forme.

Chapitre 8

Intégration de l'Hétérogénéité Non-Hiérarchique dans PTOLEMY II

Résumé -Dans ce chapitre nous intégrons notre concept de l'Hétérogénéité Non-Hiérarchique dans la plate-forme PTOLEMY II. Nous commençons par un aperçu de cette plate-forme en présentant les classes fondamentales. Dans cet aperçu, nous avons également mis en évidence le mécanisme partagé de communication entre la classe Director et la classe Receiver et nous avons détaillé les différentes phases d'un modèle PTOLEMY II. Ensuite, nous opérons à l'intégration de notre concept non-hiérarchique dans PTOLEMY II. Cette intégration est faite sur deux niveaux. Le premier niveau consiste à l'intégration des classes d'appui à la non-hiérarchie dans l'architecture de PTOLEMY II. Le deuxième niveau consiste à l'assignation des différentes phases de la non-hiérarchie dans les différentes méthodes du directeur hétérogène non-hiérarchique. Enfin nous avons clôturé le chapitre par un simple exemple explicatif

8.1 Introduction

Dans les chapitres précédents, nous avons modélisé le concept de l'hétérogénéité non-hiérarchique. Nous avons d'abord développé une étude théorique, puis, avons modélisé des composants d'appui à l'hétérogénéité non-hiérarchique. Cette modélisation est partie d'un d'une abstraction de haut niveau vers une sémantique proche de la réalisation en utilisant le formalisme UML.
Ce chapitre s'intéresse à la mise en oeuvre de ces concepts, c'est-à-dire, leur passage du formalisme UML à la réalisation et à la validation.

Il est intéressant de nous poser la question sur le comment de la validation de ces concepts dans la plate-forme PTOLEMY II choisie.
En effet, nous avons choisi de valider nos concepts par simulation. Pour cela, nous allons intégrer ces concepts dans PTOLEMY II et simuler un exemple qui rendra des résultats escomptés.

La réussite de ce challenge passe par une démarche méthodologique que nous avons adoptée. Elle consiste d'abord à présenter les caractéristiques principales de la plate-forme

PTOLEMY II. Puis, d'y intégrer les classes des composants d'appui à l'hétérogénéité non-hiérarchique que nous avons modélisés dans les chapitres précédents. Ensuite d'assigner les différentes phases théoriques de ce concept de la non-hiérarchie dans les différentes étapes de déroulement d'un modèle PTOLEMY II. Enfin, lorsque cette intégration est complètement réalisée, afin de fixer les idées, nous allons en conséquence simuler un exemple explicatif simple.

8.2 Aperçu sur PTOLEMY II

PTOLEMY II utilise l'approche objet pour permettre la modélisation, la conception et de simulation de systèmes hétérogènes. Il est programmé en JAVA, et ses composants, qui sont des acteurs, sont représentés par des classes JAVA. PTOLEMY II utilise une approche unifiée qui lui permet de supporter un grand nombre de modèles de calcul. Il est hétérogène dans le sens que plusieurs composants utilisant différents modèles de calcul peuvent donc y être utilisés conjointement.

Dans PTOLEMY II, les modèles sont des graphes où les noeuds représentent les entités et les arcs sont des relations comme montré sur la figure 8.1. Les entités sont des acteurs, et les relations ne prennent un sens que dans le contexte d'un modèle de calcul, dont l'implémentation est appelée *« domaine »* dans Ptolemy.

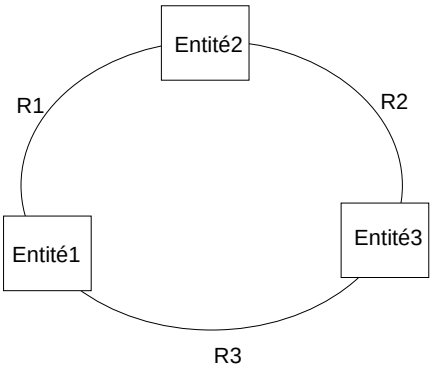


FIG. 8.1 – Graphe PTOLEMY II

8.2.1 Modèles de calcul et domaines dans PTOLEMY II

PTOLEMY II implémente un certain nombre de modèles de calcul. Ces modèles de calcul sont encapsulés dans des objets appelés domaines. Nous donnons ici, une liste non exhaustive de modèles de calcul. Leurs caractéristiques sont largement détaillées dans [13].

Communicating Séquential Processes (CSP) : dans le domaine CSP, les acteurs représentent des processus s'exécutant concurremment et implémentés comme des

threads JAVA. Ils communiquent par des actions atomiques et instantanées appelées passage de message synchrone[1]. Ces modèles correspondent bien aux applications où le partage de ressource est un élément principal, tels que les modèles de base de données client-serveur et le traitement multitâche ou le multiplexage des ressources matérielles.

Continuous Time (CT) : dans le domaine CT, les acteurs représentent les composants qui interagissent par l'intermédiaire des signaux à temps continu. Ils indiquent des relations algébriques ou différentielles entre les entrées et les sorties. Pour la bonne gestion du flot de contrôle et du flot de données, le point capital dans ce domaine demeure la recherche du point fixe, i.e., un ensemble de fonctions CT qui satisfont toutes les relations. Ce modèle est utilisé pour la spécification des circuits analogiques, mécaniques ou à micro-ondes et est également utilisé pour la spécification des systèmes de contrôle

Discrete-Events (DE) : dans le domaine DE, les acteurs communiquent par l'intermédiaire des séquences d'événements dans le temps. Un événement étant composé d'une valeur et d'une estampille de temps. Les acteurs peuvent être soit des processus qui réagissent aux événements, implémentés comme des thread JAVA, soit des fonctions qui démarrent quand de nouveaux événements sont fournis. Ce domaine est très utilisé pour la spécification des matériels numériques et pour la simulation des systèmes de télécommunications.

Process Network (PN) : dans le domaine PN, les processus communiquent par passage de message asynchrone sur des canaux qui peuvent les protéger. Dans ce modèle, les arcs représentent des séquences des valeurs de données, et les entités représentent des fonctions qui tracent des séquences d'entrée dans des séquences de sortie.

Distributed Discrete Events (DDE) : le domaine DDE peut être vu comme une variante de DE ou de PN. Il se focalise sur la fixation du point central de contrôle d'un modèle qui est un des problèmes cruciaux de la modélisation d'événements discrets. Ici, les acteurs sont des processus, implémentés comme des threads JAVA et peut avancer son temps local au minimum des temps locaux sur chacune de ses d'entrée.

Synchonous Dataflow (SDF) : le domaine SDF manipule les calculs réguliers qui opèrent sur des flots. Utilisé souvent dans le traitement des signaux, SDF est un cas particulier de PN. Dans ce modèle, le deadlock est inexistant. D'ailleurs, le programme de lancement, parallèle ou séquentiel, est calculable statiquement, faisant de SDF un formalisme extrêmement utile de spécification pour le logiciel temps réel embarqué et pour le hardware.

Discret Time (DT) : le domaine DT hérite du domaine SDF en incluant la notion d'uniformité de temps entre les données. La communication entre les acteurs est sous forme d'une séquence de données.

[1]Ce type de communication est aussi appelé communication par rendez-vous

Synchronous/Reactive (SR) : dans le modèle de calcul SR, les arcs représentent les valeurs de données qui sont mises dans la file d'attente avec les tops d'horloge globaux. Ces signaux sont discrets, mais à la différence du DT, ils n'ont pas besoin d'avoir une valeur à chaque top d'horloge. Ici, les entités représentent des relations entre les valeurs d'entrée et celles de sortie à chaque top, et sont habituellement des fonctions partielles avec certaines restrictions techniques pour assurer le déterminisme. Les modèles SR sont excellents pour des applications avec une logique de contrôle concourante et complexe.

Timed Multitasking (TM) : le domaine TM soutient la conception du logiciel temps réel concourant. Il offre un programmateur de préemption par priorité fondamental, comme dans les logiciels d'exploitation temps réel, RTOS. Dans ce domaine temporisé, chaque acteur s'exécute comme une tâche concourante quand il reçoit de nouvelles entrées, et son exécution est conduit par des événements.

Giotto : Ce domaine est conçu pour travailler avec le domaine FSM pour réaliser les modèles modaux. Il a une option de temps de déclenchement, où chaque acteur est appelé périodiquement avec une période indiquée.

Finite-State Machines (FSM) : le domaine FSM est radicalement différent de tous les autres. Ici, les entités ne représentent pas des acteurs mais plutôt des états, et les arcs représentent des transitions entre les états. L'exécution est strictement une séquence ordonnée des transitions d'état.

8.2.2 Acteur

La conception de PTOLEMY II est basée sur la méthodologie orientée acteur [80][130][105][69]. Par conséquent, un acteur a une interface composée de port d'entrée et de port de sortie et des paramètres. Certains parmi ces acteurs sont conçus pour être polymorphes de domaine, c'est-à-dire, qu'ils peuvent opérer dans divers domaines tandis que d'autres sont spécifiques de domaine, c'est-à-dire dédiés à un seul domaine.
Afin d'assurer la cohérence dans l'appellation des ports et d'éviter la reproduction de code y afférant, PTOLEMY II offre trois classes de base à savoir : les classes Sources, Sinks, et Transformer.
C'est pourquoi, du fait que la plupart des acteurs dans la bibliothèque d'acteurs héritent des ces classes de base, et pour les raisons de cohérence évoquée ci-haut, un port d'entrée est appelé *«input»* et celui de sortie est appelé *«output»*. L'utilisation de ces classes de base évite aux ports d'avoir diverses appellations. Cette uniformité favorise en conséquence la réutilisation des acteurs dans PTOLEMY II.
Le package Actor fournit deux patrons pour la communication et pour le flot d'exécution. Ce sont des patrons du fait qu'aucun mécanisme n'est réellement défini pour la communication ni pour le flot de contrôle, mais, plutôt des classes de base sont définies de sorte que les domaines puissent seulement redéfinir quelques méthodes pour leur interopérabilité. Ce

package fournit donc une infrastructure neutre indépendant du modèle de calcul

8.2.3 Interface d'un acteur

Puisque PTOLEMY II utilise l'orientation acteur, l'interface d'un acteur inclut donc les ports qui représentent ses points de communication et les paramètres utilisés pour configurer ses opérations.

Port

Les ports, faisant partie de l'interface de l'acteur, sont donc des membres publics. Ils représentent un ensemble de canaux d'entrée et de sortie par lesquels les données peuvent passer dans d'autres ports. Sa déclaration est faite avec la syntaxe :

```
Public Type_du_Port Non_du_Port
```

La plupart des ports dans les acteurs nécessitant le contrôle de type. Ces ports sont des instances de la classe TypedIOPort. Lorsqu'un port exige des services spécifiques à un domaine quelconque, il sera alors une instance d'une sous-classe d'un port spécifique. Le port est crée dans le constructeur par la syntaxe :

```
Non_du_Port = new Type_du_Port(this, ''Non_du_Port'', true, false)
```

Le premier argument du constructeur est le conteneur du port. Le deuxième représente le nom du port, qui est un String quelconque, lequel par convention, est identique au nom du membre public. Les troisième et quatrième arguments indiquent si le port est une entrée ou une sortie.

Paramètres d'un acteur

Comme pour des ports, par convention, les paramètres sont des membres publics des acteurs. Leur déclaration est faite par la syntaxe :

```
Public Parameter Non_du_Parametre
```

Et, ils sont créés dans le constructeur par la syntaxe :

```
Non_du_Parametre = new Parameter(this, ''Non_du_Parametre'')
```

On peut bien rajouter un troisième argument facultatif au constructeur qui sera une valeur par défaut pour le paramètre. Dans [13], il est donné plusieurs manières d'initialiser un paramètre. Lorsque la valeur d'un paramètre change, l'acteur est mis au courrant en appelant sa méthode `attributeChanged()`. Cette méthode est passée au paramètre qui a changé. Par contre, le changement d'un paramètre a parfois de répercussions imprévisibles.

8.2.4 Opérations de communication d'un acteur

Dans PTOLEMY II, l'opération d'écriture est traduite par la méthode `send( )`. Pour envoyer un token à travers un canal d'un port on utilise la syntaxe :

```
Non_du_Port.send(Numero_du_canal, token)
```

L'opération de lecture est traduite par la méthode `get()`. Un token peut être lu à partir d'un canal avec la syntaxe :

```
Token token = Nom_du_Port.get(Numero_du_canal)
```

Pour l'opération de test d'existence de données sur un port d'entrée, le mécanisme de PTOLEMY II revient à interroger un port d'entrée pour savoir si la méthode `get()` peut réussir, c'est-à-dire, si un token est disponible ou s'il peut être rendu disponible. Ceci est fait par la syntaxe :

```
Boolean reponse_booleen = Nom_du_Port.hasToken(Numero_du_canal)
```

De même, l'opération de test de disponibilité en réception d'un port de destination est traduite par une interrogation de ce port de sortie pour savoir si une méthode `send()` peut réussir c'est-à-dire s'il existe de la place suffisante dans le receiver pour recevoir un token donné :

```
Boolean reponse_booleen = Nom_du_Port.hasRoom(Numero_du_canal)
```

8.2.5 Opération de flot de contrôle de l'acteur

Un acteur est une entité exécutable, ainsi, son flot de contrôle ou son exécution se fait en trois phases à savoir, *Initialisation, Itération et Wrapup*, une itération étant composée d'une méthode *prefire*, d'une ou plusieurs méthodes *fire* et d'une ou plusieurs méthodes *postfire*.

Ce mécanisme d'exécution est définie dans l'interface *Executable* qui est implémentée par les Directeurs et les acteurs. Cependant, pour les acteurs, cette implémentation est faite via l'interface Actor.

8.2.6 Communication dans PTOLEMY II

Dans PTOLEMY II, un domaine définit la sémantique de communication et la séquence d'exécution des acteurs à travers deux classes ; la classe *Receiver* et la classe *Director*.
Ainsi, lorsqu'un modèle est lancé, son directeur lance les acteurs qui y sont contenus.
Le concept du couple (directeur, receiver) spécifique au domaine forme une interface entre un acteur et son environnement. Ce concept à la base de l'abstraction hiérarchique implique que tout acteur ou tout domaine dans n'importe quel domaine ou sous-domaine, puisse être embarqué dans n'importe quel autre modèle. Car, dans ce concept, tout est fondé sur des notions communes ; celle d'exécution organisée par le directeur et celle de communication régie par les receivers.

Ce concept d'interface a l'avantage de maximiser la réutilisation des composants et des modèles. Et, elle définit par voie de conséquence, deux ensembles de conventions lexicologiques. Le premier ensemble relatif à la communication, contient des conventions pour l'envoie ou la réception de l'information de et vers l'environnement d'un acteur. Le deuxième ensemble relatif au contrôle, contient des conventions pour l'exécution d'un modèle.

Par exemple, tous les receivers ont une méthode `get()`, utilisée pour obtenir un token qui y est stocké. Du fait de la généricité de la notion de receivers, il y a plusieurs implémentations différentes des ces conventions qui ne sont d'ailleurs pas toujours compatibles entre elles.

Du côté directeurs, ils ont également tous la méthode `fire()` qu'ils utilisent pour commencer une exécution. Dans cette méthode, selon le domaine, un acteur peut appeler la méthode `get()` du receiver pour obtenir un token d'entrée ou appeler complètement une autre méthode.

Considérons le domaine SDF où son ordonnanceur garantit qu'un acteur ne peut être lancé que lorsqu'il y a un token dans son receiver. Il s'ensuit qu'un acteur SDF n'ait pas besoin de vérifier la disponibilité du token avant d'appeler la méthode `get()`.
En revanche, dans le domaine DE, le directeur ne donne pas la même garantie, ainsi les acteurs devraient vérifier la disponibilité d'un token en appelant la méthode hasToken() avant d'appeler la méthode `get()`.
Par conséquent, si un acteur SDF est utilisé dans le domaine DE, il peut causer une exception en appelant la méthode `get()` sur un receiver vide, car, pour cet acteur, la vérification devrait être faite par son directeur.

Directeur

La classe Director est la classe de base pour les directeurs. Ceux-ci implémentent différents modèles de calcul dont chacun est associé à un domaine de PTOLEMY II. Cette classe fournit une implémentation par défaut d'une exécution que l'on peut redéfinir dans les

domaines spécifiques. Dans la terminologie de PTOLEMY II, on dit qu'un directeur réalise un domaine. Ainsi, quand on construit un modèle avec un directeur λ, on a donc construit un modèle « dans le domaine λ ».

Le directeur impose le modèle de communication en fournissant des receivers aux ports d'entrée et à l'intérieur des ports de sortie des acteurs composites opaques. Il implémente et contrôle la séquence d'exécution c'est-à-dire l'ordre de lancement des acteurs, la progression du temps et la taille des tampons dans la mémoire de chaque acteur.

En effet, quand un directeur est lancé, il a le contrôle complet de l'exécution, et peut lancer quelques itérations d'acteurs appropriés au modèle de calcul qu'il implémente.
Au même niveau de la hiérarchie [82], des acteurs sont contrôlés par un seul directeur commun.

Dans PTOLEMY II, les acteurs composites peuvent faire participer deux directeurs à savoir : Un directeur extérieur qu'on appelle *directeur exécutif*, contrôle l'acteur composite et un directeur intérieur qu'on appelle *directeur local*, contrôle les acteurs contenus par l'acteur composite. Celui-ci est responsable de l'exécution des composants dans l'entité composite Il effectue la programmation nécessaire, les threads d'expédition qui doivent être lancés et il génère le code qui doit être généré, etc...

Receiver

les receivers [14] [70] [71] [141], du fait qu'ils implémentent le mécanisme de communication, sont donc contenus dans des ports d'entrée et sont dédiés à chaque canal de communication comme montré sur la figure 8.2.

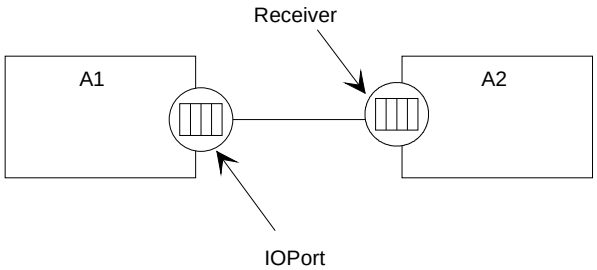


FIG. 8.2 – Receivers dans PTOLEMY II

Il s'ensuit, que lorsqu'un acteur appartient à un domaine, il acquière les receivers spécifiques à ce domaine. Ces receivers peuvent représenter un buffer, une file d'attente FIFO, un mailbox ou des points de rendez-vous selon le modèle de calcul implémenté par le directeur.
Le receiver utilisé par un port d'entrée détermine le protocole de communication. Ceci est

étroitement lié au modèle de calcul.

Un receiver est crée par la classe IOPort en appelant sa méthode protégée `_newReceiver()`. Cette méthode délègue cette création au directeur retournés par `getDirector()`, en appelant sa méthode `newReceiver()`. Ainsi, le directeur contrôle le protocole de communication, en plus de sa fonction primaire de déterminer le flot de contrôle.

En ce qui concerne les types de protocole de communication implémentés dans PTOLEMY II, il en existe deux ; la communication synchrone et la communication asynchrone. La communication asynchrone ou passage du message asynchrone est soutenu par la classe QueueReceiver qui contient une instance de FIFOQueue laquelle implémente une file d'attente FIFO. La communication synchrone ou passage de message synchrone, appelé aussi rendez-vous exige que le producteur et le consommateur soient tous les deux simultanément prêts pour le transfert de données. Le producteur et le consommateur sont deux threads séparés.

Le thread initiateur indique son souhait de rendez-vous en appelant la méthode `send()`, qui appelle à son tour la méthode `put()` du receiver approprié. Le destinataire indique son acceptation au rendez-vous en appelant la méthode `get()` sur un port d'entrée, qui appelle à son tour la méthode `get()` du receiver indiqué. Un thread qui implémente ce mécanisme doit caler jusqu'à ce que l'autre thread soit prêt à accomplir le rendez-vous.

Coopération entre Directeur et Receiver

Le domaine, par le truchement de son directeur détermine les receivers nécessaires sur leurs ports d'entrée. En d'autres termes, le directeur est également responsable de la création des receivers. C'est pourquoi, les canaux de communication entre acteurs sont implémentés par le receiver et contrôlés par le domaine, c'est-à-dire, ils sont contrôlés par le directeur. les receivers et les directeurs conjuguent une forme de coopération dans le but de mener à bien le déroulement de l'exécution d'un modèle.

Transport de données

Les données véhiculent d'un port de sortie d'un acteur à un port d'entrée d'un autre acteur via le receiver comme montré sur la figure 8.3.
Un acteur appelle la méthode `send()` de son port pour envoyer un token à un acteur à distance. Le port obtient une référence à ce receiver à distance par l'intermédiaire de IORelation. Il appelle la méthode `put()` du receiver distant et lui passe le token. L'acteur de destination recherche le token en appelant la méthode `get()` de son port d'entrée, qui appelle à son tour la méthode `get()` du receiver indiqué.

Par ailleurs, il est important de noter que les domaines fournissent des receivers spécialisés et ces receiver redéfinissent les méthodes `get()` et `put()` pour implémenter le protocole de communication concernant ce domaine.

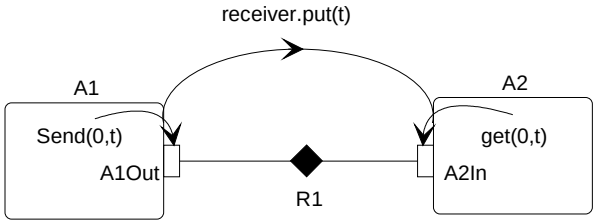


FIG. 8.3 – Transport de données

8.2.7 Entité composite opaque

Un acteur composite disposant d'un directeur local est appelé *« acteur composite opaque »*. En général, l'acteur composite au niveau supérieur est toujours opaque et n'a aucun port. Par contre s'il n'est pas au niveau supérieur, ses ports extérieurs sont opaques et cachent ses activités intérieures du reste du système tout en étant traité comme acteur atomique par son directeur exécutif.

En ce qui concerne ses entrées et ses sorties, l'acteur composite opaque délègue le transfert de ses entrées à son directeur local, et le transfert de ses sorties à son directeur exécutif. Cette organisation a son sens dans la mesure où dans un cas comme dans un autre, le directeur approprié au modèle de calcul du port de destination est celui qui manipule le transfert.

Dans une entité composite opaque, un port qui est capable de recevoir des données de l'intérieur et d'envoyer des données à l'extérieur doit donc contenir un receiver. Un tel receiver s'appelle *« inside receiver »*.

Par conséquent, un port opaque d'un acteur composite opaque doit, ainsi, être capable de stocker deux types distincts de receivers dont l'un est fourni par son directeur local et est approprié au modèle de calcul intérieur, et l'autre est fourni par son directeur exécutif et est approprié au modèle de calcul extérieur comme montré sur la figure 8.4.

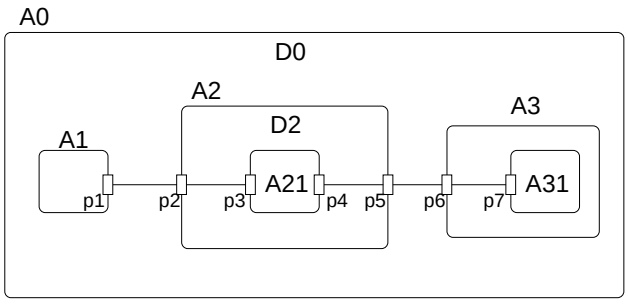


FIG. 8.4 – Directeurs et entités

Illustration 8.2.1 A0 est un acteur composite supérieur contenant les acteurs A1, A2 et A3. A2 qui contient à son tour l'acteur A21 est un acteur composite opaque car il possède un directeur D2. Par contre A3 n'ayant pas de directeur dédié est donc un acteur composite transparent. Le directeur D0 gouverne les trois acteurs A1, A2 et A3 et D2 gère l'acteur A21. Les ports p2 et p7 contiennent des receivers fournis et gérés par le directeur D0. Par contre les ports p3 et l'intérieur de p5 contiennent des receivers fournis et gérés par le directeur D2. D2 contrôle la communication entre les ports p2 et p3 et de p4 à p5 et le directeur D0 contrôle la communication de ports p1 à p2 et de p5 au port p7.

La plupart des méthodes qui accèdent à des receivers, telles que `hasToken()` ou `hasRoom()`, se réfèrent seulement aux receiver extérieurs. L'utilisation des receivers intérieurs est dédiée à la manipulation des acteurs composites opaques.
Quand une méthode d'action est appelée sur un acteur composite opaque, l'acteur composite appellera généralement la méthode correspondante de son directeur local. Cette interaction est cruciale, puisqu'elle est indépendante du domaine et tient compte de la communication entre différents modèles de calcul.

Signalons toutefois que lorsqu'un acteur composite n'a pas de directeur dédié, il est dit *« transparent »* et ne dispose en conséquence que des ports transparents.
Par définition, un port transparent est une entrée s'il est connecté à ou sur l'intérieur à l'extérieur d'un port d'entrée, ou s'il est connecté de l'intérieur à l'intérieur d'un port de sortie. C'est-à-dire, un port transparent est un port d'entrée s'il peut accepter les données qu'il peut alors juste passer à travers un port de sortie transparent.
Également, un port transparent est un port de sortie s'il est connecté à l'intérieur à l'extérieur d'un port de sortie, ou s'il est connecté à l'intérieur de l'intérieur d'un port d'entrée.

8.2.8 Manager

Dans PTOLEMY II, un Manager contrôle l'exécution globale d'un modèle en interagissant avec un acteur composite, lequel au demeurant est un modèle exécutable. L'exécution d'un modèle est implémentée par l'une de ses trois méthodes, `execute()`, `run()` et `startRun()`.

La méthode `startRun()` engendre un thread qui appelle la méthode `run()`, et puis retourne immédiatement. La méthode `run()` appelle la méthode `execute()`, attrape toutes les exceptions et les rapporte aux listeners s'il en existe, sinon, elle les rapporte à la sortie standard. La méthode `execute()`, boucle sur la méthode `iterate()` jusqu'à ce qu'elle renvoie faux.

La méthode `iterate()` à son tour appelle les méthodes `prefire()`, `fire()` et `postfire()` sur l'acteur composite supérieur. Dans le cas où l'acteur composite supérieur renvoie faux le Manager arrête son exécution. On peut aussi terminer une exécution en appelant les méthodes `terminate()` ou `finish()` sur le directeur.

8.2.9 Exécution d'un modèle

Une exécution peut se faire d'une manière contrôlée en appelant les méthodes `initialize()`, `iterate()`, et `wrapup()` sur le directeur directement.
Puisque les détails d'une exécution à travers ces trois étapes sont d'une grande importance pour la suite de ce travail, nous donnons ci-dessous un développement d'une exécution d'un modèle PTOLEMY II.

Initialisation

Un modèle exécutable de PTOLEMY II est composé d'un acteur composite supérieur[2] qui caractérise le modèle avec une instance de directeur et une instance de Manager qui lui sont dédiés. Le directeur implémente la sémantique de ce modèle de calcul pour régir l'exécution des acteurs contenus par l'acteur composite. Par conséquent, il fournit le contrôle global du flot d'exécution. Les directeurs sont donc des propriétés des acteurs composites.

– L'exécution d'un modèle dans PTOLEMY II commence par son initialisation. Le Manager lance sa méthode `initilize()`. Celle-ci appelle successivement les méthodes `preinitialize()`, `resolveTypes()` et `initialize()` de l'acteur composite supérieur.

  La méthode `preinitialize()` de cet acteur une fois lancée, valide ses attributs et appelle ensuite la méthode `preinitialize()` du directeur exécutif supérieur. Celui-ci appelle la même méthode sur les acteurs du premier niveau hiérarchique. Si ceux-ci sont des composites opaques, ils appellent également la même méthode sur leurs directeurs locaux respectifs.

  La méthode `preinitialize()` de chaque directeur local qui n'est appelée qu'une fois par exécution valide les attributs et appelle les méthodes `preinitialize()` de tous les acteurs gouvernés par ce directeur. Ici le temps n'est pas encore placé.

  A son tour la méthode `preinitialize()` de l'acteur qui également n'est exécutée qu'une seule fois avant toutes les autres méthodes d'action crée les receivers et valide les attributs de cet acteur et ceux de ses ports. Cette méthode ne peut pas produire des données de sortie puisque la résolution de type n'est pas encore faite. Les classes dérivées peuvent la redéfinir pour exécuter des fonctions additionnelles d'initialisation.
  Les acteurs atomiques situés au niveau supérieur reçoivent leur préinitialisation directement du directeur exécutif supérieur

– Ensuite, la méthode `resolveTypes()` du Manager vérifie les types sur tous les raccordements et résout les types non déclarés.

[2]Un acteur composite supérieur est aussi appelé acteur composite top-level

– Enfin, le Manager lance sa méthode `initialize()` qui appelle la même méthode sur l'acteur composite supérieur. Celui-ci invoque également la même méthode sur le directeur exécutif supérieur. Comme pour la préinitialisation, celui-ci appelle la même méthode sur les acteurs du premier niveau hiérarchique, qui, à leur tour appellent également la même méthode sur leurs directeurs locaux respectifs.

  La méthode `initialize()` de chaque directeur local place le temps courant à 0,0 ou au temps du directeur exécutif supérieur, et appelle la méthode `initialize()` de chaque acteur gouverné par ce directeur. Cette méthode peut être appelée au milieu d'une exécution, si la réinitialisation est souhaitée.

  Et, la méthode `initialize()` de chaque acteur exécute l'initialisation spécifique au domaine en appelant la méthode `initialize()` de leur directeurs respectifs en se passant en paramètre. Puisque la résolution de type a été accomplie et le temps courant est fixé à ce niveau, la méthode `initialize()` de l'acteur contenu peut produire des événements de sortie ou des événements programmés. Comme pour la préinitialisation, les acteurs atomiques situés au niveau supérieur reçoivent leur initialisation directement du directeur exécutif supérieur

Itération

Le mécanisme d'itération commence par la méthode `iterate` du Manager qui appelle une itération du modèle. Rappelons qu'une itération est composée de l'exécution des changements demandés par la méthode `requestChange()` et de la résolution de type, si nécessaire et de l'appel des méthodes d'action `prefire()`, `fire()`, et `postfire()`, dans cet ordre.
L'appel de ces deux dernières méthodes est conditionné par la valeur de retour de la méthode `prefire()`, sinon, la méthode `fire()` ne sera appelé qu'une fois, suivi la méthode `postfire()`.

En effet, la méthode `prefire()` est la seule de ces trois méthodes qui est appelée exactement une fois par itération. Elle renvoie un booléen qui indique si l'acteur souhaite être lancé. Tant que la méthode `prefire()` renvoi faux, aucune autre méthode ne sera appelée. Cette méthode sert donc à vérifier les conditions d'acceptation du lancement d'un acteur. Elle peut également être utilisée pour effectuer une opération qui se produira exactement une fois par itération.

Une fois la méthode `prefire()` retourne, intervient la méthode `fire()` qui est le point principal de l'exécution et qui est généralement responsable de la lecture des inputs et de la production des outputs. Elle peut également lire les valeurs courantes de paramètre, et la sortie peut dépendre de ces paramètres. Certains domaines appellent la méthode `fire()` à plusieurs reprises jusqu'à ce qu'un certain état de convergence soit atteint. Ainsi, elle ne met pas à jour l'état persistant, donc, ne devrait pas changer l'état de l'acteur.

Une fois appelée, la méthode `fire()` d'un acteur composite transfère les données à partir des ports d'entrée de ce composite aux ports reliés sur l'intérieur si cet acteur est opaque, et puis appelle la méthode `fire()` de son directeur local. Le transfert est fait en appelant la méthode `transferInputs()` du directeur local dont le comportement exact dépend du domaine. Le Directeur délègue cette opération à la méthode du IOPort du même nom.

Puis, la méthode `fire()` du directeur local appelle une itération sur tous les acteurs de l'acteur composite gouverné par ce directeur. Après le retour de la méthode `fire()` du directeur, elle envoie les données de sortie créées en appelant la méthode `transferOutputs()` du directeur local qui la délègue à la méthode du port du même nom. La méthode `fire()` de l'acteur atomique peut contenir les actions de première exécution.

Après la méthode `fire()`, la méthode `postfire()` vient mettre à jour l'état persistant, et déterminer si l'exécution d'un acteur est complète. La valeur de retour de la méthode `postfire()` est une booléenne qui indique au directeur si l'exécution de l'acteur est complète ou pas. Les Directeurs de PTOLEMY II étant responsables dans le sens de [81] évitent de réitérer un acteur dont la méthode `postfire()` retourne faux.

La méthode `postfire()` termine une itération, qui peut impliquer la mise à jour l'état local et peut contenir des opérations à exécuter à la fin de chaque exécution de son itération. Une itération s'exécute donc de la manière suivante ; la méthode `prefire()` de l'acteur composite supérieur appelle la méthode `prefire()` du directeur exécutif supérieur.
Ce dernier appelle la même méthode sur chacun des acteurs au niveau supérieur selon l'ordonnancement établi.

Ceux-ci une fois lancés appellent la même méthode sur leurs directeurs locaux respectifs. Si le directeur est prêt à être lancé, la méthode `prefire()` du directeur local retourne « vraie». Elle n'appelle pas `prefire()` sur les acteurs contenus. Si ce directeur n'est pas au niveau supérieur de la hiérarchie, et si le temps courant du modèle englobant est plus grand que le temps courant de ce directeur, alors le directeur intérieur met à jour le temps courant correspondant à celui du modèle englobant.

Les directeurs de domaine peuvent redéfinir cette méthode pour fournir le comportement spécifique d'un domaine. Cependant, ils devraient appeler super.`prefire()` s'ils souhaitent propager le temps.

Wrapup

La méthode `wrapup()` est utilisée pour afficher les résultats. Elle est appelée exactement une fois à la fin d'une exécution ou lorsqu'une exception qui arrête l'exécution se produit.

8.2.10 Domaines temporisés et signaux

Un acteur dont le comportement dépend du temps modèle courant devrait implémenter l'interface TimedActor. C'est une interface sans méthode. L'implémentation de cette interface signale au directeur que l'acteur dépend du temps. Les domaines qui n'ont aucune notion significative de temps ne l'implémentent pas. Un acteur peut accéder au temps modèle courant avec la syntaxe.

```
double currentTime = getDirector().getCurrentTime()
```

Un acteur peut demander une invocation à un temps futur en utilisant la méthode `fireAt()`, `fireAtCurrentTime()`, ou `fireAtRelativeTime()` du directeur. Pour un directeur correctement implémenté, ces méthodes retournent immédiatement.

Dans le domaine DE, les méthodes `fireAt()` et `fireAtRelativeTime()` prennent deux arguments, un acteur et un temps alors que la méthode `fireAtCurrentTime()` ne prend qu'un seul argument qui est un acteur. Le directeur est responsable de l'exécution d'une itération de l'acteur indiqué au temps indiqué.
Les appels de `fireAt()` sont implémentés dans la méthode `postfire()` car la demande d'un futur lancement est un état persistant. Les méthodes `fireAtCurrentTime()`, `fireAt()` ou `fireAtRelativeTime()` du directeur DE permettent des lancements différés des acteurs qui le souhaitent.

En effet, un composant peut s'auto-programmer pour être exécuté à un temps futur en plaçant un événement *« pur »*[3] sur la file d'attente d'événement, avec lui-même comme composant de destination. Quand cet événement pur est retiré de la file d'attente, le composant peut être exécuté. Cette production d'événements purs initiaux ou d'événements réguliers de sortie peut être faite avec la méthode `initialize()`

Dans le domaine CT, il existe deux types de signaux à savoir les signaux continus et les signaux d'événements discrets pour lesquels d'ailleurs le temps est continu. Ces deux types de signaux affectent directement le comportement des receivers qui les contiennent.
Dans ce domaine, un receiver peut être un CTReceiver continu ou un CTReceiver discret. Un CTReceiver continu contient un échantillon d'un signal continu au temps courant. La lecture d'un token de ce receiver ne consommera pas le token. Tandis qu'un CTReceiver discret peut ne pas contenir un événement discret. La lecture d'un événement du CTReceiver discret consommera l'événement. Les événements sont donc traités exactement une fois. Et, la lecture d'un CTReceiver discret vide n'est pas permise.

[3]Un événement étant composé d'une donnée et d'un temps. Un événement pur est un événement qui n'est composé simplement que du temps

8.3 Intégration des classes de base dans PTOLEMY II

8.3.1 Structure d'un modèle hétérogène non-hiérarchique

Dans PTOLEMY II, nous représentons un modèle hétérogène non-hiérarchique par un acteur composite disposant d'un manager et d'un directeur hétérogène non-hiérarchique que nous nommons «HDirector».
Cet acteur composite représentant le modèle, inclut les différents HICs associés aux différents regroupements d'acteurs englobés dans des objets qu'on appelle domaines[4]. Ces domaines que nous nommons Ω_i sont matérialisés par des acteurs composites opaques dépourvus de ports mais disposant chacun d'un directeur régulier qui lui est dédié comme montré sur la figure 8.5.

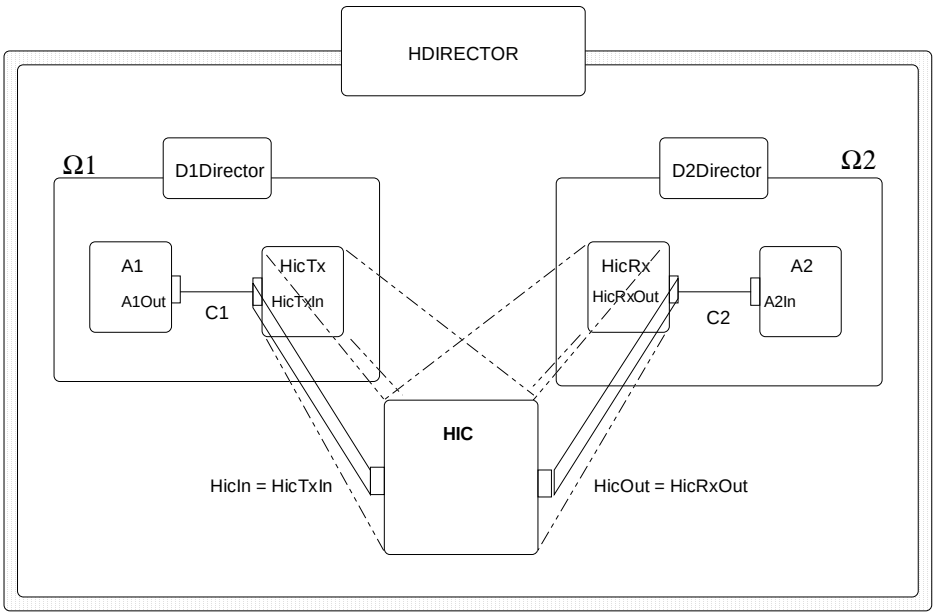


FIG. 8.5 – Les directeurs et les entités

Dans la terminologie de PTOLEMY II, HDirector est un « Directeur Exécutif » et un directeur régulier est un « Directeur Local ».

Le déroulement de l'exécution d'un modèle hétérogène non-hiérarchique est basée sur une intime coopération entre HDirector et les différents HICs du système.
En effet, cette coopération est faite de telle manière que le flot de contrôle, assuré par HDirector s'intéresse aux activations des différents domaines selon un ordonnancement approprié. Quant au flot de données qui organise la communication et dont les HICs sont des éléments pivots, il permet la circulation des données entre ces différents domaines.

[4]Dans la suite de ce document, nous utiliserons indistinctement les termes domaine et sous-système

Ainsi, nous représentons ce modèle hétérogène en transposant dans PTOLEMY II, le schéma simplifié d'un modèle à deux domaines présenté dans les chapitres précédents. Cette transposition est montré sur la figure 8.5, où HDirector gouverne directement les deux acteurs composites Ω_1 et Ω_2 et gouverne indirectement l'acteur HIC.

8.3.2 Les classes de base

Dans la spécification UML élaborée au chapitre précédent, nous avons spécifié deux classes fondamentales qui sont HicComp et HModEx et une interface que nous avons nommée *HeterogeneousBehavior*. La première classe caractérise le Composant à Interface Hétérogène et la seconde caractérise le Modèle d'Exécution Hétérogène Non-Hiérarchique.

Cependant, comme souligné dans la conclusion du chapitre précédent, afin de garder une proximité structurelle et terminologique avec la plate-forme PTOLEMY II, nous allons légèrement transformer les deux classes précitées.

En effet, en ce qui concerne les terminologies de ces classes, nous adopterons les appellations suivantes : HicActor remplace la classe HicComp et HDirector remplace la classe HModEx dans PTOLEMY II.

Sur le plan structurel, ces classes intègrent une plate-forme disposant de ses propres classes. Or d'une part, nous savons que, dans PTOLEMY II, toute classe caractérisant un acteur doit spécialiser la classe AtomicActor. Et, d'autre part nous savons également que HicActor est un acteur. Donc HicActor spécialise la classe TypedAtomicActor qui elle-même spécialise la classe AtomicActor. Ceci est lié au respect du typage dans la spécification de HIC. De même, la classe HDirector qui spécialise la classe Director.

De plus, puisque le mode opératoire de PTOLEMY II décompose sa phase d'initialisation en deux méthodes, `preinitialize()` et `initialize()`, l'opération *initialize()* présentée dans le chapitre précédent sera également décomposée en deux méthodes portant les mêmes noms.

Interface *HeterogeneousBehavior*

L'interface *HeterogeneousBehavior* impose aux classes qui spécialisent la classe HicActor de faire l'engagement d'implémenter la méthode `computeBehavior()` qui calcule le comportement d'un HIC à la frontière des différents modèles de calcul.
Cette méthode n'est pas implémentée dans la classe HicActor. Il appartient donc au concepteur du système de la redéfinir et d'y mettre en oeuvre selon sa convenance, le code destiné soit à une simple interprétation des données lorsqu'elle passe d'un domaine vers un autre et/ou le code destiné au réel calcul du comportement hétérogène à la frontière des modèles de calcul qu'utilise ce HIC.

Classe HicActor

Pour un acteur, l'opération *finish()* spécifiée au chapitre précédent n'est pas explicitement intégrée dans PTOLEMY II. Cette plate-forme utilise plutôt des variables booléennes retournées par des méthodes `prefire()` et `postfire()`. C'est pourquoi, cette opération ne sera pas représentée.

Par ailleurs, dans PTOLEMY II, le contrôle étant séparé de la communication, les méthodes relatives au contrôle d'un acteur sont déclarées dans l'interface *`Executable()`* qu'implémente la classe AtomicActor par l'intermédiaire de l'interface *`Actor()`*. Tandis que les méthodes relatives à la communication sont portées par la classe IOPort. Lorsque le modèle impose des contraintes de type, on utilise respectivement les classes TypedAtomicActor et TypedIOPort qui spécialisent respectivement les classes AtomicActor et IOPort.

De ce fait, du point de vue de son contrôle, la classe HicActor hérite directement des méthodes `initialize(), preinitialize, prefire(), fire(), postfire()` de la classe TypedAtomicActor.
Cependant, dans la classe TypedAtomicActor, ces méthodes n'offrent qu'une implémentation de base. C'est pourquoi parmi elles, les méthodes `initialize()` et `fire()` doivent obligatoirement être redéfinies afin de permettre la validation des attributs et le calcul du comportement hétérogène. De manière optionnelle et dépendant de la spécification établie par le concepteur, les méthodes `prefire()` et `postfire()` pourraient également être redéfinies pour implémenter certaines contraintes imposées par sa spécification.

Du point de vue de la terminologie, nous proposons les appellations suivantes, issue de PTOLEMY II, pour certains attributs et méthodes utilisés au chapitre précédent :

Appellation intermédiaire	Appellation dans PtolemyII
hModEx	hdirector
modEx	director
existData()	hasToken()
getModEx()	getDirector()
isFull()	hasRoom()
preCondition()	prefire()
trigger()	fire()
postCondition()	postfire()
read()	get()
write()	send()

FIG. 8.6 – Méthodes de HicActor dans PTOLEMY II

Dans l'architecture de PTOLEMY II, un port est un objet à part entière qui porte les opérations de communication. C'est pourquoi, pour se conformer à cette architecture, nous retirons de HicActor, toutes les opérations de communication qu'il portait au chapitre précédent et les intégrons, ou plutôt utilisons celles définies dans l'entité IOPort PTOLEMY II comme montré sur la figure 8.7.

Diagramme des classes

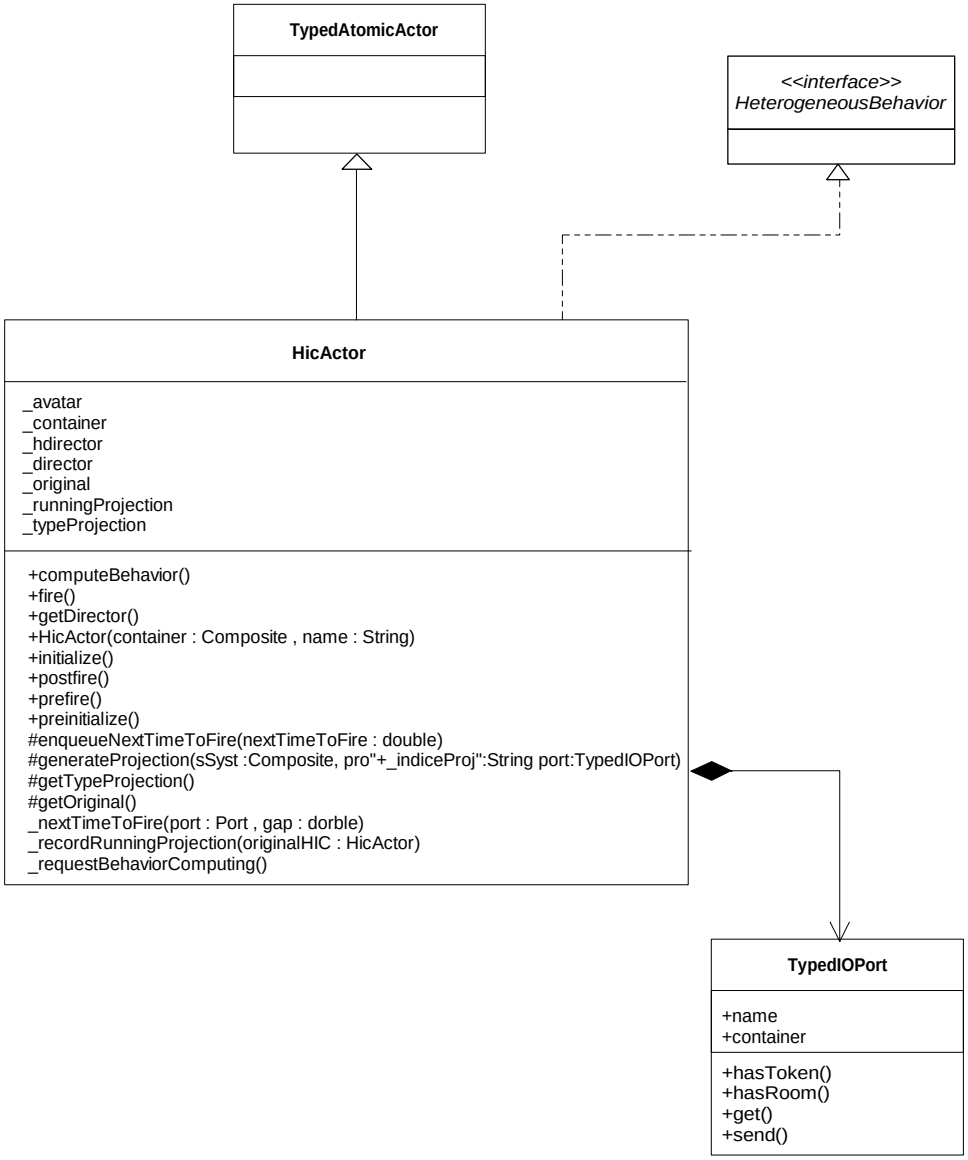


FIG. 8.7 – Diagramme de classes de HicActor

Classe HDirector

Comme pour un acteur, dans PTOLEMY II, la classe Director est la classe de base pour les directeurs. Cette classe fournit une implémentation par défaut pour le contrôle de l'exécution d'un modèle. Mais, la sémantique exacte des différents modèles de calcul est implémentée dans des directeurs qui la spécialisent. Ces directeurs travaillent en parfaite collaboration avec des receivers fournis par les directeurs et intégrés dans des ports pour assurer la tâche de communication. En outre, comme pour les acteurs, les directeurs implémentent aussi l'interface *`Executable()`*.

Et, puisque HDirector est un directeur, il spécialise donc la classe Director. De même, la classe HDirector hérite directement des méthodes `initialize(), preinitialize, prefire(), fire(), postfire()` de la classe Director.

Pour les raisons évoquées ci-dessus, nous allons redéfinir les méthodes `preinitialize()`, `initialize()` et `fire()` afin de permettre au HDirector de pouvoir prendre en charge des tâches additionnelles de création des sous-systèmes, de projection des différents HICs, de placement des acteurs dans leurs sous-systèmes associés, de déconnection et de reconnection des ports, de génération des ports virtuels dans les sous-systèmes et d'ordonnancement des sous-systèmes. Ces fonctions additionnelles sont donc des préalables structurelles et opérationnelles qui permettent à notre HDirector de supporter une exécution selon la spécification de l'approche hétérogène non-hiérarchique que nous avons élaborée. Son diagramme de classe est montré sur la figure 8.9

Il est intéressant de signaler que cette exécution hétérogène non-hiérarchique est faite en collaboration avec des HICs. Le HDirector dispose d'un contrôle direct sur les sous-systèmes et d'un contrôle indirect sur les HICs car ces derniers sont activés par leurs projections.

Comme pour la classe HicActor, du point de vue de la terminologie, nous proposons les appellations suivantes, issues de PTOLEMY II pour certains attributs et méthodes utilisés au chapitre précédent :

Appellation intermédiaire	Appellation dans PtolemyII
HModEx preCondition() trigger() postCondition()	Hdirector prefire() fire() postfire()

FIG. 8.8 – Méthodes de HDirector dans PTOLEMY II

Diagramme des classes

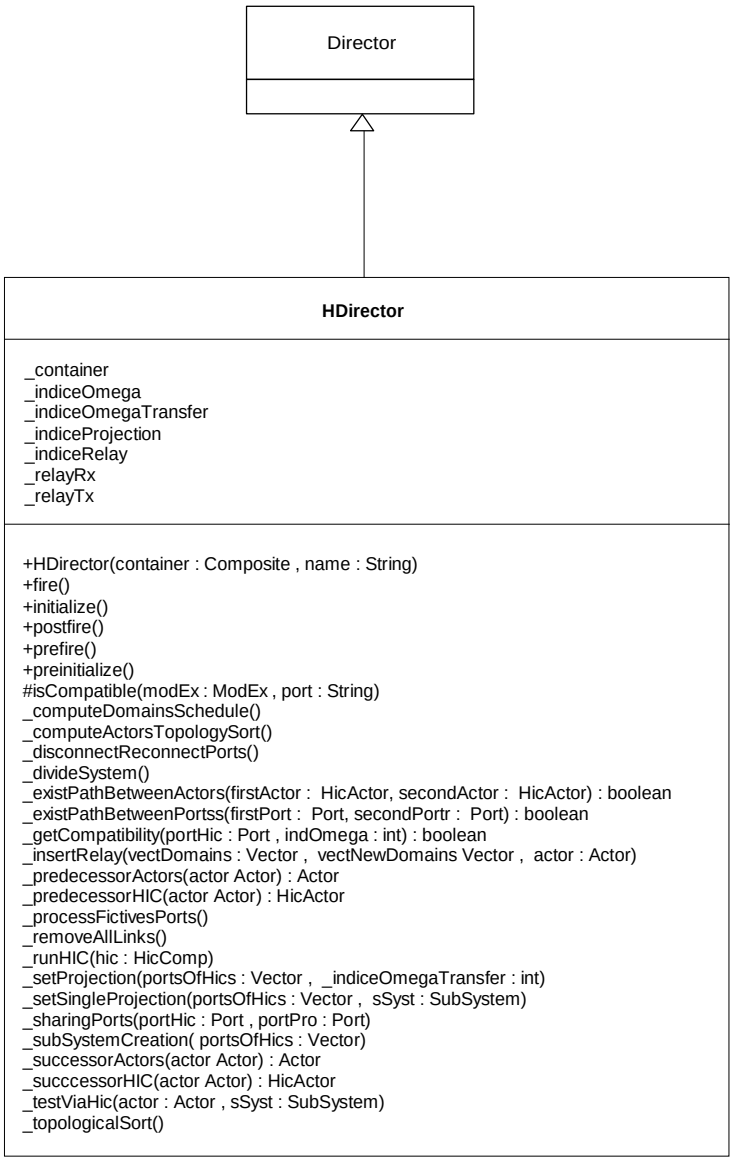


FIG. 8.9 – Diagramme des classe de HDirector

Maintenant que les différentes classes sont structurées conformément à l'architecture de la plate-forme PTOLEMY II, nous allons dans la section suivante observer l'intégration du mécanisme d'activation des différentes entités issues de HIC. Nous allons également montrer comment nous avons intégré le cas de lancement d'une projection DE positionnée en producteur dans un sous-système.

8.4 Activation et Postage du temps par HicActor

8.4.1 Activation de HicActor

La méthode `fire()` de HIC est utilisée aussi bien par lui-même que par ses projections. L'entité qui s'exécute commence par son auto-identification en obtenant son directeur à l'aide de la méthode `getDirector()`. Ensuite, connaissant le directeur qui la gouverne, elle décline sa nature pour déterminer le branchement à la méthode appropriée, celle-ci pouvant être la méthode dédiée à l'activation du HIC ou celle dédiée au calcul de son comportement hétérogène. Ce branchement est fait selon les règles suivantes :

– Si cette entité est gouvernée par un directeur régulier, on en déduit que c'est une projection. De ce fait, elle obtient la référence de son HIC original par l'intermédiaire de HDirector en utilisant l'instruction suivante :
`original = (HicActor)hdirector.htabProHic.get(this)`.
Puis, pour permettre au HIC de ne lire que les ports de l'entité qui l'a sollicité, l'entité projection s'enregistre auprès de son HIC en invoquant la méthode `recordRunningProjection()` de son HIC original.
Enfin l'entité projection lance l'activation de son HIC original en invoquant la méthode `requestBehaviorComputing()`.

– Par contre si cette entité est gouvernée par HDirector, on en déduit donc que c'est un HIC, elle active simplement sa méthode `computeBehavior()` appropriée au calcul de son comportement hétérogène.

Ce mécanisme est celui qui est implémenté dans la méthode `fire()` de HDirector. Son code est présenté dans le code suivant :

```
public void fire() throws IllegalActionException {
    _container = (TypedCompositeActor)this.getContainer();
    _director = _container.getDirector();
    //Self-identification of HIC
    //If it is a projection, require from HDirector, the
    //activation of  HIC
    if (!(_director instanceof HDirector)){
        _hdirector = (HDirector)_container.getExecutiveDirector();
        _original = (HicActor)_hdirector.htabProHic.get(this);
        _original._recordRunningProjection(this);
        _requestBehaviorComputing();
    }
    // Else, if it is a HIC, activate the method  computeBehavior()
    else if (_director instanceof HDirector){
        computeBehavior();
        return;
    }
}
```

8.4.2 Postage du temps vers un domaine DE

Dans les chapitres précédents, nous avons vu que le HDirector était complètement déchargé du mécanisme de communication entre les acteurs. De plus, les sous-systèmes en utilisant l'abstraction non-hiérarchique sont dépourvues des ports et communiquent entre eux via des canaux abstraits hétérogènes qui incluent le HIC qui les a générés.
Lorsqu'un sous-système consommateur utilise le modèle de calcul Discrete-Event, de part les caractéristiques des acteurs DE [13] [103], pour son exécution, une projection productrice s'y trouvant est obligée de s'auto-poster au préalable un événement pur dans la file d'attente de son directeur DE local. Pour cela, nous allons dédier cette tâche à son HIC original

Il s'ensuit que selon la spécification du concepteur de système, après avoir calculé son comportement, le temps de déclenchement d'une projection DE sera déterminé par son HIC original. Or, puisque le HIC partage ses ports avec ses projections, il connaît par conséquent le port de sa projection dont il doit programmer l'activation. De ce fait, il peut obtenir la référence de celle-ci par la méthode `getContainer()`. Ensuite il invoque la méthode `enqueueNextTimeToFire()` de ladite projection qui se charge de poster un événement pur dans la file d'attente de son directeur local. Nous donnons ci-dessous le code relatif à la méthode `nextTimeToFire()`.

```
protected void _nextTimeToFire(TypedIOPort port,
                               double travelGapTime)
                throws IllegalActionException {
    HicActor actorToSchedule = (HicActor)port.getContainer();
    actorToSchedule._enqueueNextTimeToFire(travelGapTime);
}
```

A l'invocation de la précédente méthode, la méthode `enqueueNextTimeToFire()` de la projection concernée obtient son directeur par la méthode `getDirector()`. Puis, de ce directeur, elle obtient son temps courrant par sa méthode `getCurrentTime()`. Ensuite elle s'auto-poste un événement pur à travers l'instruction `dedirector.fireAt(this, nextTimeToFire + currentTime)`.
Enfin elle rend la main au HIC qui a initié le postage de cet événement pur pour terminer ses activités. Nous donnons ci-dessous le code relatif à la méthode `enqueueNextTimeToFire()`.

```
protected void _enqueueNextTimeToFire(double nextTimeToFire)
                throws IllegalActionException {
    _container = (TypedCompositeActor)getContainer();
    _director = _container.getDirector();
    _dedirector = (DEDirector)_director;
    _currentTime = _dedirector.getCurrentTime();
    _dedirector.fireAt(this, nextTimeToFire + _currentTime);
    return ;
}
```

8.5 Assignation des différentes phases dans les méthodes de HDirector

Les contraintes imposées par les différentes étapes de déroulement d'un Directeur dans PTOLEMY II nous impose une organisation appropriée des différentes phases spécifiques d'un modèle d'exécution hétérogène dans les différentes méthodes de HDirector.
En effet, hormis la phase d'exécution qui se déroule pendant l'exécution effective du modèle, les phases de partitionnement et d'ordonnancement restent des phases totalement préliminaires à l'exécution réelle du modèle.

Pour cela, la phase d'exécution sera effectuée dans la méthode `fire()` de HDirector et les phases de partitionnement et d'ordonnancement seront effectuées respectivement dans les méthodes `preinitialize()` et `initialize()` de HDirector. Ce choix est justifié dans les sections qui suivent.

8.5.1 Intégration de la phase de Partitionnement dans la préinitialisation

Nous avons vu à la section 8.2.9 que c'est dans l'étape de préinitialisation caractérisée par la méthode `preinitialize()`, que le système crée des receivers et valide les attributs. De plus, dans la même étape, la résolution de type n'est pas encore faite et le temps n'est pas encore placé. Puisque le HDirector restructure le système pendant l'étape de partitionnement, il nous paraît judicieux de placer cette étape dans la méthode `preinitialize()` pour deux raisons fondamentales :

– il nous semble cohérent que la fourniture des receivers aux acteurs consommateurs par les Directeurs réguliers intervienne après la restructuration du système,

– puisque les méthodes `disconnectionReconnectionPorts()` et `processFictivePorts()` invoquées pendant l'étape du partitionnement manipule les ports, il serait imprudent de placer la restructuration du système après avoir effectué la résolution des types et validé les attributs.

Cependant, le principe de délégation des tâches interdit au HDirector d'interférer dans la gestion des sous-systèmes. La création des receivers dans les projections et dans les autres entités consommatrices ainsi que la validation de leurs attributs et de leurs ports revient naturellement aux directeurs des sous-systèmes.
Pour cela, après avoir achevé le partitionnement, le HDirector invoque la méthode `preinitilalize()` de sa super classe pour invoquer la même méthode sur les sous-systèmes. Lesquels à leur tour vont invoquer la même méthode sur leurs directeurs respectifs pour créer les receivers dans les projections et dans les autres entités consommatrices et valider leurs attributs et ceux de leurs ports.
Le code de la méthode `preinitilalize()` de HDIrector se présente comme-suit :

```
public void preinitialize() throws IllegalActionException {
    _divideSystem();
    _disconnectionReconnectionPorts();
    _processFictivePorts();
    super.preinitialize();
}
```

Il est intéressant de remarquer que pendant l'exécution de la méthode de préinitialisation, le HDIrector appelle d'abord sa méthode `divideSystem()` qui divise le système à la frontière des modèles de calcul.
Puis, du fait que lorsqu'un acteur est mis dans un domaine, il est simplement retiré du modèle original et placé dans le sous-système du nouveau modèle, mais les connexions à ses ports restent. Pour que le nouveau modèle ait le même comportement que le modèle initial, toutes les connexions seront retirées et les acteurs seront reconnectés dans leurs domaines respectifs selon leurs anciens schémas de communication. Pour cela, le modèle invoque sa méthode `disconnectionReconnectionPorts()`.
Ensuite, il peut arriver qu'un system soit divisé en sous-systèmes non-connectés. Mais, dans PTOLEMY II certains domaines tel que le SDF ne supportent pas les modèles avec des acteurs non connectés. Pour cela, le HDirector invoque sa méthode `processVirtualPorts()` qui va créer des dépendances virtuelles en connectant les projections concernées à travers des ports virtuels qui seront ignorés par des HICs.

8.5.2 Intégration de la phase d'Ordonnancement dans l'initialisation

Entre cette étape d'initialisation et celle de la préinitialisation, le manager avait lancé sa méthode `resolveType()` qui a résolut les types.

A l'étape de l'initialisation les Directeurs locaux placent le temps à 0.0. Pour cela, cette étape n'interviendra que lorsque la restructuration du système sera achevée.
Ainsi, lorsque toutes les tâches préalables sont achevées, HDirector invoque sa méthode `computeDomainsSchedule()` qui vient ordonnancer les différents sous-systèmes avant de les activer. Nous optons pour l'intégration de l'étape d'ordonnancement dans cette méthode, mais, après la méthode `initilalize()` de la super classe de HDirector. En effet, l'ordonnancement doit immédiatement précéder l'exécution et doit en effet être la dernière phase à effectuer par le HDirector avant l'exécution. La méthode `initilalize()` est :

```
public void initialize() throws IllegalActionException {
    super.initialize();
    _computeDomainsSchedule();
```

8.5.3 Intégration de la phase d'Exécution dans l'itération

Dans la phase d'itération, le HDirector appellera une seule fois la méthode `prefire()` par itération sur chaque sous-système. Comme le temps courant de HDirector est fixé à 0.0 donc le temps englobant ne sera jamais plus grand que le temps courant des directeurs locaux. Les directeurs locaux ne feront pas de mise à jour de temps à l'extérieur mais géreront leur temps courant indépendamment de l'extérieur.

Si le sous-système est prêt à être lancé, le HDirector invoque sa méthode `fire()`. Comme les sous-systèmes utilisent une abstraction non-hiérarchique, ils n'ont donc pas de ports, il n'y a donc aucune possibilité de transfert des données dans les sous-systèmes autre que le passage par des HICs. Et, c'est pendant cette méthode que les projections du sous-système lancé, solliciteront du HDirector l'activation de leur HIC.Dans le où le HIC nécessite la mise à jour et la persistance de ses données, il invoque sa méthode `postfire()`.

Ce mécanisme étant l'exécution effective du modèle, cette phase d'exécution ne peut qu'être affectée à la méthode `fire()` de HDirector.

Le modèle d'exécution commence par mettre son booléen *refire* à vraie. Ensuite, il incrémente son paramètre de compteur d'itération, `_iterationCount`, car, le nombre d'itération ne doit pas dépasser le nombré fixé par l'utilisateur.

Il lance successivement les méthodes `prefire()`, `fire()` et `postfire()` de chaque domaine. A la fin d'une itération, c'est-à-dire, d'un cycle complet d'activation des domaines, il décompte le compteur d'itération, et répète ainsi de suite jusqu'à son épuisement.

Nous donnons ci-dessous le code relatif de la méthode `fire()` de HDirector.

```
public void fire() throws IllegalActionException {
    boolean refire;
    do {
        refire = true;
        _iterationsCount++;
        for (int i = 0 ; i < vTopoTriOfProjections.size() ; i++){
            TypedCompositeActor domainToFire = (TypedCompositeActor)
                                vTopoTriOfProjections.elementAt(i);
            domainToFire.prefire();
            domainToFire.fire();
            domainToFire.postfire();
            if (_iterationsCount == _maxIterations){
                refire = false;
            }
        }
    } while (refire);
}
```

8.6 Déroulement d'un modèle hétérogène non-hiérarchique

Lors de l'exécution d'un modèle hétérogène non-hiérarchique, lorsque le manager lance sa méthode `manager.initilize()`. Celle-ci appelle successivement les méthodes `preinitialize()`, `resolveTypes()` et `initialize()` de l'acteur composite qui représente le modèle hétérogène non-hiérarchique.
De manière pratique, cet acteur composite qui représente le modèle sera une instance de la classe `TypedCompositeActor`.

A son invocation, la méthode `preinitialize()` de l'acteur composite représentant le modèle hétérogène valide ses attributs. Elle appelle ensuite la méthode `preinitialize()` de HDirector qui le gouverne.
La méthode `preinitialize()` de HDIrector restructure dynamiquement le système par la création des sous-systèmes, la projection des HICs, la déconnexion et la reconnexion des ports et la génération des ports virtuels. Ensuite elle appelle la même méthode de la classe Director qui est sa super classe pour valider ses attributs et appeler les méthodes `preinitialize()` sur tous les sous-systèmes qu'il a crée lors du partitionnement.

Puis, vient la deuxième étape, la méthode `manager.resolveTypes()` qui vérifie les types sur tous les raccordements, car, à ce niveau le nouveau système est crée et les ports sont reconnectés. Ensuite, elle résout les types non déclarés s'il y en a.

Ensuite, le manager invoque sa méthode `intialize()` qui lance la méthode `initialize()` du modèle. Celle-ci appelle la méthode `initialize()` de HDirector qui ordonnance les sous-systèmes.

Enfin, la méthode `iterate` de Manager appelle une itération, c'est-à-dire appelle successivement les méthodes `prefire()`, `fire()`, et `postfire()` sur le modèle. Ce dernier appelle dans le même ordre les mêmes méthodes de HDirector.

Dans les sections suivantes, nous allons mettre en évidence les détails de ce qui se passe dans un sous-système pendant ces différentes phases. Pour cela, nous allons considérer le cas simplifié des deux sous-systèmes utilisé à la section 8.3.1.

Nous appelons Ω, l'acteur composite représentant le modèle, Ω_1, le sous-système producteur et Ω_2, le sous-système consommateur.
Le sous-système Ω_1 est constitué d'un acteur producteur A1 et d'une projection consommatrice HicTx et du canal C1. Le port de sortie A1Out de l'acteur A1 est connecté au port d'entrée de la projection HicTxIn par le canal C1. De même le sous-système Ω_2 est constitué d'une projection productrice HicRx et d'un acteur consommateur A2 et du canal C2. Le port de sortie HicRxOut de la projection HIcRx est connecté au port d'entrée A2In de l'acteur A2 par le canal C2.

8.6.1 Preinitialisation

Cette phase est composée des deux importantes tâches assurées respectivement par les méthodes `preinitialize()` de HDirector et par celle de sa super classe, Director.
En effet, pendant cette méthode, au préalable, HDirector restructure le système en appelant successivement les méthodes `divideSystem(), disconnectionReconnectionPorts` et `processVirtualPorts()`. Lorsque cette tâche structurelle est achevée, la méthode `preinitialize()` de HDirector invoque la méthode `preinitialize()` de Director qui vient assurer les fonctions génériques que doit effectuer tout directeur pendant cette phase.

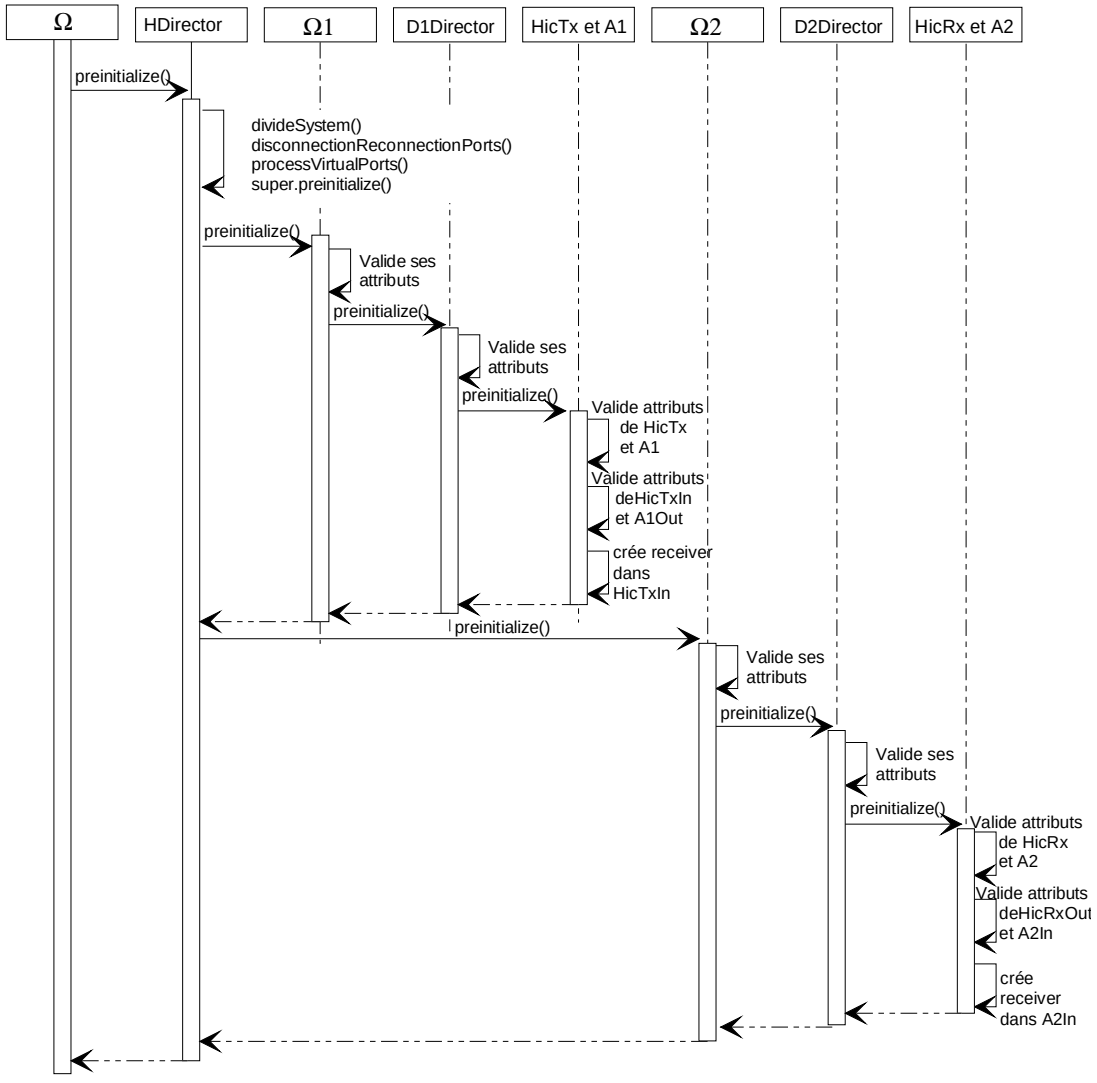


FIG. 8.10 – Digramme de séquence de la préinitialisation du système

La méthode `preinitialize()` de Director invoque la même méthode sur les sous-systèmes

Ω_i. Les Ω_i valident leurs attributs internes et lancent la preinitialisation de leurs Directeurs locaux, D1Director et D2Director. Par principe de délégation, c'est en fait aux Directeurs locaux que revient la charge de préinitialiser les acteurs dans Ω_1 et Ω_2 en appelant la méthode `preinitialize()` sur chacun d'eux.

Lorsque cette méthode est appelée sur un acteur producteur, elle valide simplement ses attributs et ceux de ses ports. Mais, lorsqu'elle est appelée sur un acteur consommateur, en plus de la validation des attributs, elle crée aussi les receivers. C'est ainsi, lorsque D1Director appelle la méthode `preinitialize()` sur la projection HicTx, cette méthode valide ses attributs et ceux de son port HicTxIn. Et, puisque cette dernière est une consommatrice, elle demande à D1Director de lui fournir un receiver pour son port HicTxIn.
De même, cette méthode appelée sur A2 par D2Director, pour la même raison que HicTx, elle valide ses attributs et ceux de son port A2In et demande à D1Director de lui fournir un receiver pour son port A2In.

8.6.2 Création des receivers dans une projection : Cas de HicD1In

Lorsque D1Director lance la méthode `preinitialize()` sur la projection HicTxIn, cette méthode invoque la méthode `_createReceiver()` de HicActor. Cette dernière lance la méthode du même nom sur le port HicTxIn qui active la méthode privée `_newReceiver()` du même port, laquelle à son tour déclenche la méthode `newReceiver()`. Comme le receiver est fourni par le directeur, il est au préalable nécessaire d'obtenir le Directeur local, D1Director qui par sa méthode `_newReceiver()` fournit au HicTxIn un receiver compatible au domaine D1.

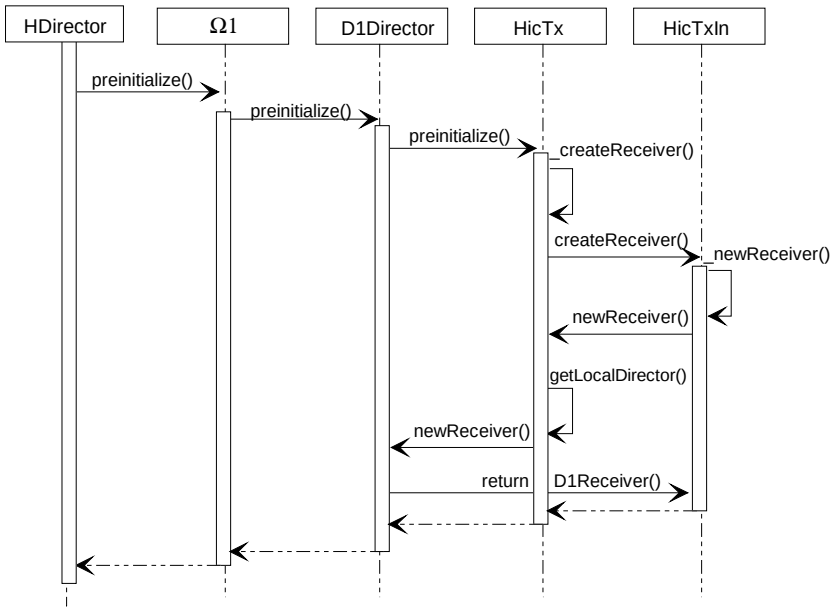


FIG. 8.11 – Digramme de séquence de création de receiver dans HicD1

8.6.3 Initialisation d'un système non-hiérarchique

Quand HDirector est initialisé, il appelle la méthode initialize() sur tous les sous-systèmes qu'il a crées, en l'occurrence Ω_1 et Ω_2. Du fait que HDirector n'interfère pas dans les mécanismes internes des domaines, le temps d'un domaine est strictement géré par son directeur. Pour cela, Ω_1 et Ω_2 appellent la même méthode sur les directeurs locaux qui placent leurs temps à 0.0 et ces derniers appellent initialize() sur tous les acteurs sous leur responsabilité. Et pendant cette initialisation, les acteurs qui ont des spécificités particulières liées à leurs domaines se verront lancer en paramètre par leur directeur local sur leur demande pour valider leurs spécificités.

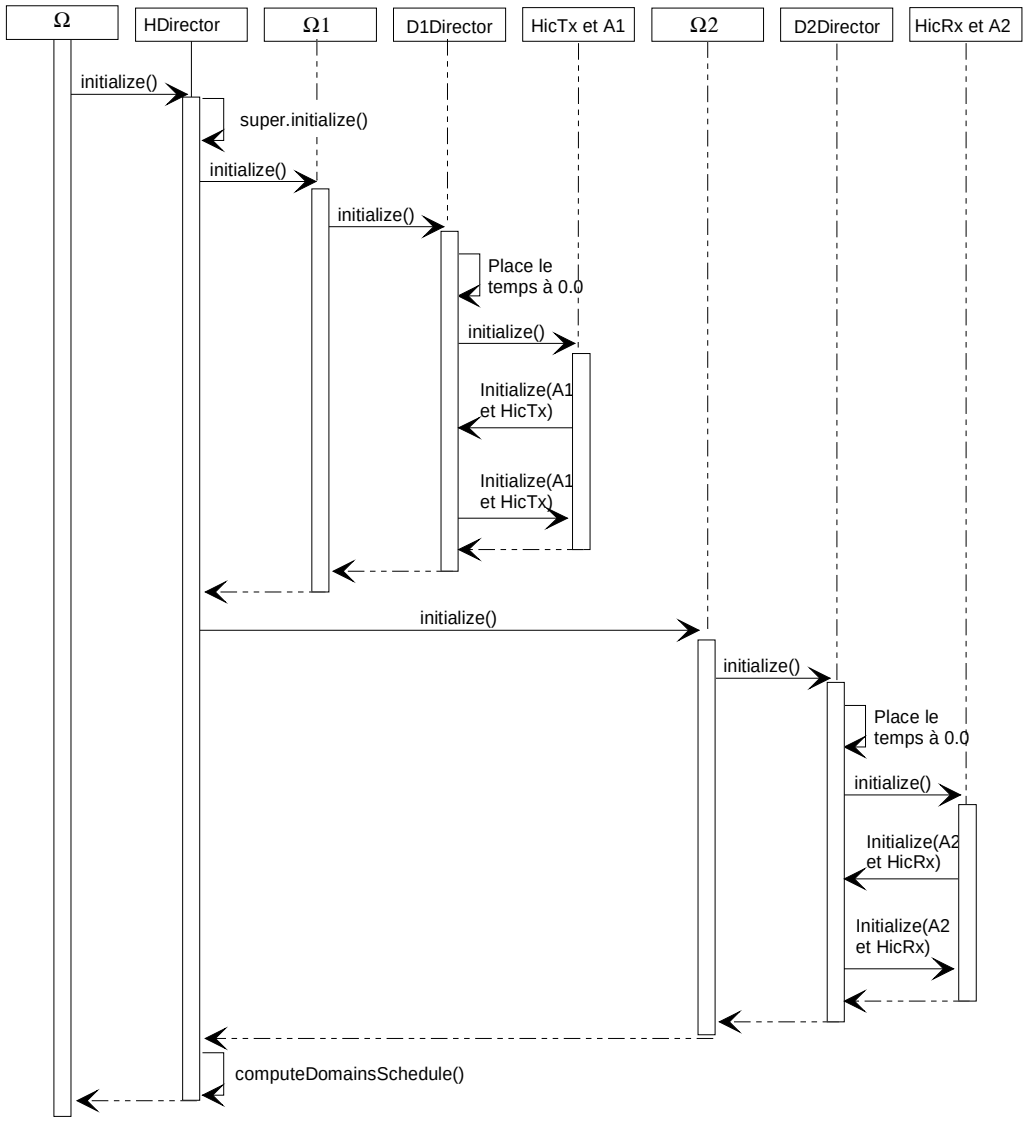


FIG. 8.12 – Digramme de séquence d'initialisation d'un système hétérogène

8.7 Exemples d'application et Simulation

Après l'intégration des classes HicActor et HDirector qui caractérisent respectivement le composant à interface hétérogène et le modèle d'exécution hétérogène non-hiérarchique dans PTOLEMY II, dans cette section, nous présentons deux simulations qui utilisent des détecteurs HICs. Il s'agit de :

– un Redresseur de signal,
– un Modulateur de signal numérique.

Ces simulations préservent la sémantique des données le long de la connexion entre les ports des différents acteurs hétérogènes qu'ils utilisent. Le changement de la sémantique entre les acteurs hétérogènes, c'est-à-dire la conversion de protocoles (si besoin il y a) et la transformation de format de données se produisent à l'intérieur des composants Detectors, Amplifier et Multiplexor qui sont des HICs.
Cependant, ces exemples sont dépourvues de toute complexité technique. Ce sont donc des exemples simples dont le but est la validation de l'approche développée dans cette thèse.

Par ailleurs, puisque l'approche non-hiérarchique est intégrée dans la plate-forme PTOLEMY II, chacun de ces modèles représentant les systèmes ci-dessus sera enveloppé dans un acteur composite qui spécialise l'acteur `TypedCompositeActor` et qui porte le nom de sa simulation, c'est-à-dire :
– `TypedCompositeActor Redresseur` pour le Redresseur de signal,
– `TypedCompositeActor Modulateur` pour le Modulateur de signal numérique.

L'Amplificateur de redressement et le Modulateur sont des HICs. Pour la communication hétérogène, ils disposent d'entrées et de sorties hétérogènes leur permettant de mettre en communication différents composants ayant des ports utilisant des modèles de calcul différents aux frontières desquels ils fournissent un comportement hétérogène.
Ces comportements sont décomposés en autant de comportements secondaire qu'il y a de modèles de calcul dans les composants Detector et Amplifier et Multiplexor. En effet, lorsqu'un de ces composants interprète une entrée, il traduit la signification de cette entrée dans le modèle de calcul associé en signification interne pour ce HIC. Lorsqu'il produit une sortie, il traduit l'information recueillie de ses entrées dans la sémantique de cette sortie selon le modèle de calcul, après une transformation éventuelle.

Ces simulations sont gouvernées par HDirector qui restructure lesdits modèles en acteurs composites au premier niveau hiérarchique selon les comportements à la frontière des différents modèles de calcul utilisés par les couples de composants (Detector et Amplifier) et (Detector et Multiplexor). Il crée des sous-systèmes et délègue le flot de contrôle local et le calcul du comportement interne des composants composites aux Directeurs locaux. Enfin il génère un ordonnancement approprié des différents acteurs composites crées et les active.

8.7.1 Redresseur de signal

Principe

Le premier système hétérogène que nous représentons est un redresseur de signal montré sur la figure 8.13. Il contient un générateur qui fournit un signal sinusoïdal. Ce signal est envoyé dans un détecteur qui réagit à ses différentes alternances.

Le détecteur capture la valeur instantanée du signal source en provenance d'un générateur de signaux. Il compare son signe à celui de la dernière valeur capturée. Si le signal n'a pas changé d'alternance, c'est-à-dire, si les deux signes sont identiques, il transmet simplement cette valeur à l'amplificateur. Dans le cas contraire, hormis le fait de transmettre cette valeur à l'amplificateur, le détecteur lui transmet également un événement de contrôle pour lui permettre de modifier son gain dont il se servira pour le redressement dudit signal.

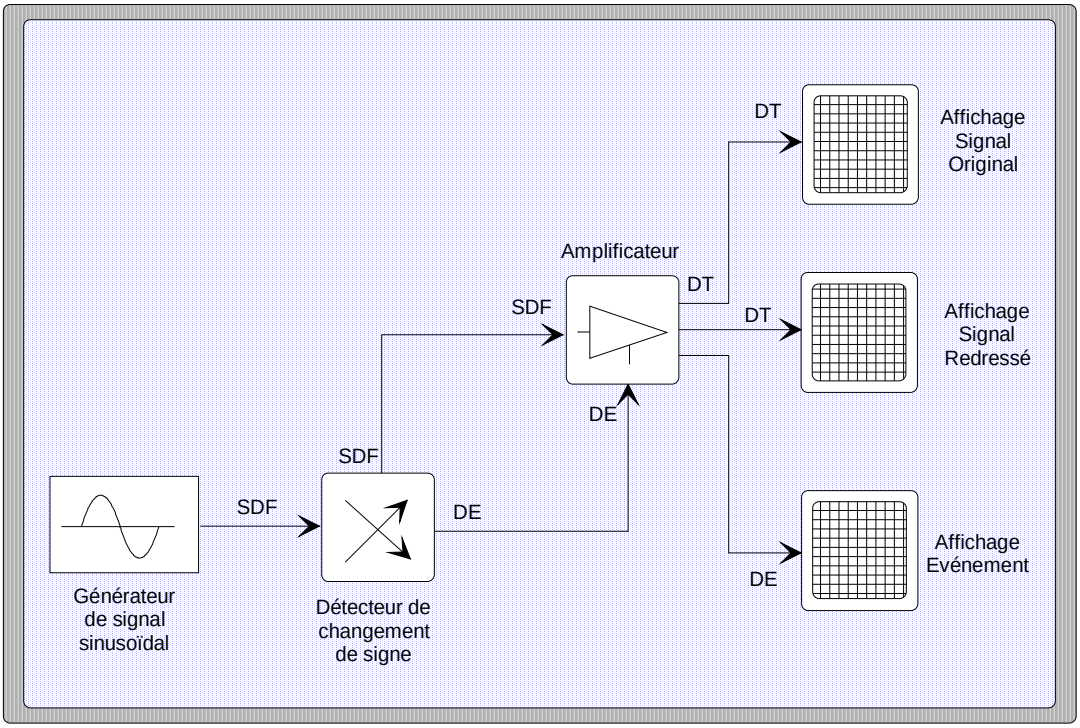


FIG. 8.13 – Exemple d'un Redresseur de signal utilisant deux HICs et plusieurs MoCs

Il est important de signaler que pour ses entrées, ce détecteur utilise le modèle de calcul Flot de Données Synchrones (Synchronous Data Flow, SDF). Et, pour ses sorties, il utilise à la fois le modèle de calcul Flot de Données (SDF) et le modèle de calcul d'Evénements Discrets (Discretes Events, DE).

Du côté de l'amplificateur, à chaque détection d'un événement de contrôle signifiant le

changement d'alternance, il change le signe de son gain. De ce fait, le signe du gain suit toujours celui des alternances. Il est positif aux alternances positives et il est négatif aux alternances négatives. Il s'ensuit que le produit du signal par le gain donnera toujours un signal positif : c'est le signal redressé qui est affiché à l'afficheur approprié.
De plus, si cette détection correspond à un début de redressement, l'amplificateur envoie également un événement à un deuxième afficheur marquant le début du redressement.

De même, signalons pour l'amplificateur, pour ses entrées, l'amplificateur utilise à la fois le modèle de calcul Flot de Données (SDF) et le modèle de calcul d'Evénenements Discrets (DE). Quant à ses sorties, il utilise à la fois le modèle de calcul Discrete Time (DT) et le modèle de calcul d'Evénenements Discrets (DE).

Composition

La composition du système dans PTOLEMY II, dans cette plate-forme, ce modèle peut être composé de six ou sept acteurs selon qu'on le représente sur l'interface graphique de PTOLEMY II appelée VERGIL comme sur la figure 8.14 ou qu'on le représente par voie d'assemblage d'acteurs en utilisant l'API Java de PTOLEMY II comme illustré sur la figure 8.15.

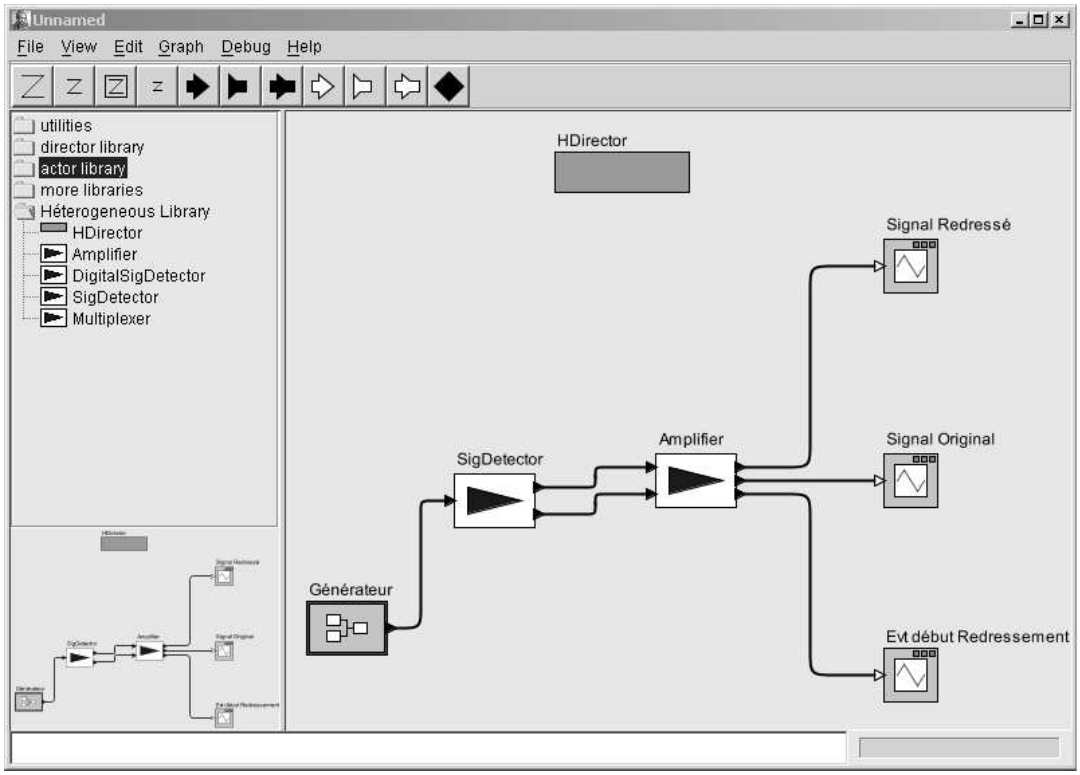


FIG. 8.14 – Représentation du Redresseur de signal dans VERGIL

Cette différence est due au fait qu'en utilisant l'API java, l'acteur `Sinewave` n'y étant pas mis en oeuvre, le générateur de signal est donc remplacé par la composition d'un acteur `Ramp` dont la sortie est connectée à l'entrée de l'acteur `TrigFunction` paramétrée sur *Sinus*.

En somme, le modèle est composé des acteurs suivants : `HDirector`, `Ramp`, `TrigFunction` paramétrée sur *Sinus*, `SigDetector`, `Amplifier` et trois `TimedPlotter` que nous décrivons dans la suite.

HDirector : le directeur hétérogène, `HDirector` restructure le système aux frontières des différents comportements hétérogènes, crée les différents modèles, les ordonnance et les active .

Ramp : acteur qui produit une séquence des valeurs numériques. Son port de sortie est connecté au port d'entrée de l'acteur `TrigFunction` pour fournir un signal sinusoïdal. Pour obtenir une courbe régulière et avoir une bonne précision dans la détection de changement d'alternance, nous avons calibré son paramètre *step* à 0.1.

TrigFunction : Cet acteur est paramétré à *Sinus*. Son entrée est connectée à la sortie de l'acteur `Ramp` qui lui fournit une séquence de données. A son tour, il produit une séquence sinusoïdale. Son port de sortie est connecté au port d'entrée de l'acteur `SigDetector` qui est un HIC.

SigDetector : Cet acteur est un HIC. Il spécialise la classe `HicActor` en implémentant le comportement requis pour le changement d'alternance. Ce comportement hétérogène est caractérisé à la frontière des modèles de calcul du Flot de Données Synchrone et d'Evénements Discrets. Cet acteur dispose donc d'un port SDF d'entrée `sdfFromSin` connecté à la sortie de l'acteur `TrigFunction`. Il dispose également de deux ports de sortie qui obéissent aux modèles de calcul SDF et DE. Son de sortie `sdfToAmpli` est connecté au port d'entrée `sdfFromDet` de l'amplificateur et lui fournit le signal original. Quant à son port de sortie `deToAmpli`, il est connecté au port d'entrée `deFromDet` de l'amplificateur pour l'envoi d'un événement lorsque l'alternance de signe est détectée.

Amplifier : cet acteur est également un HIC. Il spécialise la classe `HicActor` en implémentant le comportement requis pour le redressement du signal. Ce comportement hétérogène est caractérisé à la frontière des modèles de calcul du Flot de Données Synchrone, d'Evénements Discrets et du Discrete Time. Il dispose d'un port SDF d'entrée `sdfFromDet` connecté à la sortie SDF de l'acteur `SigDetector` et d'un autre port DE d'entrée `deFromDet` connecté à la sortie DE de l'acteur `SigDetector`. Ses deux ports DT de sortie, `dtToOriginalPlot` et `dtToRecifiedPlot` sont connectés respectivement à l'afficheur du signal original et à l'afficheur du signal redressé. Quant à son dernier port de sortie DE `deToEventPlot` est connecté à l'afficheur de l'événement représentant la détection du début d'un redressement.

TimedPlotter : Les entrées de chacun des trois acteurs `TimedPlotter` sont tous connectés aux sorties de l'acteur `Amplifier`.

```
package ptolemy.domains.heterogeneous;

import ptolemy.actor.TypedCompositeActor;
import ptolemy.actor.gui.PtolemyApplet;
import ptolemy.actor.lib.*;
import ptolemy.actor.lib.conversions.*;
import ptolemy.actor.lib.gui.*;
import ptolemy.data.expr.Parameter;
import ptolemy.domains.de.kernel.DEDirector;
import ptolemy.kernel.util.IllegalActionException;
import ptolemy.kernel.util.NameDuplicationException;
import ptolemy.kernel.util.Workspace;
import ptolemy.domains.sdf.kernel.SDFDirector;
import ptolemy.domains.de.lib.*;
import ptolemy.data.type.BaseType;
import ptolemy.data.*;


public class Redresseur extends TypedCompositeActor {
    public Redresseur(Workspace workspace)
    throws IllegalActionException, NameDuplicationException {
    super(workspace);

    //Instanciation de HDirector
    HDirector hdirector = new HDirector(this, "hdirector");
    setDirector(hdirector);

    //Instanciation des acteurs du système
    TrigFunction sinus = new TrigFunction(this, "sinus");
    Ramp pente = new Ramp(this, "pente");
    SigDetector detector = new SigDetector(this, "detector");
    Amplifier ampli = new Amplifier(this, "ampli");
    TimedPlotter signalOriginal =
            new TimedPlotter(this, "signal Original");
    TimedPlotter signalRedresse =
            new TimedPlotter(this, "signal Redressé");
    TimedPlotter signalEvent = new TimedPlotter(this,
            "Détection d'Alternances Positive-Négative");

    //Connexion des acteurs
    connect (pente.output, sinus.input);
    connect (sinus.output, detector.sdfFromSin);
    connect (detector.sdfToAmpli, ampli.sdfFronDet);
    connect (detector.deToAmpli, ampli.control);
    connect (ampli.dtToOriginalPlot, signalOriginal.input);
    connect (ampli.dtToRectifiedPlot, signalRedresse.input);
    connect (ampli.deToEventPlot, signalEvent.input);

    }
}
```

FIG. 8.15 – Code représentant le système en utilisant l'API Java de PTOLEMY II

Comportement du HIC SigDetector

Le comportement hétérogène de l'acteur `SigDetector` est à la frontière des modèles de calcul SDF et DE. Ce comportement se résume de la manière suivante :

- lorsqu'il est activé par sa projection qui porte son port d'entrée `sdfFromSin`, il se comporte en consommateur. Il lit la donnée à partir de ce port, extrait sa valeur numérique et la place dans une variable. Et, il programme l'activation de sa projection DE portant son port `deToAmpli`.

- lorsqu'il est activé par sa projection qui porte son port de sortie `sdfToAmpli`, il se comporte en producteur. De ce port, il expédie la donnée lue précédemment au port `sdfFromDet` de l'acteur `Amplifier`

- lorsqu'il est activé par sa projection qui porte son port de sortie `deToAmpli`, il commence par tester si le changement d'alternance s'est produit.
  Dans l'affirmatif, de ce port `deToAmpli`, il envoie un événement à l'acteur `Amplifier` sur son port DE `control` pour lui signaler d'inverser le signe de son gain. Ensuite, il inverse le signe de la variable représentant l'alternance.

```
 if(sdfFromSin.getContainer() == _runningProjection){
    _tokenFromSin = sdfFromSin.get(0);
    _valueOfTokenFromSin =
        (((DoubleToken)_tokenFromSin).doubleValue());
    _nextTimeToFire(deToAmpli, 0);
}

if(sdfToAmpli.getContainer() == _runningProjection){
    sdfToAmpli.send(0, _tokenFromSin);
}

if(deToAmpli.getContainer() == _runningProjection){
    if(_valueOfTokenFromSin*lastSigneDet < 0){
        deToAmpli.send(0, new IntToken(lastSigneDet));
        previousSigneDet = -previousSigneDet;
        }
      }
}
```

FIG. 8.16 – Code représentant le comportement du HIC SigDetector

Comportement du HIC Amplifier

De même, le comportement hétérogène de l'acteur `Amplifier` est à la frontière des modèles de calcul SDF, DE et DT. Ce comportement se résume de la manière suivante :

– lorsqu'il est activé par sa projection qui porte son port `sdfFromDet`. Il se comporte en consommateur. Il lit la donnée à partir de ce port, extrait sa valeur numérique et la place dans une variable.

– lorsqu'il est activé par sa projection qui porte son port `control`, il consomme la donnée DE présente sur ce port pour vider son receiver. Cette consommation est importante car, elle permet à cet acteur de n'être activé qu'après détection de changement d'alternance. Ensuite, il incrémente le pas de changement d'alternance, il calcule la différence de temps de détection entre ce redressement et le précédent. Puis, il programme l'activation de sa projection DE qui porte le port `deToEventPlot` et il met à jour la variable représentant le temps de redressement.

```
if(sdfFronDet.getContainer() == _runningProjection){
    _sdfTokenFromDet = sdfFronDet.get(0);
    _valueOfsdfTokFromDet =
           (((DoubleToken)_sdfTokenFromDet).doubleValue());
}

if((control.getContainer() == _runningProjection) &&
                              (control.hasToken(0))){
    //Lecture à vide pour enlever le token du port control
    control.get(0);
    gain = -gain;
    _downCrossing = _downCrossing + 1;
    _offset = +Math.PI*100*(1 + (2*_downCrossing));
    _period = (_offset - _base);
    _nextTimeToFire(deToEventPlot, 0);
}

if(dtToOriginalPlot.getContainer() == _runningProjection){
    dtToOriginalPlot.send(0, _sdfTokenFromDet);
}

if(dtToRectifiedPlot.getContainer() == _runningProjection){
    sortie = gain*_valueOfsdfTokFromDet;
    dtToRectifiedPlot.send(0, new DoubleToken(sortie));
}

if(deToEventPlot.getContainer() == _runningProjection){
    deToEventPlot.send(0, new DoubleToken(1), _period);
    _base = _offset;
}
```

FIG. 8.17 – Code représentant le comportement du HIC Amplifier

– lorsqu'il est activé par sa projection qui porte son port `dtToOriginalPlot`, il se comporte en producteur. De ce port, il envoie la donnée original à l'acteur `OriginalPlot`.

– lorsqu'il est activé par sa projection qui porte son port `dtToRectifiedPlot`, il se comporte en producteur et calcule le signal redressé en multipliant le gain par la valeur du signal reçu. Après, de ce port il envoie le résultat à l'acteur `RectifiedPlot`.

– lorsqu'il est activé par sa projection qui porte son port `deToEventPlot`, il se comporte en producteur. Il envoie un événement de détection à partir de son port `deToEventPlot` à l'afficheur `EventPlot`.

Résultat de la simulation

La figure 8.18 montre le résultat de la simulation. Les afficheurs en haut, au milieu et en bas montrent respectivement le signal original, le signal redressé et les instants de début de redressement.

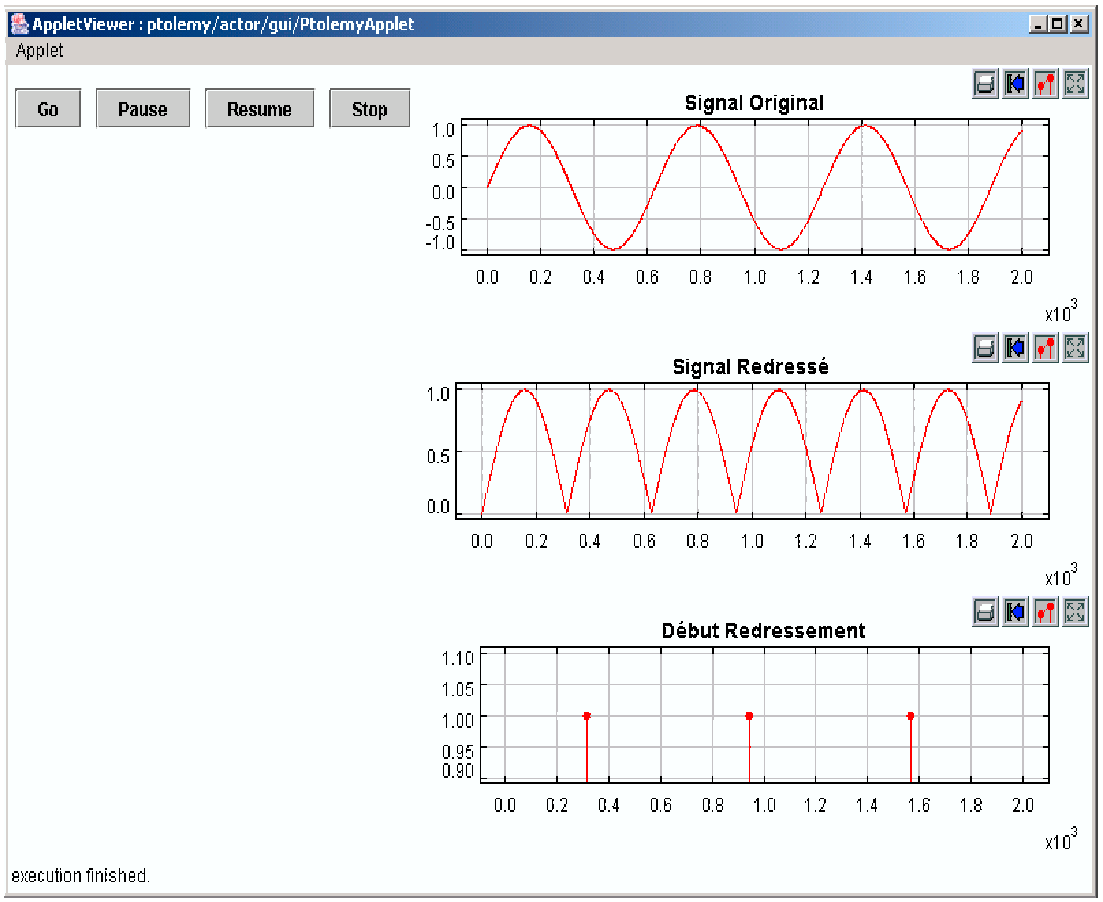


FIG. 8.18 – Simulation d'un Redresseur de signal utilisant plusieurs MoCs

8.7.2 Cas d'un signal sinusoïdal bruité

Nous injectons un bruit perturbateur dans le signal pour observer le résultat du redressement. Pour cela, nous additionnons ce signal sortant du générateur avec un bruit généré par l'acteur `Gaussian` qui fournit une séquence des valeurs d'une variable aléatoire gaussienne distribué. Nous fixons sa moyenne à zéro et son écart-type à 0.1. Le résultat devient :

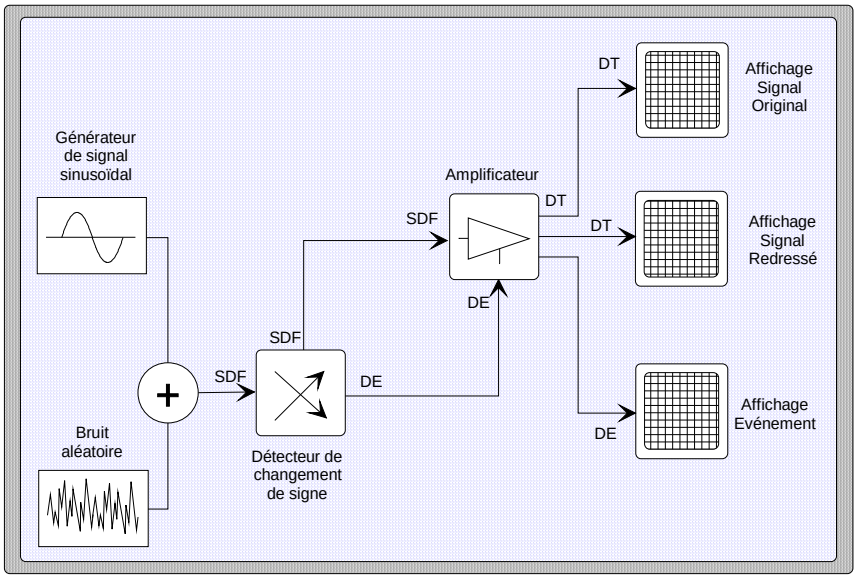


FIG. 8.19 – Exemple d'un Redresseur de signal bruité utilisant plusieurs MoCs

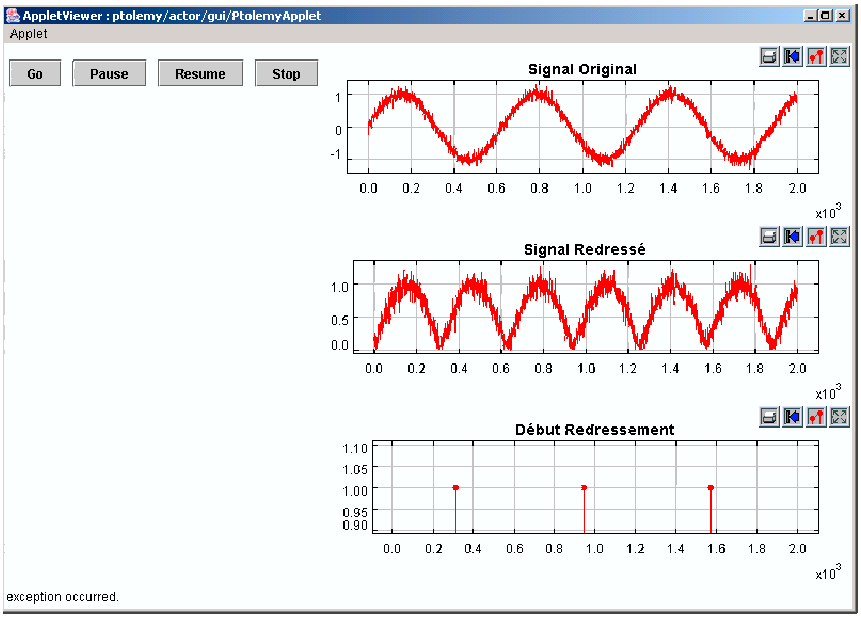


FIG. 8.20 – Simulation d'un Redresseur de signal bruité utilisant plusieurs MoCs

8.7.3 Modulateur d'Amplitude

Principe

Le principe de fonctionnement du modulateur de signal numérique reste quasi- identique à celui du redresseur de signal à la seule différence qu'ici, le premier HIC dispose de deux ports SDF d'entrée, d'un port SDF de sortie et de deux ports DE de sortie
Quant au deuxième HIC, il dispose d'un port SDF d'entrée, de deux ports d'entrée DE et de deux ports de sortie DT et de port DT de sortie.

Le premier HIC qui reçoit une porteuse SDF et un signal modulant numérique SDF doit détecter les fronts montants et les fronts descendants du signal modulant numérique.
Si le front est montant, il envoi un événement sur le port `On` d'entrée DE du deuxième HIC pour lui signaler de garder la valeur de l'amplitude de la porteuse. C'est cette valeur qui sera affichée sur l'affichage du signal modulant.
Si le front est descendant, il envoi un événement sur le port `Off` d'entrée DE du deuxième HIC pour lui signaler de moduler l'amplitude de la porteuse. C'est cette valeur modulée qui sera affichée sur l'affichage du signal modulant. Comme pour le premier exemple, il est intéressant de remarquer le premier HIC utilise simultanément les deux modèles de calcul SDF et DE. De même, le deuxième modèle de calcul utilise les MoCs SDF, DE et DT.

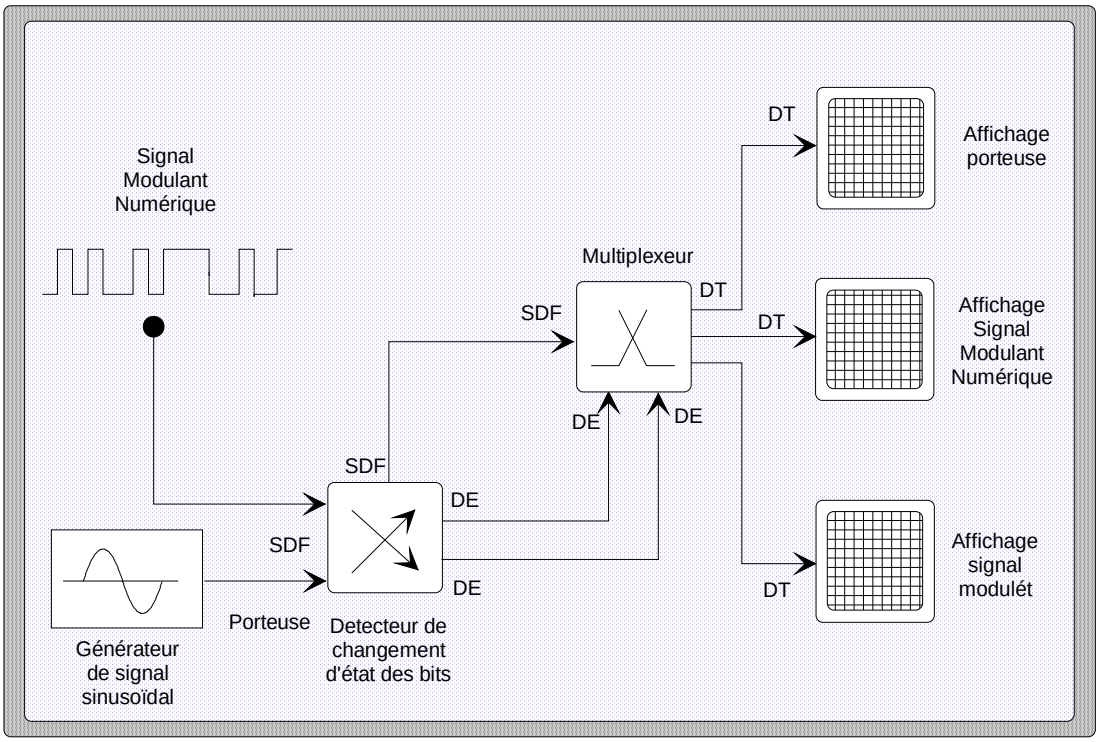


FIG. 8.21 – Exemple d'un Modulateur d'Amplitude utilisant plusieurs MoCs

Résultat de la simulation

La figure 8.22 montre le résultat de la simulation du modulateur d'amplitude. Les afficheurs en haut, au milieu et en bas montrent respectivement la porteuse, le signal modulant numérique et le signal modulé.

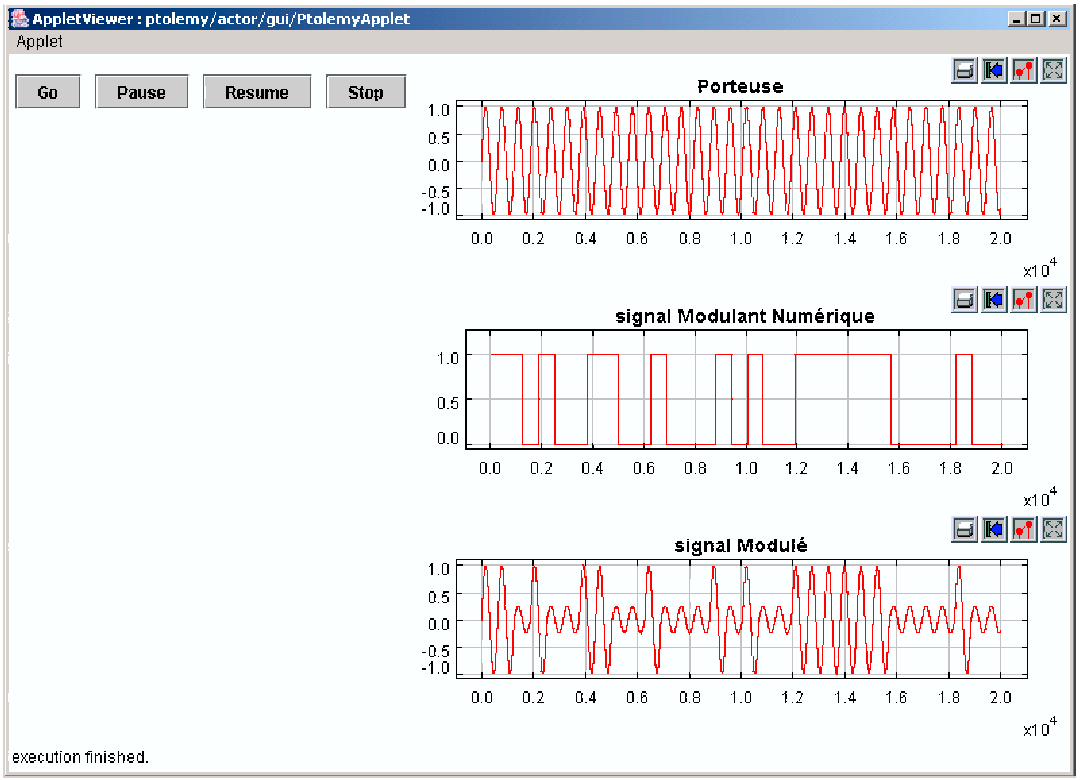


FIG. 8.22 – Simulation d'un Modulateur d'Amplitude utilisant plusieurs MoCs

8.7.4 Cas d'un Modulateur de phase

En gardant le même principe que le modulateur d'amplitude, ici, le deuxième HIC va plutôt moduler la phase du signal.
En effet, de même, le premier HIC qui reçoit une porteuse et un signal modulant numérique doit détecter les fronts montants et les fronts descendants du signal modulant numérique. Si le front est montant, il envoi un événement sur le port `On` d'entrée DE du deuxième HIC pour lui signaler de garder la valeur de la phase de la porteuse. C'est cette valeur qui sera affichée sur l'affichage du signal modulant.
Si le front est descendant, il envoi un événement sur le port `Off` d'entrée DE du deuxième HIC pour lui signaler de moduler la phase de la porteuse. C'est également cette valeur

modulée qui sera affichée sur l'affichage du signal modulant.

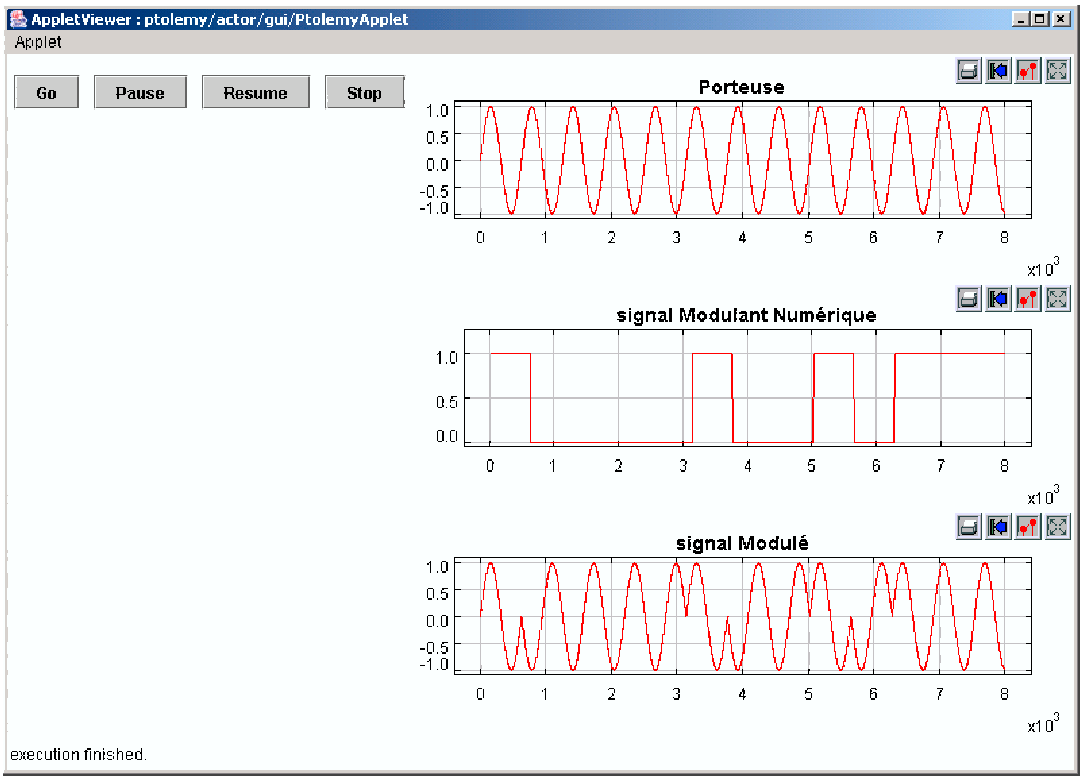


FIG. 8.23 – Simulation d'un Modulateur de phase utilisant plusieurs MoCs

8.8 Conclusion partielle

Dans ce chapitre nous avons intégré notre approche de l'Hétérogénéité Non-Hiérarchique dans la plate-forme PTOLEMY II.
Nous avons d'abord intégré les classes d'appui à la non-hiérarchie, ensuite avons assigné les différentes phases du concept dans les différentes méthodes du directeur hétérogène non-hiérarchique.
En effet, en ce qui concerne les classes des composants d'appui que nous avons modélisées sous le formalisme UML au chapitre précédent, ont été intégrées après leur adaptation. Ce qui nous a permis de générer deux classes fondamentales à savoir : la classe HicActor et la classe HDirector

La difficulté dans cette intégration a résidée dans la bonne compréhension du fonctionnement détaillée de cette plate-forme, à partir de l'initialisation jusqu'à l'exécution.

Par ailleurs, l'une des clés de réussite de cette intégration est notre décomposition minutieuse du mécanisme d'exécution d'un modèle dans PTOLEMY II. Ceci nous a permis de dégager avec clarté les différents pavés d'exécution, leurs contraintes, les préalables requis du point de vue de la création des receivers, de la résolution des types etc. . . . Ainsi, de pouvoir aisément intégrer chaque phase de modélisation dans chaque phase de HDirector

Pour la validation, nous avons présenté deux exemples simples des systèmes hétérogènes intégrant deux HICs et utilisant plusieurs modèles de calcul au même niveau de la hiérarchie. Ces systèmes génèrent des signaux sinusoïdaux qui passent à travers deux HICs en changeant de domaine à chaque fois qu'il en sortait. Et, à l'autre extrémité ils en ressortent chacun redressé ou modulé.

Cependant, afin de se rapprocher de la pratique, nous avons présenté deux exemples dans lesquels cette approche est appliquée.
Au demeurant, notre approche de l'Hétérogénéité Non-Hiérarchique a été bien intégrée et validée par simulation dans PTOLEMY II.

Chapitre 9

Conclusion et perspectives

9.1 Conclusion

Cette thèse a été réalisée au sein de l'équipe « Génie Logiciel et Méthodes Avancées de Développement », spécialement dans le groupe de Recherche sur la Modélisation hétérogène du Service Informatique de l'Ecole Supérieure d'Electricité (SUPELEC).
Elle s'inscrit dans le cadre de la modélisation des systèmes hétérogènes et particulièrement celle des systèmes embarqués. Son étendue technologique est très vaste, car, elle concerne la modélisation, la conception, la simulation et la validation de systèmes logiciels et matériels que l'on trouve essentiellement dans les technologies de pointe telles que les appareils de communication, les appareils médicaux, les systèmes de détection, les systèmes de navigation, les équipements militaires, les équipements spatiaux etc. . . .

Le deuxième chapitre de ce document a mis en évidence la problématique de cette étude. Cette problématique repose sur le constat qu'actuellement les outils génériques utilisés pour la modélisation des systèmes hétérogènes accusent quelques limites, entre autre, l'impossibilité d'utiliser les composants fonctionnant à la frontière de plusieurs modèles de calcul, c'est-à-dire, possédant des entrées et des sorties obéissant à des sémantiques différentes. De plus, la sémantique de passage de données d'un domaine vers un autre n'est pas explicite ou encore l'obligation de changer de niveau hiérarchique à chaque changement de modèle de calcul.

La conjonction de ces limites accusées par ces outils de modélisation nuit à la modularité, à la réutilisabilité et à la maintenabilité des applications logicielle et des composants.

Le but poursuivi par cette étude était donc de proposer une approche de modélisation reposant sur le principe de modularité, utilisant des composants réutilisables et assurant une facilité de maintenabilité. Cette approche devait permettre d'obtenir un modèle hétérogène « plat » qui permettra de changer de modèle sans changer de niveau hiérarchique et sans altération du comportement du système original. En conséquence, faire interagir deux ou plusieurs composants hétérogènes au même niveau hiérarchique. Elle devait également permettre de modéliser, de concevoir et de générer des composants susceptibles de travailler à la frontière de plusieurs modèles de calcul, i.e., ayant des entrées et des sorties obéissant simultanément à différentes sémantiques. Enfin cette approche devait également permettre de donner au concepteur du système, la flexibilité lui permettant de concevoir explicitement le comportement du système à la frontière de plusieurs modèles de calcul comme partie intégrante de son modèle, i.e., donner au concepteur du système le contrôle entier du comportement de son système à la limite des différents modèles de calcul.

Pour cela, à travers cette étude, nous avons proposé une approche hétérogène non-hiérarchique qui a permis de changer de modèle sans changer de niveau hiérarchique et d'obtenir un modèle hétérogène « plat » sans altération du comportement du système original. Ainsi, il s'en est suivi de la possibilité de faire interagir plusieurs composants hétérogènes au même niveau hiérarchique. Ce qui a induit la possibilité de modéliser, de concevoir et de générer des composants susceptibles de travailler à la frontière de plusieurs modèles de calcul, c'est-à-dire, ayant des entrées et des sorties obéissant à différentes sémantiques. De plus, elle a donné au concepteur du système, la flexibilité lui permettant de concevoir explicitement, selon la spécification souhaitée, le comportement du système à la frontière de plusieurs modèles de calcul comme une partie intégrante de son modèle.

Le développement de cette étude a été segmenté en trois étapes à savoir, la présentation des outils de modélisation et celle de l'approche théorique non-hiérarchique, la modélisation des composants d'appui à l'hétérogénéité non-hiérarchique et l'intégration, la validation et la simulation dans PTOLEMY II.

Dans l'étape concernant le développement de l'approche non-hiérarchique, nous avons montré que le principe fondateur de cette approche reposait sur deux composants de base à savoir, le composant à interface hétérogène (HIC) et le modèle d'exécution hétérogène. Afin de séparer clairement les préoccupations, dans cette approche, nous avons séparé le flot de contrôle du flot de données. Le flot de contrôle est assuré par le modèle d'exécution hétérogène et le flot des données est assuré par le composant à interface hétérogène. Dans le second flot, la communication est gérée par les projections de HIC et le calcul de comportement hétérogène est fait par le composant à interface hétérogène lui-même.

Le composant à interface hétérogène dispose donc d'entrées et de sorties hétérogènes

lui permettant de mettre en communication des composants ayant des ports utilisant des modèles de calcul différents. Il est également capable de fournir un comportement hétérogène aux frontières des différents modèles de calcul.

Le modèle d'exécution hétérogène restructure le système en le divisant à la frontière des différents modèles de calcul utilisés par le composant à interface hétérogène. Ensuite, il crée des sous-systèmes et délègue leur flot de contrôle et le calcul de leur comportement locaux à leurs modèles d'exécution réguliers respectifs. Puis, il génère l'ordonnancement approprié de ces différents sous-systèmes sans tenir compte des sémantiques de leurs modèles de calcul respectifs. Enfin il les active.
Cependant le fait pour le modèle d'exécution hétérogène de ne pas connaître les modèles de calcul utilisés par ses sous-systèmes lui donne la capacité de supporter tous les modèles d'exécution homogènes. Pour la communication, étant donné que les sous systèmes ne sont pas connectés, nous avons remplacé l'abstraction hiérarchique par l'abstraction non-hiérarchique. Ceci a engendré des « canaux abstraits hétérogènes » qui ont facilité la communication entre les sous-systèmes non-connectés.

Cette configuration nous a permis de mettre à « plat » le système hétérogène et d'isoler le modèle d'exécution de la connaissance des modèles de calcul utilisés par les sous-systèmes.

Signalons par ailleurs qu'une conséquence de cet algorithme de partitionnement du système hétérogène de manière non-hiérarchique est qu'il ne peut pas y avoir des boucles dans le graphe du système. C'est le prix à payer pour supporter n'importe quel modèle de calcul.
En effet, l'ordonnancement d'un système qui contient des boucles dépend de la sémantique du modèle de calcul utilisé par ce système. Une manière de casser des boucles de dépendance est d'insérer le retard dans la boucle. Cependant la sémantique du retard est elle-même très dépendante de la sémantique du modèle de calcul du système.
Par conséquent, si nous considérons les sous-ensembles et leurs modèles de calcul en tant que boîtes noires, nous ne pouvons pas permettre des boucles de dépendance entre les sous-ensembles. Toutefois, si une boucle est locale à un sous-ensemble et si le modèle de calcul correspondant soutient des boucles, la boucle est acceptée et sa sémantique sera donnée par le modèle de calcul du sous-ensemble.

Dans l'étape concernant la modélisation des composants d'appui à l'hétérogénéité non-hiérarchique, nous avons opté de modéliser ces concepts à un niveau d'abstraction élevé pour permettre leur intégration dans n'importe quelle plate-forme de modélisation.
Puis, au niveau d'abstraction inférieur, dans le but de se rapprocher de la réalisation, nous avons utilisé le langage UML comme formalisme intermédiaire.
Avec ce formalisme, nous avons représenté le composant à interface hétérogène (HIC) et le modèle d'exécution hétérogène. Cette représentation est faite de manière visuelle par une vue statique à l'aide des diagrammes des classes et par des vues dynamiques à l'aide des diagrammes d'état et des séquences.

Comme pour la modélisation abstraite, cette spécification intermédiaire est faite sans tenir compte des spécificités des différentes plates-formes. En effet, nous n'avons fait aucune hypothèse sur le mécanisme fondateur de cette théorie en termes de communication, ni en termes du comportement, ni encore en termes de contrôle du système. Cette abstraction permet au concepteur d'adapter les principes énoncés à des spécificités de sa plate-forme.

En somme, nous avons conçu le modèle d'exécution hétérogène qui supporte les HICs. Ce modèle d'exécution peut être implémenté sur différentes plates-formes, car, il ne nécessite pas de modification des modèles d'exécution existants.

Enfin vient l'étape d'intégration et de la validation de notre approche d'Hétérogénéité Non-Hiérarchique dans la plate-forme PTOLEMY II.
En ce qui concerne cette intégration, nous avons d'abord intégré les classes d'appui à la non-hiérarchie dans PTOLEMY II, ensuite avons assigné les différentes phases du concept dans les différentes méthodes du directeur hétérogène non-hiérarchique nommé HDirector.

La difficulté dans cette intégration a résidée dans la bonne compréhension du fonctionnement détaillée de cette plate-forme, à partir de l'initialisation jusqu'à l'exécution.

Par ailleurs, l'une des clés de réussite de cette intégration a été notre décomposition minutieuse du mécanisme d'exécution d'un modèle dans PTOLEMY II afin de pouvoir aisément intégrer chaque phase de modélisation dans chaque phase de HDirector. Ceci nous a en effet permis de dégager avec clarté les différentes étapes d'exécution, leurs contraintes, les préalables requis du point de vue de la création des receivers, de l'initialisation des paramètres des acteurs dans certains domaines, de la résolution des types etc. . .

Nous avons enfin validé cette approche par simulation de quelques exemples d'application. Cette validation a conclu à la satisfaction sur la problématique développée dans son introduction.

La première contribution de cette thèse est le développement d'une approche non-hiérarchique basée sur deux composants d'appui disposant de capacité de réutilisabilité.
En effet, lors d'une nouvelle instanciation du composant à interface hétérogène, le concepteur du système implémente son code en fonction d'une nouvelle spécification relative à son comportement à la frontière des modèles de calcul qu'il utilise. Cette spécification« sur mesure » lui donne son caractère de composant flexible et réutilisable.
Quant au modèle d'exécution non-hiérarchique, sa non-dépendance des modèles de calcul utilisés par les sous-systèmes, lui donne une possibilité d'évolutivité indépendante desdits modèles de calcul. Ainsi, un retrait ou une spécification d'un nouveau modèle de calcul, i.e., une nouvelle extension du système à un nouveau domaine supplémentaire ne remet nullement en cause sa spécification. Ce qui garanti sa réutilisabilité et sa maintenabilité.
Au niveau du système, cette modularité est également dans la génération des sous-systèmes.

La deuxième contribution de cette étude est l'annihilation de l'obligation d'introduire artificiellement des niveaux hiérarchiques pour autoriser des changements de MoC.
En effet, ceci permet une modélisation plus naturelle des composants à interface hétérogène en donnant au concepteur plus de contrôle sur la sémantique des interactions entre les modèles de calcul. Ce découplage entre les changements de domaine et de la structure hiérarchique du modèle contribue à la modularité et à la maintenabilité des modèles.

La troisième contribution peut être observée sur le plan pratique, car, cette étude a abouti à l'élaboration d'un nouveau domaine « Hétérogène » que nous avons baptisé « Flat Heterogeneous Domain ». Ce domaine introduit la prise en charge de l'hétérogénéité non-hiérarchique pouvant supporter les différents domaines au même niveau hiérarchique.
Dans PTOLEMY II, il se présente sous forme d'une bibliothèque incluant des composants d'appui à l'hétérogénéité non-hiérarchique.

9.2 Perspectives

Cette thèse a concerné la modélisation non-hiérarchique des systèmes impliquant plusieurs modèles de calcul. L'approche développée pour cette modélisation reste applicable dans plusieurs environnements de modélisation. Cependant, les résultats obtenus permettent d'envisager l'ouverture de quelques perspectives très intéressantes pour cette axe de recherche.

A cet effet, deux orientations se profilent : une première orientation à moyen terme, relative à la plate-forme de modélisation que nous avons utilisée et une deuxième orientation à long terme, relative l'approche non-hiérarchique que nous avons proposée.

Pour les travaux à moyen terme, il est par la suite prévu de faciliter la construction des modèles hétérogènes en envisageant d'étendre ce travail sur l'interface graphique de la plate-forme PTOLEMY II appelée VERGIL. Cette extension passera par l'intégration de deux concepts suivants :
- la prise en compte par VERGIL des aspects liés à la restructuration dynamique des domaines
- la prise en compte par VERGIL de l'abstraction non-hiérarchique, en d'autres terme, lui faire supporter les canaux de communication abstraits.

Pour les travaux à long terme, actuellement dans le groupe de recherche sur la modélisation hétérogène de SUPELEC, les travaux en cours sur la méthode de conception des composants polymorphes de domaines ont donné naissance à la spécification d'un composant d'adaptation de sémantique.
En effet, la modélisation hétérogène non-hiérarchique peut aussi s'appliquer au niveau de la conception des composants élémentaires d'un système. Il s'agit alors d'adapter la sémantique d'un noyau comportemental à celle du modèle de calcul dans lequel il doit fonctionner.

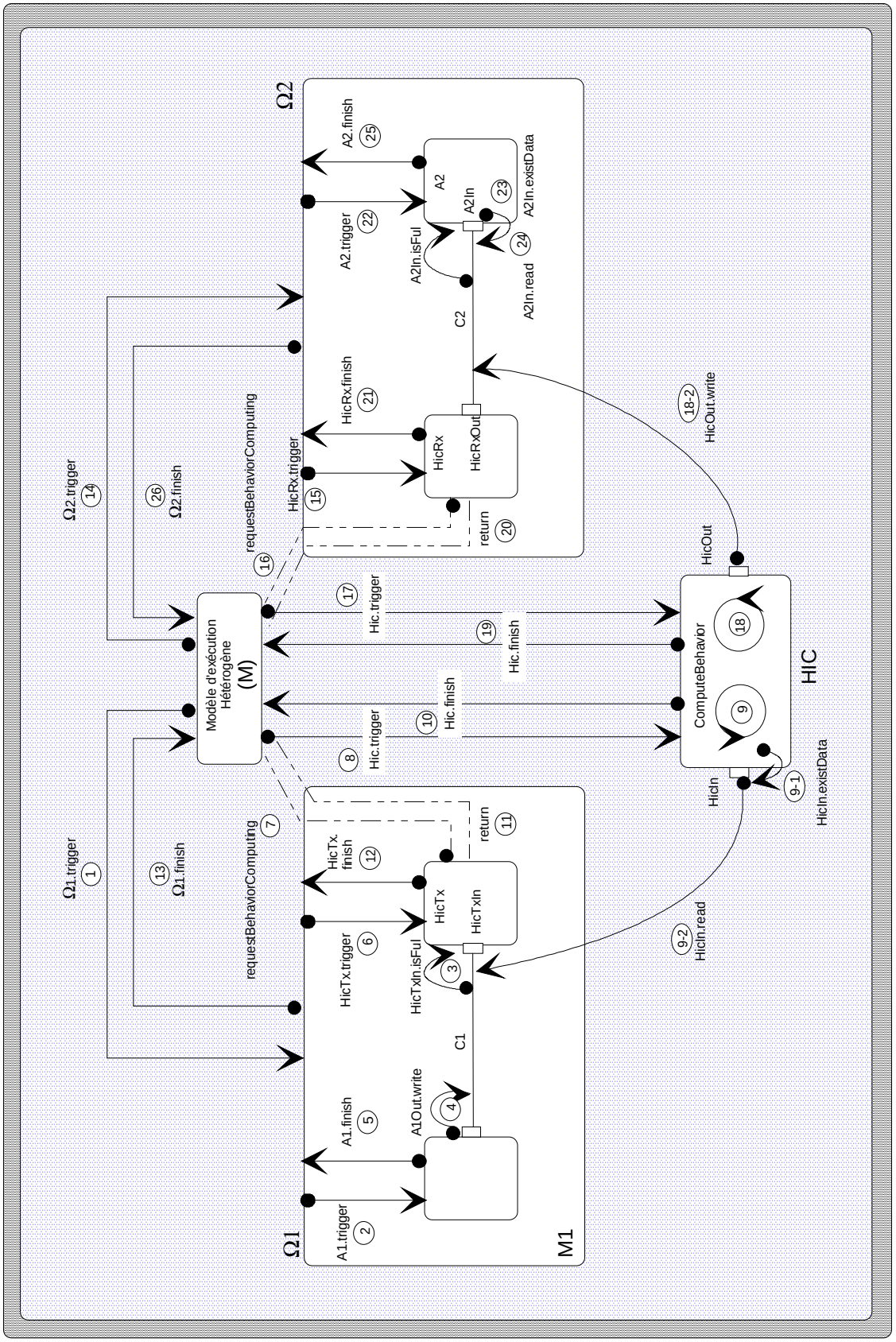
Pour cela, l'approche réactive synchrone et le langage ESTEREL ont été utilisés pour coder le comportement qui peut ensuite être utilisé dans des modèles de calcul tels que le modèle à événements discrets et le modèle à flot de données synchrones.
Il serait donc intéressant d'enrichir la spécification de ce composant afin de pouvoir l'utiliser dans la non-hiérarchie au niveau des systèmes hétérogènes, et ce, en collaboration avec le modèle d'exécution hétérogène non-hiérarchique.

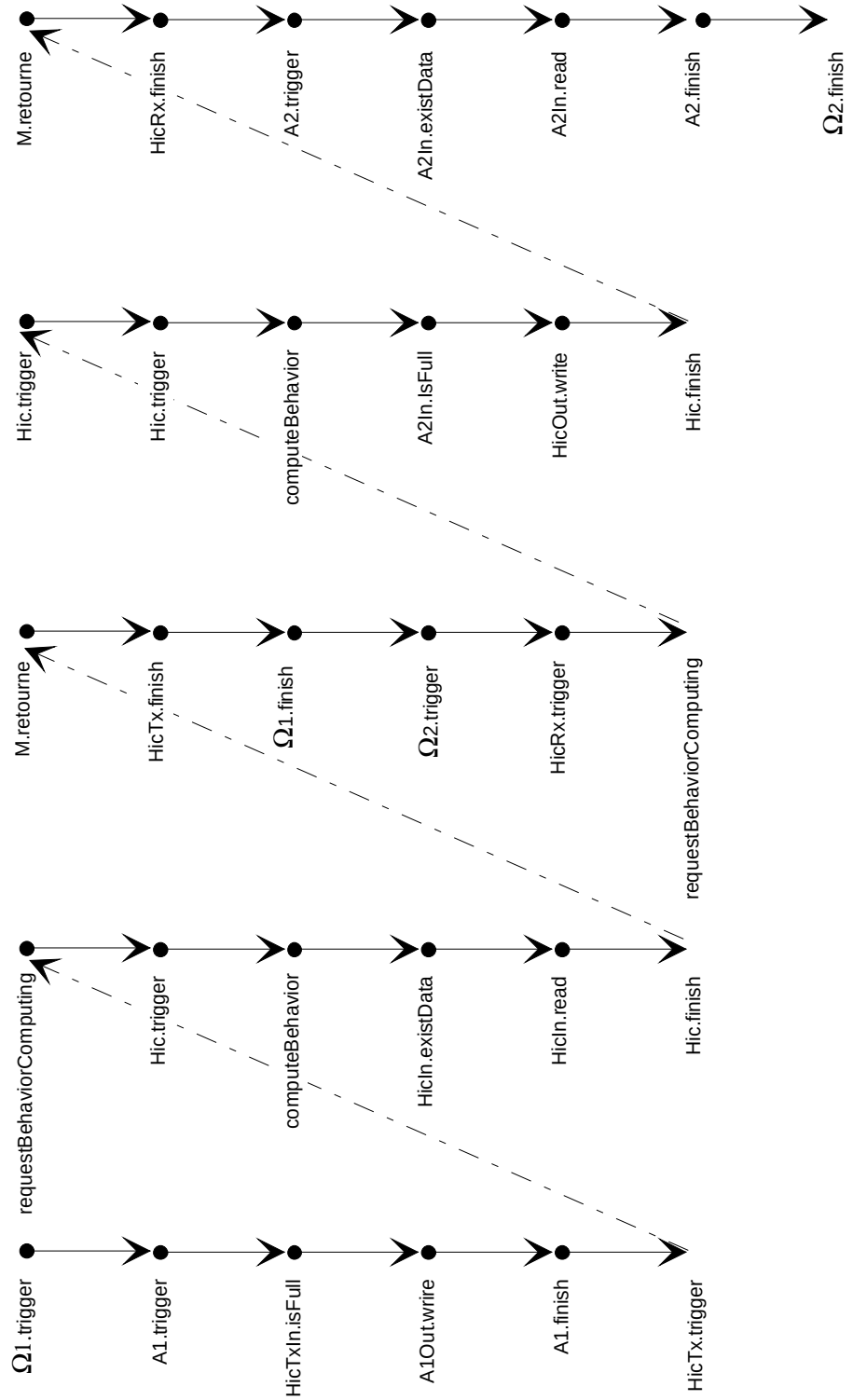
Par ailleurs, puisque l'algorithme utilisé par le modèle d'exécution hétérogène non-hiérarchique ne supporte pas de boucles dans le graphe du système, nous pouvons dans l'avenir explorer la suggestion qu'avec certaines hypothèses sur le comportement des acteurs, cet algorithme pourrait supporter les boucle.

Quatrième partie

Annexes

Exécution complète et diagramme de Hasse d'un modèle à deux MoCs


Ω2
Ω1
M1
HIC
Modèle d'exécution Hétérogène (M)
Ω1.trigger
Ω1.finish
Ω2.trigger
Ω2.finish
requestBehaviorComputing
A1.trigger
A1.finish
A1Out.write
C1
HicTx.trigger
HicTx.finish
HicTxIn.isFul
HicTx
HicTxIn
A2.trigger
A2.finish
A2In.isFul
A2In.read
A2In.existData
A2
A2In
C2
HicRx.trigger
HicRx.finish
HicRx
HicRxOut
return
Hic.trigger
Hic.finish
HicOut
HicOut.write
HicIn
HicIn.read
HicIn.existData
ComputeBehavior


Ω1.trigger
A1.trigger
HicTxIn.isFull
A1Out.wrire
A1.finish
HicTx.trigger
requestBehaviorComputing
Hic.trigger
computeBehavior
HicIn.existData
HicIn.read
Hic.finish
M.retourne
HicTx.finish
Ω1.finish
Ω2.trigger
HicRx.trigger
requestBehaviorComputing
Hic.trigger
Hic.trigger
computeBehavior
A2In.IsFull
HicOut.write
Hic.finish
M.retourne
HicRx.finish
A2.trigger
A2In.existData
A2In.read
A2.finish
Ω2.finish

Code source de HicActor

```
1  /* An Heterogeneous Interfaces Component (HIC) communicates by its
2   * inputs and outputs ports according to the semantics of its
3   * associated domains.
4   */
5
6  package ptolemy.domains.heterogeneous;
7
8  import ptolemy.actor.*;
9  import ptolemy.data.*;
10 import ptolemy.data.type.*;
11 import ptolemy.data.expr.Parameter;
12 import ptolemy.domains.csp.kernel.*;
13 import ptolemy.domains.ct.kernel.*;
14 import ptolemy.domains.dde.kernel.*;
15 import ptolemy.domains.de.kernel.*;
16 import ptolemy.domains.dt.kernel.*;
17 import ptolemy.domains.gr.kernel.*;
18 import ptolemy.domains.pn.kernel.*;
19 import ptolemy.domains.sdf.kernel.*;
20 import ptolemy.domains.sr.kernel.*;
21 import ptolemy.kernel.*;
22 import ptolemy.kernel.util.*;
23
24 import java.util.Iterator;
25
26 //////////////////////////////////////////////////////////////////////////
27 //// HicActor
28
29 /**
30 The interpretation of the data fron one domain to another domain is
31 explicitly stated in the behavior of the Heterogeneous Interfaces
32 Component.  A HIC is in charge of the management of the data flows
33 between different MoCs. In this component, we define the semantics
34 of the communication between actors that obey different MoCs.
35 Since HICs are at the boundary of several MoCs, they must obey the
36 semantics of several domains. However, to be usable with existing
37 Models of Computation, they must behave exactly as actors of a MoC
38 from the point of view of the corresponding domain. Everything that
39 does not belong to a Models of Computation must be masked when a HIC
40 is considered as an actor of this MoC.
41 /*
42 @author : Mokhoo MBOBI
43 Ecole Supérieure d'Electricité (SUPELEC France)
44 Computer Sciences Department
45 Email : Mokhoo.MBOBI@supelec.fr
46 */
```

```
47 public class HicActor extends TypedAtomicActor
48          implements HeterogeneousBehavior{
49
50     /** Construct an HicActor with the specified container and name.
51      *  @param container The composite actor to contain this one.
52      *  @param name The name of this actor.
53      *  @exception IllegalActionException If the entity cannot be
54      *   contained by the proposed container.
55      *  @exception NameDuplicationException If the container already
56      *   has an actor with this name.
57      */
58     public HicActor(CompositeEntity container, String name)
59             throws NameDuplicationException, IllegalActionException  {
60         super(container, name);
61     }
62
63     /** Construct an HicActor in the default workspace with an empty
64      *  string as its name. Increment the version number of the
65      *  workspace.
66      *  The object is added to the workspace directory.
67      */
68     public HicActor() {
69     }
70
71     ////////////////////////////////////////////////////////////////
72     ////                         public methods                 ////
73
74     /** Clone the HicActor into the specified workspace.
75      *  @param workspace The workspace for the new object.
76      *  @return A new actor.
77      *  @exception CloneNotSupportedException If a derived class has
78      *   has an attribute that cannot be cloned.
79      */
80     public Object clone(Workspace workspace)
81         throws CloneNotSupportedException {
82         HicActor newObject = (HicActor)(super.clone(workspace));
83         return newObject;
84     }
85
86     /** This method is used by the HIC to compute the behavior of the
87      *  model in the bundary of different Models of Computation.
88      *  It does nothing in this base class.
89      *  Each HIC must override this method to provide an heterogeneous
90      *  behavior at the bundary of the model of computation that it
91      *  uses
92      */
```

```
93
94      public void  computeBehavior() {
95     }
96
97     /** This method is used both by the HIC and by its projections.
98      *  This methode self-identify the nature of HIC
99      *  (HIC or projection).
100     *  If it is a projection, it requires from HDirector, the
101     *  activation of its HIC generator.
102     *  Else, if it is a HIC, it simplely activates the compute of
103     *  the behavior by the method computeBehavior().
104     *  N.B. Don't override this method.
105     */
106    public void fire() throws IllegalActionException {
107        _container = (TypedCompositeActor)this.getContainer();
108        _director = _container.getDirector();
109
110        //Self-identification (HIC or projection)
111        //If it is a projection, require from HDirector, the
112        //activation of  HIC
113
114        if (!(_director instanceof HDirector)){
115            _hdirector = (HDirector)_container
116                    .getExecutiveDirector();
117            _original = (HicActor)_hdirector.htabProHic.get(this);
118            _original._recordRunningProjection(this);
119            _requestActivateHIC();
120        }
121
122        // Else, if it is a HIC, activate the method  computeBehavior()
123
124        else if (_director instanceof HDirector){
125            computeBehavior();
126        }
127        return;
128    }
129
130    ////////////////////////////////////////////////////////////////
131    ////                     protected methods                  ////
132
133    /** This method is used by the HIC to enqueue a pure event in the
134     *  FIFO queue of DEDirector to program the firing of its
135     *  producer projection at the times determined by itself.
136     *  N.B. Don't override this method.
137     */
138    protected void _enqueueNextTimeToFire(double nextTimeToFire)
```

```
139         throws IllegalActionException {
140         _container = (TypedCompositeActor)getContainer();
141         _director = _container.getDirector();
142         _dedirector = (DEDirector)_director;
143         _currentTime = _dedirector.getCurrentTime();
144         _dedirector.fireAt(this, nextTimeToFire + _currentTime);
145         return ;
146     }
147 
148     /** This method is used by the HIC to generate its differents
149      *  projections. This method must be overriden by add only tese
150      *  two following lines. First for to create the projection as :
151      * (_avatar =(HicActor) new (name of HIC)(tdo, namePro).
152      *  And, to activate the method _setting port as :
153      *  (_endGenerateProjection(tdo, namePro, pfh, _avatar, this))
154      */
155     protected void _generateProjection(TypedCompositeActor tdo,
156             String namePro, TypedIOPort pfh)
157                     throws IllegalActionException{
158     }
159 
160     /** This method is used by the projection for to say if it is
161      *  consummer or if it is producer.
162      **/
163     protected String  _getTypeProjection() throws IllegalActionException{
164         return _typeProjection;
165     }
166 
167     /** This method is used by the projection  for to get its original
168      *  HIC
169      **/
170     protected HicActor _getOriginal(){
171         _container = (TypedCompositeActor)this.getContainer();
172         _hdirector = (HDirector)_container.getExecutiveDirector();
173         _original = (HicActor)_hdirector.htabProHic.get(this);
174         return _original;
175     }
176 
177     /* This method is used by the HIC after the method
178      * generateProjection. It set the compatibles ports of projection
179      * in its domain.
180      * Moreover, if the ports on which the projection is based is a
181      * producer port, it keep only the producer ports.
182      * if the ports on which the projection is based is a consumer
183      * port, it keep only the consumer ports.
184      */
```

```
185     protected void _generateProjectionEnd (TypedCompositeActor tdo,
186             String namePro, TypedIOPort pfh, HicActor _avatar,
187         HicActor thisHIC) throws IllegalActionException {
188         _hdirector = (HDirector)this.getDirector();
189         _hdirector.htabProHic.put(_avatar, thisHIC);
190         _hdirector.vectPro.addElement(_avatar);
191         try{
192             if((pfh.isInput()) && (pfh instanceof DEIOPort)){
193                 TypedAtomicActor actor = (TypedAtomicActor)pfh.getContaine
194                 Iterator ports = actor.portList().iterator();
195                 while (ports.hasNext()) {
196                     TypedIOPort port = (TypedIOPort)ports.next();
197                     _avatar._typeProjection = "Rx";
198                 }
199             }
200             else if ((pfh.isInput() && (!(pfh instanceof DEIOPort)))){
201                 _avatar._typeProjection = "Rx";
202             }
203             else if (pfh.isOutput()){
204                 _avatar._typeProjection = "Tx";
205             }
206         }
207         catch (Exception e){
208             System.out.println ("generateProjection2 failed");
209         }
210         //Get the compatible ports of this projection
211         Iterator ports = _avatar.portList().iterator();
212         try{
213             while (ports.hasNext()) {
214                 TypedIOPort port = (TypedIOPort)ports.next();
215 
216                 //Check the compatibility between the port and the
217                 //domain in which it is projected.
218                 //If it is not compatible and it have not the same
219                 //sense (consumer or producer) as the port that
220                 //generate this domain, remove it
221 
222                 if (((_hdirector._isCompatible(tdo.getDirector().
223                         getClass().toString(),
224                             port.getClass().toString())))
225                         && ((pfh.isInput())  == (port.isInput()))){
226 
227                     //Replace the HIC port by the port of the
228                     //projection in the port table contained by
229                     //HDirector, end share them.
230                     //Them, in the reconnection phase, the port which
```

```
231                     //was connected to HIC will be connected to its
232                     //projection in this domain.
233
234                     if (pfh.isInput()){
235                         Iterator inputPorts =
236                         this.inputPortList().iterator();
237                         while (inputPorts.hasNext()) {
238                             TypedIOPort inputP =
239                             (TypedIOPort)inputPorts.next();
240                             //If these ports are the same
241                             if ((inputP.getName()).
242                                 equals (port.getName())){
243                                 _hdirector.
244                                 _replacePortsHicByPortsProjection
245                                 (inputP ,port);
246
247                                 //Put them in the hashtab
248                                 //(HIC input-projection input)
249
250                                 _hdirector.htabPiPa.put(inputP, port);
251                             }
252                         }
253                     }
254                     else if (pfh.isOutput()){
255                         Iterator outputPorts =
256                         this.outputPortList().iterator();
257                         while (outputPorts.hasNext()) {
258                             TypedIOPort outputP =
259                             (TypedIOPort)outputPorts.next();
260                             //If these ports are the same
261                             if ((outputP.getName()).
262                                 equals (port.getName())){
263                                 _hdirector.
264                                     _replacePortsHicByPortsProjection
265                                         (outputP , port);
266                                 //Put them in the hashtab
267                                 //(HIC output-projection output)
268                                 _hdirector.htabPaPi.put(outputP,port);
269                             }
270                         }
271                     }
272                 }
273             }
274         }
275         catch (Exception e){
276             System.out.println ("Creating avatar failed");
```

```
277         }
278     }
279 
280     /** This method is used by the HIC to schedule the firing of a
281      *  DE projection which is producer. It gives the interval of time
282      *  of firing.
283      *  N.B : this gap must to be a positf number.
284      */
285     protected void _nextTimeToFire(TypedIOPort port, double travelGapTime)
286         throws IllegalActionException {
287         HicActor actorToSchedule = (HicActor)port.getContainer();
288         actorToSchedule._enqueueNextTimeToFire(travelGapTime);
289     }
290 
291     ////////////////////////////////////////////////////////////////
292     ////                     private methods                    ////
293 
294     /** This method is used by the projections for to require the
295      ** activation of their original HIC.
296      **/
297     private void  _requestActivateHIC() throws IllegalActionException{
298         _hdirector._runHic(_original);
299         return;
300     }
301 
302     /** This method is used by the projections the to record to their
303      *  original HIC, before asking its activation
304      **/
305     private void _recordRunningProjection(HicActor actor){
306         _runningProjection = actor;
307     }
308 
309 
310     ////////////////////////////////////////////////////////////////
311     ////
312     ////                       protected variables
313 
314     protected HicActor _avatar;
315     protected Director _director;
316     protected HDirector _hdirector;
317     protected HicActor _original;
318     protected HicActor _runningProjection;
319 
320     ////////////////////////////////////////////////////////////////
321     ////
322     ////                        private variables                ////
```

```
323
324     private TypedCompositeActor _container;
325     private double _currentTime;
326     private DEDirector _dedirector;
327     private String _typeProjection;
328 }
```

Code source de HDirector

```
1  /* A HDirector governs the execution of a Non-Hierarchical
2  * Heterogeneity
3  system. */
4
5
6  package ptolemy.domains.heterogeneous;
7
8  import ptolemy.actor.*;
9  import ptolemy.data.*;
10 import ptolemy.data.expr.*;
11 import ptolemy.data.type.Type;
12 import ptolemy.data.type.BaseType;
13 import ptolemy.domains.ct.kernel.CTCompositeActor;
14 import ptolemy.domains.ct.kernel.CTEmbeddedDirector;
15 import ptolemy.domains.de.kernel.DEDirector;
16 import ptolemy.domains.de.kernel.DEIOPort;
17 import ptolemy.domains.dt.kernel.DTDirector;
18 import ptolemy.domains.sdf.kernel.SDFDirector;
19 import ptolemy.domains.sdf.kernel.SDFIOPort;
20 import ptolemy.kernel.CompositeEntity;
21 import ptolemy.kernel.Port;
22 import ptolemy.kernel.util.*;
23
24 import java.io.*;
25 import java.lang.*;
26 import java.util.Hashtable;
27 import java.util.LinkedList;
28 import java.util.Iterator;
29 import java.util.Vector;
30
31
32
33
34 ////////////////////////////////////////////////////////////////////////
35 //// HDirector
36
37 /**
38 * This Non-Hierarchical Heterogeneous Director (HDirector) is able to
39 * delegate the computation of the behavior of different components to
40 * their respective Model of Computation, and to handle differents
41 * Heterogenous Interface Components (HICs).
42
43 * This HDIrector operates in three phases :
44 *   - Partitioning of the system into subsystems
45 *   - Scheduling of the subsystems
46 *   - Execution
```

```
47 * In its preinitialization phase, this Director divides the model at
48 * the border of the domains, and thus creates homogenous sub-models
49 * which are the opaque CompositeActor.
50 * The Heterogeneous Interface Components, (HicActor), which are at the
51 * border of several Model of Computation, are projected onto each
52 * subsystem to which some of their ports belong, and the other actors
53 * are transferred to their associated subsystems.
54 * The HDirector copies the connections from the original system to the
55 * subsystems and generates virtual dependencies between the different
56 * projections of a HIC on the subsystems.
57 * Then in its initialization phase, this Director schedules the
58 * activation of the subsystems and delegates their internal scheduling
59 * to their regular MoC.
60 * Finally, In its iteration phase, it executes the system in accordance
61 * to the scheduling.
62 */
63
64 /*
65 @author : Mokhoo MBOBI
66 Ecole Supérieure d'Electricité (SUPELEC France)
67 Computer Sciences Department
68 Email : Mokhoo.MBOBI@supelec.fr
69 */
70
71
72 ////////////////////////////////////////////////////////////////////
73 public class HDirector extends Director {
74
75     /** Construct an HDirector in the default workspace with an empty
76      *  string as its name.
77      *  The HDirector is added to the list of objects in the workspace.
78      *  Increment the version number of the workspace.
79      */
80     public HDirector() {
81         this(null);
82     }
83
84     /** Construct an  HDirector in the  workspace with an empty name.
85      *  The HDirector is added to the list of objects in the workspace.
86      *  Increment the version number of the workspace.
87      *  @param workspace The workspace of this object.
88      */
89     public HDirector(Workspace workspace) {
90         super(workspace);
91         _initParameters();
92     }
```

```
93      /** Construct a non-hierchical heterogeneous director (HDirector)
94       *  in the given container with the given name.
95       *  The container argument must not be null, or a
96       *  NullPointerException will be thrown. If the name argument is
97       *  null, then the name is set to the empty string.
98       *  Increment the version number of the workspace.
99       *  @param container CompositeActor which is in the top of level
100      *  in the hierarchy.
101      *  @param name Name of this non-hierchical heterogeneous director.
102      *  @exception IllegalActionException occurs if there is tentation
103      *  of crating the non-hierchical heterogeneous director in the
104      *  CompositeActor which is not in the top of level in the
105      *  hierarchy.
106      *  @exception NameDuplicationException If the top level
107      *  CompositeActor already contains a heterogeneous director
108      */
109     public HDirector(CompositeEntity container, String name)
110         throws IllegalActionException, NameDuplicationException {
111         super(container, name);
112         _initParameters();
113     }
114
115
116     ////////////////////////////////////////////////////////////
117     ////                       parameters                     ////
118
119     /** A Parameter representing the number of times that postfire may
120      *  be called before it returns false. If the value is less than
121      *  or equal to zero, then the execution will never run and an
122      *  an IllegalActionException will be thrown. So this is the
123      *  maximum number of iterations. The default value is 1000, of
124      *  type IntToken.
125      */
126     public Parameter maxIterations;
127
128     /** The start time of the simulation. The default value is 0.0,
129      *  of type DoubleToken.
130      */
131     public Parameter startTime;
132
133     /** The stop time of the simulation. The default value is
134      *  Double.MAX_VALUE, of type DoubleToken.
135      */
136     public Parameter stopTime;
137
138
```

```
139     ////////////////////////////////////////////////////////////////
140     ////                       public methods                   ////
141
142     /** React to a change in an attribute.
143      *  If the changed attribute matches a parameter of this
144      *  non-hierchical heterogeneous director (maxIterations,
145      *  startTime and stopTime), then the corresponding local copy of
146      *  the parameter value will be updated.
147      *  @param attribute The changed parameter.
148      *  @exception IllegalActionException If the parameter set is not
149      *  valid.
150      */
151     public void attributeChanged(Attribute attribute)
152         throws IllegalActionException {
153         if (_debugging) _debug("Updating HDirector parameter: ",
154             attribute.getName());
155
156         if (attribute == startTime) {
157             _startTime = ((DoubleToken)startTime.getToken()).
158                     doubleValue();
159         }
160
161         else if (attribute == stopTime) {
162             _stopTime = ((DoubleToken)stopTime.getToken()).
163                         doubleValue();
164         }
165
166         else if (attribute == maxIterations) {
167             int value = ((IntToken)maxIterations.getToken())
168                         .intValue();
169             if (value < 1) {
170                 throw new IllegalActionException(this,
171                         "Cannot set a zero or negative
172                                         iteration number");
173             }
174             _maxIterations = value;
175         }
176
177         else {
178             super.attributeChanged(attribute);
179         }
180     }
181
182
183
184
```

```
185     /** Invoke an iteration on all of the CompositeActor of the
186      *  heterogeneous model.
187      *  Each CompositeActor which is a sub-domain is iterated
188      *  repeatedly (prefire(), fire(), postfire()), until its
189      *  prefire() method returns false.
190      */
191     public void fire() throws IllegalActionException {
192         boolean refire;
193         //The simulation loops until reaching _maxIterations which
194         //is the maximum number of iterations
195         do {
196             refire = true;
197             _iterationsCount++;
198             _currentTime = _currentTime + 1;
199             //It fires the different domains one by one
200             for (int i = 0 ; i < vTopoTriOfProjections.size() ; i++){
201                 TypedCompositeActor domainToFire =
202                     (TypedCompositeActor)vTopoTriOfProjections
203                         .elementAt(i);
204                 domainToFire.prefire();
205                 domainToFire.fire();
206                 domainToFire.postfire();
207                 if (_iterationsCount == _maxIterations){
208                     refire = false;
209                 }
210             }
211         } while (refire);
212     }
213 
214     /** This method is overrided to perform additional
215      *  initialization functions according to the scheduling of the
216      *  system.
217      *  This method should be invoked once per execution, after the
218      *  preinitialization method, but before any iteration.
219      *  First, this method invokes the initialize() method of its
220      *  super class for to set the current time to 0.0 and to
221      *  initialize all composite actors and their embedded actors.
222      *  Then, it compute the schedule, by performing the topological
223      *  sort on all the projections of HicActor, so that composite
224      *  actors can can be properly fired
225      */
226     public void initialize() throws IllegalActionException {
227         super.initialize();
228         _computeDomainsSchedule();
229     }
230 
```

```
231     /* This method is overrided to perform additional
232      * preinitialization functions according to the partitionning
233      * of the system.
234      */
235     public void preinitialize() throws IllegalActionException {
236         _divideSystem();
237         _disconnectionReconnectionPorts();
238         _processVirtuelPorts();
239         super.preinitialize();
240     }
241
242
243
244     ////////////////////////////////////////////////////////////////
245     ////                       protected methodes                 ////
246
247     /* This method gives the compatibility between a given port and
248      * a given director
249      */
250     protected boolean _isCompatible(String director, String port){
251         boolean _compatibility = false;
252         if((_getCompatiblePortOf(director)).equals(port)){
253             _compatibility = true;
254         }
255         return _compatibility;
256     }
257
258     /* This method replace the ports of the HIC by the ports of the
259      * projection
260      */
261     protected void _replacePortsHicByPortsProjection(TypedIOPort
262                         portHic, TypedIOPort portPro){
263         for (int b = 0 ; b < tabInPorts.length ; b++){
264             if (portHic == tabInPorts[b]){
265                 tabInPorts[b] = portPro;
266             }
267         }
268         for (int b = 0 ; b < tabOutPorts.length ; b++){
269             if (portHic == tabOutPorts[b]){
270                 tabOutPorts[b] = portPro;
271             }
272         }
273         return;
274     }
275
276
```

```
277     /** This method runs the HIC which is passed in parameter.
278      */
279     protected void _runHic(HicActor hic){
280         try{
281             hic.fire();
282         }
283         catch (Exception e){
284             System.out.println (" Error _runHic()");
285         }
286     }
287 
288     ////////////////////////////////////////////////////////////////
289     ////                         private methodes                 ////
290 
291     /** This method set the actor in the created domain
292      */
293     public void _actorSetting(TypedAtomicActor actor){
294         try{
295             TypedCompositeActor domain = (TypedCompositeActor)
296                         vectOmega.elementAt(_indiceOmega);
297             actor.setContainer(domain);
298             //Increment the index of tabStoreActors and put actor
299             //into for to avoid its reusse
300             vectActorsPlaced.addElement(actor);
301         }
302         catch (Exception e){
303             System.out.println (" Error _actorSetting");}
304         return;
305     }
306 
307     /* This method checks the consecutive HICs for th insert a
308      * repeater between them
309      */
310     private void _checkConsecutiveHics(Actor actor) {
311         Iterator outports = actor.outputPortList().iterator();
312         while (outports.hasNext()) {
313             TypedIOPort outPort = (TypedIOPort) outports.next();
314             Iterator inPorts =
315               outPort.deepConnectedInPortList().iterator();
316             while (inPorts.hasNext()) {
317                 TypedIOPort inPort = (TypedIOPort)inPorts.next();
318                 if(inPort.getContainer() instanceof HicActor){
319                     _effectiveInsertRepeater(outPort, inPort);}
320             }
321         }
322     }
```

```
323     /* This method computes the topological sort of the actors
324      */
325     private void _computeActorsTopologicalSort(){
326         // inputDeg contains the account of input ports of each
327         // actor
328         inputDeg = new int[vectActors.size()];
329         for (int indH = 0 ; indH < vectActors.size() ; indH++){
330             vectPredActor.removeAllElements();
331             TypedAtomicActor myActor =
332                 (TypedAtomicActor)vectActors.elementAt(indH);
333             //Takes the vector of the pedecessors of the actor
334             //myActor
335             vectPredecessors.removeAllElements();
336             vectPredActor = _uniquePredecessorActors(myActor);
337             inputDeg[indH] = vectPredActor.size();
338             //If the vector containing the input ports of this actor
339             //is empty, then it has no predecessor. So, it is an
340             //initial vertex or a node of zero order and enqueue it
341             if (inputDeg[indH] == 0){
342                 vTopoTriOfActors.addElement(myActor);
343             }
344         }
345         do{
346             //Checks the node of zero order
347             for (int i = 0 ; i < vTopoTriOfActors.size() ; i++){
348                 Actor actor = (Actor)vTopoTriOfActors.elementAt(i);
349                 vectSuccActor.removeAllElements();
350                 vectSuccActor = _uniqueSuccessorActors(actor);
351                 for(int iv = 0 ; iv < vectSuccActor.size() ; iv++){
352                     TypedAtomicActor successor =
353                         (TypedAtomicActor)vectSuccActor
354                                 .elementAt(iv);
355                     //Takes the index of the successor of ector in
356                     //vectActors
357                     int intRang = vectActors.indexOf(successor);
358                     inputDeg[intRang] = inputDeg[intRang] - 1;
359                     //If it becomes a node of zero order (initial
360                     //or end) enqueue it
361                     if (inputDeg[intRang] == 0){
362                         vTopoTriOfActors.addElement(successor);
363                     }
364                 }
365             }
366         }
367         while(vTopoTriOfActors.size() == 0);
368     }
```

```
369     /** This method should be invoked once per execution, after the
370      *  preinitialization method, but before any iteration.
371      *  First, this method invokes the initialize() method of its
372      *  super class for to set the current time to 0.0 and to
373      *  initialize all composite actors and their embedded actors.
374      *  Then, it compute the schedule, by performing the
375      *  topological sort on all the projections of HicActor,
376      *  so that composite actors can can be properly fired
377      */
378     private void _computeDomainsSchedule()
379                     throws IllegalActionException {
380         //vectQueuePro is the queue of the projections
381         Vector vectQueuePro = new Vector();
382 
383         //inputDegPro contains the amount of input port of each
384         //projection which are compatible to its domain
385 
386         inputDegPro = new int[vectPro.size()];
387         for (int indH = 0 ; indH < vectPro.size() ; indH++){
388             HicActor projection = (HicActor)vectPro.elementAt(indH);
389             String direction = projection._getTypeProjection();
390             //For the projections consummer
391             if(direction.equals("Rx")){
392                 vectOutPortsPredecessorsHics.removeAllElements();
393                 //Take the vector of the first HIC predecessours.
394                 Vector pHics =
395                             _portsOfFirstPredecessorHics(projection);
396                 //The total of the input of projection equals the
397                 //size of the vector containing the ports of its
398                 //predecessors. This is because the vector contains
399                 //all the output ports of its predecessors connected
400                 //to its inputs. By corresponding, we have the amount
401                 //of its inputs. So, the total of inputs of
402                 //projection equals the total of its inputs ports
403                 inputDegPro[indH] = pHics.size();
404                 //If inputDegPro contains the total of imputs of
405                 //projection is empty, then it have not a
406                 //predecessor The this HIC is a vertex which have
407                 //a node of zero order among the HICs, so it is
408                 //enqueued
409                 if (inputDegPro[indH] == 0){
410                     vectQueuePro.addElement(projection);
411                 }
412             }
413 
414 
```

```
415             //Else for the producter projections
416             else if(direction.equals("Tx")){
417                 //Check the producter projections
418                 int countInputPro = 0;
419                 for(int j = 0 ; j < vectPro.size() ; j++){
420                     actorP = (HicActor)vectPro.elementAt(j);
421                     String directionP = actorP._getTypeProjection();
422                     /Check its consummers which are considered as
423                     //the predecessors
424                     if(directionP.equals("Rx")){
425                         //If the producter and the consummer have the
426                         //same HIC, adds the input account.
427                         //Knowing that a predecessor is considered as
428                         //an input
429                         if(actorP._getOriginal() ==
430                             projection._getOriginal()){
431                             countInputPro++;
432                         }
433                     }
434                 }
435                 inputDegPro[indH] = countInputPro;
436             }
437         }
438         do{
439             //Check all the nodes of zero order
440             for (int i = 0 ; i < vectQueuePro.size() ; i++){
441                 HicActor actor = (HicActor)vectQueuePro.elementAt(i);
442                 String directionbis = actor._getTypeProjection();
443                 if(directionbis.equals("Rx")){
444                     //Check the projections
445                     for(int j = 0 ; j < vectPro.size() ; j++){
446                         HicActor actorP = (HicActor)vectPro
447                             .elementAt(j);
448                         String directionP =
449                                 actorP._getTypeProjection();
450                         //Check the producers which are considered as
451                         //its successors
452                         if(directionP.equals("Tx")){
453                             //If the producter and the consummer have
454                             //the same HIC, adds the input account.
455                             //Knowing that a predecessor is considered
456                             //as an input
457                             if(actorP._getOriginal() ==
458                                     actor._getOriginal()){
459
460
```

```
461                                 //decrements inputDegPro.
462                                 inputDegPro[j] = inputDegPro[j] - 1;
463                                 //If its becomes a node of zero order
464                                 //(outgoing vertex), put it in the
465                                 //queue
466                                 if (inputDegPro[j] == 0){
467                                      vectQueuePro.addElement(actorP);
468                                 }
469                             }
470                         }
471                     }
472                 }
473                 else if(directionbis.equals("Tx")){
474                     vectInPortsSuccessorsHics.removeAllElements();
475                     //Take the HICs first successor vector.
476                     Vector sHics = _portsOfFirstSuccessorHics(actor);
477                     if(!(sHics.isEmpty())){
478                         for(int h = 0 ; h < sHics.size() ; h++){
479                             TypedIOPort port =
480                                 (TypedIOPort)sHics.elementAt(h);
481                             TypedAtomicActor actorS =
482                                 (TypedAtomicActor)port
483                                         .getContainer();
484                             //Take the index of actorS in vectPro
485                             for(int j = 0 ;
486                                         j < vectPro.size() ; j++){
487                                 TypedAtomicActor actorP =
488                                 (TypedAtomicActor)vectPro
489                                         .elementAt(j);
490                                 if(actorP == actorS){
491                                     inputDegPro[j] =
492                                         inputDegPro[j] - 1;
493                                 }
494                                 //If its becomes a node of zero order
495                                 //(outgoing vertex), put it in the
496                                 //queue
497                                 if (inputDegPro[j] == 0){
498                                     vectQueuePro.addElement(actorS);
499                                 }
500                             }
501                         }
502                     }
503                 }
504             }
505         }
506
```

```
507         while(vectQueuePro.size() == 0);
508             for (int ind = 0 ; ind < vectQueuePro.size() ; ind++){
509             TypedAtomicActor actorForDomain =
510                 (TypedAtomicActor)vectQueuePro.elementAt(ind);
511             TypedCompositeActor domain =
512                 (TypedCompositeActor)actorForDomain.getContainer();
513             if (!vTopoTriOfProjections.contains(domain)){
514                 vTopoTriOfProjections.addElement(domain);
515             }
516         }
517     }
518
519     /* This process the differerd actors. This actor will be placed
520      * in the same domain that the first actor that is already placed
521      * and which have a path to it.
522      */
523    private void _differedPlacement(TypedAtomicActor actorD){
524         boolean actorPlaced = false;
525         LinkedList successors = new LinkedList();
526         Iterator outports = actorD.outputPortList().iterator();
527         while (outports.hasNext()) {
528             TypedIOPort outPort = (TypedIOPort) outports.next();
529             Iterator inPorts =
530                 outPort.deepConnectedInPortList().iterator();
531             while (inPorts.hasNext()) {
532                 TypedIOPort inPort = (TypedIOPort)inPorts.next();
533                 TypedAtomicActor post =
534                     (TypedAtomicActor)inPort.getContainer();
535                 if (vectActorsPlaced.contains(post)) {
536                     TypedCompositeActor domain =
537                         (TypedCompositeActor)post.getContainer();
538                     try{
539                         actorD.setContainer(domain);
540                         vectActorsPlaced.addElement(actorD);
541                     }
542                     catch (Exception e){
543                         System.out.println ("differedPlacement
544                                                 failed");
545                     }
546                     break;
547                 }
548             }
549         }
550     }
551
552
```

```
553     /** For the new model to behave like the initial one, all
554      *  connections are cancelled and actors are then connected
555      *  inside each domain according to the communication channels of
556      *  the initial model before the system is run.
557      */
558     private void _disconnectionReconnectionPorts()
559         throws IllegalActionException {
560         //Ports disconnection
561         for (int inp = 0 ; inp < tabInPorts.length ; inp++){
562             tabInPorts[inp].unlinkAll();
563         }
564         for (int ou = 0 ; ou < tabOutPorts.length ; ou++){
565             tabOutPorts[ou].unlinkAll();
566         }
567         //Ports reconnection
568         for (int inp = 0 ; inp < tabInPorts.length ; inp++){
569             for (int ou = 0 ; ou < tabOutPorts.length ; ou++){
570                 if (allPortsAdjacentMatrix[inp][ou] == 1 ){
571                     if (tabInPorts[inp].getContainer()
572                             .getContainer()
573                                 == tabOutPorts[ou].getContainer().
574                                     getContainer()){
575                         TypedCompositeActor omegaForConnnexion =
576                             (TypedCompositeActor)tabInPorts[inp].
577                                 getContainer().getContainer();
578                         omegaForConnnexion.connect(tabOutPorts[ou],
579                         tabInPorts[inp]);
580                         break;
581                     }
582                 }
583             }
584         }
585     }
586
587     /** Since the original system contains regular actors, HICs
588      *  and communication channels that link the ports of actors and
589      *  HICs, when there is a path between two HICs, the output of
590      *  the producer HIC, the input of the consumer HIC and all the
591      *  actors will be put into the same subsystem which are the
592      *  CompositeActor. The two HICs at both extremities of the path
593      *  will be put on this compositeActor and the inputs of the
594      *  producer HIC and the outputs of the consumer HIC are masked
595      *  after the projection. This partitioning method begin by first
596      *  performing a topological sort on the actors of the system.
597
598
```

```
599      *  Then, for each actor, in an order which is compatible with the
600      *  topological sort, it looks for a subsystem that uses the MoC
601      *  of the actor. If such a subsystem exists, it puts the actor
602      *  there if the dependencies allow it, else a new subsystem is
603      *  created to host the actor. The condition on the dependencies
604      *  must be respected so that the subsystems that result from the
605      *  partitioning can be scheduled.
606      *  These conditions ensure that there won't be cross dependencies
607      *  between subsystems.
608      *  If those conditions is hold for more than one subsystem for a
609      *  given actor, in which case we choose to put the actor in the
610      *  subsystem that already contains a projection of the HIC which
611      *  belongs to the same segment as the actor if any, or we will
612      *  put it in the most recently created compatible subsystem.
613      */
614     private void _divideSystem() throws IllegalActionException {
615         _insertRepeaters();
616         //Hashtab (Projection - HIC)
617         htabProHic = new Hashtable(_totalPortsHics);
618         //Hashtab (input of HIC - input projection)
619         htabPiPa = new Hashtable(_totalPortsHics);
620         //Hashtab (output of HIC - output projection)
621         htabPaPi = new Hashtable(_totalPortsHics);
622
623         //Setting of 3 vectors of actors, inputs ports and outputs
624         //ports
625         Iterator completeModel =
626             container.deepEntityList().iterator();
627         while(completeModel.hasNext()){
628             TypedAtomicActor actor =
629                     (TypedAtomicActor)completeModel.next();
630             vectActors.addElement(actor);
631             //Check inputs and outputs ports.
632             Iterator portsOfModel = actor.portList().iterator();
633             while (portsOfModel.hasNext()) {
634                 TypedIOPort port = (TypedIOPort)portsOfModel.next();
635                 if(port.isInput()){
636                     vectInPorts.addElement(port);
637                 }
638                 else if(port.isOutput()){
639                     vectOutPorts.addElement(port);
640                 }
641             }
642         }
643
644
```

```
645         //Tables of all inputs and outputs ports
646         tabInPorts = new TypedIOPort[vectInPorts.size()];
647         tabOutPorts = new TypedIOPort[vectOutPorts.size()];
648         int v = -1; int w = -1;
649         Iterator TheCompleteModel =
650              container.deepEntityList().iterator();
651         while(TheCompleteModel.hasNext()){
652             TypedAtomicActor actor =
653                 (TypedAtomicActor)TheCompleteModel.next();
654             Iterator portsOfModel = actor.portList().iterator();
655             while (portsOfModel.hasNext()) {
656                 TypedIOPort port = (TypedIOPort)portsOfModel.next();
657                 if(port.isInput()){
658                     v++;
659                     tabInPorts[v] = port;
660                 }
661                 else if(port.isOutput()){
662                     w++;
663                     tabOutPorts[w] = port;
664                 }
665             }
666         }
667         //When the communication channel between 2 actors is absent,
668         //this adjacent matrix = zero else = 1
669         //NB : index of this matrix is the index of actors in vectActor
670
671         actorAdjacentMatrix = new
672               int[vectActors.size()][vectActors.size()];
673
674         //When the communication channel between 2 ports is absent,
675         //this adjacent matrix = zero else = 1
676         //NB : index of this matrix is the index of in  vectInports
677         //and in vectOutPorts
678
679         portsAdjacentMatrix = new
680                         int[vectInPorts.size()][vectOutPorts.size()];
681
682         //Table of differents connections between differents ports
683         //with the ports of the projections with replace the ports of
684         //Hics
685
686         allPortsAdjacentMatrix = new
687                         int[vectInPorts.size()][vectOutPorts.size()];
688
689
690
```

```
691         //Adjacent matrix
692         for(int i = 0 ; i < vectActors.size() ; i++){
693             TypedAtomicActor actor =
694                     (TypedAtomicActor)vectActors.elementAt(i);
695             Iterator inports = actor.inputPortList().iterator();
696             while (inports.hasNext()) {
697                 TypedIOPort in = (TypedIOPort) inports.next();
698                 Iterator outPorts =
699                         in.deepConnectedOutPortList().iterator();
700                 while (outPorts.hasNext()) {
701                     TypedIOPort out = (TypedIOPort)outPorts.next();
702                     //index of input port
703                     int indInPort = vectInPorts.indexOf(in);
704                     //index of the actor which carry this port
705                     int indInActor = vectActors.indexOf(actor);
706                     //index of output port
707                     int indOutPort = vectOutPorts.indexOf(out);
708                     //index of the actor which carry this port
709                     int indOutActor =
710                     vectActors.indexOf(out.getContainer());
711                     //Fill up the adjacent matrix (dual sense)
712                     actorAdjacentMatrix[indInActor][indOutActor] = 1;
713                     //Fill up the adjacent matrix (single sense)
714                     portsAdjacentMatrix[indInPort][indOutPort] = 1;
715                 }
716             }
717         }
718         allPortsAdjacentMatrix = portsAdjacentMatrix;
719         _computeActorsTopologicalSort();
720
721         //for each actor, in an order which is compatible with the
722         //topological sort, it looks for a subsystem that uses the MoC
723         //of the actor. If such a subsystem exists, it puts the actor
724         //there if the dependencies allow it, else a new subsystem is
725         //created to host the actor.
726
727         for (int indTr = 0 ; indTr < vTopoTriOfActors.size() ; indTr++){
728             TypedAtomicActor actor =
729                 (TypedAtomicActor)vTopoTriOfActors.elementAt(indTr);
730             //Reinitialization of all vectors
731             vectFirstHics.removeAllElements();
732             vectOutPortsPredecessorsHics.removeAllElements();
733             vectInPortsSuccessorsHics.removeAllElements();
734             //Determination of two vectors contains :
735             //First vector : all HICs intial vertex and
736             //Second vector : all HICs end vertex
```

```
737             Vector preHics = _portsOfFirstPredecessorHics(actor);
738             Iterator preHic = preHics.iterator();
739             while (preHic.hasNext()) {
740                 TypedIOPort pH = (TypedIOPort)preHic.next();
741                 vectFirstHics.addElement(pH);
742             }
743             Vector succHics = _portsOfFirstSuccessorHics(actor);
744             Iterator succHic = succHics.iterator();
745             while (succHic.hasNext()) {
746                 TypedIOPort sH = (TypedIOPort)succHic.next();
747                 vectFirstHics.addElement(sH);
748             }
749
750             //Process of the first actor from the topological sort.
751             //This actor must create the first domain.
752             //N.B : we call Omega those domains
753             //When Omega does'nt exist
754
755             if (vectOmega.isEmpty()){
756                 _subSystemCreation(vectFirstHics);
757                 _actorSetting(actor);
758             }
759
760             // For all remains actors, if this actor is not a HIC
761             else if ((!(vectOmega.isEmpty())) &&
762                       (!(actor instanceof HicActor))){
763                 //The set of its vertex (initial and end) must not be
764                 //empty
765                 if(!(vectFirstHics.isEmpty())){
766                     //Looking for a domain that uses the MoC of this
767                     //actor. If such a subsystem exists, it puts the
768                     //actor there.
769                     //So, we must check the compatibility between the
770                     //first port of the first HIC in the set
771                     //(vectFirstHics) and all domains which allready
772                     //exist.
773                     //Initialisation
774                     boolean pathViaHic= false;
775                     boolean pathViaHicHic= false;
776                     boolean newDomainNotRequired = false;
777                     boolean compatibility = false;
778                     //Initialisation the vectors in which we put the
779                     //domains that uses the MoC of this actor.
780                     Vector vectDomains = new Vector();
781                     Vector vectNewDomains = new Vector();
782
```

```
783                     //Check the domains that uses the MoC of this actor.
784                     //and first put them in vectDomains.
785                     //Then, check the domain which have no path from its
786                     //actors to the considered actor that goes through
787                     //a HIC.
788                     //If there is, put them in vectNewDomains
789                     //Else if there is not i.e. create a new dommain
790                     //N.B : The test is done on the first port in
791                     //vectFirstHics
792                     for (int indi = 0 ; indi < vectOmega.size() ; indi++)
793                         TypedIOPort port =
794                             (TypedIOPort)vectFirstHics.firstElement();
795                         //If the domain indexed by indi of
796                         //tabDynamicOmega uses the MoC of this actor
797                         //then, put compatibility to true.
798                         if( _getCompatibility(port, indi)){
799                             compatibility = true;
800                             vectDomains.add(vectOmega.elementAt(indi));
801                         }
802                     }
803                     //If there is et least one donain that uses the MoC
804                     //of this actor
805                     if(compatibility){
806                         //Check among the domain in vectDomains those
807                         //that have no path to the considered actor that
808                         //goes through a HIC.
809                         //If there is, put them in vectNewDomains and
810                         //put newDomainNotRequired to true
811                         //Else if there is not, put newDomainNotRequired
812                         //to false for to create a new domain
813                         for (int indV = 0 ; indV <
814                             vectDomains.size() ; indV++){
815                             //Check the existence of the path
816                             lastDomain = (TypedCompositeActor)
817                                     vectDomains.elementAt(indV);
818                             pathViaHic = _testViaHic(actor, lastDomain);
819                             if (!(pathViaHic)){
820                                 vectNewDomains.add(lastDomain);
821                                 newDomainNotRequired = true;
822                             }
823                         }
824                         //If a new doamin is really required
825                         if (!(newDomainNotRequired)){
826                             _subSystemCreation(vectFirstHics);
827                             _actorSetting(actor);
828
```

```
829                         //Now actor is placed, check if there is
830                         //actor which precedes this actor and
831                         //which is allready placed in other
832                         //domain that have a  path to the
833                         //considered actor that goes through a HIC.
834                         //That domains are those that in
835                         //vectDomains but not in vectNewDomains.
836                         //If there is such actors put a relay for
837                         //each of them.
838                         _insertRelay(vectDomains,
839                             vectNewDomains, actor);
840                     }
841                     //Else if a new domain is not required, put this
842                     //actor in the existing domain that use the Moc
843                     //of this actor and that have no path to the
844                     //considered actor that goes through a HIC.
845                     //If those conditions is hold for more than one
846                     //domains, in which case we choose to put this
847                     //actor in the domain that already contains a
848                     //projection of the HIC which belongs to the
849                     //same segment as the actor if any, or we will
850                     //put it in the most recently created
851                     //compatible domain.
852                     else if (newDomainNotRequired){
853                         try{
854                             TypedIOPort portS = (TypedIOPort)
855                             vectFirstHics.firstElement();
856                             //Found its HIC
857                             TypedAtomicActor actorS =
858                                 (TypedAtomicActor)portS.
859                                     getContainer();
860                             //Check among the vectors able to
861                             //receive this actor
862                             Iterator domains =
863                                 vectNewDomains.iterator();
864                             while (domains.hasNext()) {
865                                 boolean foundLastDomain =false;
866                                 TypedCompositeActor domain =
867                                 (TypedCompositeActor)
868                                         domains.next();
869                                 //On itère sur le domaine
870                                 Iterator actorsInOmega =
871                                 domain.deepEntityList().
872                                         iterator();
873
874
```

```
875                             while (actorsInOmega.hasNext()){
876                                 TypedAtomicActor actorInOmega =
877                                     (TypedAtomicActor)
878                                 actorsInOmega.next();
879                                 if(actorInOmega
880                                     instanceof HicActor){
881                                     //If HIC is already exist
882                                     //in this domain, ten the
883                                     //domain is the first
884                                     //founded
885                                     TypedAtomicActor original =
886                                         (TypedAtomicActor)
887                                             htabProHic.
888                                             get(actorInOmega);
889                                     if ((original.getName().
890                                         toString()).equals
891                                             (actorS.getName().
892                                                 toString())){
893                                         lastDomain = domain;
894                                         foundLastDomain =true;
895                                         break;
896                                     }
897                                 }
898                             }
899                             if(foundLastDomain){
900                                 break;
901                             }
902                             //Else this domain will be the
903                             //last one
904                             if(!(foundLastDomain)){;
905                                 lastDomain = (TypedCompositeActor
906                                 vectNewDomains.lastElement();
907                             }
908                         }
909                         actor.setContainer(lastDomain);
910                         _setSingleProjection(vectFirstHics,
911                             lastDomain);
912                         vectActorsPlaced.addElement(actor);
913                     }
914                     catch (Exception e){
915                         System.out.println(" Error ");
916                     }
917
918
919
920
```

```
921                             //Now again actor is placed, check if
922                             //there is actor which precedes this actor
923                             //and which is allready placed in other
924                             //domain that have a  path to the
925                             //considered actor that goes through a HIC.
926                             //That domains are those that in
927                             //vectDomains but not in vectNewDomains.
928                             //If there is such actors put a relay for
929                             //each of them.
930                             _insertRelay(vectDomains,
931                             vectNewDomains, actor);
932                         }
933                     }
934                     //If there is no domain that use the MoC of this
935                     //then, creates a new domain
936                     else if(!(compatibility)){
937                         _subSystemCreation(vectFirstHics);
938                         _actorSetting(actor);
939                     }
940                 }
941                 //The set of its vertex (initial and end) is empty
942                 //store this actor and
943                 // Since it not produce data toward a HIC, it not
944                 //usefull to create a new domain for it because it
945                 //will constrainds the actors which are connected to
946                 //it to be placed in the same domain that it.
947                 //So, since it has no incoming data dependancies we
948                 //chose to postpone it seting. It will be placed in
949                 //the same domain that the first actor that is
950                 //already placed and which have a path to it.
951                 else if(vectFirstHics.isEmpty()){
952                     vectDiferedActor.addElement(actor);
953                 }
954             }
955             //Process of differed acors if there is one.
956                 if(!(vectDiferedActor.isEmpty())){
957                 Iterator diferedActor = vectDiferedActor.iterator();
958                 while (diferedActor.hasNext()) {
959                     TypedAtomicActor dActor = (TypedAtomicActor)
960                          diferedActor.next();
961                     _differedPlacement(dActor);
962                 }
963             }
964         }
965     }
966
```

```
967     /* This method disconnect the two ports of HICs, create the new
968      * repeater and connect it to the port of HIC
969      */
970     private void _effectiveInsertRepeater(TypedIOPort out,
971                     TypedIOPort in) {
972         try{
973             //Disconnect the two ports
974             out.unlink(0);
975             in.unlink(0);
976             //Create a new repeater
977             _indiceRepeater++;
978             TypedCompositeActor model =
979             (TypedCompositeActor)this.getContainer();
980             Repeater repeater =(Repeater) new
981             Repeater(model, "repeater"+_indiceRepeater);
982             //On lui met les ports des deux HICs
983             TypedIOPort portIn = (TypedIOPort)in.clone();
984             portIn.unlinkAll();
985             TypedIOPort portOut = (TypedIOPort)out.clone();
986             portOut.unlinkAll();
987             portIn.setContainer(repeater);
988             portOut.setContainer(repeater);
989             //Set the two ports of the two HICs to it
990             model.connect(out, portIn);
991             model.connect(portOut, in);
992         }
993         catch (Exception e){
994             System.out.println (" Error insertRepeater");
995         }
996     }
997
998     /*This method checs if there is a path between two given actors
999      */
1000    private boolean _existPathBetweenActors(TypedAtomicActor
1001                    firstActor, TypedAtomicActor secondActor){
1002        int i , j , somme;
1003        boolean pathFound = false;
1004        boolean valMatriceEqualZero = false;
1005        int matIntermediaire[][];
1006        matIntermediaire = actorAdjacentMatrix;
1007        //If the matrix is not zero and the path is not found
1008        //(Is the crossing line-colun is diffŽérent to zero ?
1009        while ((!(valMatriceEqualZero)) && (!(pathFound))){
1010            //Found the index of those two actors
1011            int _rx = vectActors.indexOf(firstActor);
1012            int _tx = vectActors.indexOf(secondActor);
```

```
1013            //Checs if there is a path with a length = 1
1014            if ( matIntermediaire[_rx][_tx] == 1 ){
1015                pathFound = true;
1016            }
1017            //If there is no path whith a length = 1 then multiply
1018            //matIntermediaire by the actorAdjacentMatrix for to
1019            //check the path whith a length = 2 and so on.
1020            if (!( matIntermediaire[_rx][_tx] == 1 )){
1021                //Set the sum to zero
1022                somme = 0;
1023                matIntermediaire = _multiplyMatrix(matIntermediaire,
1024                    actorAdjacentMatrix);
1025                for (i = 0 ; i < matIntermediaire.length ; i++){
1026                    for (j = 0 ; j < matIntermediaire.length ; j++){
1027                        somme = somme + matIntermediaire[i][j];
1028                    }
1029                }
1030                //Is the matrix = 0 ?
1031                //If there is, valMatriceEqualZero = true then there
1032                //is not a path between _rx et _tx
1033                if (somme == 0){
1034                    valMatriceEqualZero = true;
1035                }
1036                else if(somme != 0){
1037                    pathFound = true;
1038                }
1039            }
1040        }
1041        return pathFound;
1042    }
1043
1044    /*This method checs if there is a path between two given ports
1045     */
1046    private boolean _existPathBetweenPorts(TypedIOPort firstPort ,
1047                           TypedIOPort secondPort){
1048        boolean pathFound = false;
1049        int matIntermediairePorts[][];
1050        matIntermediairePorts = portsAdjacentMatrix;
1051        //Check the path between two ports
1052        //(Is the crossing line-colun is diffŽérent to zero ?
1053        int _tx = vectOutPorts.indexOf(secondPort);
1054        int _rx = vectInPorts.indexOf(firstPort);
1055        //Checs if there is a path with a length = 1 between
1056        //_tx and _rx
1057        if ( matIntermediairePorts[_rx][_tx] == 1 ){
1058            pathFound = true; }
```

```
1059            return pathFound;
1060        }
1061
1062        /* This method gives the compatibility between a given port and
1063         * a given domain
1064         */
1065        private boolean _getCompatibility(TypedIOPort portH,
1066                                          int indOmega){
1067            TypedIOPort port = (TypedIOPort)portH;
1068            boolean is = false;
1069            if (port == null){
1070                is = false;
1071            }
1072            else {
1073                if (!(port == null)){
1074                    TypedCompositeActor domain =
1075                    (TypedCompositeActor)vectOmega.elementAt(indOmega);
1076                    is =_isCompatible(domain.getDirector().getClass().
1077                            toString(), port.getClass().toString());
1078                }
1079            }
1080            return is;
1081        }
1082
1083         /* This method return the port that use its MoC
1084         */
1085        private String _getCompatiblePortOf(String director){
1086            String port = null;
1087            if (director.equals
1088                    ("class ptolemy.domains.sdf.kernel.SDFDirector")){
1089                port = "class ptolemy.domains.sdf.kernel.SDFIOPort";
1090            }
1091            else if (director.equals
1092                    ("class ptolemy.domains.de.kernel.DEDirector")){
1093                port = "class ptolemy.domains.de.kernel.DEIOPort";
1094            }
1095            else if (director.equals
1096                    ("class ptolemy.domains.dt.kernel.DTDirector")){
1097                port = "class ptolemy.actor.TypedIOPort";
1098            }
1099            else if (director.equals
1100                    ("class ptolemy.domains.ct.kernel.CTEMixedDirector")){
1101                port = "class ptolemy.actor.TypedIOPort";
1102            }
1103            return port;
1104        }
```

```
1105      /** Create and initialize all parameters to their default values.
1106       */
1107     private void _initParameters() {
1108         try {
1109             _stopTime = java.lang.Double.MAX_VALUE;
1110             _startTime = 0.0;
1111             //_maxIterations = 2000;
1112             //_maxIterations = 8000;
1113             _maxIterations = 20000;
1114             _iterationsCount = 0;
1115             startTime = new Parameter(
1116                     this, "startTime", new DoubleToken(0.0));
1117             startTime.setTypeEquals(BaseType.DOUBLE);
1118             stopTime = new Parameter(this, "stopTime",
1119                     new DoubleToken(_stopTime));
1120             stopTime.setTypeEquals(BaseType.DOUBLE);
1121             maxIterations = new Parameter(this, "maxIterations",
1122                     new IntToken(_maxIterations));
1123             maxIterations.setTypeEquals(BaseType.INT);
1124         }
1125         catch (IllegalActionException e) {
1126             throw new InternalErrorException(
1127                     "Parameter creation error.");
1128         }
1129         catch (NameDuplicationException ex) {
1130             throw new InvalidStateException(this,
1131                     "Parameter name duplication.");
1132         }
1133     }
1134
1135     /* This method insert the relay in both of domains which contained
1136      * two differents actors connected but placed in those differents
1137      * domains
1138      */
1139     private void _insertRelay(Vector vectDomains,
1140         Vector vectNewDomains, TypedAtomicActor actorRx){
1141         boolean existPred = false;
1142         Vector vectOut = new Vector();
1143         Vector vectIn = new Vector();
1144         //Take the domain of actorRx
1145         TypedCompositeActor domainR =
1146                   (TypedCompositeActor)actorRx.getContainer();
1147         //Check the domain which use the same MoC as actorRx
1148         for(int iT = 0 ; iT < vectDomains.size() ; iT++){
1149             TypedCompositeActor domainT =
1150                (TypedCompositeActor)vectDomains.elementAt(iT);
```

```
1151            boolean foundDomain = false;
1152            //Check the domain which use the same MoC as actorRx but
1153            //which have no path to actorRx that goes through a HIC.
1154            //In fact, take the complement of vectNewDomains in
1155            //vectDomains
1156            for(int iTT = 0 ; iTT < vectNewDomains.size() ; iTT++){
1157                if (domainT == vectNewDomains.elementAt(iTT)){
1158                    foundDomain = true;
1159                }
1160            }
1161            //If this domain is concerned
1162            if (!(foundDomain)){
1163                //Check all actors and take which are predecesso
1164                //of actor
1165                Iterator domains = domainT.deepEntityList().iterator();
1166                while (domains.hasNext()) {
1167                    TypedAtomicActor actorInDom =
1168                        (TypedAtomicActor)domains.next();
1169                    //Here, HICs is not considered yet
1170                    if(!(actorInDom instanceof HicActor)){
1171                        //Put their connected ports in the vectors.
1172                        //For this, check all the inputs ports of
1173                        //of actorRx
1174                        Iterator inputPorts =
1175                            actorRx.inputPortList().iterator();
1176                        while (inputPorts.hasNext()) {
1177                            existPred = false;
1178                            TypedIOPort inputP =
1179                            (TypedIOPort)inputPorts.next();
1180                            //For each input of actorRx, check all
1181                            //the   output ports of actorInDom
1182                            Iterator outputPorts =
1183                            actorInDom.outputPortList().iterator();
1184                                while (outputPorts.hasNext()) {
1185                                    TypedIOPort outputP =
1186                                        (TypedIOPort)outputPorts.next();
1187                                    //Is there a direct path between the
1188                                    //two actors ?
1189                                    if (_existPathBetweenPorts(inputP,
1190                                            outputP)){
1191                                        //Insert the two ports in the two
1192                                        //vectors
1193                                        vectIn.add(inputP);
1194                                        vectOut.add(outputP);
1195                                        existPred = true;
1196                                    }
```

```
1197                                }
1198                            }
1199                        }
1200                    }
1201                    if(existPred){
1202                    //Set the relay in both of domains
1203                    try{
1204                        _indiceRelay++;
1205                        relayTx =(RelayTx) new
1206                                RelayTx(domainT, "relay"+
1207                                        _indiceRelay);
1208                    }
1209                    catch (Exception e){
1210                        System.out.println (" Set RelayTx failed");
1211                    }
1212                    try{
1213                        _indiceRelay++;
1214                        relayRx =(RelayRx) new
1215                            RelayRx(domainR,  "relay"+_indiceRelay);
1216                        htabRelayRelay.put(relayTx, relayRx);
1217                        htabRelayOriginal.put(relayRx, actorRx);
1218                    }
1219                    catch (Exception e){
1220                        System.out.println (" Set RelayRx failed");
1221                    }
1222                    //Clone the inputs ports of actorRx which are
1223                    //connected to actorInDomain and set them to
1224                    // RelayTx
1225                    for(int iR = 0 ; iR < vectIn.size() ; iR++){
1226                        try{
1227                            TypedIOPort portRx = (TypedIOPort)vectIn.
1228                                    elementAt(iR);
1229                            TypedIOPort portTx = (TypedIOPort)vectOut.
1230                                    elementAt(iR);
1231                            TypedIOPort portIn =
1232                                (TypedIOPort)portRx.clone();
1233                            TypedIOPort portOut =
1234                            (TypedIOPort)portTx.clone();
1235                            portIn.setContainer(relayTx);
1236                            portOut.setContainer(relayRx);
1237                            _replacePortsHicByPortsProjection
1238                                (portRx, portIn);
1239                            domainR.connect(portOut, portRx);
1240                        }
1241
1242
```

```
1243                        catch (Exception e){
1244                            System.out.println ("Replace port failed");
1245                        }
1246                    }
1247                }
1248            }
1249        }
1250    }
1251
1252    /* This method insert the repeaters between two successives HICs in
1253     * the begin of the partitionning stape
1254     */
1255    private void _insertRepeaters() throws IllegalActionException {
1256        _workspace.getReadAccess();
1257        container = (TypedCompositeActor)getContainer();
1258        //Hashtab (RelayTx - RelayRx)
1259        htabRelayRelay = new Hashtable();
1260        //Hashtab (RelayTx - Original)
1261        htabRelayOriginal = new Hashtable();
1262        Iterator model = container.deepEntityList().iterator();
1263        Vector vectPredOfActor = new Vector();
1264        //Insert the Repeater between two successives HICs
1265        Iterator nativeModel = container.deepEntityList().iterator();
1266        while(nativeModel.hasNext()){
1267            TypedAtomicActor actor =
1268                (TypedAtomicActor)nativeModel.next();
1269            if(actor instanceof HicActor){
1270                vectHics.addElement(actor);
1271                _checkConsecutiveHics(actor);
1272                //Sum of HICs ports in the system
1273                Iterator allPortsOfHics = actor.portList().iterator();
1274                while (allPortsOfHics.hasNext()) {
1275                    //empty extract for account of HICs ports
1276                    allPortsOfHics.next();
1277                    _totalPortsHics++;
1278                }
1279            }
1280        }
1281    }
1282
1283    /* Compute the product of two matrix
1284     */
1285    private int[][] _multiplyMatrix (int M1[][] , int M2[][]){
1286        int i , j , k;
1287        int produit[][];
1288
```

```
1289          produit = new int[M1.length][M2.length];
1290          for (i = 0 ; i < M1.length ; i++){
1291              for (j = 0 ; j < M2.length ; j++){
1292                  //Compute of (M1.M2)[i][j]
1293                  produit[i][j] = 0;
1294                  for (k = 0 ; k < M1[0].length ; k++){
1295                      produit[i][j] = produit[i][j] + M1[i][k]*M2[k][j];
1296                  }
1297              }
1298          }
1299          return produit;
1300      }
1301
1302      /* This method create the domains (Omegas) according to the ports
1303       * in the vector portsOfHics
1304       */
1305      private void _subSystemCreation(Vector portsOfHics){
1306          _indiceOmega++;
1307          //Checking for the process separating (SDF, DE from DT from CT)
1308          TypedIOPort port = (TypedIOPort)portsOfHics.firstElement();
1309          if ((port instanceof SDFIOPort) ||
1310                      (port instanceof DEIOPort)){
1311              //Create an Omega whithout Director
1312              try{
1313                  TypedCompositeActor domain = (TypedCompositeActor)
1314                  new TypedCompositeActor(container,
1315                          "omega"+_indiceOmega);
1316                  vectOmega.addElement(domain);
1317                  //if the port portForOmega of HIC is a SDFIOPort
1318                  if ((port instanceof SDFIOPort)){
1319                      try {
1320                          SDFDirector director = new SDFDirector(domain,
1321                                  "director"+_indiceOmega);
1322                          _setProjection(portsOfHics, +_indiceOmega);
1323                      }
1324                      catch (Exception e){
1325                      System.out.println ("subSystemCreation
1326                                                  SDF failed");
1327                      }
1328                  }
1329                  //if the port portForOmega of HIC is a DEIOPort
1330                  if (port instanceof DEIOPort){
1331                      try {
1332                          DEDirector director = new DEDirector(domain,
1333                                      "director"+_indiceOmega);
1334                          _setProjection(portsOfHics, +_indiceOmega);  }
```

```
1335                    catch (Exception e){
1336                        System.out.println ("subSystemCreation
1337                                                        DE failed");
1338                    }
1339                }
1340            }
1341            catch (Exception e){
1342                System.out.println ("subSystemCreation  failed");
1343            }
1344        }
1345
1346        //if the port portForOmega of HIC neither a SDFIOPort nor a
1347        //DEIOPort, then there is a generique port that uses CT
1348
1349        else {
1350            try {
1351                TypedCompositeActor domain = (TypedCompositeActor)
1352                new TypedCompositeActor (container, "omega"+_indiceOmega)
1353                vectOmega.addElement(domain);
1354                DTDirector director = new DTDirector(domain,
1355                                    "director"+_indiceOmega);
1356                _setProjection(portsOfHics, +_indiceOmega);
1357            }
1358            catch (Exception e){
1359                System.out.println ("subSystemCreation CT failed");
1360            }
1361            try{
1362            }
1363            catch (Exception e){
1364                System.out.println ("setOmega failed");
1365            }
1366        }
1367        return;
1368    }
1369
1370     /* This method gives the port of the first predecessor which is a
1371      * HIC
1372     */
1373    private Vector _portsOfFirstPredecessorHics(TypedAtomicActor actor){
1374        LinkedList predecessors = new LinkedList();
1375        Iterator inports = actor.inputPortList().iterator();
1376        while (inports.hasNext()) {
1377            boolean firstHic = false;
1378            TypedIOPort inPort = (TypedIOPort) inports.next();
1379            Iterator outPorts =
1380                inPort.deepConnectedOutPortList().iterator();
```

```
1381            while (outPorts.hasNext()) {
1382                TypedIOPort outPort = (TypedIOPort)outPorts.next();
1383                TypedAtomicActor pre =
1384                    (TypedAtomicActor)outPort.getContainer();
1385                if (pre instanceof HicActor){
1386                    //If there is no HIC from this port
1387                    if (!(vectPortInitial.contains(inPort))){
1388                        //Marks this port for to avoid to reuse it
1389                        vectPortInitial.addElement(inPort);
1390                        firstHic = true;
1391                        if (!vectOutPortsPredecessorsHics.
1392                                contains(outPort)) {
1393                            if(vectOutPortsPredecessorsHics.isEmpty()){
1394                                vectOutPortsPredecessorsHics.
1395                                        addElement(outPort);
1396                            }
1397                        else if(!(vectOutPortsPredecessorsHics.
1398                            isEmpty())){
1399                                boolean trouve = false;
1400                                int vIPH =
1401                                vectOutPortsPredecessorsHics.size();
1402                                for (int i = 0 ; i < vIPH ; i++){
1403                                    TypedIOPort port = (TypedIOPort)
1404                                    vectOutPortsPredecessorsHics.
1405                                    elementAt(i);
1406                                    if ((outPort.getContainer()) ==
1407                                        (port.getContainer())) {
1408                                        trouve = true;
1409                                    }
1410                                }
1411                                if(!(trouve)){
1412                                    vectOutPortsPredecessorsHics.
1413                                    addElement(outPort);
1414                                }
1415                            }
1416                        }
1417                    }
1418                }
1419                if(firstHic){
1420                    break;
1421                }
1422                else if(!(firstHic)){
1423                    _portsOfFirstPredecessorHics(pre);
1424                }
1425            }
1426        }
```

```
1427            vectPortInitial.removeAllElements();
1428            return vectOutPortsPredecessorsHics;
1429        }
1430
1431        /* This method gives the port of the first successor which is a
1432         * HIC
1433         */
1434        private Vector _portsOfFirstSuccessorHics(TypedAtomicActor actor){
1435            LinkedList successors = new LinkedList();
1436            Iterator outports = actor.outputPortList().iterator();
1437            while (outports.hasNext()) {
1438                boolean firstHic = false;
1439                TypedIOPort outPort = (TypedIOPort) outports.next();
1440                Iterator inPorts =
1441                outPort.deepConnectedInPortList().iterator();
1442                while (inPorts.hasNext()) {
1443                    TypedIOPort inPort = (TypedIOPort)inPorts.next();
1444                    TypedAtomicActor post =
1445                        (TypedAtomicActor)inPort.getContainer();
1446                    if (post instanceof HicActor){
1447                        //If there is no HIC from this port
1448                        if (!(vectPortInitial.contains(outPort))){
1449                            //Marks this port for to avoid to reuse it
1450                            vectPortInitial.addElement(outPort);
1451                            firstHic = true;
1452                            if (!(vectInPortsSuccessorsHics.
1453                                    contains(inPort))) {
1454                                if(vectInPortsSuccessorsHics.isEmpty()){
1455                                    vectInPortsSuccessorsHics.
1456                                        addElement(inPort);
1457                                }
1458                            else if(!(vectInPortsSuccessorsHics.
1459                                        isEmpty())){
1460                                    boolean trouve = false;
1461                                    int vISH =
1462                                    vectInPortsSuccessorsHics.size();
1463                                    for (int i = 0 ; i < vISH ; i++){
1464                                        TypedIOPort port =
1465                                            (TypedIOPort)
1466                                            vectInPortsSuccessorsHics.
1467                                                    elementAt(i);
1468                                        if ((inPort.getContainer()) ==
1469                                                (port.getContainer())){
1470                                            trouve = true;
1471                                        }
1472                                    }
```

```
1473                                    if(!(trouve)){
1474                                    vectInPortsSuccessorsHics.
1475                                    addElement(inPort);
1476                                }
1477                            }
1478                        }
1479                    }
1480                }
1481                if(firstHic){
1482                    break;
1483                }
1484                else if(!(firstHic)){
1485                    _portsOfFirstSuccessorHics(post);
1486                }
1487            }
1488        }
1489        vectPortInitial.removeAllElements();
1490        return vectInPortsSuccessorsHics;
1491    }
1492
1493    /* HDirector generates virtual dependencies between the different
1494     * projections of a HIC on the domain.
1495     * This is because some domains need this dependencies for to
1496     * build a schedule
1497     */
1498    private void _processVirtuelPorts(){
1499        Vector vectQueueProCompact = vectPro;
1500        for(int h1 = 0 ; h1 < vectOmega.size() ; h1++){
1501        TypedCompositeActor omega =
1502                (TypedCompositeActor)vectOmega.elementAt(h1);
1503        TypedIOPort outputPortTx = null;
1504        TypedIOPort outputPortRx = null;
1505        for (int h2 = 0 ; h2 < vectQueueProCompact.size() ; h2++){
1506            icActor hic =
1507                (HicActor)vectQueueProCompact.elementAt(h2);
1508                //If we are in this domain
1509                if(hic.getContainer() == omega){
1510                    //If this projection is a producer, it must have
1511                    //two virtuels ports (one input and one output)
1512                    //to ensure the dependancy whith an other
1513                    //projection of an other HIC
1514                try{
1515                    if(hic._getTypeProjection() == "Tx"){
1516                        try{
1517
1518
```

```
1519                        //Create first a port input and check
1520                        //if another projection precedes it,
1521                        //so has an output port. So connect
1522                        //this output to this input.
1523                        TypedIOPort inputPortTx =
1524                                new TypedIOPort(hic,
1525                                    "inputPortTx", true, false);
1526                        if(!(outputPortTx == null)){
1527                            omega.connect(outputPortTx,
1528                                inputPortTx);
1529                            //Set the same type
1530                            inputPortTx.
1531                                    setTypeSameAs(outputPortTx);
1532                        }
1533                        //Create its ouput port for to ensure
1534                        //the connection to another future
1535                        //projrction
1536                        outputPortTx = new TypedIOPort(hic,
1537                                    "outputPortTx", false, true);
1538                    }
1539                    catch (Exception e){
1540                        System.out.println
1541                            ("Creating virtuel portTx failed  "
1542                                    +hic);
1543                    }
1544                }
1545                //Else if this projection is a producer, it
1546                //must have two virtuel ports (input and
1547                //output) to ensure the dependancy whith an
1548                //other projection of an other HIC
1549
1550                else if(hic._getTypeProjection() == "Rx"){
1551                    try{
1552                        //On crée d'abord un port input et
1553                        //observe si une autre projection
1554                        //l'a précédé ; donc dispose d'un
1555                        //port output, ainsi on connecte cet
1556                        //output à cet input qui vient d'être
1557                        // crée
1558                        TypedIOPort inputPortRx =
1559                                new TypedIOPort(hic,
1560                        "inputPortRx", true, false);
1561                        if(!(outputPortRx == null)){
1562                            omega.connect(outputPortRx,
1563                                inputPortRx);
1564
```

```
1565                                //On calibre les types à
1566                                //l'identique
1567                                inputPortRx.setTypeSameAs
1568                                    (outputPortRx);
1569                            }
1570                            //Create first a port input and check
1571                            //if another projection precedes it, so
1572                            //has an output port. So connect this
1573                            //output to this input.
1574                            outputPortRx = new TypedIOPort(hic,
1575                                    "outputPortRx", false, true);
1576                            }
1577                            catch (Exception e){
1578                                System.out.println
1579                                    ("Creating virtuel portRx failed
1580                                            " +hic);
1581                            }
1582                        }
1583                    }
1584                    catch (Exception e){
1585                        System.out.println ("Creating virtuel port
1586                                                failed  ");
1587                    }
1588                }
1589            }
1590        }
1591    }
1592
1593    /* This method creates the projection in the domain indexed by
1594     * _indiceOmegaTransfer
1595     */
1596    private void _setProjection(Vector portsOfHics,
1597                int _indiceOmegaTransfer)
1598                        throws IllegalActionException {
1599        Iterator portOfHics = portsOfHics.iterator();
1600        while (portOfHics.hasNext()) {
1601            TypedIOPort firstPH = (TypedIOPort)portOfHics.next();
1602            HicActor hic = (HicActor)(firstPH.getContainer());
1603            _indiceProjection++;
1604            TypedCompositeActor domain = (TypedCompositeActor)
1605                 vectOmega.elementAt(_indiceOmegaTransfer);
1606            hic._generateProjection(domain,
1607                    "pro"+_indiceProjection, firstPH);
1608        }
1609    }
1610
```

```
1611      /* This method creates the projection in the domain if this
1612       * projection doesn't exist in this domain
1613       */
1614      private void _setSingleProjection(Vector portsOfHics,
1615                      TypedCompositeActor domain)
1616                           throws IllegalActionException {
1617          Iterator portOfHics = portsOfHics.iterator();
1618          while (portOfHics.hasNext()) {
1619              TypedIOPort firstPH = (TypedIOPort)portOfHics.next();
1620              HicActor hicForGenerate =
1621                           (HicActor)(firstPH.getContainer());
1622              //The projection is created only if it doen't exist in this
1623              //domain. So, compare their names
1624              boolean existProjection = false;
1625              Iterator actorsInOmega =
1626                            domain.deepEntityList().iterator();
1627              while (actorsInOmega.hasNext()){
1628                  TypedAtomicActor actorInOmega =
1629                  (TypedAtomicActor)actorsInOmega.next();
1630                  if(actorInOmega instanceof HicActor){
1631                      TypedAtomicActor original =
1632                          (TypedAtomicActor)htabProHic.
1633                               get(actorInOmega);
1634                      if((original.getName().toString()).
1635                          equals(hicForGenerate.getName().toString())){
1636                          existProjection = true;
1637                      }
1638                  }
1639              }
1640              if(!(existProjection)){
1641              _indiceProjection++;
1642              hicForGenerate._generateProjection(domain,
1643                                "pro"+_indiceProjection, firstPH);
1644              }
1645          }
1646      }
1647
1648       /* This method checks if there is any path from a given actor to
1649        * a given domain that goes through a HIC.
1650       */
1651      private boolean _testViaHic(TypedAtomicActor actor,
1652                  TypedCompositeActor omega) {
1653          boolean existPath = false;
1654          for (int indHic = 0 ; indHic < vectHics.size() ; indHic++){
1655
1656
```

```
1657            //The HIC must precede actor
1658            int rangHic =
1659            vTopoTriOfActors.indexOf(vectHics.elementAt(indHic));
1660            int rangActor = vTopoTriOfActors.indexOf(actor);
1661            if (rangHic < rangActor){
1662                //If there is a path between actor and the HIC
1663                TypedAtomicActor actorTempo =
1664                    (TypedAtomicActor)vectHics.elementAt(indHic);
1665                if (_existPathBetweenActors(actor, actorTempo)){
1666                    //Checks Omega's actors
1667                    Iterator actorsInOmega =
1668                        omega.deepEntityList().iterator();
1669                    while (actorsInOmega.hasNext()){
1670                        TypedAtomicActor actorInOmega =
1671                        (TypedAtomicActor)actorsInOmega.next();
1672                        if (!(actorInOmega instanceof HicActor)){
1673                            if (!(actorInOmega instanceof RelayTx)){
1674                                if (!(actorInOmega instanceof RelayRx)){
1675                                    //actorInOmega must precede the HIC
1676                                    int rangActorInO =
1677                                    vTopoTriOfActors.
1678                                            indexOf(actorInOmega);
1679                                    if (rangActorInO < rangHic){
1680                                        //Is there a path between a Hic
1681                                        //and actorInOmega ?
1682                                        if (_existPathBetweenActors
1683                                            (actorTempo, actorInOmega)){
1684                                            existPath = true;
1685                                        }
1686                                    }
1687                                }
1688                            }
1689                        }
1690                    }
1691                }
1692            }
1693        }
1694        return existPath;
1695    }
1696
1697     /* This method gives the predecessor of a given actor
1698      */
1699    private Vector _uniquePredecessorActors(Actor actor) {
1700        Iterator inports = actor.inputPortList().iterator();
1701        while (inports.hasNext()) {
1702            TypedIOPort inPort = (TypedIOPort) inports.next();
```

```
1703            Iterator outPorts =
1704            inPort.deepConnectedOutPortList().iterator();
1705            while (outPorts.hasNext()) {
1706                TypedIOPort outPort = (TypedIOPort)outPorts.next();
1707                Actor pre = (Actor)outPort.getContainer();
1708                if (!vectPredecessors.contains(pre)) {
1709                    vectPredecessors.addElement(pre);
1710                    _uniquePredecessorActors(pre);
1711                }
1712            }
1713        }
1714        return vectPredecessors;
1715    }
1716
1717     /* This method gives the successor of a given actor
1718     */
1719    private Vector _uniqueSuccessorActors(Actor actor) {
1720        Iterator outports = actor.outputPortList().iterator();
1721        while (outports.hasNext()) {
1722            TypedIOPort outPort = (TypedIOPort) outports.next();
1723            Iterator inPorts =
1724              outPort.deepConnectedInPortList().iterator();
1725            while (inPorts.hasNext()) {
1726                TypedIOPort inPort = (TypedIOPort)inPorts.next();
1727                Actor post = (Actor)inPort.getContainer();
1728                if (!vectSuccessors.contains(post)) {
1729                    vectSuccessors.addElement(post);
1730                    _uniqueSuccessorActors(post);
1731                }
1732            }
1733        }
1734        return vectSuccessors;
1735    }
1736
1737
1738    ////////////////////////////////////////////////////////////////
1739    ////                         public variables                   ////
1740
1741    public Hashtable htabProHic = null;
1742    public Hashtable htabRelayRelay = null;
1743    public Hashtable htabRelayOriginal = null;
1744
1745    public Vector vectPro = new Vector();
1746
1747
1748
```

```
1749    ////////////////////////////////////////////////////////////////
1750    ////                         protected variables                ////
1751
1752    protected Hashtable htabPiPa = null;
1753    protected Hashtable htabPaPi = null;
1754
1755    ////////////////////////////////////////////////////////////////
1756    ////                         private variables                  ////
1757
1758    private int _indiceOmega = -1;
1759    private int _indiceProjection = -1;
1760    private int _indiceRelay = -1;
1761    private int _indiceRepeater = -1;
1762    private int _iterationsCount = (0);
1763    private int _maxIterations;
1764    private int _totalPortsHics = 0;
1765
1766    private double _startTime;
1767    private double _stopTime;
1768
1769    private HicActor actorP;
1770    private RelayRx relayRx;
1771    private RelayTx relayTx;
1772
1773    private TypedCompositeActor container;
1774    private TypedCompositeActor lastDomain;
1775
1776    private int[] inputDeg;
1777    private int[] inputDegPro;
1778    private TypedIOPort[] tabInPorts;
1779    private TypedIOPort[] tabOutPorts;
1780    private int[][] actorAdjacentMatrix;
1781    private int[][] allPortsAdjacentMatrix;
1782    private int[][] portsAdjacentMatrix;
1783
1784    private Vector vectActors = new Vector();
1785    private Vector vectActorsPlaced = new Vector();
1786    private Vector vectDiferedActor = new Vector();
1787    private Vector vectFirstHics = new Vector();
1788    private Vector vectHics = new Vector();
1789    private Vector vectInPorts = new Vector();
1790    private Vector vectInPortsSuccessorsHics = new Vector();
1791    private Vector vectOmega = new Vector();
1792    private Vector vectOutPorts = new Vector();
1793    private Vector vectOutPortsPredecessorsHics = new Vector();
1794    private Vector vectPredActor = new Vector();
```

```
1795      private Vector vectPredecessors = new Vector();
1796      private Vector vectPortInitial = new Vector();
1797      private Vector vectSuccActor = new Vector();
1798      private Vector vectSuccessors = new Vector();
1799      private Vector vTopoTriOfActors = new Vector();
1800      private Vector vTopoTriOfProjections = new Vector();
1801
1802  }
1803
```

Interface HeterogeneousBehavior

```
1  /* Interface HeterogeneousBehavior
2   */
3
4  /*
5  @author : Mokhoo MBOBI
6  Ecole Supérieure d'Electricité (SUPELEC France)
7  Computer Sciences Department
8  Email : Mokhoo.MBOBI@supelec.fr
9  */
10
11
12 package ptolemy.domains.heterogeneous;
13
14 ////////////////////////////////////////////////////////
15 //// HeterogeneousBehavior
16
17 public interface HeterogeneousBehavior {
18
19     /** Computes the heterogeneous behavior between two MoCs
20      */
21     public void computeBehavior();
22
23 }
24
```

Code source de l'Applet qui lance la simulation

```
/* Applet which allow the simulation of the model
 * named Heterogeneous.class in Ptolemy II
 */

/*
@author : Mokhoo MBOBI
Ecole Supérieure d'Electricité (SUPELEC France)
Computer Sciences Department
Email : Mokhoo.MBOBI@supelec.fr
*/


<HTML>
<HEAD>
<META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=iso-8859-1">
<TITLE>Heterogeneous</TITLE>
</HEAD>
<BODY>
<H1>Heterogeneous</H1>
<APPLET
 code = "ptolemy/actor/gui/PtolemyApplet"
 codebase = "../../.."
 width = "800"
 height = "300"
 >
 <PARAM NAME = "modelClass" VALUE =
 "ptolemy.domains.heterogeneous.Heterogeneous" >
 <PARAM NAME = "controls" VALUE = "buttons, topParameters" >
 <PARAM NAME = "orientation" VALUE = "horizontal" >
No Java Plug-in support for applet, see
 <a
 href="http://java.sun.com/products/plugin/"><code>http://java.sun.com/pr
 oducts/plugin/</code></a>
</APPLET>
<p>This applet uses the <code>controls</code> and
<code>orientation</code> PtolemyApplet applet parameter.
<p>This applet will not work under Netscape 4.x.
</BODY>
</HTML>

```

Code source incluant différents modèles simulés

```
1  /* Since the graphical interface of Ptolemy II (Vergil) does
2  * not support flat heterogeneous systems yet, then, a
3  * heterogeneous system will be built by assembling actors using
4  * the Java API of Ptolemy II.
5   */
6
7
8   package ptolemy.domains.heterogeneous;
9
10 import ptolemy.actor.TypedCompositeActor;
11 import ptolemy.actor.gui.PtolemyApplet;
12 import ptolemy.actor.lib.*;
13 import ptolemy.actor.lib.conversions.*;
14 import ptolemy.actor.lib.gui.*;
15 import ptolemy.data.expr.Parameter;
16 import ptolemy.domains.de.kernel.DEDirector;
17 import ptolemy.kernel.util.IllegalActionException;
18 import ptolemy.kernel.util.NameDuplicationException;
19 import ptolemy.kernel.util.Workspace;
20 import ptolemy.domains.sdf.kernel.SDFDirector;
21 import ptolemy.domains.de.lib.*;
22 import ptolemy.data.type.BaseType;
23 import ptolemy.data.*;
24
25
26 /*
27 @author : Mokhoo MBOBI
28 Ecole Supérieure d'Electricité (SUPELEC France)
29 Computer Science Department
30 Email : Mokhoo.MBOBI@supelec.fr
31 */
32
33
34 public class Heterogeneous extends TypedCompositeActor {
35     public Heterogeneous(Workspace workspace)
36     throws IllegalActionException, NameDuplicationException {
37     super(workspace);
38
39     //Instanciation de HDirector
40     HDirector hdirector = new HDirector(this, "hdirector");
41     setDirector(hdirector);
42
43     //REDRESSEMENT D'UN SIGNAL SINUSOIDAL
44     //Instanciation des acteurs du système
45
46
```

```
47      TrigFunction sinus = new TrigFunction(this, "sinus");
48      Ramp pente = new Ramp(this, "pente");
49      SigDetector detector = new SigDetector(this, "detector");
50      Amplifier ampli = new Amplifier(this, "ampli");
51      TimedPlotter signalOriginal = new TimedPlotter(this,
52                                      "Signal Original");
53      TimedPlotter signalRedressé = new TimedPlotter(this,
54                                      "Signal Redressé");
55      TimedPlotter signalEvent = new TimedPlotter(this,
56                                      "Début Redressement");
57
58      //Connexion des acteurs
59      connect (pente.output, sinus.input);
60      connect (sinus.output, detector.sdfFromSin);
61      connect (detector.sdfToAmpli, ampli.sdfFronDet);
62      connect (detector.deToAmpli, ampli.control);
63      connect (ampli.dtToOriginalPlot, signalOriginal.input);
64      connect (ampli.dtToRedressePlot, signalRedressé.input);
65      connect (ampli.deToEventPlot, signalEvent.input);
66
67      /*
68      //REDRESSEMENT D'UN SIGNAL SINUSOIDAL BRUITE
69      //Instanciation des acteurs du système
70      AddSubtract add = new AddSubtract(this, "add");
71      TrigFunction sinus = new TrigFunction(this, "sinus");
72      Ramp pente = new Ramp(this, "pente");
73      Gaussian gs = new Gaussian(this, "gs");
74      SigDetector detector = new SigDetector(this, "detector");
75      Amplifier ampli = new Amplifier(this, "ampli");
76      TimedPlotter signalOriginal = new TimedPlotter(this,
77                                        "Signal Original");
78      TimedPlotter signalRedressé = new TimedPlotter(this,
79                                        "Signal Redressé");
80      TimedPlotter signalEvent = new TimedPlotter(this,
81                                        "Début Redressement");
82
83      //Connexion des acteurs
84      connect (pente.output, sinus.input);
85      connect (sinus.output, add.plus);
86      connect (gs.output, add.minus);
87      connect (add.output, detector.sdfFromSin);
88      connect (detector.sdfToAmpli, ampli.sdfFronDet);
89      connect (detector.deToAmpli, ampli.control);
90      connect (ampli.dtToOriginalPlot, signalOriginal.input);
91      connect (ampli.dtToRedressePlot, signalRedressé.input);
92      connect (ampli.deToEventPlot, signalEvent.input); */
```

```
93      //MODULATION D'AMPLITUDE
94      //Instanciation des acteurs du système
95      DiscreteRandomSource drs = new DiscreteRandomSource(this, "drs");
96      TrigFunction sinus = new TrigFunction(this, "sinus");
97      Ramp pente = new Ramp(this, "pente");
98      DigitalSigDetector detector = new DigitalSigDetector(this,
99                                                  "detector");
100     Multiplexer multiplex = new Multiplexer(this, "multiplex");
101     //TimedPlotter porteuse = new TimedPlotter(this, "Porteuse");
102     //TimedPlotter signalModulantNumerique =  new TimedPlotter(this,
103                                     "signal Modulant Numérique");
104     //TimedPlotter signalModule = new TimedPlotter(this,
105                                     "signal Modulé");
106     TimedPlotter porteuse = new TimedPlotter(this, "Carrier Signal");
107     TimedPlotter signalModulantNumerique =  new TimedPlotter(this,
108                                     "Digital Modulating Signal");
109     TimedPlotter signalModule = new TimedPlotter(this,
110                                     "Modulated Signal");
111
112     //Connexion des acteurs
113     connect (pente.output, sinus.input);
114     connect (sinus.output, detector.sdfFromSin);
115     connect (drs.output, detector.sdfFromRandom);
116     connect (detector.sdfToMultiplex, multiplex.sdfFronDet);
117     connect (detector.deToOff, multiplex.off);
118     connect (detector.deToOn, multiplex.on);
119     connect (multiplex.dtToOriginalPlot, porteuse.input);
120     connect (multiplex.dtToDigitalPlot, signalModulantNumerique.input);
121     connect (multiplex.dtToModulPlot, signalModule.input);
122
123
124     /*
125     //MODULATION DE PHASE
126     //Instanciation des acteurs du système
127     DiscreteRandomSource drs = new DiscreteRandomSource(this, "drs");
128     TrigFunction sinus = new TrigFunction(this, "sinus");
129     Ramp pente = new Ramp(this, "pente");
130     DigitalSigDetector detector = new DigitalSigDetector(this, "detector"
131     Multiplexer multiplex = new Multiplexer(this, "multiplex");
132     TimedPlotter porteuse = new TimedPlotter(this, "Porteuse");
133     TimedPlotter signalModulantNumerique =  new TimedPlotter(this, "signal
134     TimedPlotter signalModule = new TimedPlotter(this, "signal Modulé");
135
136
137
138
```

```
139     //Connexion des acteurs
140     connect (pente.output, sinus.input);
141     connect (sinus.output, detector.sdfFromSin);
142     connect (drs.output, detector.sdfFromRandom);
143     connect (detector.sdfToMultiplex, multiplex.sdfFronDet);
144     connect (detector.deToOff, multiplex.off);
145     connect (detector.deToOn, multiplex.on);
146     connect (multiplex.dtToOriginalPlot, porteuse.input);
147     connect (multiplex.dtToDigitalPlot, signalModulantNumerique.input);
148     connect (multiplex.dtToModulPlot, signalModule.input);
149     */
150
151 }
152
```

Code source du Détecteur dans la simulation du Redresseur

```
1  /* Détecteur de signal numérique
2   */
3  /*
4
5  @author : Mokhoo MBOBI
6  Ecole Supérieure d'Electricité (SUPELEC France)
7  Computer Sciences Department
8  Email : Mokhoo.MBOBI@supelec.fr
9  */
10
11
12 package ptolemy.domains.heterogeneous;
13
14 import ptolemy.actor.*;
15 import ptolemy.actor.lib.*;
16 import ptolemy.data.*;
17 import ptolemy.data.type.*;
18 import ptolemy.data.expr.Parameter;
19 import ptolemy.domains.ct.kernel.*;
20 import ptolemy.domains.de.kernel.*;
21 import ptolemy.domains.sdf.kernel.*;
22 import ptolemy.kernel.*;
23 import ptolemy.kernel.util.*;
24
25 import java.lang.Boolean;
26 import java.util.Iterator;
27 import java.util.Vector;
28
29
30 ////////////////////////////////////////////////////////////////
31 //// HicActor
32 /**
33 */
34 public class SigDetector extends HicActor
35     implements SequenceActor, TimedActor{
36
37     /** Construct an actor with the specified container and name.
38      *  @param container The composite actor to contain this one.
39      *  @param name The name of this actor.
40      *  @exception IllegalActionException If the entity cannot
41      *  be contained by the proposed container.
42      *  @exception NameDuplicationException If the container
43      *  already has an actor with this name.
44      */
45
46
```

```
47      public SigDetector(CompositeEntity container, String name)
48              throws NameDuplicationException, IllegalActionException  {
49          super(container, name);
50          sdfFromSin = new SDFIOPort(this, "sdfFromSin", true, false);
51          sdfToAmpli = new SDFIOPort(this, "sdfToAmpli", false, true);
52          deToAmpli = new DEIOPort(this, "deToAmpli", false, true);
53
54          sdfToAmpli.setTypeEquals(BaseType.DOUBLE);
55          deToAmpli.setTypeEquals(BaseType.DOUBLE);
56      }
57
58      ////////////////////////////////////////////////////////////////
59      ////                       ports and parameters                 ////
60      /**
61       */
62
63      public SDFIOPort sdfFromSin;
64
65      public SDFIOPort sdfToAmpli;
66      public DEIOPort deToAmpli;
67
68
69      ////////////////////////////////////////////////////////////////////
70      ////                          public methods                      ////
71
72      public void initialize() throws IllegalActionException {
73          container = (TypedCompositeActor)this.getContainer();
74          _director = container.getDirector();
75          if (_director instanceof HDirector){
76              _hdirector = (HDirector)_director;
77              sdfFromSin = (SDFIOPort)_hdirector.htabPiPa
78                              .get(sdfFromSin);
79              sdfToAmpli = (SDFIOPort)_hdirector.htabPaPi
80                              .get(sdfToAmpli);
81              deToAmpli = (DEIOPort)_hdirector.htabPaPi
82                              .get(deToAmpli);
83          }
84      }
85
86      public void  computeBehavior() {
87          try{
88              if((sdfFromSin.getContainer() == _runningProjection)
89                              && (sdfFromSin.hasToken(0))){
90                  _tokenFromSin = sdfFromSin.get(0);
91                  _valueOfTokenFromSin = (((DoubleToken)_tokenFromSin)
92                              .doubleValue());
```

```
93                  _nextTimeToFire(deToAmpli, 0);
94              }
95
96              if(sdfToAmpli.getContainer() == _runningProjection){
97                  sdfToAmpli.send(0, _tokenFromSin);
98              }
99
100             if(deToAmpli.getContainer() == _runningProjection){
101                 if(_valueOfTokenFromSin == 0){
102                 }
103                 else if(_valueOfTokenFromSin*lastSigneDet < 0){
104                     deToAmpli.send(0, new IntToken(lastSigneDet));
105                     lastSigneDet = -lastSigneDet;
106                 }
107             }
108         }
109         catch (Exception e){
110             System.out.println ("computedBehavior failed");
111         }
112     }
113
114     ////////////////////////////////////////////////////////////////
115     ////                         protected methods                  ////
116
117     protected void _generateProjection(TypedCompositeActor tdo,
118                 String namePro, TypedIOPort pfh)
119                             throws IllegalActionException {
120         try{
121             _avatar =(HicActor) new SigDetector(tdo, namePro);
122             _generateProjectionEnd(tdo, namePro, pfh, _avatar, this);
123         }
124         catch (Exception e){
125             System.out.println ("generateProjection SigDetector failed");
126         }
127     }
128
129     ////////////////////////////////////////////////////////////////
130     ////
131     ////                      private variables                    ////
132
133     private double _valueOfTokenFromSin;
134     private int lastSigneDet = 1;
135     private Token _tokenFromSin;
136     private TypedCompositeActor container;
137 }
```

Code source de l'Amplificateur dans la simulation du Redresseur

```
/* Amplifier
 */

/*
@author : Mokhoo MBOBI
Ecole Supérieure d'Electricité (SUPELEC France)
Computer Sciences Department
Email : Mokhoo.MBOBI@supelec.fr
*/

package ptolemy.domains.heterogeneous;

import ptolemy.actor.*;
import ptolemy.actor.lib.*;
import ptolemy.data.*;
import ptolemy.data.type.*;
import ptolemy.data.expr.Parameter;
import ptolemy.domains.ct.kernel.*;
import ptolemy.domains.de.kernel.*;
import ptolemy.domains.sdf.kernel.*;
import ptolemy.kernel.*;
import ptolemy.kernel.util.*;

import java.lang.Boolean;
import java.util.Iterator;
import java.util.Vector;


//////////////////////////////////////////////////////////////////
//// Amplifier
/**
*/
public class Amplifier extends HicActor
    implements SequenceActor, TimedActor{

    /** Construct an actor with the specified container and name.
     *  @param container The composite actor to contain this one.
     *  @param name The name of this actor.
     *  @exception IllegalActionException If the entity cannot
     *  be contained by the proposed container.
     *  @exception NameDuplicationException If the container already
     *  has an actor with this name.
     */
    public Amplifier(CompositeEntity container, String name)
        throws NameDuplicationException, IllegalActionException  {
        super(container, name);
```

```
47          sdfFronDet = new SDFIOPort(this, "sdfFronDet", true, false);
48          control = new DEIOPort(this, "control", true, false);
49
50          deToEventPlot = new DEIOPort(this, "deToEventPlot",
51                          false, true);
52          dtToOriginalPlot = new TypedIOPort(this, "dtToOriginalPlot",
53                          false, true);
54          dtToRedressePlot = new TypedIOPort(this, "dtToRedressePlot",
55                          false, true);
56
57          deToEventPlot.setTypeEquals(BaseType.DOUBLE);
58          dtToOriginalPlot.setTypeEquals(BaseType.DOUBLE);
59          dtToRedressePlot.setTypeEquals(BaseType.DOUBLE);
60      }
61
62      ////////////////////////////////////////////////////////////////
63      ////                       ports and parameters                 ////
64
65      public SDFIOPort sdfFronDet;
66      public DEIOPort control;
67
68      public DEIOPort deToEventPlot;
69      public TypedIOPort dtToOriginalPlot;
70      public TypedIOPort dtToRedressePlot;
71
72      ////////////////////////////////////////////////////////////////
73      ////                         public methods                     ////
74
75      public void initialize() throws IllegalActionException {
76          container = (TypedCompositeActor)this.getContainer();
77          _director = container.getDirector();
78          if (_director instanceof HDirector){
79              //Initialisation des ports du HIC
80              _hdirector = (HDirector)_director;
81              sdfFronDet = (SDFIOPort)_hdirector.htabPiPa.
82                              get(sdfFronDet);
83              control = (DEIOPort)_hdirector.htabPiPa.get(control);
84              deToEventPlot = (DEIOPort)_hdirector.
85                              htabPaPi.get(deToEventPlot);
86              dtToOriginalPlot = (TypedIOPort)_hdirector.
87                              htabPaPi.get(dtToOriginalPlot);
88              dtToRedressePlot = (TypedIOPort)_hdirector.
89                              htabPaPi.get(dtToRedressePlot);
90          }
91      }
92
```

```
93      //Behavior of Amplifier
94      public void computeBehavior() {
95          try{
96              if((sdfFronDet.getContainer() == _runningProjection) &&
97                                          (sdfFronDet.hasToken(0))){
98                  _sdfTokenFromDet = sdfFronDet.get(0);
99                  _valueOfsdfTokFromDet =
100                         (((DoubleToken)_sdfTokenFromDet)
101                         .doubleValue());
102             }
103             if((control.getContainer() == _runningProjection) &&
104                                         (control.hasToken(0))){
105
106                 ontrol.get(0);
107                 gain = -gain;
108                 _downCrossing = _downCrossing + 1;
109                 _nextTimeToFire(deToEventPlot, 0);
110             }
111
112             if(dtToOriginalPlot.getContainer() == _runningProjection){
113                     dtToOriginalPlot.send(0, _sdfTokenFromDet);
114             }
115
116             if(dtToRedressePlot.getContainer() == _runningProjection){
117                 sortie = gain*_valueOfsdfTokFromDet;
118                 dtToRedressePlot.send(0, new DoubleToken(sortie));
119             }
120
121             if(deToEventPlot.getContainer() == _runningProjection){
122                 //offset pour le signal sinusoŽidal
123                 _offset = +Math.PI*100*(1 + (2*_downCrossing));
124                 _period = (_offset - _base);
125                 deToEventPlot.send(0, new DoubleToken(1), _period);
126                 _base = _offset;
127             }
128         }
129         catch (Exception e){
130             System.out.println ("computedBehavior failed");
131         }
132     }
133
134
135
136
137
138
```

```
139     ////////////////////////////////////////////////////////////////
140     ////                         protected methods                  ////
141
142     protected void _generateProjection(TypedCompositeActor tdo,
143                    String namePro, TypedIOPort pfh)
144                                  throws IllegalActionException {
145         try{
146             _avatar = (HicActor) new Amplifier(tdo, namePro);
147             _generateProjectionEnd(tdo, namePro, pfh, _avatar, this);
148         }
149         catch (Exception e){
150             System.out.println ("generateProjectionDESDF failed");
151         }
152     }
153
154
155     ////////////////////////////////////////////////////////////////
156     ////
157     ////
158     ////              private variables                    ////
159
160     private int _downCrossing = -1;
161     private double gain = 1;
162     private double _offset = 0;
163     private double _base = 0;
164     private double _period;
165     private double sortie;
166     private double _valueOfsdfTokFromDet = 0;
167     private Token _sdfTokenFromDet;
168     private TypedCompositeActor container;
169 }
170
```

Code source du Détecteur dans la simulation du Modulateur

```
1  /* Détecteur de signal numérique
2   */
3  /*
4
5  @author : Mokhoo MBOBI
6  Ecole Supérieure d'Electricité (SUPELEC France)
7  Computer Sciences Department
8  Email : Mokhoo.MBOBI@supelec.fr
9  */
10
11 package ptolemy.domains.heterogeneous;
12
13 import ptolemy.actor.*;
14 import ptolemy.actor.lib.*;
15 import ptolemy.data.*;
16 import ptolemy.data.type.*;
17 import ptolemy.data.expr.Parameter;
18 import ptolemy.domains.ct.kernel.*;
19 import ptolemy.domains.de.kernel.*;
20 import ptolemy.domains.sdf.kernel.*;
21 import ptolemy.kernel.*;
22 import ptolemy.kernel.util.*;
23
24 import java.lang.Boolean;
25 import java.util.Iterator;
26 import java.util.Vector;
27
28 ////////////////////////////////////////////////////////////////
29 //// HicActor
30 /**
31 */
32 public class DigitalSigDetector extends HicActor
33     implements SequenceActor, TimedActor{
34
35     /** Construct an actor with the specified container and name.
36      *  @param container The composite actor to contain this one.
37      *  @param name The name of this actor.
38      *  @exception IllegalActionException If the entity cannot be
39      *   contained by the proposed container.
40      *  @exception NameDuplicationException If the container already
41      * has an actor with this name.
42      */
43     public DigitalSigDetector(CompositeEntity container, String name)
44             throws NameDuplicationException, IllegalActionException  {
45         super(container, name);
46         sdfFromSin = new SDFIOPort(this, "sdfFromSin", true, false);
```

```
47          sdfFromRandom = new SDFIOPort(this, "sdfFromRandom",
48                          true, false);
49          sdfToMultiplex = new SDFIOPort(this, "sdfToMultiplex",
50                          false, true);
51          deToOn = new DEIOPort(this, "deToOn", false, true);
52          deToOff = new DEIOPort(this, "deToOff", false, true);
53
54          sdfToMultiplex.setTypeEquals(BaseType.DOUBLE);
55          deToOn.setTypeEquals(BaseType.DOUBLE);
56          deToOff.setTypeEquals(BaseType.DOUBLE);
57      }
58
59      ////////////////////////////////////////////////////////////////
60      ////                       ports and parameters                 ////
61      /**
62       */
63
64      public SDFIOPort sdfFromSin;
65      public SDFIOPort sdfFromRandom;
66      public SDFIOPort sdfToMultiplex;
67      public DEIOPort deToOn;
68      public DEIOPort deToOff;
69
70
71      ////////////////////////////////////////////////////////////////
72      ////                         public methods                     ////
73
74      public void initialize() throws IllegalActionException {
75          container = (TypedCompositeActor)this.getContainer();
76          _director = container.getDirector();
77          if (_director instanceof HDirector){
78              _hdirector = (HDirector)_director;
79              sdfFromSin = (SDFIOPort)_hdirector.htabPiPa
80                              .get(sdfFromSin);
81              sdfFromRandom = (SDFIOPort)_hdirector.htabPiPa
82                              .get(sdfFromRandom);
83              sdfToMultiplex = (SDFIOPort)_hdirector.htabPaPi
84                             .get(sdfToMultiplex);
85              deToOn = (DEIOPort)_hdirector.htabPaPi.get(deToOn);
86              deToOff = (DEIOPort)_hdirector.htabPaPi.get(deToOff);
87          }
88      }
89
90
91
92
```

```
93      public void  computeBehavior() {
94          try{
95              if((sdfFromSin.getContainer() == _runningProjection)
96                                      && (sdfFromSin.hasToken(0))){
97                  _tokenFromSin = sdfFromSin.get(0);
98                  _valueOfTokenFromSin = (((DoubleToken)_tokenFromSin)
99                                      .doubleValue());
100                 _nextTimeToFire(deToOn, 0);
101
102                 _tokenFromRandom = sdfFromRandom.get(0);
103                 _valueOfTokenFromRandom =
104                         (((IntToken)_tokenFromRandom).intValue());
105             }
106
107             if(sdfToMultiplex.getContainer() == _runningProjection){
108                 sdfToMultiplex.send(0, _tokenFromSin);
109             }
110
111             if(deToOn.getContainer() == _runningProjection){
112                 if(_valueOfTokenFromSin == 0){
113                 }
114                 //Détection du passage au positif
115                 else if((lastSigneDet < 0)
116                             && (_valueOfTokenFromSin > 0)){
117                     if(_valueOfTokenFromRandom == 0){
118                         deToOff.send(0, new IntToken(0));
119                     }
120                     else if(_valueOfTokenFromRandom == 1){
121                         deToOn.send(0, new IntToken(1));
122                     }
123                 }
124                 lastSigneDet = _valueOfTokenFromSin;
125             }
126         }
127         catch (Exception e){
128             System.out.println ("computedBehavior failed");
129         }
130     }
131
132
133     ////////////////////////////////////////////////////////////
134     ////                       protected methods                ////
135
136      protected void _generateProjection(TypedCompositeActor tdo,
137                  String namePro, TypedIOPort pfh)
138                             throws IllegalActionException {
```

```
139         try{
140             _avatar =(HicActor) new DigitalSigDetector(tdo, namePro);
141             _generateProjectionEnd(tdo, namePro, pfh, _avatar, this);
142         }
143         catch (Exception e){
144             System.out.println ("generateProjection Detector failed");
145         }
146     }
147 
148     ////////////////////////////////////////////////////////////////
149     ////
150     ////                     private variables                  ////
151 
152     private double _valueOfTokenFromSin;
153     private int _valueOfTokenFromRandom;
154     private double lastSigneDet;
155     private Token _tokenFromSin;
156     private Token _tokenFromRandom;
157     private TypedCompositeActor container;
158 }
159 
```

Code source de l'Ampli-Multiplexeur dans la simulation du Modulateur

```
1  /* Amplificateur-Multiplexeur
2   */
3  /*
4
5  @author : Mokhoo MBOBI
6  Ecole Supérieure d'Electricité (SUPELEC France)
7  Computer Sciences Department
8  Email : Mokhoo.MBOBI@supelec.fr
9  */
10
11 package ptolemy.domains.heterogeneous;
12
13 import ptolemy.actor.*;
14 import ptolemy.actor.lib.*;
15 import ptolemy.data.*;
16 import ptolemy.data.type.*;
17 import ptolemy.data.expr.Parameter;
18 import ptolemy.domains.ct.kernel.*;
19 import ptolemy.domains.de.kernel.*;
20 import ptolemy.domains.sdf.kernel.*;
21 import ptolemy.kernel.*;
22 import ptolemy.kernel.util.*;
23
24 import java.lang.Boolean;
25 import java.util.Iterator;
26 import java.util.Vector;
27
28 //////////////////////////////////////////////////////////////////
29 //// Multiplexer
30 /**
31 */
32 public class Multiplexer extends HicActor
33     implements SequenceActor, TimedActor{
34
35     /** Construct an actor with the specified container and name.
36      *  @param container The composite actor to contain this one.
37      *  @param name The name of this actor.
38      *  @exception IllegalActionException If the entity cannot
39      *  be contained by the proposed container.
40      *  @exception NameDuplicationException If the container already
41      *  has an actor with this name.
42      */
43     public Multiplexer(CompositeEntity container, String name)
44             throws NameDuplicationException, IllegalActionException  {
45         super(container, name);
46         sdfFronDet = new SDFIOPort(this, "sdfFronDet", true, false);
```

```
47          on = new DEIOPort(this, "on", true, false);
48          off = new DEIOPort(this, "off", true, false);
49
50          dtToOriginalPlot = new TypedIOPort(this, "dtToOriginalPlot",
51                          false, true);
52          dtToDigitalPlot = new TypedIOPort(this, "dtToDigitalPlot",
53                          false, true);
54          dtToModulPlot = new TypedIOPort(this, "dtToModulPlot",
55                          false, true);
56
57      dtToOriginalPlot.setTypeEquals(BaseType.DOUBLE);
58      dtToModulPlot.setTypeEquals(BaseType.DOUBLE);
59      dtToDigitalPlot.setTypeEquals(BaseType.DOUBLE);
60          }
61
62      ////////////////////////////////////////////////////////////////
63      ////                       ports and parameters                 ////
64      /** The input port is a SDFIOPort and the output is the DEIOPort.
65       */
66
67      public SDFIOPort sdfFronDet;
68      public DEIOPort off;
69      public DEIOPort on;
70
71      public TypedIOPort dtToOriginalPlot;
72      public TypedIOPort dtToModulPlot;
73      public TypedIOPort dtToDigitalPlot;
74
75
76      ////////////////////////////////////////////////////////////////
77      ////                         public methods                   ////
78
79      public void initialize() throws IllegalActionException {
80          container = (TypedCompositeActor)this.getContainer();
81          _director = container.getDirector();
82          if (_director instanceof HDirector){
83              //Initialisation des ports du HIC
84              _hdirector = (HDirector)_director;
85              sdfFronDet = (SDFIOPort)_hdirector.htabPiPa
86                              .get(sdfFronDet);
87              off = (DEIOPort)_hdirector.htabPiPa.get(off);
88              on = (DEIOPort)_hdirector.htabPiPa.get(on);
89              dtToOriginalPlot = (TypedIOPort)_hdirector.
90                              htabPaPi.get(dtToOriginalPlot);
91              dtToModulPlot = (TypedIOPort)_hdirector.
92                              htabPaPi.get(dtToModulPlot);
```

```
93              dtToDigitalPlot = (TypedIOPort)_hdirector.
94                              htabPaPi.get(dtToDigitalPlot);
95          }
96      }
97
98      public void computeBehavior() {
99          try{
100             if((sdfFronDet.getContainer() == _runningProjection) &&
101                                     (sdfFronDet.hasToken(0))){
102                 _sdfTokenFromDet = sdfFronDet.get(0);
103                 _valueOfsdfTokFromDet =
104                         (((DoubleToken)_sdfTokenFromDet)
105                                         .doubleValue());
106             }
107             if((on.getContainer() == _runningProjection)){
108                 if(on.hasToken(0)){
109                     on.get(0);
110                     _amplitude = 1;
111                 }
112                 else if(off.hasToken(0)){
113                     off.get(0);
114                     _amplitude = 0.25;
115                     //Pour la modulation de phase on met
116                     //l'amplitude à -1
117                     //_amplitude = -1;
118                 }
119             }
120
121             if(dtToOriginalPlot.getContainer() == _runningProjection){
122                 dtToOriginalPlot.send(0, _sdfTokenFromDet);
123             }
124
125             if(dtToDigitalPlot.getContainer() == _runningProjection){
126                 if (_amplitude == 1){
127                     dtToDigitalPlot.send(0, new DoubleToken(1));
128                 }
129                 else if (_amplitude < 1){
130                     dtToDigitalPlot.send(0, new DoubleToken(0));
131                 }
132             }
133
134             if(dtToModulPlot.getContainer() == _runningProjection){
135                 sortie = _amplitude*_valueOfsdfTokFromDet;
136                 dtToModulPlot.send(0, new DoubleToken(sortie));
137             }
138         }
```

```
139         catch (Exception e){
140             System.out.println ("computedBehavior failed");
141         }
142     }
143 
144 
145     ////////////////////////////////////////////////////////////////
146     ////                         protected methods                ////
147 
148 
149     protected void _generateProjection(TypedCompositeActor tdo,
150                     String namePro, TypedIOPort pfh)
151                                     throws IllegalActionException {
152         try{
153             _avatar = (HicActor) new Multiplexer(tdo, namePro);
154             _generateProjectionEnd(tdo, namePro, pfh, _avatar, this);
155         }
156         catch (Exception e){
157             System.out.println ("generateProjection Multiplexer failed");
158         }
159     }
160 
161 
162     ////////////////////////////////////////////////////////////////
163     ////
164     ////
165     ////              private variables                    ////
166 
167     private double _amplitude = 1;
168     private double sortie;
169     private double _valueOfsdfTokFromDet = 0;
170     private Token _sdfTokenFromDet;
171     private TypedCompositeActor container;
172 }
173 
```

Bibliographie

[1] G. Agha, *« Abstracting Interaction Patterns : A Programming Paradigm for Open Distributed Systems »*, in Formal Methods for Open Object-based Distributed Systems, IFIP Transactions, E. Najm and J.-B. Stefani, Eds., Chapman and Hall, 1997.

[2] G. Agha, S. Frolund, W. Kim, R. Panwar, A. Patterson, and D. Sturman, *« Abstraction and modularity mechanisms for concurrent computing »*, IEEE Parallel and Distributed Technology : Systems and Applications, 1(2) :3-14, May 1993.

[3] G. Agha, I. A. Mason, S. F.Smith, and C. L. Talcott, *« A foundation for actor computation »*, Journal of Functional Programming, 7(1) :1-72, 1997.

[4] P. Alexander, C. Kong, D. Barton, P. Ashenden et C. Menon, *« Rosetta, Strawman Version 0.1 »*, September 2002.

[5] L. de Alfaro and T. A. Henzinger, *« Interface Theories for Component-based Design »*, in the Proceedings of the First International Workshop on Embedded Software, EMSOFT, 2001.

[6] R. Alur, T. Dang, J. Esposito, R. Fierro, Y. Hur, F. Ivancic, V. Kumar, I. Lee, P. Mishra, G. Pappas, and O. Sokolsky, *« Hierarchical Hybrid Modeling of Embedded Systems »*, University of Pennsylvania, Availlable on line at http ://www.seas.upenn.edu/hybrid/

[7] R. Alur, T.A Henzinger, G. Lafferriere and G.J. Pappas *« Discrete Abstractions of Hybrid Systems »*, Proceedings of the IEEE, Vol. 88, N°. 7, July 2000.

[8] M. Auguin, *« Co-Conception de Systèmes Spécialisés sur Composant »*, Ecole Thématique sur les Systèmes Enfouis, I3S, Université de Nice Sophia Antipolis - CNRS, novembre 2000, Available on-line at http ://www.ens-lyon.fr/LIP/Ecoles-Archi/auguin.pdf

[9] M. Auguin, *« Systèmes sur Puce : vers l'exploration en milieu complexe »*, Ecole Thématique sur les Systèmes Enfouis, I3S, Université de Nice Sophia Antipolis, CNRS, INRIA, novembre 2003, Available on-line at http ://www-sop.inria.fr/intech/embarques/I3S_overview.pdf

[10] F. Balarian and al., *« Hardware/Software codesign of emeded system, The Polis approach »*, Kluwer Academic publishers, 1997.

[11] A. Benveniste and G. Berry, *« The synchronous approach to reactive and real-time systems »*, Proceedings of the IEEE, 79(9), pages 1270-1280, 1991.

[12] P.G. Basset, *« The theory of practice of adaptive reusse »*, In SSR, page 2-9, 1997.

[13] S.S. Bhattacharyya, E. Cheong, J. Davis II, M. Goel, Bart Kienhuis, C. Hylands, E.A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, B. Vogel, W. Williams, Y. Xiong, Yang Zhao, H. Zheng, *« heterogeneous concurrent Modeling and design in java, Volume I : Introduction to Ptolemy I »*, Memorandum UCB/ERL M01/12 Department of Electrical Engineering and Computer Sciences University of California at Berkeley, March 15, 2001, Available on-line at Availlale on line at http ://ptolemy.eecs.berkeley.edu/papers/

[14] S.S. Bhattacharyya, E. Cheong, J. Davis II, M. Goel, Bart Kienhuis, C. Hylands, E.A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, B. Vogel, W. Williams, Y. Xiong, Yang Zhao, H. Zheng, *« heterogeneous concurrent Modeling and design in java, Volume II : Software Architechture »*, Memorandum UCB/ERL M01/12 Department of Electrical Engineering and Computer Sciences University of California at Berkeley, March 15, 2001, Available on-line at Availlale on line at http ://ptolemy.eecs.berkeley.edu/papers/

[15] S.S. Bhattacharyya, E. Cheong, J. Davis II, M. Goel, Bart Kienhuis, C. Hylands, E.A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, B. Vogel, W. Williams, Y. Xiong, Yang Zhao, H. Zheng, *« heterogeneous concurrent Modeling and design in java, Volume III : Domains »*, Memorandum UCB/ERL M01/12 Department of Electrical Engineering and Computer Sciences University of California at Berkeley, March 15, 2001, Available on-line at Availlale on line at http ://ptolemy.eecs.berkeley.edu/papers/

[16] G. Berry et L. Cosserat, *« The Esterel synchronous programming language and its mathematical semantics, Language for synthesys »*, Ecole Nationale Supérieure des Mines de Paris, 1984.

[17] G. Berry, *« Hardware implemantation of pure Esterel »*, Proceedings of the ACM Workshop on formal methods in VLSI Design, 1991.

[18] G. Berry, *« The fundation of Esterel »*, MIT Press, In G. PlotKin, C. Stirling and M. Tofle, editors, Proof, Language and Interactions : Essays in Honour of Robin Milner edition, 1998.

[19] G. Berry, *« The Esterel v5 Language Primer »*, Version 5.21, Release 2.0, Centre de Mathématiques Appliquées, Ecole Nationale Supérieure des Mines de PAris, Avril, 1999.

[20] G. Berry and G. Gonthier *« The Esterel Synchronous Programming Language : Design, Semantics, Implementation »*, Science of Computer Programming, 19(2), pages 87-152, 1992.

[21] W. Boggs and M. Boggs, *« Mastering UML with Rational Rose »*, Sybex, 2002.

[22] F. Boniol, *« Etude d'une sémantique de la réactivité : Variation autour du modèle synchrone et applications aux systèmes embarqués »*, Ph.D. Thesis Ecole Nationale Supérieure de l'Aéronautique et de l'espace, Decembre, 1997.

[23] F. Boniol, *« Une approche synchrone multi-formalismes pour la conception de systèmes temps-réel distribués »*, Technique et science informatiques, Hermès, volume 17, n°9/1998, pages 1099-1128.

[24] G. Booch, J. Rumbaugh, I. Jacobson, *« The Unified Modeling Language User Guide »* Addison Wesley Professional, ISBN : 0-201-57168-4, 1999.

[25] G. Booch, I. Jacobson and J. Rumbaugh, *« The Unified Software Development Process »*, Addison-Wesley Pub Co, 1999.

[26] F. Boulanger, *« Intégration de modules synchrones dans un langage à Objets »*, Ph.D. Thesis Université Paris XI, Decembre, 1994.

[27] F. Boulanger, M. Mbobi and M. Feredj, *« Flat Heterogeneous Modeling »*, to appear in the Proceedings of the conference of Internet Processing Systems Interdisciplinaries, Venice, Italy, November 10 to 15, 2004

[28] F. Boussinot et R. De Simone, *« The Esterel language »*, Proceedings of the IEEE, 79(9), 1991.

[29] J. Buck and R. Vaidyanathan. *« Heterogenous modeling and simulation of embedded systems in el Greco »*, Proceedings of the 8th international workshop on Hardware/software codesign, San Diego, California, USA, Pages : 142-146, 2000, ISBN :1-58113-268-9.

[30] W-T. Chang, S. Ha, and E.A. Lee, *« Heterogenous simulation - mixing discrete-event models with dataflow »*, Journal of VLSI Signal Processing 15, 127-144, Kluwer Academic Publishers, 1997.

[31] J. Buck, S. Ha, E.A Lee, D.G Messerschmitt. *« Ptolemy : A Framework for Simulating and Prototyping Heterogeneous Systems »*, Invited paper in the International Journal of Computer Simulation, special issue on Simulation Software Development, August 31, 1992.

[32] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic And R. De Simone, *« The Synchronous Languages 12 Years Later »*, Proceedings of the IEEE , Volume : 91 Issue : 1 , Jan 2003.

[33] J. Chevalier, O. Benny, M. Rondonneau, G. Bois, E. M. Aboulhamid, F.-R. Boyer, SPACE : Electrical Engineering Department, Ecole Polytechnique de Montréal, Québec, Canada, *« A Hardware/Software SystemC modeling platform including an RTOS »*, Forum on specification and Design Languages (FDL), Frankfurt, Germany, 2003.

[34] E. Christen, K. Bakalar, E. Moser, *« Analog and Mixed-Signal Modeling using the VHDL-AMS Language »*, 36th Design Automation Conference, New Orleans, June 1999.

[35] P. Coste, F. Hessel, P. Le Marrec, Z. Sugar, M. Romdhani, R. Suescun, N. Zergainoh, A.A. Jerraya, *« Multilanguage design of Heterogeneous system »*, International Conference on Hardware Software Codesign Proceedings of the seventh international workshop on Hardware/software codesign Rome, Italy, 1999, Pages : 54-58, ISBN :1-58113-132-1.

[36] Coware N2C Homepage, Coware Inc., 2004, Availlable on line at http ://www.coware.com.

[37] J.M. Daveau, *« Spécification système et synthèse de la communication pour le Co-Design Logiciel/Matériel »*, Ph.D. Thesis INPG, TIMA Laboratory, Decembre, 1997.

[38] A. Dasdan, D. Ramanathan and R.K. Gupta *« A timing driven design and validation methodology for embedded real time systems »*, ACM Transactions of Design Automation of Electronic Systems, Vol. 3(4), October, 1998.

[39] J.M. Daveau et al., *« Protocol selection and interface generation for Sw-Hw codesign »*, IEEE Trans. On VLSI Systems, Vol. 5, no. 1, pg. 136-144, 1997.

[40] Dome « Guide, Honeywell Inc., 2000, Availlable on line at http ://www.htc.honeywell.com/dome/.

[41] Doulos, « System C Golden Reference Guide, Fevrier 2002.

[42] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, Xiaojun Liu, Jozsef Ludvigy, Stephen Neuendorffer, Sonia Sachs, Yuhong Xiong, *« Taming Heterogeneity-the Ptolemy Approach »*, EECS Department, University of California at Berkeley, Berkeley, CA, in Proceedings of the IEEE, 2003.

[43] H. Elmqvist, F. E. Cellier, and M. Otter, *« Object-oriented modeling of hybrid systems »*, in Proceedings of Eur. Simulation Symp., 1993, pp. 31–41.

[44] H. Elmqvist, S. E. Mattsson, and M. Otter,, *« Modelica—The new object-oriented modeling langugae, »*, in Proceedings of 12th Eur. Simulation Multiconf., 1998, pp. 127–131.

[45] R. Esser, J.W. Janneck, *« Moses - A tool suite for visual modeling of discrete-event systems »*, Proceedings of Symposium on Visual/Multimedia Approaches to Programming and Software Engineering, Stresa, Italy, September, 2001.

[46] M Feredj. F. Boulanger, Mbobi, *« An Approach for Domain-Polymorph Components Design »*, to appear in the Proceedings of The 2004 IEEE International Conference on Information Reuse and Integration (IEEE-IRI 2004), November 8-10, 2004, Las Vegas, Nevada, USA, IEEE Catalog Number 04EX974, ISBN 2004113902, pages 145 à 150

[47] D.D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, S. Zhu., *« SpecC : Specification Language and Methodology »*, Kluwer Academic Publishers, Dordrecht, Hardbound, ISBN 0-7923-7822-9, March 2000.

[48] A. Ghosh, J. Kunkel, S. Liao, *« Hardware Synthesis from C/C++ »*, Proceedings of the DATE99 conference, pp.382-383, March, 1999.

[49] A. Girault, B. Lee, and E. A. Lee, Fellow, IEEE, *« Hierarchical Finite State Machines with Multiple Concurrency Models »*, Proceedings of the DATE99 conference, pp.382-383, March99.

[50] Glass, *« Real-time : the lost world of software debugging and testing »*, Communication of the ACM, Mai 1980.

[51] Goel, *« Process Network in Ptolemy II »*, Technical Memorandum UCB/ERL M98/69, EECS Department, University of California at Berkeley, December 16, 1998.

[52] M. Gondran, M. Minoux. *« Graphes et algorithmes »* Eyrolles, Paris, 1979.

[53] N. Halbwachs and F. Lagnier and C. Ratel *« Programming and Verifying Real-Time Systems by Means of the Synchronous Data-Flow Language LUSTRE »*, IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems, 18(9), September, 1992.

[54] N.Halbwachs, P.Caspi, P.Raymond and D.Pilaud. *« The synchronous dataflow programming language Lustre »*, Proceedings of the IEEE, 79(9), September 1991.

[55] R. Hamouche, *« Modélisation des systèmes embarqués à base de composants et d'aspects »*, Ph.D. Thesis, Laboratoire des Méthodes Informatiques, Université d'Evry Val d'essone, Juin, 2004.

[56] D. Harel, *« Statecharts a Visual Formalism for Complex Systems »*, Department of Applied Mathematics, The Weizmann Institute of Science, Rehovot, Israell, in Science for computer programming 8, pg. 231-274, North-Holland, 1987.

[57] D. Harel, A. Naamad *« The Statemate semantics of Statecharts »*, ACM Transactions on Software Engineering and Methodology, Vol 5, N°4, October 1996, Pages 283-333.

[58] C. Hewitt, *« Viewing control structures as patterns of passing messages »*, Journal of Artificial Intelligence, 8(3) :323-363, June 1977.

[59] C. Hoare, *« Communicating Sequential Processes »*, Prentice Hall, 1985.

[60] IEEE DASC 1076.1 Working Group, *« VHDL-A Design Objective Document »*, version 2.3, http ://www.vhdl.org/analog/ftp_files/requirements/DOD_v2.3.txt

[61] A.A Jerraya et al, *« Multilanguage specification for system design and codesign »*, Availlable on line at http ://tima.imag.fr/publications/files/rr/mss_54.pdf

[62] K. Keutzer, S. Malik, A.R. Newton, J.M. Rabaey and A. Sangiovanni-Vincentelli., *« System-Level Design : Orthogonalization of Concerns and Platform-Based Design »*, IEEE transactions on computer aided design of integrated circuits and systems, VOL.19,NO.12,December 2000, Pages 1507-1522.

[63] L. Lagadec, *« Abstraction, modélisation et outils de CAO pour les architectures réconfigurables »*, Ph.D. Thesis, Université Renne 1, Décembre 2000.

[64] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi, *« The Generic Modeling Environment »*, Proceedings of IEEE International Workshop on Intelligent Signal Processing (WISP2001), May, 2001, Budapest, Hungary.

[65] B. Lee and E.A. Lee, *« Hierarchical Concurrent Finite State Machines in Ptolemy »*, University of California at Berkeley, Proceeding of International Conference on Application of Concurrency to System Design, p. 34-40, Fukushima, Japan, March 1998.

[66] B. Lee, *« Specification and Design of Reactive Systems, »*, Ph.D. Thesis, Memorandum UCB/ERL M00/29, Electronics Research Laboratory, University of California, Berkeley, May, 2000.

[67] B. Lee and E. A. Lee, *« Interaction of Finite State Machines and Concurrency Models »*, University of California at Berkeley, Proceeding of Thirty Second Annual Asilomar Conference on Signals, Systems, and Computers, Pacific Grove, California, November 1998.

[68] E.A. Lee, *« Computing for Embedded Systems »*, IEEE Instrumentation and Measurement, Technology Conference, Budapest, Hungary, May 21-23, 2001.

[69] E.A. Lee, S. Neuendorffer, M.J. Wirthlin, *« Actor-oriented design of embedded hardware and software systems »*, Invited paper, Journal of Circuits, Systems, and Computers, Version 2, November 20, 2002.

[70] E.A. Lee and Y. Xiong, *« System-Level Types for Component-Based Design »*, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, First Workshop on Embedded Software, EMSOFT2001, Lake Tahoe, CA, USA, Oct. 8-10, 2001.

[71] E.A. Lee and Y. Xiong, *« A Behavioral Type System and Its Application in Ptolemy II »*, To appear in Formal Aspects of Computing Journal, special issue on Semantic Foundations of Engineering Design Languages, November 10, 2003. Availlale on line at http ://ptolemy.eecs.berkeley.edu/papers/

[72] E.A. Lee and A. Sangiovanni-Vincentelli, *« A Framework for Comparing Models of Computation »*, IEEE Transactions on computer-aided design of integrated circuits and systems, Vol. 17, no. 12, December 1998.

[73] E.A Lee and S. Neuendorffer, *« Classes and Subclasses in Actor-Oriented Design »*, Invited paper, Conference on formal methods and models for codesign, MEMOCODE, San Diego, California, June 22-25, 2004. Availlale on line at http ://ptolemy.eecs.berkeley.edu/papers/

[74] E.A. Lee, *« Model-Driven Development - From Object-Oriented Design to Actor-Oriented Design »*, Extended abstract of an invited presentation at Workshop on Software Engineering for Embedded Systems : From Requirements to Implementation (a.k.a. The Monterey Workshop) Chicago, Sept. 24, 2003.

[75] E.A. Lee and S. Neuendorffer, *« Actor-oriented models for codesign »*, Balancing Re-Use and Performance, To appear in : Formal methods and models for system, Chapter 2, Kluwer 2004.

[76] E. A. Lee and T. M. Parks, *« Dataflow Process Networks »*, Proceedings of the IEEE, vol. 83, no. 5, pp. 773-801, May, 1995

[77] E. A. Lee and D. G. Messerschmitt, *« Synchronous Data Flow »*, Proceedings of the IEEE, vol. 75, no. 9, September, 1987.

[78] E. A. Lee and D. G. Messerschmitt, *« Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing »*, IEEE Trans. on Computers, January, 1987

[79] E.A. Lee and A. Sangiovanni-Vincentelli, *« Comparing Models of Computation »*, Proceedings of ICCAD, pages 10-14, November 1996.

[80] J. Liu, J. Eker, X. Liu, J. Reekie, E.A. Lee, *« Actor-Oriented Control System Design : A Responsible Framework Perspective »*, IEEE Transaction on Control System Technology, special issue on Computer Automated Multi-Paradigm Modeling. March 2003, Availlale on line at http ://ptolemy.eecs.berkeley.edu/papers/

[81] J. Liu, *« Responsible Frameworks for Heterogeneous Modeling and Design of Embedded Systems »*, Ph.D. thesis, Technical Memorandum UCB/ERL M01/41 Electronics Research Laboratory, University of California, Berkeley, December 20th, 2001.

[82] J. Liu and E. A. Lee, *« A Component-Based Approach to Modeling and Simulating Mixed-Signal and Hybrid Systems »*, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720, ACM Trans. on Modeling and Computer Simulation special issue on computer automated multi-paradigm modeling, Volume 12, Issue 4, pp. 343-368, October 2002.

[83] J. Liu, X. Liu, T-K J. Koo, B. Sinopoli, S. Sastry and E.A. Lee, *« A Hierarchical Hybrid System Model and Its Simulation »*, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720 Availlale on line at http ://ptolemy.eecs.berkeley.edu/papers/

[84] J. Liu and E. A. Lee, *« Component-Based Hierarchical Modeling of Systems with Continuous and Discrete Dynamics »*, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720 Proceedings of the 2000 IEEE International Symposium on the Computer-Aided Control Design, Anchorage, Alaska, USA, September 25-27, 2000, pp 95-100.

[85] X. Liu, J. Liu, J. Eker, E.A. Lee *« Heterogeneous Modeling and Design of Control Systems »*, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720 Software-Enable Control : Information technology for Dynamical Systems Tariq Samad and Garry Balas (eds.), Wiley-IEEE Press, April 2003.

[86] J. Liu, S. Jefferson and E.A. Lee *« Motivating Hierarchical Run-Time Models in Measurement and Control Systems »*, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720, Proceedings of the American Control Conference Arlington, VA June 25-27, 2001.

[87] J. Liu, X. Liu and E.A. Lee *« Modeling Distributed Hybrid Systems in Ptolemy II »*, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720, Proceedings of the American Control Conference Arlington, VA June 25-27, 2001.

[88] J. Liu and E.A. Lee, *« On the Causality of Mixed-Signal and Hybrid Models »*, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720, 6th International Workshop on Hybrid Systems, Computation and Control, (HSCC '03), April 3-5, 2003, Prague, Czech.

[89] J. Liu, *« Continuous Time and Mixed-Signal Simulation in Ptolemy II »*, MS Report, UCB/ERL Memorandum M98/74, Dept. of EECS, University of California, Berkeley, CA 94720, December 1998.

[90] D. Luckman and al., *« Specification and analysis of systems of System architecture using RAPID »*, IEEE on Software Engineering, special issue on software architecture, 21(4), pp 336-355, April, 1995.

[91] D. Luckman and J. Vera, *« An event-based Architecture Definition Language »*, IEEE transaction on Software Engineering 21(9), pp 717-734, Sept, 1995.

[92] N. Lynch, R. Segala, F. Vaandrager, and H.B. Weinberg, *« Hybrid I/O automata »*, MIT Laboratory for Computer Science, March 2003. Availlable on line at http ://www.csail.mit.edu/research/abstracts/abstracts03/theory/43lynch.pdf

[93] D. Maliniak, *« Design Languages Vie For System-Level Dominance »*, Electronic Design, October 2001,pp.53-60.

[94] J. Martin, *« Design of real-time systems »*, Prentice Hall, Englewood Cliffs, NJ, 1965.

[95] S. E. Mattsson and M. Anderson, *« The ideas behind omola »*, in CACSD 92 : IEEE Symp. Comput. Aided Control Syst. Design, 1992, pp. 23–29.

[96] M. Mbobi, *« Modélisation hybride dans ptolemy II »*, Supelec-Ecole des Mines de Paris, Technical report, Septembre 2002.

[97] M. Mbobi, F. Boulanger and M. Feredj, *« Non-hierarchical heterogeneity »*, International Conference on Computer, Communication and Control Technologies, July 31, August 1-2, 2003, Orlando, Floride-USA. International Institute of Information and Systemics, Volume III, ISBN 980-6560-05-01, pages 430 à 435.

[98] M. Mbobi, F. Boulanger and M. Feredj, *« Execution Model for Non-Hierarchical Heterogeneous Modeling »*, to appear in the Proceedings of The 2004 IEEE International Conference on Information Reuse and Integration (IEEE-IRI 2004), November 8-10, 2004, Las Vegas, Nevada, USA, IEEE Catalog Number 04EX974, ISBN 2004113902, pages 139 à 144

[99] N. Modvidovic, D. S Rosenblum, *« Domains of concern in software architecture and architecture description languages »*, Proceedings of the USENIX Conference on Domain-Specific Languages October 15-17, 1997, Santa Barbara, California.

[100] M. Mokhtari and M. Marie. *« Engineering Applications of MATLAB5.3 and SIMULINK 3 »*, Springer Verlag, 2000.

[101] P. R. Moorby, D.E Thomas, *« The Verilog Hardware Description Language »*, Handcover, May 1988.

[102] G.E. Moore, *« Cramming more components onto integrated circuits »*, Research and Development Laboratories, Fairchild Semiconductor division of Fairchild Camera and Instrument Corp., Electronics, Volume 38, Number 8, April 19, 1965.

[103] L. Muliadi, *« Discrete Event Modeling in Ptolemy II »*, Technical Report, Dept. of EECS, University of California, Berkeley, CA 94720, May 1999. Availlale on line at http ://ptolemy.eecs.berkeley.edu/papers/

[104] P.A Muller et N. Gaertner, *« Modélisation Objet avec UML »*, Eyrolles, Paris, 2001 ISBN 2-212-09122-2.

[105] S. Neuendorffer, *« Automatic Specialization of Actor-oriented Models in Ptolemy II »*, Technical Memorandum UCB ERL M02/41, Electronics Research Laboratory, University of California at Berkeley, December 25, 2002. Availlale on line at http ://ptolemy.eecs.berkeley.edu/papers/

[106] S. Neuendorffer, *« Implementation Issues in Hybrid Embedded Systems »*, Technical Memorandum UCB ERL MEMO M03/22, Electronics Research Laboratory, University of California at Berkeley, June 24, 2003 Availlale on line at http ://ptolemy.eecs.berkeley.edu/papers/

[107] S. Neuendorffer and E.A. Lee, *« Hierarchical Reconfiguration of Dataflow Models »*, Invited paper, Conference on formal methods and models for codesign, MEMOCODE, San Diego, California, June 22-25, 2004.

[108] Object Management Group (OMG), *« CORBA Services, Common Object Services Specification »*, Availlable on line at http : //www.omg.org/technology/documents/corbaservices_spec_catalog.htm Mars 2004.

[109] Object Management Group (OMG), *« Common Object Request Broker Architecture »*,
Availlable on line at http ://www.omg.org/technology/documents/spec_catalog.htm

[110] Object Management Group (OMG), *« Unified Modeling Language : Superstructure »*, Interim FTF Report of the UML 2.0 Superstructure Finalization Task Force to the Platform, Technical Committee of the Object Management Group, 11 January 2004 Availlable on line at http ://www.omg.org/docs/ptc/04-01-11.pdf

[111] G J. Pappas and S. Simic, *« Consistent Abstractions of Affine Control Systems »*, IEEE, Transactions on automatic control Vol. 47, NO. 5, May 2002 745.

[112] T. M. Parks, *« Bounded Scheduling of Process Networks »*, Technical Report UCB/ERL-95-105. PhD Dissertation. EECS Department, University of California. Berkeley, California 94720. December 1995.

[113] Polis Homepage, Berkeley University, Hardware/Software design group, 2004, Availlable on line at http ://www-cad.eecs.berkeley.edu/ polis/

[114] RAPIDE Project Homepage, Stanford University, 2004, Availlable on line at http ://pavg.stanford.edu/rapide

[115] H.J Reekie and E.A. Lee, *« Lightweight Component Models for Embedded Systems »*, Technical Memorandum UCB ERL M02/30, Electronics Research Laboratory, University of California at Berkeley, October 2002, Availlale on line at http ://ptolemy.eecs.berkeley.edu/papers/

[116] Rosetta, 2004, Availlable on line at http ://www.sldl.org/

[117] RNTL, Rapport du groupe de travail « système embarqué et temps réel, codéveloppement », 2001, Availlale on line at http ://www.telecom.gouv.fr/rtnl

[118] S. Saracco, J. R. W. Smith, and R. Reed, *« Telecommunications Systems Engineering Using SDL »*, North-Holland - Elsevier, 1989.

[119] SDL, *« Computer Networks and ISDN Systems »*, CCITT SDL, 1987.

[120] L. Séméria, A. Ghosh, *« Methodology for Hardware/Software Co-verification in C/C++ »*, Proceedings of IEEE International High Level Design Validation and Test Workshop HLDVT'99, pp. 67-72, San Diego, Nov. 99.
And Proceedings of Asia and South Pacific Design Automation Conference ASP-DAC'00, pp. 405-408, Yokohama, January 2000.

[121] R. Sessions, COM and DCOM : *« Microsofts Vision for Distributed Objects »*, New York, NY : JohnWiley and Sons, Decembre, 1997, ISBN : 047119381X.

[122] F. Simonot-Lion, J.P. Elloy, Y. Trinquet. *« AIL_ Transport : Un langage de description d'architecture électronique embarquée dans l'automobile »*, Veille Technologique, 45, juin, 2002, p. 34-36.

[123] N. Smyth, *« Communicating Sequential ProcessesDomain in Ptolemy II »*, Master's Report, UCB/ERL Memorandum M98/70, Dept. of EECS, University of California, Berkeley, CA 94720, December 1998.

[124] Specc, Availlable on line at http ://www.specc.gr.jp/eng/index.htm

[125] Stewart, R.A. Volpe, and P.K. Khosla, *« Design of dynamically reconfigurable real-time software using port-based objects »*, IEEE Trans. on Software Engineering, 23(12), 1997, pp.759-776.

[126] Superlog, 2004, Availlable at http ://www.synopsys.com/products/hlv/hlv.html

[127] Svarstad, G. Nicolescu, and A.A. Jerraya, *« A Model for Describing Communication between Aggregate Objects in the Specification and Design of Embedded Systems »*, Munich, Germany, Pages : 77-85, 2001, ISBN :0-7695-0993-2.

[128] S. Swan, *« An Introduction to System Level Modeling in SystemC 2.0 »*, White paper of OSCI, Mai 2001.

[129] SystemC Transaction Level Modeling Working Group Charter., June 2003, Availlable on line at http ://www.systemc.org/

[130] C. Szyperski, *« Component Software : Beyond Object-Oriented Programming »*, Addison-Wesley Pub., Jan. 1998.

[131] P. Tabuada, G.J. Pappas and P. Lima, *« Hybrid Abstractions : A Search and Rescue Case Study »*, Proceedings of the European Control Conference 2001.

[132] P. Tabuada, G.J. Pappas and P. Lima, *« Compositional Abstractions of Hybrid Control Systems »*, Proceedings of the 40th IEEE Conference on Decison and Control, Orlando, Florida USA, December 2001.

[133] UML, Availlable on line at http ://www-306.ibm.com/software/rational/uml/

[134] F. Vahid and D. Gajski, *« Specification Partitioning For System Design »*, Proceedings of the IEEE Design Automation Conference, pp. 219-224, June 1992.

[135] A. Cataldo, C. Hylands, E.A. Lee, J. Liu, X. Liu, S. Neuendorffer, H. Zheng, Technical Memorandum UCB/ERL M03/31, University of California, Berkeley, CA 94720, July 17, 2003. *« HyVisual : A Hybrid System Visual Modeler »*, http ://ptolemy.eecs.berkeley.edu/publications/papers/

[136] D. Verkest et al, *« Co-Ware, A design environment for heteregeneous Hardware/Software systems »*, Design automation for embedded systems, vol 1, no 4, pg. 357-386, 1996.

[137] VHDL, Institute of Electrical and Electronically Engineers, IEEE, Standard VHDL Language, Reference manual, STD 1076-1993. IEEE, 1993.

[138] Whitaker, *« The Simulation of Synchronous Reactive Systems In Ptolemy II »*, Master's Report, Memorandum UCB/ERL M01/20, Electronics Research Laboratory, University of California, Berkeley, May 2001.

[139] G. Vidal-Naquet, F. Boulanger, H. Delebecque, *« Intégration de modules synchrones dans un langage à objets »*, Conférence Real Time Systems, Paris, 1994

[140] M. Von der Beeck, *« A Comparison of Statecharts Variants »*, in Proceedings of Formal Techniques in Real Time and Fault Tolerant Systems, LNCS 863, pp. 128-148, Springer-Verlag, 1994.

[141] Y. Xiong, *« An Extensible Type System for Component-Based Design »*, Technical Report, Ph.D in Engineering. Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Spring 2002.

[142] J.S. Young and al, *« Design and specification of embedded systems in Java using successive, formal rafinement »*, Proceedings of Of 35th DCA, pg. 70-74, 1998.

[143] J.Zhu, D. Gajski, *« OpenJ : An extensive system level design language »*, Proceedings of DATE, 1999.