Université Paris XI

École Supérieure d'Électricité

UFR Scientifique d'Orsay

N° d'ordre : 8101

UNIVERSITÉ PARIS SUD

UFR SCIENTIFIQUE D'ORSAY

THÈSE

Présentée pour obtenir le grade de

DOCTEUR EN SCIENCES

DE L'UNIVERSITÉ PARIS XI ORSAY

Spécialité : Informatique

Préparée au département informatique de Supélec à Gif-sur-Yvette

par

MOHAMED FEREDJ

SUJET :

Étude des Méthodes de Conception de Composants Domaine-Polymorphes

Soutenue publiquement le 02 décembre 2005 devant la commission d'examen

Rapporteur	M. Charles ANDRÉ	Université Nice Sophia-Antipolis
Examinateur	M. Frédéric BOULANGER	Supélec
Rapporteur	M. Paul CASPI	Université Grenoble I
Examinateur	M. Jean-Paul SANSONNET	CNRS
Examinateur	M. François TERRIER	CEA Saclay
Directeur de thèse	M. Guy VIDAL-NAQUET	Université Paris-Sud Orsay et Supélec

À

ma défunte mère, mon père, ma tante et
son défunt époux,

mes sœurs, mes frères, mes nièces,
mes neveux, toute ma famille,

et ma chère fiancée.

Remerciements

C'est avec une grande joie que je formule mes remerciements qui témoignent par écrit ma reconnaissance à tous ceux qui m'ont manifesté leur soutien et leur confiance tout au long de ces années de thèse.

D'abord, j'adresse mes sincères remerciements à Monsieur OLIVIER FRIEDEL, ancien chef du département informatique de Supélec et actuellement directeur des études de Supélec, pour m'avoir accueilli au sein du département informatique et m'avoir donné tous les moyens pour la réalisation de ma thèse. Je tiens aussi à remercier Madame YOLAINE BOURDA qui lui a succédé à la tête du département informatique, de m'avoir assuré les mêmes conditions, notamment le prolongement du financement de ma thèse.

C'est avec un grand respect que j'adresse mes sincères reconnaissance et remerciements à Monsieur GUY VIDAL-NAQUET, Professeur à l'université de Paris-Sud et détaché au département informatique de Supélec, pour m'avoir fait l'honneur de diriger ma thèse. Ses conseils et ses remarques étaient toujours valorisants.

J'adresse mes sincères reconnaissances à Monsieur PAUL CASPI, Professeur à l'université de Grenoble 1 et Directeur de Recherche au CNRS, et Monsieur CHARLES ANDRÉ, Professeur à l'université de Nice Sophia-Antipolis, d'avoir accepté la lourde tâche de rapporter sur ma thèse. Je leur présente mes remerciements les plus chaleureux pour m'avoir honoré par cette acceptation. Qu'ils trouvent ici l'expression de ma profonde gratitude.

Je tiens à remercier Monsieur JEAN-PAUL SANSONNET, Directeur de recherche au CNRS, et Monsieur FRANÇOIS TERRIER, Directeur de recherche au CEA d'avoir accepté d'examiner ma thèse.

Je tiens à exprimer mes sincères reconnaissance et remerciements à Monsieur FRÉDÉRIC BOULANGER, professeur au département informatique de Supélec, de m'avoir proposé ce sujet de thèse, ce qui m'a permis de découvrir un domaine qui est à la fois difficile et passionnant, c'est pourquoi je lui doit beaucoup mon goût pour les « Acteurs ». Je le remercie aussi pour ses conseils et ses orientations.

Et bien sûr sans oublier mon collègue Monsieur MOKHOO MBOBI, Docteur en sciences de l'université Paris-Sud XI et actuellement en Post-Doc, qui m'a traité tout au long de ma thèse comme un petit frère. Je lui exprime du fond de mon cœur mes sincères reconnaissance et remerciements pour ses conseils et ses remarques qui étaient toujours valorisants. Grâce à ses compétences transversales, qui sont comme par hasard celles nécessaires à l'étude des systèmes embarqués, j'ai toujours trouvé chez lui des réponses bien détaillées à mes questions. Qu'il trouve ici l'expression de ma profonde gratitude.

Je tiens à remercier Monsieur STEVE NEUENDORFFER, ancien doctorant du département EECS de l'université de Berkeley en Californie et actuellement chercheur à Xilinx Research Labs, de m'avoir donné les détails nécessaires sur l'implémentation de l'interface graphique Vergil de Ptolemy II lors de l'intégration de mes concepts dans celle-ci.

Je remercie tout le personnel du département informatique de Supélec pour les bons moments que nous avons passés tout au long de cette thèse. Un grand merci à notre secrétaire, Madame EVELYNE FAIVRE, et notre technicien, Monsieur GILBERT DAHAN, qui étaient toujours présents quant j'avais besoin d'eux.

Je ne dois jamais oublier ma tante et son défunt époux, qui ne savent ni lire ni écrire. Sans eux je ne serai jamais arrivé à ce niveau d'étude. Je leur présente du fond de mon cœur ma sincère reconnaissance d'avoir supporté de m'élever malgré leurs modestes moyens. Je les remercie surtout de m'avoir appris, depuis mon enfance, d'être responsable et de ne compter que sur moi-même.

Mes remerciements vont droit à ma chère défunte mère et mon père qui ont fait tout pour que je réussisse dans mes études. Ils étaient toujours là, bien que je n'aie pas vécu avec eux. Je remercie aussi mes frères et mes sœurs de m'avoir soutenu et encouragé tout au long de cette thèse.

J'ajoute à ces remerciements tous mes amis qui m'ont encouragé et soutenu, d'une façon directe ou indirecte. Ils avaient toujours confiance en ma réussite.

Finalement, une tendre pensée à ma fiancée HASSIBA KELLALI, qui a supporté toutes ces années de ma thèse et toutes les barrières géographiques sans cesser de m'encourager et de me soutenir, tout en croyant fermement que j'y arriverai.

Il est plus difficile de dire les choses

difficiles simplement que de

dire les choses difficiles

difficilement.

Résumé

Les systèmes embarqués sont utilisés dans plusieurs domaines d'application et mélangent plusieurs technologies, ce qui fait d'eux des systèmes hétérogènes complexes. Ainsi, ils sont composés de sous systèmes dont chacun obéit à un modèle de calcul qui lui est approprié. Les modèles de calcul sont des lois qui régissent les interactions des systèmes.

En effet, la conception d'un système doit faire coexister les modèles de calcul régissant ses composants au sein du modèle qui le représente, ce qui revient à faire communiquer des composants obéissants à des modèles de calcul différents. Pour cela, on peut utiliser l'approche hiérarchique ou l'approche non hiérarchique. L'approche hiérarchique structure un système en niveaux hiérarchiques dont chacun contient des composants qui obéissent au même modèle de calcul. On est ainsi obligé de changer de niveau hiérarchique pour passer d'un modèle de calcul à un autre.

Au contraire, l'approche non hiérarchique permet d'utiliser plusieurs modèles de calcul à un même niveau de la hiérarchie en les séparant en sous ensembles homogènes interfacés par des composants à interfaces hétérogènes.

Actuellement, pour exécuter un composant atomique, dont le comportement est spécifié suivant la sémantique d'un modèle de calcul, dite sémantique de spécification, sous différents modèles de calcul, on peut utiliser soit l'approche hiérarchique, soit l'approche non hiérarchique, soit des composants domaine-spécifiques. Un composant domaine-spécifique étant un composant qui est conçu pour fonctionner selon un modèle de calcul particulier, et dont le fonctionnement correct n'est donc garanti que pour le domaine correspondant.

Cependant, ces approches présentent plusieurs désavantages. L'approche hiérarchique mène à une explosion verticale des niveaux hiérarchiques, nuit à la modularité du modèle, et n'explicite pas les interactions entre les modèles de calcul. L'approche non hiérarchique oblige à spécifier la sémantique des interactions entre modèles de calcul deux-à-deux, ce qui conduit à une explosion combinatoire en fonction du nombre de modèles de calcul supportés. Quant aux composants domaine-spécifiques, il sont coûteux car ils sont générés à partir d'une même spécification pour chacun des modèles de calcul dans lesquels on veut les utiliser. Ils ne sont donc pas réutilisables dans d'autres modèles de calcul, et ont un coût de mise-à-jour élevé.

Pour éviter ces problèmes, nous proposons dans cette thèse un nouveau modèle de composant, appelé composant domaine-polymorphe. Un composant domaine-polymorphe est un composant atomique ayant la capacité d'exécuter son comportement interne suivant la sémantique de spécification tout en garantissant un fonctionnement correct sous différents modèles de calcul. En découplant la sémantique de spécification et celle d'exécution, cette approche offre une bonne modularité. L'adaptation à la sémantique d'exécution étant automatique, ces composants sont facilement réutilisables et paramétrables afin d'expliciter les interactions entre modèles de calcul, ce qui augmente la productivité et facilite la maintenabilité ainsi que le processus de validation.

Mots Clés : Systèmes Embarqués, Modèles de Calcul, Conception Hétérogène, Ingénierie Logicielle, Composants, Acteurs.

Abstract

Embedded systems are used in numerous application domains and mix several technologies, what makes them complex and heterogeneous systems. They are composed of subsystems that are designed using a model of computation (MoC) that suits the needs of their designers. The MoCs are the laws that govern the interaction of the components of a subsystem.

The model of system must allow the MoCs that govern its components to coexist and interact. Therefore, components that obey different MoCs must be able to communicate. Two main approaches are possible to make MoCs interact : the hierarchical approach or the non-hierarchical approach. The hierarchical approach structures the system into hierarchical levels where each level contains components that obey the same MoC. Therefore, each change of MoC implies a new level in the hierarchical structure of the system. On the contrary, the non-hierarchical approach allows the use of several MoCs at the same level of the hierarchy, and therefore supports Heterogeneous Interface Components. Such components have terminals that obey different MoCs and can be considered as bridges between MoCs.

In order to use an atomic component, the behavior of which is specified according to a given MoC, in a model that obeys a different MoC, we can use one of the approaches above or build a domain-specific component (DSC). Such components are designed for use in a specific MoC, and their correct behavior is not guaranteed in other domains.

However, these approaches present several disadvantages. The hierarchical approach creates hierarchical levels that do not reflect the structure of the system, impairs reusability, and make the semantics of the interaction between MoCs implicit. The non-hierarchical approach requires that the semantics of the interactions between MoCs be specified for each pair of MoCs, what leads to a combinatorial explosion of the specifications versus the number of MoCs used. Last, DSCs are expensive because they are specific to a MoC and must be generated from the same specification for each MoC under which we want to use them.

To overcome these problems, we propose a new model of component, that we call “domain-polymorph component”. Such a component is atomic and is able to execute its core behavior, specified under a given MoC, under different host MoCs. By decoupling the semantics of the specification and the semantics of the execution context, our approach provides good modularity. The adaptation to the semantic of execution is automatic, so the domain-polymorph components are easily reusable. It is possible to customize their adaptation to the host MoC in order to get explicit control on the interactions between MoCs, what facilitates the maintainability and the process of validation of the systems.

Keywords : Embedded Systems, Models of Computation, Heterogeneous Design, Software Engineering, Components, Actors.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction Générale

Les systèmes embarqués qui autrefois étaient utilisés essentiellement pour des besoins collectifs tels que le militaire, le spatial et l'aéronautique sont aujourd'hui de plus en plus présents dans notre environnement pour des utilisations personnelles telles que le micro-onde, le téléphone portable, le chronomètre, etc.

Cette présence a été confirmée par de récentes études scientifiques [2] [3] dont les chiffres révèlent 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 l'an 2004 était 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 la vie courante [12].

Dans [12], l'auteur affirme que la croissance de l'utilisation des systèmes embarqués est due aux raisons suivantes :

- La digitalisation, car, de nombreux appareils qui étaient dans le passé analogiques sont devenus numériques, et en conséquence, sont devenus programmables, ce qui nécessite un savoir-faire et des outils spécifiques ;
- La conjonction entre l'accroissement des performances et la baisse des prix des processeurs dictés par la loi de Moore [4]. 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 ;
- 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 devenu un noeud d'un réseau dont ils augmentent l'utilité. En retour, ces réseaux engendrent des synergies qui ouvrent de nouvelles opportunités pour ces appareils.

Un système embarqué est défini comme un dispositif matériel comportant des parties logicielles et dont le fonctionnement est par nature en temps réel [7]. Il est embarqué (enfoui) dans un dispositif plus large constituant son environnement mais qui n'est pas forcément un ordinateur [8].

Un système embarqué est au minimum composé de deux composants dont le premier est un contrôleur qui supervise le deuxième dédié au traitement de données. De plus, il réagit aux stimuli provenant de son environnement à travers ses capteurs et ses actionneurs. Les capteurs permettent de

[1] L'utilité d'un réseau est proportionnelle au carré du nombre de ses utilisateurs.

détecter les variations de l'environnement et transmettent ensuite les données concernant ces variations au système. Quant aux actionneurs, ils permettent d'agir sur l'environnement en transformant les réponses du système en actions.

Un système embarqué est en interaction continue avec son environnement. Cette interaction implique des communications avec cet environnement ainsi que d'autres communications entre les sous-systèmes qui le composent.

En ce qui concerne leur utilisation, les systèmes embarqués sont utilisés dans plusieurs secteurs qui généralement combinent différents domaines techniques, tels que la mécanique, l'électronique analogique, l'électronique numérique, etc. Chacun des sous-systèmes les composant peut appartenir à un domaine technique différent des autres. Cette diversité d'appartenance implique que chaque sous-système obéisse à un ensemble de lois physiques ou d'axiomes régissant ses interactions avec le reste de son environnement. Cet ensemble est appelé *« Modèle de Calcul », ou Model of Computation - (MoC).* Actuellement, il existe de nombreux modèles de calcul comme le modèle de calcul à flots de données synchrones (SDF), le modèle réactif synchrone (SR), les machines à états finies (FSM), les événements discrets (DE), le temps continu (CT), etc. De cette pluralité de MoCs sur les systèmes embarqués découle leur hétérogénéité.

Cette hétérogénéité, tout en complexifiant les systèmes embarqués, représente un défi à la fois économique et scientifique. Cette analyse est détaillée dans [12]. Ainsi, des sauts technologiques s'imposent car l'industrie doit fabriquer de plus en plus vite des produits de plus en plus complexes, et ce, dans le respect absolu de certaines contraintes liées au poids, au volume, à la rapidité de calcul, au coût, etc. Par exemple, dans l'avion Airbus A340, les systèmes embarqués dédiés aux gestions du pilotage automatique, des commandes de vol, de freinage etc., sont implémentés par plus de 100 calculateurs répartis dans l'avion et interconnectés via des bus numériques [1]. Ces calculateurs doivent être suffisamment rapides pour fournir des réponses avant la date d'échéance. Ils doivent également être de volume réduit pour faciliter leur intégration tout en étant extrêmement légers pour ne pas augmenter le poids total de l'avion.

Par ailleurs, les outils de modélisation, de spécification, de conception et de simulation des systèmes embarqués doivent à la fois permettre d'exprimer la structure, les comportements et les propriétés de ces systèmes embarqués pour les modéliser, les concevoir et les simuler le plus fidèlement possible, et offrir de meilleurs mécanismes de modularité et de réutilisabilité pour augmenter la productivité et faciliter la maintenabilité ainsi que le processus de validation.

1.1 Conception des systèmes embarqués hétérogènes

La conception des systèmes embarqués hétérogènes consiste à élaborer leurs modèles exécutables, qui décrivent leurs comportements et leurs propriétés pertinentes, afin de répondre aux stimuli provenant de leurs environnements. Elle 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.

A toutes les étapes de la conception, un système embarqué hétérogène peut être vu comme un ensemble de composants dont chacun a sa propre interface de communication et obéit à un modèle de calcul qui lui est approprié. En effet, la conception d'un système embarqué hétérogène doit organiser et interconnecter l'ensemble des composants, ce qui entraîne une mise en communication des composants obéissant à des modèles de calcul différents. La conception des systèmes embarqués hétérogènes doit donc faire face au problème de l'hétérogénéité.

Actuellement, dans le but de pallier les difficultés de l'hétérogénéité et de pouvoir faire coexister plusieurs composants obéissant à des modèles de calcul différents au sein d'un même modèle représentant un système, la conception peut se baser sur l'une des trois approches hétérogènes. Il s'agit de l'approche amorphe [20] [64][82], l'approche hiérarchique [108] [27] [33] [25] [35] [36] [37] [38] [32] [73] et l'approche non hiérarchique [30] [31] [39] [12] [71] [72].

Conception hétérogène amorphe

L'hétérogénéité amorphe permet de faire coexister plusieurs modèles de calcul au même niveau. Ainsi, les composants obéissant à des modèles de calcul différents peuvent communiquer directement. Pour cela, l'approche amorphe implique qu'un composant doit incorporer les caractéristiques de multiples modèles de calcul et avoir donc de multiples interfaces de communication.

En général, cette approche ne se concentre que sur un petit nombre de modèles de calcul qui sont souvent le temps continu et le temps discret pour les systèmes mixant des signaux analogiques et numériques, et les machines à états finis et le temps continu pour les systèmes hybrides.

Puisque elle n'utilise qu'un nombre réduit de modèles de calcul, l'hétérogénéité amorphe fait l'union de ces modèles de calcul afin d'avoir une conjonction entre eux [73]. Ainsi, les composants obéissent simultanément à plusieurs modèles de calcul. Par exemple, un convertisseur de signal analogique/numérique peut être conçu comme un composant avec des ports d'entrée obéissant au modèle de calcul temps continu et des ports de sortie obéissant au modèle de calcul temps discret.

VHDL-AMS [44] et Simulink [50] font partie des outils de modélisation de conception et/ou de simulation basés sur l'approche amorphe.

L'avantage de l'hétérogénéité amorphe c'est qu'elle permet l'intégration complète des modèles de calcul, c'est à dire qu'il n'y a pas de frontières entre les modèles de calcul. Cependant, elle souffre de certaines faiblesses que nous citons comme suit :

- Les protocoles de communication interagissent d'une façon imprévisible avec le flot de contrôle car il n'y a pas une séparation claire entre le flot de contrôle et la communication ;
- La conception est moins compréhensible, ce qui rend la validation plus difficile ;
- Il est impossible d'ajouter de nouveaux modèles de calcul, l'ensemble des modèles de calcul utilisables n'est donc pas ouvert.

Conception hétérogène hiérarchique

Contrairement à l'approche amorphe, avec l'approche hiérarchique, un composant n'obéit qu'à un seul modèle de calcul différent des autres. En conséquence, la communication entre les différents composants ne peut avoir lieu de façon directe. Cela provient de l'incompatibilité des interfaces et protocoles de communication utilisés par les composants. L'approche hiérarchique impose de placer chaque MoC différent dans un niveau différent de la hiérarchie. Ainsi, les composants appartenant au même niveau hiérarchique forment un réseau de composants interconnectés et obéissent au même modèle de calcul, voir figure 1.1.

Lorsqu'un composant est un schéma de composants, c'est-à-dire un composant raffiné en un schéma de composants, on l'appelle composant composite. Par contre, lorsque son contenu est une description de son comportement sous forme d'un ensemble d'opérations (qui peuvent être spécifiées par un langage de haut niveau), on l'appelle composant atomique.

Pour que les données passent d'un niveau hiérarchique à un autre (au niveau des frontières), il faut détailler d'une manière très fine comment les données, dont le format dépend du modèle de calcul du niveau hiérarchique, sont communiquées avec fidélité entre les différents niveaux et aussi spécifier les interfaces au niveau des frontières entre les niveaux hiérarchiques. Malheureusement, à notre connaissance, les outils de modélisation et de conception actuels, qui se basent sur l'approche hiérarchique pour combiner différents modèles de calcul, *n'offrent pas un passage explicite entre les niveaux hiérarchiques* [12], tel est le cas de Ptolemy II [8][9][10], El Greco [41], OMOLA [40], SystemC [43], ROSETA [45], POLIS [46], DYMOLA [47], MODELICA [48], SpecC [42].

L'hétérogénéité hiérarchique a beaucoup d'avantages :

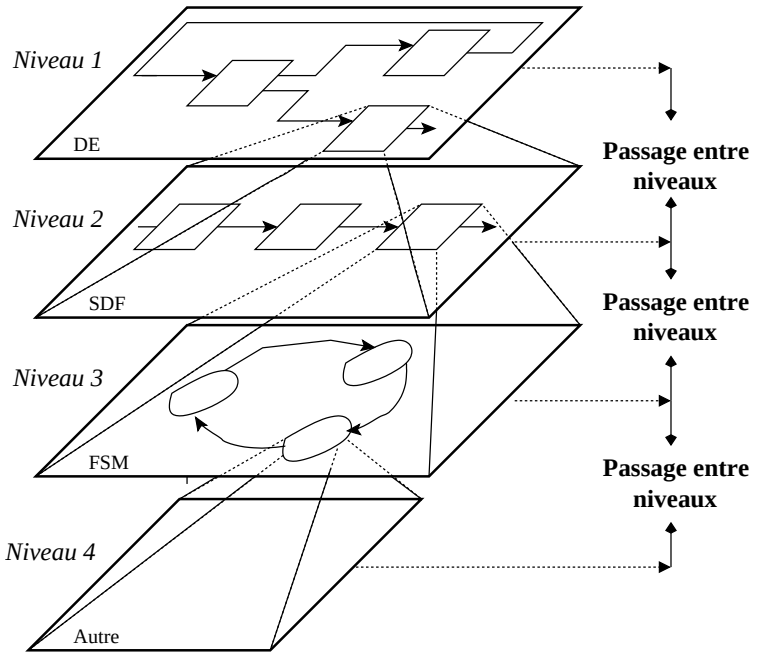


FIG. 1.1: Hétérogénéité hiérarchique

- Elle permet de contrôler et de maîtriser la complexité des systèmes. Elle considère un système complexe comme un ensemble composé de parties élémentaires ;
- Du point de vue syntaxique, elle permet d'abstraire un réseau de composants obéissant au même modèle de calcul en un seul composant (composant composite). Ce qui permet aux concepteurs de voir le système au niveau de détail désirable ;
- Elle permet d'organiser l'hétérogénéité d'une façon structurée qui permet ainsi de faciliter des raffinements successifs ;
- L'ensemble des modèles de calcul est ouvert : il est possible d'ajouter de nouveaux modèles de calcul.

Cependant, elle possède plusieurs limites. Dans [12], ces limites sont classées suivant trois considérations :

- Du point de vue des outils de modélisation et de conception, 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 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 ;
- Pour le concepteur du système, il ne peut pas explicitement intervenir sur les transformations des données qui se produisent à la frontière de deux niveaux hiérarchiques (lors du passage d'un modèle de calcul à un autre).

Conception hétérogène non hiérarchique

L'approche non hiérarchique permet d'avoir un modèle hétérogène plat qui donne la possibilité de changer de modèle de calcul sans changer de niveau hiérarchique, et cela sans altération du sys-

tème original. Elle permet aussi la conception de composants susceptibles de travailler à la frontière de plusieurs modèles de calcul (composants avec des ports d'entrée/sortie hétérogènes). De plus, le concepteur a la possibilité d'expliciter le passage des données et de contrôler le comportement du système au niveau des frontières, ce qui fait partie de la conception des systèmes.

La conception des systèmes embarqués hétérogènes basée sur l'approche non hiérarchique consiste d'abord à concevoir chaque-sous système du système d'une manière indépendante. Ensuite, suivant leurs relations d'interconnexion au niveau du système original, elle interconnecte directement les composants homogènes (ceux obéissant au même modèle de calcul) tandis que les autres composants (n'obéissant pas au même modèle de calcul) sont interconnectés à travers des composants à interface hétérogène (HICs) [30]. A cette étape, la conception basée sur l'approche non hiérarchique permet d'avoir un modèle hétérogène plat [31][12] représentant le système, c'est-à-dire un modèle permettant la cœxistance de différents modèles de calcul au même niveau.

Avec cette approche, l'hétérogénéité sera gérée à l'exécution. Donc, au lancement de l'exécution du modèle hétérogène plat, l'approche non hiérarchique transforme le modèle original (modèle plat) en ajoutant un niveau à sa hiérarchie et en construisant des sous-ensembles homogènes à ce niveau afin d'isoler les différents modèles de calcul. Ainsi, deux couches en résulte, une couche supérieure et une couche secondaire contenant les sous-ensembles homogènes. Pour contrôler la couche supérieure, l'approche non hiérarchique propose un modèle de calcul hétérogène permettant l'ordonnancement des sous ensembles homogènes et la délégation du calcul de leurs comportements à leurs modèles de calcul.

Comme le composant HIC a des ports de communication obéissants à des modèles de calcul différents, le modèle d'exécution hétérogène effectue une tâche cruciale lors de la transformation. Cette tâche consiste à projeter le composant HIC sur chacun des sous-système qui l'utilisent. Les composants représentant le composant HIC dans les sous-ensembles homogènes sont appelés les projections du HIC [12], voir figure 1.2.

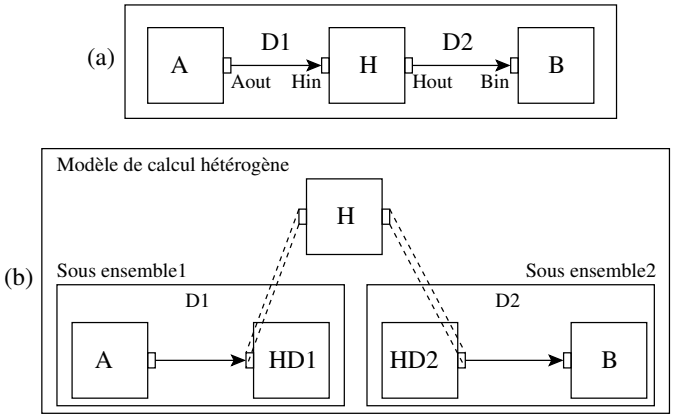


FIG. 1.2: Hétérogénéité non hiérarchique : (a) Le modèle hétérogène plat. (b) Partitionnement du modèle et projection du HIC : Système final comme exécuté.

Le composant projection d'un HIC appartenant à un sous ensemble homogène est un composant ayant une structure et un comportement qui respectent les caractéristiques du modèle de calcul de son sous-ensemble homogène (c'est-à-dire que la projection du HIC est un composant comme les autres composants de son sous-ensemble homogène) et son comportement est calculé par ce même modèle de calcul. Aussi, parmi l'ensemble de ports du HIC, un composant projection ne garde que les ports obéissant au modèle de calcul de son sous-ensemble homogène tandis que les autres ports obéissant à d'autres modèles de calcul sont cachés.

Le composant HIC n'est pas connecté à ses composants projections par contre il partage les mêmes ports de communication avec eux. C'est grâce à ce partage que les données passent d'un

sous ensemble homogène à un autre, en partant du composant projection du sous ensemble de départ, ensuite en passant par le composant HIC et finalement en arrivant au composant projection du sous ensemble d'arrivée. Ainsi, le composant HIC est vu comme un canal intelligent.

Avec l'approche non hiérarchique, l'ensemble de modèles de calcul est ouvert : il est possible d'ajouter de nouveaux modèles de calcul.

Malgré ses avantages, l'approche non hiérarchique est moins puissante que l'approche amorphe en ce qui concerne les boucles hétérogènes. Actuellement, avec l'approche hétérogène non hiérarchique il est impossible d'utiliser les boucles hétérogènes alors que c'est possible avec l'approche amorphe.

1.2 Motivations de nos travaux

Un système embarqué hétérogène peut être vu comme un ensemble de composants (contrôleur et environnement) disposants chacun de sa propre interface de communication et obéissants à un modèle de calcul qui leur est approprié. En effet, la conception d'un tel système doit organiser et interconnecter l'ensemble de ces composants, ce qui entraîne une mise en communication entre des modèles de calcul différents utilisés respectivement par le contrôleur et son environnement.

Le composant contrôleur d'un système embarqué doit être spécifié (ou décrit) suivant une sémantique orientée contrôle (dite sémantique de spécification), par exemple, la sémantique DE (Discrete Event), SR (Synchronous Reactive) ou FSM (Finite State Machines). Le composant dédié au traitement de données doit également être spécifié suivant une autre sémantique orientée traitement de données, par exemple, la sémantique SDF (Synchronous Dataflow), DDF(Dynamic DataFlow), etc.

Donc, un composant contrôleur spécifié selon un modèle de calcul ne peut être utilisé que dans un environnement qui utilise le même modèle de calcul. Par contre, dans le cas où celui-ci est utilisé dans d'autres environnements qui implémentent des modèles de calcul différents, ceci peut entraîner des incompatibilités de communication.

Du point de vue de la modélisation et de la conception, cette logique reste la même pour un composant atomique modélisé dans un domaine utilisant une sémantique différente de sa sémantique de spécification.

Actuellement, pour utiliser le même composant atomique dans plusieurs domaines utilisant des sémantiques différentes de la sienne, on est obligé de gérer l'hétérogénéité entre la sémantique du composant atomique et celle de chacun des domaines (c'est à dire de conserver la sémantique du composant atomique au sein de différents domaines). Pour cela, pour chaque domaine, on doit traduire le composant atomique en un composant domaine-spécifique dont le comportement interne représente le comportement du composant atomique. Un composant domaine-spécifique a une structure et des ports de communication qui sont spécifiques au modèle de calcul implémenté par son domaine.

Cependant, cette solution doit spécifier une interface à l'intérieur de chaque composant domaine-spécifique pour gérer l'hétérogénéité entre la sémantique de spécification du comportement interne et celle du domaine. La gestion de l'hétérogénéité entre la sémantique interne et celle du domaine revient à conserver la sémantique interne et respecter la sémantique du domaine (dite aussi sémantique externe).

L'utilisation des composants domaine-spécifiques pour conserver une sémantique au sein de différents modèles de calcul présente plusieurs désavantages, à savoir : une mise en œuvre très lourde, l'exigence que le concepteur ait une parfaite connaissance des caractéristiques des modèles de calcul, car la gestion de l'hétérogénéité entre la sémantique de spécification et celle du domaine est de sa responsabilité, la non réutilisabilité des composants que ce soit dans le même domaine ou dans différents domaines, la non lisibilité du code des composants, des mises à jour des composants très coûteuses, nombre de composants explosif.

Par ailleurs, il est possible d'utiliser les approches hiérarchique et non hiérarchique pour gérer l'hétérogénéité entre la sémantique de spécification et celle du domaine :

- Soit en se basant sur l'approche hiérarchique pour encapsuler le composant atomique dans un composant composite implémentant le modèle de calcul approprié à la sémantique de spécification. Ensuite, par l'abstraction hiérarchique, on utilise le composant composite au sein du domaine. Ainsi, le composant composite obéit au modèle de calcul du domaine et exécute le composant atomique suivant son modèle de calcul interne.
  Cette solution présente quelques désavantages donnés dans la sous section « conception hétérogène hiérarchique » ;
- Soit en se basant sur l'approche non hiérarchique pour utiliser le modèle de calcul approprié à la sémantique de spécification au même niveau que celui du domaine. Pour cela, on interconnecte le composant atomique avec les composants du domaine à travers un HIC. Ce dernier possède des ports obéissant au modèle de calcul approprié à la sémantique de spécification et des ports obéissant au modèle de calcul du domaine. Ainsi, à l'exécution on aura deux sous ensembles homogènes au même niveau. Le premier implémente le modèle de calcul approprié à la sémantique de spécification et contient le composant atomique et une projection de HIC. Quant au deuxième, il implémente le modèle de calcul du domaine et contient les composants de celui-ci et une projection du HIC. En effet, c'est le composant HIC qui gèrera l'hétérogénéité entre les deux modèles de calcul.
  Bien que cette solution résolve le problème de l'hétérogénéité entre la sémantique de spécification et celle du domaine, elle présente certaines faiblesses, notamment l'obligation d'utiliser des composants HIC, ce qui augmente le nombre de composants utilisés dans la conception.

La question qui se pose est *comment rendre possible l'exécution d'un composant atomique, dont le comportement est spécifié suivant la sémantique d'un modèle de calcul, au sein de différents domaines implémentant des modèles de calcul différents, et cela, sans passer ni par l'approche hiérarchique ni par l'approche non hiérarchique, ni par les composants domaine-spécifiques ?*

Cette thèse apporte une réponse à cette question en proposant une approche de conception de composants domaine-polymorphes [196]. Un composant domaine-polymorphe est un composant atomique, dynamique, réutilisable et qui a la capacité de s'adapter à différents modèles de calcul tout en garantissant un comportement interne en accord avec la sémantique de spécification.

1.3 Contributions

Notre principale contribution dans cette thèse est la proposition d'un nouveau modèle de composant, dit composant domaine-polymorphe, qui apporte des avantages pour la modélisation et la conception des systèmes embarqués hétérogènes.

En particulier, ce modèle de composant est très utile pour la modélisation et la conception de la partie contrôle des systèmes embarqués hétérogènes, car cette partie a pour rôle de guider les comportements desdits systèmes indépendamment des autres modèles de calcul.

Notre approche de conception de composant domaine-polymorphe a pris la sémantique réactive synchrone comme sémantique de spécification et les composants domaine-spécifiques comme moyen d'utilisation des comportements synchrones sous différents modèles de calcul. Ainsi, un composant domaine-spécifique ayant un comportement synchrone est baptisé composant domaine-spécifique synchrone (CDS Synchrone). Ensuite, en partant de la structure abstraite commune aux composants domaine-spécifiques synchrones, notre approche propose des modèles de composants intermédiaires jusqu'à parvenir au modèle de composant domaine-polymorphe.

Comme le montre la figure 1.3, quelle que soit la sémantique de spécification du comportement interne, un composant domaine-polymorphe comporte quatre couches :

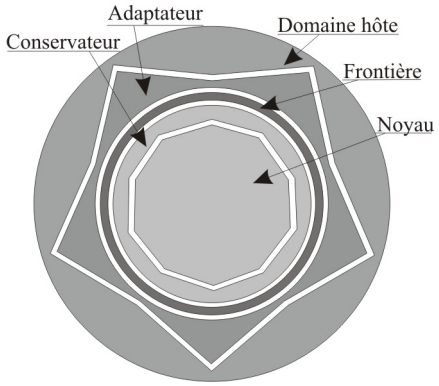


FIG. 1.3: Vue abstraite pour les composants domaine-polymorphes.

- Couche Noyau qui spécifie le comportement interne du composant domaine-polymorphe suivant la sémantique de spécification et représente une entité à exécuter. Par exemple, elle représente un module réactif synchrone ;
- Couche Conservateur qui fournit l'environnement d'exécution adéquat à la sémantique de la couche Noyau ;
- Couche Frontière permettant d'expliciter le passage des données entre la sémantique du Noyau et la sémantique du domaine hôte. Cette explicitation du passage fait partie de la modélisation et la conception des systèmes ;
- Couche Adaptateur permettant au composant domaine-polymorphe de fonctionner correctement dans le domaine hôte.

Notre proposition donne aux concepteurs des systèmes embarqués hétérogènes plusieurs avantages, nous citons :

- La réutilisabilité : le composant domaine-polymorphe est réutilisable sous plusieurs modèles de calcul, même dans le cas où un modèle de calcul n'existait pas au moment de sa conception, ce qui implique que la conception de la couche Noyau se fait une fois pour toute ;
- La conception de la couche Noyau ne nécessite que la connaissance des caractéristiques de la sémantique de spécification ;
- La sémantique du passage des données entre les deux modèles de calcul est explicite et fait partie de la conception du système. En effet, les concepteurs auront le choix entre différentes possibilités pour générer des données internes à partir des données externes et vice versa. Ces possibilités peuvent être l'échantillonnage, l'interprétation, la détection, une simple conversion, etc.

Tous les concepts conçus dans cette thèse ont été intégrés dans Ptolemy II [8] [9] [10], qui est une plateforme de modélisation, de conception et de simulation des systèmes embarqués hétérogènes développée par DEECS [2] de l'université de Berkeley en Californie, et validés par simulation. Les résultats de la simulation ont montré le bon fonctionnement des composants domaine-polymorphes.

Comme nous avons choisi la sémantique réactive synchrone pour la spécification des comportements internes, nous avons développé un traducteur permettant de générer des composants Ptolemy II à partir des modules Esterel. Les composants générés sont de deux types :

- Composant domaine-spécifique synchrone. Dans ce cas, l'hétérogénéité entre la sémantique de spécification et celle du domaine hôte est gérée automatiquement lors de la phase de traduction ;
- Composant Noyau synchrone à enfouir dans un composant Conteneur faisant partie de l'ensemble de composants du composant domaine-polymorphe. Dans ce cas, l'hétérogénéité entre la sémantique de spécification et celle du domaine hôte est gérée par l'architecture de composant domaine-polymorphe.

[2] Department of Electrical Engineering and Computer Sciences.

1.4 Organisation du mémoire

Ce mémoire se compose de sept chapitres dont les cinq premiers sont consacrés à l'étude théorique et les deux derniers sont consacrés à l'intégration, la validation et la simulation dans Ptolemy II. Voici le détail de l'ensemble des chapitres :

1. Le premier chapitre est un état de l'art consacré à la conception des systèmes embarqués. Nous commençons d'abord par donner les définitions de tous les concepts nécessaires pour cette thèse. Ensuite, nous présentons les approches de conception des systèmes embarqués hétérogènes qui existent actuellement. Finalement, nous présentons la problématique traitée dans cette thèse, et cela, après avoir présenté la méthodologie acteur que nous avons choisie pour la réalisation de notre travail ;
2. Le deuxième chapitre est consacré à la conception de composants domaine-spécifiques synchrones. Nous présentons d'abord les deux techniques permettant d'utiliser le comportement synchrone dans des composants domaine-spécifiques puis nous justifions le choix de la technique à utiliser. Ensuite, pour plusieurs modèles de calcul, nous donnons la spécification de l'interface gérant l'hétérogénéité entre le comportement synchrone du composant domaine-spécifique et le modèle de calcul du domaine hôte. Finalement, nous dégageons les règles générales pour la conception des composants domaine-spécifiques ayant des comportements spécifiés suivant d'autres sémantiques ;
3. Dans le troisième chapitre, nous exposons les faiblesses engendrées par l'utilisation des composants domaine-spécifiques pour exécuter un comportement interne suivant la sémantique de spécification sous différents modèles de calcul. Ensuite, nous donnons la définition et les propriétés du composant domaine-polymorphe ainsi que les objectifs souhaités et les contraintes à respecter lors de la conception de ce dernier. Nous terminons par la définition du polymorphisme du point de vue génie logiciel suivie par des explorations et des discussions des technologies du le génie logiciel jugées utiles pour ce polymorphisme ;
4. Le quatrième chapitre est consacré à la conception du composant domaine-polymorphe. Il détaille toutes les étapes de la conception ainsi que toutes les améliorations et les spécifications permettant d'avoir le modèle final de composant domaine-polymorphe ;
5. Dans le cinquième chapitre, nous montrons plutôt comment utiliser le composant domaine-polymorphe pour la conception de systèmes embarqués hétérogènes. Pour cela, nous avons donné la spécification du composant domaine-polymorphe pour les modèles de calcul SR (Synchronous Reactive), DE (Discrete Event) et SDF (Synchronous Data Flow). En effet, pour chaque modèle de calcul, nous avons spécifié l'adaptateur et le conservateur du composant domaine-polymorphe. Finalement, nous avons présenté des cas de mélange des modèles de calcul ci-dessus, à base de composants domaine-polymorphes ;
6. Le sixième chapitre est consacré à l'intégration de nos concepts dans Ptolemy II. Nous commençons d'abord par un aperçu de Ptolemy II et nous détaillons ensuite comment le composant domaine-polymorphe a été intégré dans Ptolemy II ainsi que dans Vergil. Ce dernier est l'interface graphique de Ptolemy II ;
7. Finalement, dans le dernier chapitre, nous exposons la validation de nos concepts par simulation. Pour cela, nous avons pris le problème de l'atelier de production automatisé, proposé par FZI (Forschungszentrum Informatik de Karlsruhe), comme exemple à concevoir à base de composants domaine-polymorphes ;

A L'annexe A présente notre outil de traduction de modules Esterel en composants, en montrant comment nous générons des composants domaine-spécifiques synchrones et des composants Noyaux synchrones à partir de modules Esterel.

Première partie

Conception des Systèmes et Composants Hétérogènes

Chapitre 2

Conception des Systèmes Embarqués Hétérogènes

Ce chapitre commence par la définition des systèmes temps réel, des systèmes embarqués ainsi que des modèles de calcul. Ces définitions permettent, ensuite, de présenter les approches qui existent actuellement et sur lesquelles la conception des systèmes doit se baser pour faire face au problème de l'hétérogénéité. Finalement, nous exposons la problématique que cette thèse traite, et cela après avoir présenté les outils ainsi que la méthodologie de conception les plus utilisés actuellement.

2.1 Système temps réel

Il existe plusieurs définitions décrivant les systèmes temps réel. Toutes ces définitions ont la notion de temps comme élément déterminant, parmi elles, nous citons :

- Un système temps réel est un système dont la fonction est de contrôler son environnement en recevant des données, en les traitant et en produisant des résultats ou une action en un temps suffisamment rapide [5] ;
- Un système temps réel est tout système qui doit répondre à des entrées dans un délai de temps fini et spécifié [6] ;
- Un système temps réel est 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.

Un système temps réel est donc un système qui est généralement destiné à observer et agir sur son environnement, qui lui est extérieur, tout en respectant un délai de réponse fini et bien spécifié.

En effet, la contrainte de temps est très importante et tout dépassement ou retard par rapport au délai de réponse conduit donc à un déphasage du fonctionnement du système par rapport à son environnement, ce qui implique des erreurs de traitement. Selon l'environnement dans lequel le système évolue, les erreurs de traitement conduisent à des conséquences de plus ou moins graves. Par exemple, lors de l'atterrissage d'un avion, le système d'indication d'altitude doit fournir des valeurs exactes et à un délai très précis pour que les réactions du pilote, qui dépendent de l'altitude, aient lieu au bon moment, sinon tout retard de réponse conduira à l'écrasement de l'avion.

L'interaction d'un système temps réel avec son environnement se passe à travers des capteurs et des actionneurs. En effet, la réaction du système passe par trois étapes : *acquisition des données* à travers les capteurs, *traitement (dit aussi calcul)*, et finalement *envoi de réponse* à travers les actionneurs. La réalisation de ces trois étapes ne doit pas dépasser un certain délai.

La contrainte de temps permet de distinguer deux types de systèmes temps réel :

Les systèmes temps réel à contraintes souples : Ce type de système est moins exigeant et tolère certaines violations raisonnables de contraintes de temps. Autrement dit, avec ce type de système, une faible probabilité de ne pas respecter des contraintes temporelles peut être tolérée. En effet, les tâches temps réel souples peuvent répondre avec un certain retard sachant que la cohérence du système en sera dégradée, sans induire des conséquences catastrophiques sur l'environnement contrôlé. Par exemple, pour un système vidéo, si une image est affichée en retard ou une trame de données transmises sur une ligne téléphonique est perdue, cela ne remet pas en cause le fonctionnement correct de l'ensemble du système.

Les systèmes temps réel à contraintes strictes ou critiques : Ce type de système ne tolère aucune violation de contrainte de temps et la gestion de ce dernier est très stricte. Autrement dit, ce sont des systèmes soumis à des contraintes de fiabilité et le non respect d'une contrainte temporelle cause une erreur de fonctionnement pouvant entraîner des dégâts catastrophiques : perte de vies humaines, destruction de matériel, impact financier, etc. Par exemple, le contrôle de processus industriels sensibles (dans les centrales nucléaires, aéronautique, etc.), la bourse, la médecine assistée par ordinateur font partie des systèmes à contraintes strictes.

2.2 Systèmes embarqués

Un système embarqué, dit aussi système enfoui, est un dispositif matériel comportant des parties logicielles dont le fonctionnement est par nature en temps réel [7]. Il est embarqué (enfoui) dans un dispositif plus large, qui n'est pas forcément un ordinateur et constitue son environnement [8]. Il réagit aux stimuli provenant de cet environnement à travers ses capteurs et ses actionneurs, voir figure 2.1. Les capteurs permettent donc de détecter les variations de l'environnement et transmettent ensuite les données concernant ces variations au système, et les actionneurs permettent d'agir sur l'environnement en transformant les réponses en actions.

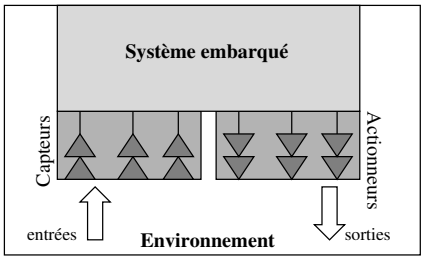


FIG. 2.1: Système embarqué dans son environnement

Les systèmes embarqués sont de plus en plus présents dans notre environnement et, selon leur utilisation, sont classés en deux catégories : systèmes embarqués personnels et systèmes embarqués collectifs. Un système embarqué personnel est un système enfoui dans des dispositifs utilisés par une personne : micro-onde, téléphone portable, chronomètre, etc. Par contre, un système embarqué collectif est un système destiné a être enfoui dans des dispositifs à usage collectif : avion, tour de contrôle de navigation aérienne, bateau, etc.

2.2.1 Exemple d'un système embarqué

La figure 2.2 montre un exemple de système embarqué. Il s'agit d'un contrôleur de la pression et de la température des pneus [29]. Ce système est créé par deux ingénieurs toulousains, Philippe Lefaure et Dominique Luce.

Le système est composé de deux capteurs, un pour chaque roue, qui pèsent à peine 24 g. Ils sont suffisamment compacts pour s'installer dans les jantes de moto et communiquent par radio avec un afficheur installé au guidon. Le système indique non seulement la pression du pneu, compensée en fonction de la température extérieure, mais aussi la température du pneu. Il y a aussi une détection de crevaison lente par la mesure des variations de pression.

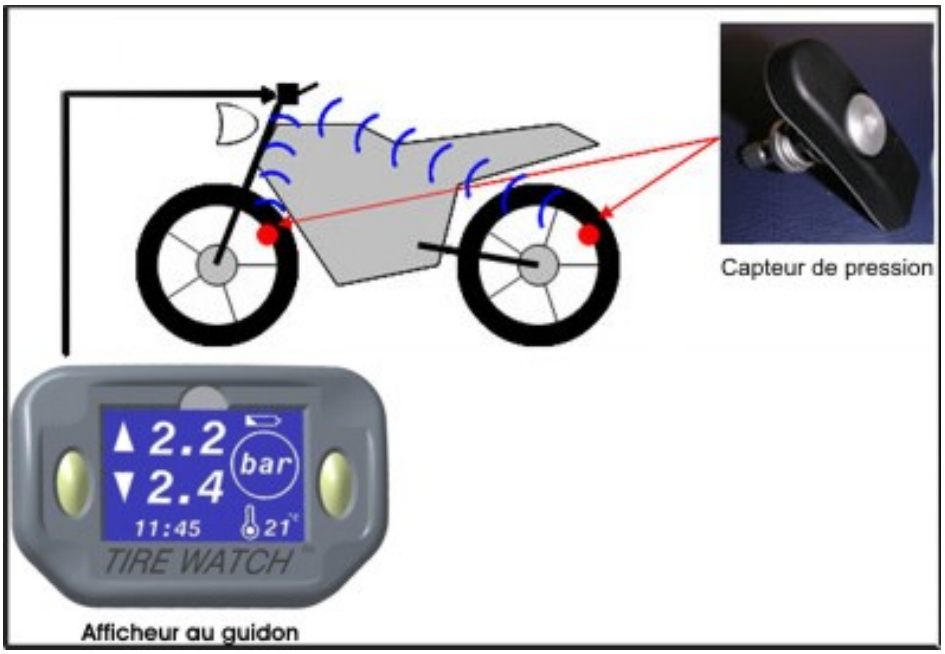


FIG. 2.2: Un système embarqué pour contrôler la pression et la température des pneus.

Pour chaque roue, un module électronique est disposé dans le fond de jante, sous le pneu. Il est maintenu par un collier ou intégré à la valve de gonflage. Ce module intègre des capteurs électroniques qui mesurent en permanence la température et la pression à l'intérieur du pneu. Un émetteur radio transmet l'information à un calculateur qui traite ces informations et les affiche pour le conducteur.

2.3 Conception des systèmes embarqués

Dans cette section, nous présentons les concepts indispensables pour la modélisation et la conception des systèmes embarqués.

2.3.1 Modèle, modélisation et conception

Un modèle peut être mathématique, dans ce cas, il peut être vu comme un ensemble d'assertions concernant les propriétés d'un système, à savoir sa fonctionnalité, ses dimensions physiques, etc. [8]. Il peut aussi être constructif, dans ce cas, il définit une procédure informatique qui simule l'ensemble des propriétés du système.

Les modèles constructifs, appelés aussi *modèles exécutables*, sont souvent utilisés pour décrire le comportement d'un système afin de répondre aux stimuli provenant de son environnement [8].

Des modèles d'exécution 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. Ceci est particulièrement vrai pour les logiciels embraqués [8].

Dans [11], l'auteur voit un modèle comme une abstraction d'un système qui est plus simple que celui-ci. L'abstraction signifie la négligence des détails de réalisation du système : seuls ses propriétés pertinentes et son comportement sont considérés. Il ajoute ensuite, que quelle que soit leur forme, les modèles partagent tous la 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 des charges, à détecter les erreurs de conception avant la réalisation du système, et éventuellement contribuera à déterminer certains paramètres du système tels que les paramètres de contrôle/commande. L'analyse et la validation sont donc des étapes primordiales, notamment dans les systèmes embarqués. L'analyse et la validation doivent donc garantir que le système, une fois réalisé, se comporte correctement d'un point de vue fonctionnel et temporel.

Actuellement, il existe deux manières pour valider un modèle :

- **La validation par preuve formelle :** Cette manière de valider fait appel à des méthodes formelles, comme le model checking [13] ;
- **La validation par simulation :** Par contre, cette manière consiste à exécuter le modèle et observer son comportement. Elle fait appel à des environnements logiciels, qui sont des environnements de simulation capables de reproduire un contexte d'exécution proche de l'environnement du système à valider, et d'exprimer les différentes conditions dans lesquelles peut se retrouver le système. Pour ce faire, les environnements de simulation adoptent généralement un modèle de temps logique, appelé temps de simulation. Ce dernier fournit une référence temporelle avec des unités pouvant être élastiques afin de simuler certaines conditions ou situations réelles du système.
  Certains environnements de simulation intègrent des outils de visualisation afin d'offrir un meilleur rendu du comportement d'un modèle et permettent de donner une vue globale du fonctionnement prévisionnel d'un système.

La modélisation consiste à partitionner et représenter formellement un système à haut niveau en plusieurs sous-systèmes sans prendre en considération les détails d'implémentation (dits détails de réalisation) [8].

La conception est par contre l'acte de définition d'un système ou d'un sous-système [8][20]. Dans [12], l'auteur définit la conception comme 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.

Dans le cadre de cette thèse, nous définissons la conception comme *une activité créatrice qui consiste à élaborer une implémentation (modèle exécutable) en partant des besoins exprimés, des moyens existants et des possibilités technologiques*.

Pour montrer que la modélisation et la conception sont corrélées, nous définissons cette dernière en fonction de la première comme suit : La conception est la modélisation raffinée jusqu'à obtenir un modèle exécutable sur la plateforme ciblée. De plus, dans [8], les auteurs affirment que la modélisation et la conception sont évidemment fortement liées. Dans certaines circonstances, les modèles peuvent être immuables, dans le sens qu'ils décrivent des sous-systèmes, des contraintes ou des comportements qui sont extérieurement imposés à une conception. Par exemple, ils peuvent décrire un système mécanique qui est déjà conçu, mais qui doit être commandé par le système électronique à concevoir.

2.3.2 Modèles 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 [12].

Un modèle exécutable, vu dans la section 2.3.1, est construit sous un modèle de calcul (*Model of Computation, MoC*), qui est l'ensemble de *lois physiques* régissant l'interaction des composants dans le modèle. Généralement, un modèle de calcul est un ensemble de règles qui sont plus abstraites. Ces règles fournissent un cadre dans lequel les concepteurs construisent un modèle. L'ensemble des règles qui régissent les interactions entre les composants est appelé *la sémantique du modèle de calcul*. Ainsi, un modèle de calcul peut avoir plus d'une sémantique car il peut exister des ensembles distincts de règles qui imposent les mêmes contraintes sur le comportement [8].

Un modèle de calcul peut être temporisé, dans ce cas il implémente une notion de temps, comme il peut être non temporisé, dans ce cas il n'en implémente pas.

Dans tous les cas, un modèle de calcul doit expliciter le type du comportement du système pendant son traitement (cyclique, réactif, concurrent, séquentiel, etc.). Il doit aussi expliciter le mécanisme d'interaction, qui est un protocole de communication, du système avec son environnement. Ce mécanisme peut être le rendez-vous, le passage de messages, l'échange d'événements, etc.). Finalement, il doit définir le format des données échangées : requêtes, flux, etc.

Actuellement, il existe plusieurs modèles de calcul utiles pour concevoir les systèmes embarqués [14] et d'autres sont en cours de spécification. Cependant, la spécification de nouveaux modèles de calcul est influencée par la variété des domaines d'application des systèmes embarqués et la croissance de leur complexité.

Le choix du modèle de calcul pour la conception se fait en fonction du type de système à concevoir. Donc, un mauvais choix de modèle de calcul lors de la conception d'un système embarqué va sûrement mener le concepteur à une implémentation moins fiable, pour ne pas dire incorrecte. Par exemple, pour la conception d'un système purement informatique qui a comme fonction la transformation d'une structure de données en une autre, c'est la sémantique impérative (commune aux langages C, C++, Java, etc.) qui sera adéquate. Par contre, pour la conception d'un système mécanique, c'est la sémantique temps continu qui sera adéquate afin de prendre en compte la manipulation du temps continu et la concurrence.

Dans les sous-sections suivantes nous présentons les modèles de calcul suivants : Flot de données (DF : Data Flow), Flot de données synchrones (SDF : Synchronous Data Flow), Temps Continu (CT : Continuous Time), Evénements 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 FD (Flot de Données) est adapté aux systèmes qui interagissent avec leur environnement en échangeant des flots de données [15] [16] [17] [18] [10]. Chaque entrée du système est équipée d'une file d'attente permettant la réception des suites de données. Une fois que les entrées sont lues, le système effectue un traitement, produit de manière asynchrone des suites de données, qui seront déposées sur les files d'attente de sortie, et termine par la mise à jour de son état.

Le modèle de calcul SDF (Flot de Données Synchrones) est un cas particulier du modèle de calcul Flot de Données [76] [18]. Il est adéquat pour la représentation des systèmes qui consomment et

produisent des suites de données de tailles fixes. Avec ce modèle de calcul, il est possible de spécifier les conditions permettant d'éviter au système de rentrer dans des interblocages ou de congestionner car les tailles des suites de données consommées et produites sont connues d'avance. Avec le modèle de calcul Flot de Données Synchrones, les composants du système peuvent être ordonnancés statiquement [76].

Listing 2.1: Comportement suivant le modèle de calcul SDF

```
initialiser <etats>;
repeter
  attendre <N donnees>;
  calculer <sorties>;
  modifier <etats>;
jusqua <arret systeme>
```

Evénements Discrets

Le modèle de calcul Evénements Discrets [19][10][79] est adapté aux systèmes qui réagissent à des événements discrets ordonnés dans le temps. Un événement est un couple d'une donnée et d'une estampille de temps, dite aussi date [14]. En effet, un événement peut être produit par une variation de l'environnement à une date donnée, mais peut aussi correspondre à un top d'horloge dans le cas d'un système piloté par une horloge.

L'arrivée d'un événement provoque l'activation du système. Dans ce cas, ce dernier effectue son traitement et met à jour l'environnement par génération d'événements à la date courante (événements instantanés) ou dans le futur. Par contre, en absence d'événements, le système se retrouve dans un état oisif où il ne fait rien jusqu'à la variation de l'environnement.

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 d'événements du système totalement ordonné.

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, etc.

Listing 2.2: Comportement suivant le modèle de calcul DE

```
initialiser <etats>;
repeter
  attendre <evenement(s)>;
  calculer <sorties>;
  modifier <etats>;
jusqua <arret systeme>
```

Temps Continu

Le modèle de calcul Temps Continu [122][10][78] est adapté aux systèmes utilisés pour la spécification des circuits analogiques, mécaniques, etc. Ces systèmes sont dits à dynamique continue et leurs composants interagissent par l'intermédiaire de signaux à temps continu. En effet, ce type de système est souvent représenté par des équations différentielles. Ainsi, sa simulation consiste à résoudre numériquement les équations différentielles. Les techniques usuelles d'intégration déterminent un pas d'intégration permettant d'obtenir une précision donnée sur la valeur des fonctions du temps. Lorsque

des événements sont déclenchés par des conditions sur la valeur de ces fonctions, la précision sur la date des événements discrets est obtenue par itération de l'intégration sur des pas de plus en plus petits.

Listing 2.3: Comportement suivant le modèle de calcul CT

```
initialiser <etats>;
repeter
  repeter
    calculer <point fixe>;
  jusqua <point fixe trouve>
  calculer <sorties>;
  modifier <etats>;
jusqua <arret systeme>
```

Synchrone/Réactif

Le modèle de calcul Synchrone/Réactif [97][99][21][22][23][10][24] est adapté aux systèmes qui sont fortement synchronisés avec leur environnement. En effet, le rythme de fonctionnement de ce type de système dépend toujours de la variation de son environnement, et cela, pour qu'il soit (le système) toujours capable de répondre à tous les stimuli de son environnement. Ce type de système est présenté en détail dans le chapitre suivant.

Processus Séquentiel Communicants

Le modèle de calcul Processus Séquentiel Communicants [119][120][80][10] est adapté aux systèmes s'exécutant concurremment et pouvant avoir besoin de se synchroniser pour communiquer à des points précis dans leur calcul. Ces points de synchronisation son appelés *rendez-vous*.

Ce type de système est utilisé dans des applications avec partage de ressources. Le problème *Dining Philosophers*, proposé par Edsger et W. Dijkstra en 1965, est un bon exemple de système à représenter sous le modèle de calcul CSP.

Au cours de 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. En effet, le système ne quitte pas son point de rendez-vous tant que les systèmes concernés par ce rendez-vous ne sont pas présents.

Listing 2.4: Comportement suivant le modèle de calcul CSP

```
initialiser <etats>;
repeter
  attendre <rendez-vous>;   //pour lire
  calculer <sorties>;
  modifier <etats>;
  attendre <rendez-vous>;   //pour ecrire
jusqua <arret systeme>
```

Machines à Etats Finis

Le modèle de calcul Machine à Etats Finis est adapté aux systèmes dont les états sont discrets et finis [10] et dont l'espace d'état peut être modélisé explicitement.

Les systèmes qui se comportent suivant ce modèle de calcul sont des systèmes qui disposent d'une logique de contrôle, notamment dans les contextes critiques où ces systèmes doivent assurer un changement de mode, tel que le passage en mode dysfonctionnement d'une partie du système.

En effet, au cours de son traitement, ce type de système transite d'un état à un autre et son comportement est guidé par des événements. L'arrivée d'un ou plusieurs événements permet au système d'exécuter une ou plusieurs actions. La transition d'un état à un autre est conditionné par des expressions booléennes, appelées *gardes*.

Le modèle FSM a été enrichi afin de prendre en compte des contraintes posées par la complexification des systèmes embarqués (par exemple les traitements concurrents), tel que le modèle Machines à Etats Finis Concurrents [26][25][27].

2.4 Conception hétérogène

Actuellement, la plupart des systèmes embarqués sont des systèmes hétérogènes, c'est-à-dire, des systèmes caractérisés par la coexistence d'un grand nombre de sous systèmes de types et de fonctions divers.

La figure 2.3, prise de [28], illustre un système typique hétérogène. Il est composé de plusieurs sous systèmes où chacun d'eux peut avoir une partie matérielle ou logicielle. Par exemple, un système peut contenir des sous systèmes programmables comme des microprocesseurs, ainsi que des sous systèmes analogiques comme des convertisseurs (Analogique/Numérique, Numérique/Analogique) et des capteurs ou encore des sous systèmes mécaniques.

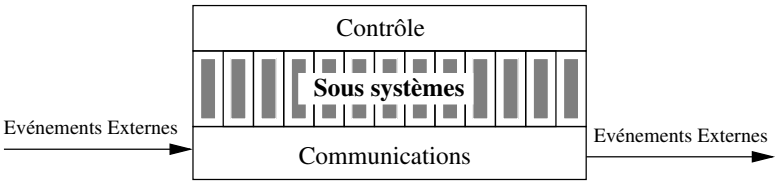


FIG. 2.3: Système hétérogène

L'activation de l'ensemble de sous systèmes est prise en charge par un modèle d'exécution et suivant un ordonnancement approprié. Tandis que la couche communication permet à ces sous systèmes de communiquer des données.

A toutes les étapes de la conception, un système embarqué hétérogène peut être vu comme un ensemble de modules communicants dont chacun est représenté par un modèle sous un modèle de calcul qui lui est approprié.

Chaque module a sa propre interface de communication. Cette dernière est vue comme un ensemble contenant deux sous ensembles [12]. Le premier sous ensemble contient des ports de communication permettant au module de communiquer avec l'extérieur. Le deuxième sous ensemble regroupe l'ensemble d'opérations applicables sur les ports et spécifiant les interactions entre le module et son extérieur. Lorsque ces opérations sont effectuées de l'intérieur du module contenant le port, on les appelle opérations internes, et lorsqu'elles sont effectuées par un module externe, on les appelle

opérations externes. Par exemple, Ptolemy II [8][9][10] définit un ensemble d'opérations pour manipuler les ports : hasToken(), get() et send() sont des opérations internes, et hasRom() est une opération externe.

Les modules sont ainsi interconnectés par des canaux de communication. Ces derniers sont des unités qui servent comme un conteneur des communications entre les modules. La figure 2.4 montre deux modules, Module1 et Module2, interconnectés via deux canaux de communication, C1 et C2.

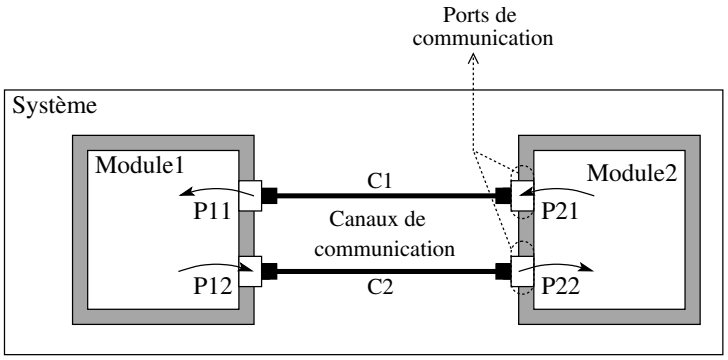


FIG. 2.4: Modules communicant via des canaux de communication.

Un canal de communication offre un ensemble de primitives de communication (services) qui sont appelés par les modules pour communiquer. Il peut avoir plusieurs formes qui dépendent du niveau d'abstraction des communications entre les modules. Au niveau application, il a la forme d'un canal abstrait et peut être représenté par un RPC (appel de procédure à distance), dans ce cas il offre des primitives de communication de haut niveau (par exemple send, receive, put, get, etc.). Au niveau physique, il est sous forme d'un bus physique, dans ce cas, il offre des primitives de communication spécifiques au bus physique.

Les primitives offertes par un canal de communication permettent d'encapsuler les détails de communication et leur réalisation est assurée par un protocole de communication. La définition des protocoles de communication est réalisée dans une bibliothèque de communication. Cette dernière peut contenir plusieurs réalisations différentes du même protocole parmi les lesquelles le processus de synthèse choisit celle permettant d'avoir les primitives de communication adéquates au canal [12].

Comme un système embarqué est par sa nature hétérogène, la conception d'un tel système ne s'échappera pas de l'organisation du système et l'interconnexion de l'ensemble de ses modules, ce qui entraîne une mise en communication des modules obéissant aux modèles de calcul différents, ce qui implique la non utilisation du même protocole de communication. En effet, *la conception des systèmes embarqués doit faire face au problème d'hétérogénéité*.

Pour faire face à ce problème, c'est-à-dire de faire coexister des modèles de calcul différents au sein du même modèle représentant un système, la communauté scientifique propose des approches hétérogènes pour concevoir des systèmes embarqués hétérogènes complexes. En général, il existe trois approches : l'approche amorphe [20][64][82], dans ce cas il s'agit de la conception hétérogène amorphe, l'approche hiérarchique [108] [27] [33] [25] [35] [36] [37] [38] [32] [73], dans ce cas il s'agit de la conception hétérogène hiérarchique, et l'approche non hiérarchique [30] [31] [39] [12] [71] [72], dans ce cas il s'agit de la conception hétérogène non hiérarchique.

2.4.1 Conception hétérogène amorphe

L'hétérogénéité amorphe permet de faire coexister plusieurs modèles de calcul au même niveau. Ainsi, les composants obéissant à des modèles de calcul différents peuvent communiquer directement. Pour cela, l'approche amorphe implique qu'un composant doit incorporer les caractéristiques de multiples modèles de calcul et avoir donc de multiples interfaces de communication.

En général, cette approche ne se concentre que sur un petit nombre de modèles de calcul qui sont souvent le temps continu et le temps discret pour les systèmes mixant des signaux analogiques et discrets, et les machines à états finis et le temps continus pour les systèmes hybrides.

Puisque elle n'utilise qu'un nombre réduit de modèles de calcul, l'hétérogénéité amorphe fait l'union de ces modèles de calcul afin d'avoir une conjonction entre eux [73]. Ainsi, les composants obéissent simultanément à plusieurs modèles de calcul. Par exemple, un convertisseur de signal analogique/numérique qui peut être conçu comme un composant avec des ports d'entrée obéissant au modèle de calcul temps continu et des ports de sortie obéissant au modèle de calcul temps discret.

VHDL-AMS [44] et Simulink [50] font partie des outils de modélisation, de conception et/ou de simulation basés sur l'approche amorphe.

L'avantage de l'hétérogénéité amorphe c'est qu'elle permette l'intégration complète des modèles de calcul, c'est à dire qu'il n'y a pas de frontières entre les modèles de calcul. Cependant, elle souffre de pas mal de faiblesses que nous citons comme suit :

- Les protocoles de communication s'interagit d'une façon imprévisible avec le flot de contrôle car il n'y a pas une séparation claire entre le flot de contrôle et la communication ;
- Conception moins compréhensible, ce qui rend la validation plus difficile ;
- Pas de possibilité d'ajouter de nouveaux modèles de calcul. Donc, l'ensemble de modèles de calcul n'est pas ouvert.

2.4.2 Conception hétérogène hiérarchique

La conception hétérogène hiérarchique consiste à concevoir chaque module du système d'une manière indépendante. Au niveau conceptuel, le système peut être vu comme un schéma de blocks ou de composants reliés entre eux où chaque composant représente un module du système. Mais, le fait que le système est hétérogène, les composants conçus ont des sémantiques différentes car chacun d'eux obéit a un modèle de calcul différent des autres. Par conséquence, la communication entre les différents composants ne peut être lieu d'une façon directe. Cela revient à la non compatibilité des interfaces et protocoles de communication utilisés par les composants. En effet, l'approche hiérarchique impose de placer chaque composant différent dans un niveau différent de la hiérarchie, voir figure 2.5.

Pour que les données passent d'un niveau hiérarchique à un autre, il faut détailler d'une manière très fine comment les données, dont le format dépend du modèle de calcul du niveau hiérarchique, sont communiquées avec fidélité entre les différents niveaux et aussi de spécifier les interfaces au niveau des frontières entre les niveaux hiérarchiques.

Actuellement, la plupart des plateformes de modélisation et de conception des systèmes embarqués hétérogènes utilisent un nombre limité de modèles de calcul, qui sont généralement temps continu, événements discrets (périodique ou non périodique), flots de données et les machines à états finis. Cependant, il existe des plateformes qui supportent un nombre extensible de modèles de calcul, tel est le cas de Ptolemy II [8][9][10].

Par contre, toutes les plateformes se basent sur l'approche hiérarchique afin de combiner différents modèles de calcul. Cependant, elles *n'offrent pas un passage explicite entre les niveaux hiérarchique*, tel est le cas de Ptolemy II [8][9][10], El Greco [41], OMOLA [40], SystemC [43], ROSETA [45], POLIS [46], DYMOLA [47], MODELICA [48], SpecC [42].

La figure 2.6 montre la représentation d'un système embarqué hétérogène suivant l'approche hiérarchique. Le niveau 1 de la hiérarchie ne comporte que les composants obéissant au modèle de calcul A et les composants composites encapsulants les autres composants, qui obéissent à d'autres modèles de calcul différents de A. Ainsi, les composants composites sont raffinés par d'autres schémas de composants dont chacun d'eux sera placé dans un niveau vers le bas de la hiérarchie, tel est le cas

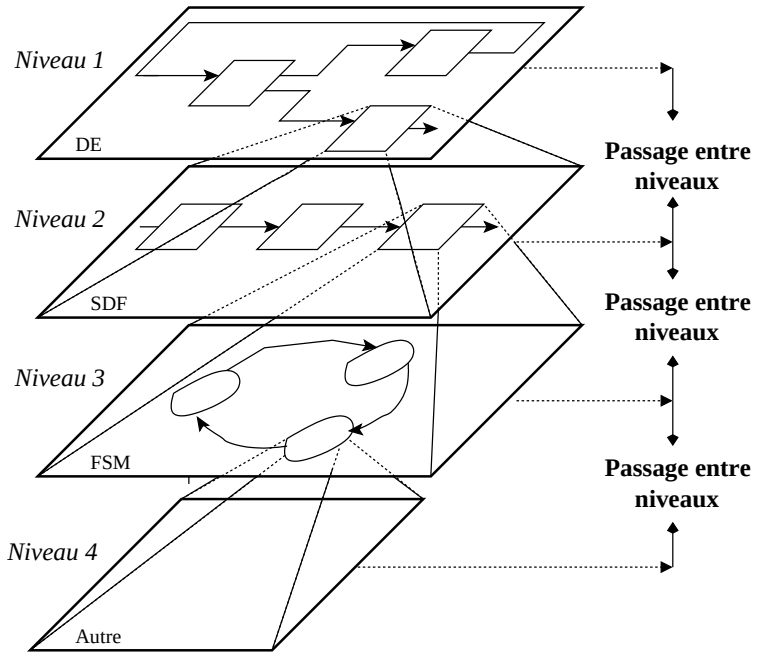


FIG. 2.5: Conception basée sur l'approche hiérarchique

du composant X (resp. Y) raffiné au niveau 2 de la hiérarchie par un schéma de composants sous le modèle de calcul B (resp. C).

Lorsque le contenu d'un composant représentant un module du système est un schéma de composants, c'est-à-dire un composant raffiné en un schéma de composants, on l'appelle composant composite, tel est le cas des composants X et Y sur la figure 2.6. Par contre, lorsque son contenu est une description de son comportement sous forme d'un ensemble d'opérations (qui peuvent être spécifiées par un langage de haut niveau), on l'appelle composant atomique.

L'hétérogénéité hiérarchique a beaucoup d'avantages. Dans [12], l'auteur a détaillé ces derniers comme suit :

- Elle permet de contrôler et de maîtriser la complexité des systèmes. Elle considère un système complexe comme un ensemble composé des parties élémentaires ;
- De point de vue syntaxique, elle permet d'abstraire un réseau de composants obéissant au même modèle de calcul dans un seul composant (composant composite). Ce qui permet aux concepteurs de voir le systèmes au niveau de détail désirable ;
- Elle permet d'organiser l'hétérogénéité d'une façon structurée qui permet ainsi de faciliter des raffinements successifs ;
- L'ensemble de modèles de calcul est ouvert. Donc, il est possible d'ajouter de nouveaux modèles de calcul.

Cependant, elle possède plusieurs limites. Dans [12], on trouve la citation de ces limites classées suivant trois considérations :

- Du point de vue des outils de modélisation et de conception, 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 ;

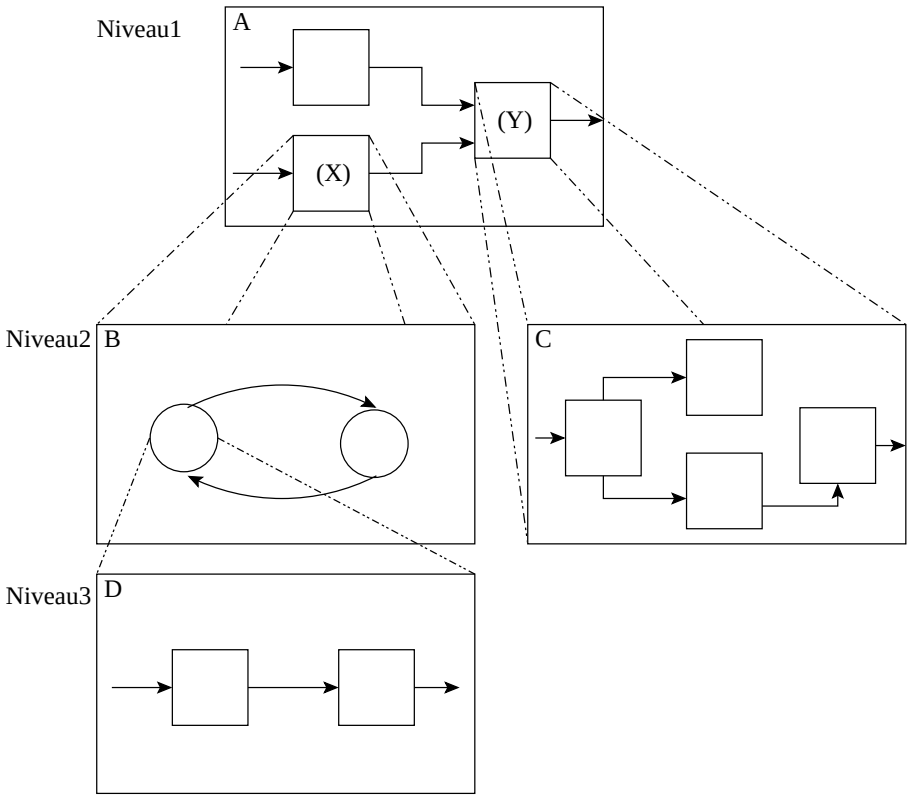


FIG. 2.6: Exemple de conception hétérogène hiérarchique

– 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 ;
– Pour le concepteur du système, il ne peut pas explicitement intervenir sur les transformations des données qui se produisent à la frontière de deux niveaux hiérarchiques(lors du passage d'un modèle de calcul à un autre).
  Le passage implicite des données entre les modèles de calcul est expliqué comme suit :
  Comme tout système hétérogène met en communication différents modèles de calcul dont la représentation est souvent non claire, nous adoptons la même heuristique que celle adoptée par [12] et qui consiste à considérer un système élémentaire composé que de deux composants atomiques, A et B, dont chacun obéit à un modèle de calcul, respectivement D1 et D2, voir la figure 2.7(a). En effet, la conception de ce système suivant l'approche hiérarchique consiste à créer deux composants composites : Le composant global représentant le système et le composant BH. Le premier composant composite, qui est placé au premier niveau de la hiérarchie, contient le composant A et le composant composite BH ainsi les deux sont gouvernés par le modèle de calcul D1. Le deuxième composant BH, qui a un port d'entrée BHIn obéissant au modèle de calcul D1, encapsule le composant B qui est placé au deuxième niveau hiérarchique et gouverné par le modèle de calcul D2, voir figure 2.7(b).
  Le transfert des données du port AOut du composant A au port BIn du composant B se passe comme suit : d'abord, le modèle d'exécution approprié au modèle de calcul D1 s'occupe du transfert des données du port AOut au port BHIn en fournissant la sémantique de communication approprié. Ensuite, le composant BH délègue le transfert des données arrivées sur son port BHIn à son modèle d'exécution interne. Finalement, ce dernier transfert les données du port BHIn au port BIn. Ptolemy II fait partie des plateformes qui se base sur ce concept de transfert des données entre les niveaux hiérarchiques.

De plus, l'approche hiérarchique pose le problème du choix du modèle de calcul du niveau supé-

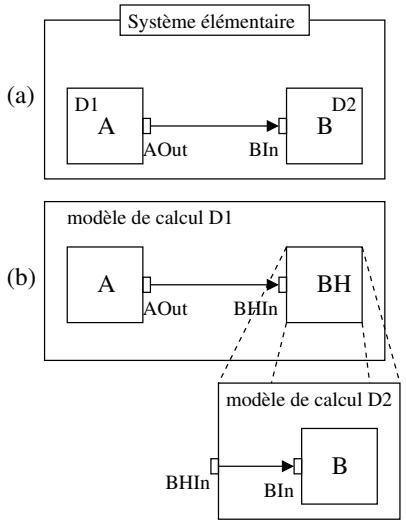


FIG. 2.7: (a) Système élémentaire. (b) La conception de (a) suivant l'approche hiérarchique.

rieur de la hiérarchie permettant de gouverner le comportement global du système.

Un système peut être dominé par le contrôle (réaction aux événements). D'après [108], on utilise l'un des modèles de calcul : DE, FSM, SR, etc. ou dominé par les données (les sorties dépendent fonctionnellement des entrées), dans ce cas on utilise l'un des modèles de calcul : PN, SDF, DF, etc. Mais, d'après [12], aucun modèle de calcul, qui est destiné à régir le comportement global du système, qu'il soit orienté contrôle ou orienté traitement des données ne peut entièrement satisfaire car, d'après l'auteur, le contrôle dépend des données entrantes et le traitement des données dépend du contrôle.

2.4.3 Conception hétérogène non hiérarchique

Le but de l'hétérogénéité non hiérarchique [12] n'est pas de bannir l'approche hiérarchique mais plutôt de donner des solutions aux limites de l'approche hiérarchique tout en permettant d'avoir un modèle hétérogène plat qui permet de changer de modèle de calcul sans changer de niveau hiérarchique, et cela, sans altération du système original. Elle permet aussi la conception des composants susceptibles de travailler à la frontière de plusieurs modèles de calcul (composants avec des ports d'entrée/sortie hétérogènes). De plus, le concepteur a la possibilité d'expliciter le passage des données et de contrôler le comportement du système au niveau des frontières, ce qui fait partie de la conception du système.

La conception des systèmes embarqués hétérogènes basée sur l'approche non hiérarchique consiste d'abord à concevoir chaque module du système en composant d'une manière indépendante. Ensuite, suivant leurs relations d'interconnexion au niveau du système original, elle interconnecte directement les composants homogènes (ceux obéissant au même modèle de calcul) tandis que les autres composants (ceux n'obéissant pas au même modèle de calcul) elle les interconnecte à travers des composants d'interfaces hétérogènes (HICs) [30]. A cette étape, la conception basée sur l'approche non hiérarchique permet d'avoir un modèle hétérogène plat [31][12] représentant le système, c'est-à-dire un modèle permettant la coexistance de différents modèles de calcul au même niveau.

La figure 2.8(a) montre un modèle hétérogène plat du système élémentaire de la figure 2.7(a). Les deux composants A et B sont interconnectés à travers le HIC H qui a un port d'entrée, Hin, obéissant au modèle de calcul D1 et un port de sortie, Hout, obéissant au modèle de calcul D2.

L'hétérogénéité sera gérée à l'exécution. Donc, au lancement de l'exécution du modèle hétérogène plat, l'approche non hiérarchique transforme le modèle original (modèle plat) en ajoutant un niveau à sa hiérarchie et construisant des sous ensembles homogènes à ce niveau afin d'isoler les différents

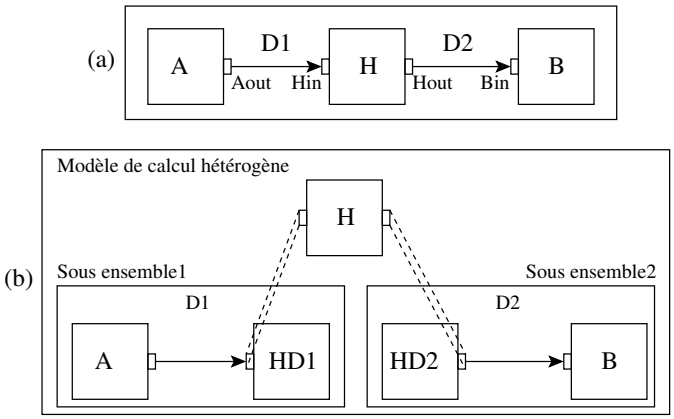


FIG. 2.8: (a) Le modèle hétérogène plat. (b) Partitionnement du modèle et projection du HIC : Système final comme exécuté.

modèles de calcul. Ainsi, deux couches se résultent, une couche supérieure et une couche secondaire contenant les sous ensembles homogènes. Pour contrôler la couche supérieure, l'approche non hiérarchique propose un modèle de calcul hétérogène permettant l'ordonnancement des sous ensembles homogènes et la délégation du calcul de leurs comportements à leurs modèles de calcul.

Comme le composant HIC a des ports de communication obéissants aux modèles de calcul différents, le modèle d'exécution hétérogène effectue une tâche cruciale lors de la transformation. Cette tâche consiste à projeter le composant HIC sur chacun des sous-ensembles qu'ils utilisent. Les composants représentant le composant HIC dans les sous ensembles homogènes sont appelés les projections du HIC [12].

Le composant projection d'un HIC appartenant à un sous ensemble homogène est un composant ayant une structure et un comportement qui respectent les caractéristiques du modèle de calcul de son sous ensemble homogène (c'est-à-dire que la projection du HIC est un composant comme les autres composants de son sous ensemble homogène) et son comportement est calculé par ce même modèle de calcul. Aussi, parmi l'ensemble de ports du HIC, un composant projection ne garde que les ports obéissant au modèle de calcul de son sous ensemble homogène tandis que les autres ports obéissant à d'autres modèles de calcul sont cachés.

La figure 2.8(b) montre le système final après la construction des sous ensembles homogènes et la projection du HIC H, c'est le résultat de la transformation du modèle hétérogène plat de la figure 2.8(a). Le composant A et le composant projection HD1 représentant le HIC sont isolés dans sous ensemble1 et gouvernés par le modèle de calcul D1. De même, le composant B et le composant projection HD2 représentant le HIC sont isolés dans sous ensemble2 et gouvernés par le modèle de calcul D2.

Avec l'approche non hiérarchique, l'ensemble de modèles de calcul est ouvert. Donc, il est possible d'ajouter de nouveaux modèles de calcul.

Malgré ses avantages, l'approche non hiérarchique est moins puissante que l'approche amorphe en ce qui concerne les boucles hétérogènes. Actuellement, avec l'approche hétérogène non hiérarchique il est impossible d'utiliser les boucles hétérogènes alors qu'il est possible avec l'approche amorphe.

2.5 Outils de conception des systèmes hétérogènes

Nous savons maintenant qu'un système embarqué hétérogène est composé d'un ensemble de modules communicants dont chacun d'eux obéit à un modèle de calcul. Aussi, nous avons dit que la

conception d'un tel système consiste à concevoir chaque module à part et l'intégration totale dudit système doit se baser sur l'approche hiérarchique ou l'approche non hiérarchique.

Pour qu'une conception puisse tenir compte des détails de l'implémentation et des contraintes à respecter tout au long des étapes de la conception, il faut que l'outil utilisé dans cette conception soit capable d'exprimer tous les détails et toutes les contraintes afin de satisfaire les besoins fixés au départ. Dans [12], l'auteur affirme qu'un langage idéal de description doive couvrir tout le flot de conception, tout en restant exécutable à tous les niveaux d'abstraction. Il ajoute, ce langage doit également tenir compte des aspects fondamentaux du système tels que la concurrence et l'aspect temporel. Malheureusement, ce langage n'existe pas.

En effet, la conception des différents modules du système peut impliquer l'utilisation de plusieurs langages où chaque module sera décrit par un langage approprié permettant la bonne spécification de ses contraintes (par exemple, certains langages sont spécialisés sur la description matérielle, d'autres sur la spécification système, etc.). Donc, le choix du langage approprié dépend de plusieurs critères (type du système à concevoir, la culture du concepteur, etc.). Dans [70], les auteurs ont donné une classification de différents langages ainsi que des critères de choix.

Dans le cas où la conception du système a utilisé différents langages, l'intégration de l'ensemble de modules dans un modèle (représentation) unifié(e) est nécessaire. Ainsi, cette approche de conception est coûteuse malgré qu'elle permette l'intégration de nouveaux langages de spécification.

Généralement, pour la conception et la simulation des systèmes embarqués hétérogènes, ce sont les plateformes de modélisation, de conception et de simulation qui sont utilisées.

Actuellement, il existe plusieurs outils dédiés à la conception des systèmes embarqués hétérogènes. Nous citons :

Music

L'environnement Music est conçu pour la conception de systèmes hétérogènes multilangages spécifiés au niveau système [56]. 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.

Polis

L'environnement Polis [51][52] permet la synthèse et la validation de systèmes temps réel orientés 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. Il utilise le modèle de calcul CFSM (machine d'états finis pour la coconception). Une CFSM est comme une FSM sauf qu'au niveau de communication la première (CFSM) utilise une communication GASL (globalement asynchrone localement synchrone) et la deuxième utilise une communication synchrone.

Metropolis

Metropolis [49] est un environnement de conception qui fournit une infrastructure pour la conception des systèmes embarqués hétérogènes à des niveaux d'abstraction différents.

Cossap

Cossap [55] est un environnement de conception de systèmes de traitement du signal proposé par Synopsys. Partant d'un modèle de spécification du système à haut niveau (saisi graphiquement par composition d'éléments de biblithèque), le découpage produit un modèle VHDL pour la partie matérielle et un modèle logiciel en C ou assembleur.

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 [53][54]. 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.

Simulink

Simulink [50] est environnement interactif pour la modélisation, l'analyse et la simulation des systèmes dynamiques. Il est une extension du langage MATLAB permettant la représentation des systèmes sous forme de schémas de blocks. Il utilise deux modèles de calcul : modèle de calcul à temps continu et modèle de calcul à temps discret.

Ptolemy II

Ptolemy II [8][9][10], proposé par le département EECS (Electrical Engineering and Computer Sciences) de l'université de Berkeley en Californie, est une plateforme de modélisation, de conception et de simulation des systèmes embarqués hétérogènes. C'est un projet de recherches non finalisé. Il est orienté composant et utilise une approche unifiée. Les composants (entités ou blocks), appelés *acteurs*, sont programmés en Java.

L'avantage de Ptolemy II est non seulement l'utilisation de plusieurs modèles de calcul mais aussi la permission aux utilisateurs d'intégrer de nouveaux modèles de calcul définis par eux même. Cela revient à sa structure qui est ouverte à l'ajout de nouveaux modèles de calcul.

LabVIEW

LabVIEW [57] de National Instruments est un environnement de développement graphique basé sur le langage de programmation G. Ce dernier se base sur le modèle de calcul flot de données.

LabVIEW est plus particulièrement destiné à l'acquisition de données et au traitement du signal. Il offre de larges possibilités de communication entre l'ordinateur et le monde physique (par cartes d'acquisitions analogiques ou numériques, etc.) ainsi que d'importantes bibliothèques mathématiques permettant de traiter les signaux mesurés.

SPW

SPW (Signal Processing Worksystem) de Cadence [58] est un outil de conception permettant la capture, la simulation et la vérification des signaux numériques. Il est particulièrement destiné à la conception des systèmes de communication numérique avec ou sans fil. Il utilise le modèle de calcul flot de données.

Easy5

EASY5 [59], développé par la compagnie Boeing, est un outil de modélisation, de conception et de simulation des systèmes dynamiques caractérisés par des équations différentielles et algébriques.

EASY5 utilise des outils de simulation non linéaires puissants et offre une librairie de blocks (composants : additionneur, diviseur, intégrateur, etc.) primitifs qui peuvent être utilisés dans l'assemblage des modèles représentants des systèmes.

Dans le cadre de cette thèse, nous avons choisi la plateforme Ptolemy II pour valider par simulation nos concepts. Les raisons justifiant notre choix sont les suivantes :

1. Ptolemy II utilise une approche unifiée, c'est-à-dire, il utilise un seul langage pour la modélisation et la conception des composants obéissant à différents modèles de calcul. Cela facilite l'intégration des différents modules (représentés par les composants) du système ;
2. Ptolemy II utilise un nombre important de modèles de calcul, ce qui n'est pas le cas avec les autres outils de conception qui, en général, proposent qu'un seul modèle de calcul. Cette raison justifie largement notre choix car il nous faut un bon nombre de modèles de calcul pour tester et montrer l'utilité du composant domaine-polymorphe.

2.6 Méthodologie de conception orientée acteur

2.6.1 Introduction

La conception basée composant a été considérée comme une importante approche pour la conception des systèmes embarqués hétérogènes complexes [61]. Cette approche consiste à décomposer un système en différents sous ensembles qui soient individuellement à la fois facilement maniable et spécifique à un domaine [12], ce qui permet aux concepteurs de diviser et dominer efficacement le problème de conception [62].

Actuellement, il existe plusieurs modèles de composant logiciel bien utilisés par la communauté académique que par la communauté industrielle, mais ne sont pas adéquats à la conception des systèmes embarqués hétérogènes, tel est le cas de CORBA de l'OMG (Object Management Group), qui est orienté vers tout ce qui est traitement spécifique à l'entreprise, de DCOM de Microsoft, qui est orienté vers tout ce qui est application de bureau (desktop application) et de JavaBeans de Sun MicroSystem, qui est orienté vers tout ce qui est application internet. Ces standards sont des modèles orientés objet distribués et ne permettent pas de prendre en compte certaines caractéristiques des systèmes embarqués, tel que l'hétérogénéité, la concurrence, etc. C'est pourquoi Edward Lee a proposé un nouveau modèle de composant dédié à la modélisation et la conception des systèmes embarqués hétérogènes, c'est le modèle d'acteur [60].

La modélisation et la conception orientées acteur [60] [63] [64] [74] est une méthodologie pour la conception au niveau système qui a évolué à travers plusieurs années de recherches.

La première apparition du terme acteur dans la conception des systèmes remonte à l'année 1970 où Carl Hewitt de MIT[1] l'a utilisé dans la description du concept des agents intelligents dont la communication est basée sur le passage de message asynchrone [65]. Ensuite, le terme a évolué à travers le travail de Gul Agha et d'autres qui ont développé une théorie formelle pour la description des systèmes concurrents qui combinent le passage de message asynchrone avec la mise à jour d'état interne [67] [66]. Les acteurs proposés par Gul Agha sont des acteurs autonomes dont chacun a son propre thread de contrôle indépendant et communique avec les autres par passage de message asynchrone.

Dans cette section, nous présentons le modèle d'acteur proposé par Edward Lee. Ce modèle d'acteur est lié aux travaux de Carl Hewitt et Gul Agha, mais n'a pas besoin d'avoir son propre thread de contrôle et la communication n'a besoin d'être forcement asynchrone [63].

[1] Massachusetts Institute of Technology

2.6.2 Concepts

Le principe de la méthodologie acteur consiste à décomposer un système du point de vue de ses *actions* [62] en unités élémentaires de fonctionnalité, appelées acteurs.

Un acteur est donc un composant ayant une interface bien définie qui différentie son état interne de son comportement et définit comment dudit acteur interagit avec son environnement. L'interface d'un acteur inclut des ports, qui représentent ses points de communication, et des paramètres, qui servent à la configuration de son comportement, voir figure 2.9. Dans [62], les auteurs définissent un acteur comme une encapsulation d'actions paramétrables effectuées sur des données d'entrée et produisent des données de sortie résultantes de son comportement.

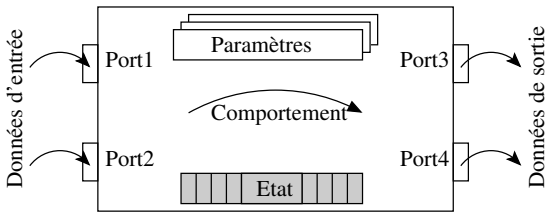


FIG. 2.9: Architecture d'acteur montrant ses paramètres, son état et ses ports de communication.

Les acteurs sont interconnectés via des canaux de communication qui leurs permettent de faire passer des données d'un port à un autre. Ils peuvent être composés avec d'autres acteurs afin de former des *acteurs composites* ou *modèles*. La figure 2.10 montre trois acteurs, dont deux sont atomiques et un est composite, interconnectés formant le modèle global. Le composant composite, qui est un sous modèle, est composé d'un acteur atomique et a des ports de communication externes permettant à son acteur interne de communiquer implicitement avec l'extérieur. Donc, le modèle global est un composant composite mais n'ayant pas des ports de communication externes car il est au niveau supérieur. En effet, un modèle peut être composé d'acteurs atomiques et/ou des sous modèles (acteurs composites).

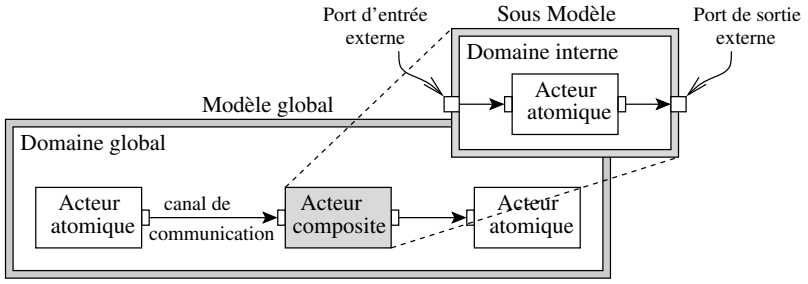


FIG. 2.10: Composition d'acteurs.

Comme pour les acteurs, un modèle peut aussi définir une interface externe, appelée l'abstraction hiérarchique [64][8]. Cette interface est composée des ports et des paramètres externes qui sont distincts des ports et des paramètres des différents acteurs composants le modèle. Les ports externes du modèle peuvent être interconnectés par des canaux à d'autres ports externes d'un autre modèle ou aux ports des acteurs composants le modèle. Les paramètres externes d'un modèle peuvent être utilisés pour déterminer les valeurs des paramètres des acteurs à l'intérieur du modèle [69].

Nous définissons ici la notion de *domaine* qui fait partie du nom du modèle de composant que nous allons proposer. Donc, un domaine est une zone dans laquelle les acteurs (atomiques et/ou composites) résident. Un domaine peut être *domaine global*, dit aussi *domaine principal*, dans ce cas, il est la zone dans laquelle les acteurs du modèle global résident, comme peut être *domaine interne*, dit aussi *sous*

domaine ou *domaine local*, dans ce cas, il est la zone dans laquelle les acteurs du sous modèle ou de l'acteur composite résident, voir figure 2.10.

Un domaine implémente un modèle de calcul qui régit les interactions de ses acteurs. Ainsi, sa sémantique est celle du modèle de calcul qui l'implémente. En effet, les domaines se diffèrent au niveau sémantique, c'est pourquoi on ajoute à leurs noms les noms des modèles de calcul qui les implémentent. Par exemple, un domaine contenant des acteurs gouvernés par le modèle de calcul CT est appelé domaine CT.

Un acteur s'exécute dans le domaine dans lequel il réside. Son exécution est donc contrôlée par son domaine à travers d'un ensemble d'opérations, dites opérations de contrôle (voir figure 2.11), qui sont généralement identiques à tous les acteurs. Ces opérations permettrent l'initialisation, l'activation et l'arrêt de l'acteur. En effet, le domaine et les acteurs dans lequel ils résident représentent un système.

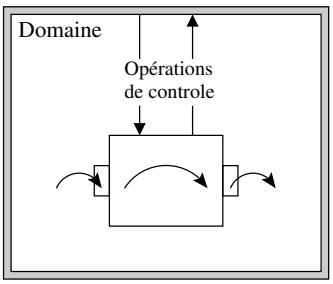


FIG. 2.11: Acteur contrôlé par son domaine.

2.6.3 Intérêts

Actuellement, l'orientation acteur est très répondue. Elle est utilisée par la plupart des plateformes de modélisation et de conception des systèmes embarqués hétérogènes. Par exemple la plateforme Ptolemy II est une plateforme orientée acteur.

Le succès de l'orientation acteur est du à :

– L'orientation acteur permet, par encapsulation, de cacher le comportement interne et l'état d'un acteur derrière son interface, ce qui permet de les rendre invisibles de l'extérieur. Cela permet donc de séparer le comportement interne de l'acteur de ses interactions avec les autres acteurs. En effet, la conception orientée acteur d'un système rend les propriétés comportementales des modèles de calcul indépendamment des propriétés des comportements internes des acteurs [68]. La séparation du comportement interne de l'acteur de son interaction permet donc la spécification de celui-ci par d'autres langages, tel que le langage Java, C/C++, etc, d'une façon indépendante des propriétés des modèles de calcul, à condition que les interfaces entre le comportement interne de l'acteur et ses ports doivent être spécifiées ;
– L'orientation acteur permet d'établir efficacement la séparation entre le flot de contrôle et le flot des données, les deux flots seront détaillés dans la section 2.6.4.

2.6.4 Structures de l'acteur et du modèle

Dans notre équipe, [12] a présenté les structures de l'acteur et du modèle. Nous reprenons donc intégralement cette présentation dans cette partie.

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.

Un acteur dispose d'un ensemble de variables noté A.X. Cet ensemble peut être subdivisé en deux sous-ensembles à savoir : le sous-ensemble de variables d'interface appelées ports et le sous-ensemble de variables internes noté S et contenant ses paramètres et ses états.

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. Nous avons vu qu'un modèle hiérarchique peut aussi définir une interface. De la même manière, un 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.

Définition *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 :

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

Où

$$M.S = \{\{\bigcup A_i\} \bigcup \{\bigcup C_i\}\}$$

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 :

$$M.Ctrl = \bigcup A_i.Ctrl$$

Avec $\alpha.Ctrl$ 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 ;
- *finish_trigger()*, l'opération utilisée par un acteur pour indiquer qu'il n'a plus d'opérations à exécuter pour un trigger, c'est-à-dire elle permet à un acteur de signaler la fin de ses activités.

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

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

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

$$A_i.trigger \subset A_i.Ctrl$$

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 :

$$M.Clbk = \bigcup A_i.Clbk$$

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

Après avoir choisi la plateforme pour intégrer et valider nos concepts, et, avoir présenté la méthodologie acteur sur laquelle la plateforme choisie se base, nous signalons donc que le modèle de composant domaine-polymorphe que nous allons concevoir dans cette thèse est un acteur atomique.

2.6.5 Primitives abstraites

Nous savons maintenant que tout modèle de calcul définit la sémantique de flot de contrôle sur les acteurs et la sémantique de communication entre leurs ports.

Au niveau abstrait un acteur est vu comme un composant abstrait ayant une interface composée de paramètres et de ports, et peut être atomique comme peut être composite. Tandis que tous les modèles de calcul définissent leurs sémantiques à travers un même ensemble de primitives.

En effet, cet ensemble de primitives n'est pas l'union des sémantiques, mais plutôt leur intersection [64]. Il est composé de primitives abstraites de flot de contrôle et de primitives abstraites de communication. Nous reprenons ici la même présentation de ces primitives donnée par [12].

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.

Primitives abstraites de communication

Les primitives abstraites présentées dans [64] 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.

2.7 Problématique traitée dans cette thèse

Un système embarqué hétérogène peut être vu comme un ensemble de composants (contrôleur et environnement) disposant chacun de sa propre interface de communication et obéissant à un modèle de calcul qui lui est approprié. En effet, la conception d'un tel système doit organiser et interconnecter l'ensemble de ces composants, ce qui entraîne une mise en communication entre des modèles de calcul différents utilisés respectivement par le contrôleur et son environnement.

Le composant contrôleur d'un système embarqué doit être spécifié (ou décrit) suivant une sémantique orientée contrôle (dite sémantique de spécification), par exemple, la sémantique DE (Discrete Event), SR (Synchronous Reactive) ou FSM (Finite State Machines). Le composant dédié au traitement de données doit également être spécifié suivant une autre sémantique orientée traitement de données, par exemple, la sémantique SDF (Synchronous Dataflow), DDF(Dynamic DataFlow), etc.

Donc, un composant contrôleur spécifié utilisant un modèle de calcul ne peut être utilisé que dans un environnement qui utilise le même modèle de calcul. Par contre, dans le cas où celui-ci est utilisé dans d'autres environnements qui implémentent des modèles de calcul différents, ceci peut entraîner des incompatibilités de communication.

Du point de vue de la modélisation et de la conception, cette logique reste la même pour un composant atomique modélisé dans un domaine utilisant une sémantique différente de sa sémantique de spécification.

Actuellement, pour utiliser le même composant atomique dans plusieurs domaines utilisant des sémantiques différentes de la sienne, on est obligé de gérer l'hétérogénéité entre la sémantique du composant atomique et celle de chacun des domaines (c'est à dire de conserver la sémantique du composant atomique au sein de différents domaines). Pour cela, pour chaque domaine, on doit traduire le composant atomique en un composant domaine-spécifique dont le comportement interne représente le comportement du composant atomique. Un composant domaine-spécifique a une structure et des ports de communication qui ne sont spécifiques qu'au modèle de calcul implémenté par son domaine.

Cependant, cette solution doit spécifier une interface à l'intérieur de chaque composant domaine-spécifique pour gérer l'hétérogénéité entre la sémantique de spécification du comportement interne et celle du domaine. La gestion de l'hétérogénéité entre la sémantique interne et celle du domaine revient à conserver la sémantique interne et respecter la sémantique du domaine (dite aussi sémantique externe).

L'utilisation des composants domaine-spécifiques pour conserver une sémantique au sein de différents modèles de calcul présente plusieurs désavantages, à savoir : une mise en œuvre très lourde, une exigence au concepteur d'avoir une parfaite connaissance des caractéristiques des modèles de calcul car la gestion de l'hétérogénéité entre la sémantique de spécification et celle du domaine est à sa (concepteur) responsabilité, la non réutilisabilité des composants que ce soit dans le même domaine ou dans différents domaines, la non lisibilité du code des composants, des mises à jour des composants très coûteuses, nombre de composants explosif.

Par ailleurs, il est possible d'utiliser les approches hiérarchique et non hiérarchique pour gérer l'hétérogénéité entre la sémantique de spécification et celle du domaine :

- En se basant sur l'approche hiérarchique pour encapsuler le composant atomique dans un composant composite implémentant le modèle de calcul approprié à la sémantique de spécification. Ensuite, par l'abstraction hiérarchique, on utilise le composant composite au sein du domaine. Ainsi, le composant composite obéit au modèle de calcul du domaine et exécute le composant atomique suivant le modèle de calcul interne.
  Cette solution présente quelques désavantages donnés dans la sous section conception hétérogène hiérarchique ;
- En se basant sur l'approche non hiérarchique pour utiliser le modèle de calcul approprié à la sémantique de spécification au même niveau que celui du domaine. Pour cela, on interconnecte le composant atomique avec les composants du domaine à travers un HIC. Ce dernier possède des ports obéissant au modèle de calcul approprié à la sémantique de spécification et des ports obéissant au modèle de calcul du domaine. Ainsi, à l'exécution on aura deux sous ensembles homogènes au même niveau. Le premier implémente le modèle de calcul approprié à la sémantique de spécification et contient le composant atomique et une projection de HIC. Quant au deuxième, il implémente le modèle de calcul du domaine et contient les du domaine et une projection du HIC. En effet, c'est le composant HIC qui gèrera l'hétérogénéité entre les deux modèles de calcul.
  Malgré que cette solution résout le problème d'hétérogénéité entre la sémantique de spécification et celle du domaine, elle représente certaines faiblesses, notamment l'obligation d'utilisation des composants HIC, ce qui fait augmenter le nombre de composants utilisés dans la conception.

En effet, la question qui se pose est *comment rendre possible l'exécution d'un composant atomique, dont le comportement est spécifié suivant la sémantique d'un modèle de calcul, au sein de différents domaines implémentant des modèles de calcul différents, et cela, sans passer ni par l'approche hiérarchique ni par l'approche non hiérarchique, ni par les composants domaine-spécifiques ?*

Cette thèse apporte donc une réponse à cette question en proposant une approche de conception de composants domaine-polymorphes [196]. Un composant domaine-polymorphe est un composant atomique, dynamique, réutilisable et a la capacité de s'adapter aux différents modèles de calcul des domaines tout en garantissant un comportement interne suivant la sémantique de spécification.

2.8 Conclusion

Dans ce chapitre nous avons présenté les systèmes embarqués qui sont souvent des systèmes temps réel et hétérogènes. Après avoir présenté les modèles de calcul les plus utilisés, nous avons abordé les approches utilisées actuellement dans la conception de ces systèmes : l'approche amorphe et l'approche hiérarchique, et récemment l'approche non hiérarchique. Ces approches permettrent de résoudre le problème d'hétérogénéité au niveau de communication entre les différents composants des systèmes. Ensuite, des outils dédiés à la conception de ces systèmes ont été présentés ainsi que la méthodologie acteur sur laquelle la majorité desdits outils sont basés.

Ce chapitre nous a permis de présenter la problématique traitée dans cette thèse et de choisir la plateforme (l'outil) de conception pour l'intégration et la validation de nos concepts ainsi que la méthodologie et les primitives à utiliser.

Dans le chapitre suivant, qui représente une étape clé et de départ de la conception du composant domaine-polymorphe, nous montrerons comment utiliser les composant domaine-spécifiques pour exécuter un comportement suivant sa sémantique de spécification sous différents modèles de calcul.

Chapitre 3

Conception des Composants Domaine-Spécifiques Synchrones

3.1 Introduction

Nous avons vu dans le chapitre précédent qu'actuellement tous les systèmes embarqués sont hétérogènes et impliquent l'utilisation des modèles de calcul différents pour les modéliser et les concevoir.

En général, tout système est composé d'au moins deux sous-systèmes : l'un pour le contrôle et l'autre pour le traitement des données. Et, dans tous les cas, la présence d'une partie de contrôle au sein du système est indispensable afin de contrôler ses autres parties. Ainsi, le fonctionnement du système implique la coopération de ses différentes parties, ce qui permettra à la partie contrôle d'obtenir des informations et d'agir sur les autres parties du système.

La partie contrôle du système sera représentée par un composant dont le comportement doit être spécifié suivant une sémantique orientée contrôle (par exemple SR, FMS ou DE). En effet, pour faire communiquer le composant contrôle avec le reste du système, nous devons utiliser l'approche hiérarchique ou l'approche non hiérarchique.

Actuellement, il existe une seule façon pour faire communiquer un composant atomique, dont le comportement est spécifié suivant une sémantique donnée, avec des composants obéissant à d'autre modèle de calcul, et cela, sans passer par les approches hiérarchique et non hiérarchique. Cette façon de faire réside dans l'utiliser des composants domaine-spécifiques.

Dans ce chapitre, nous avons choisi la sémantique synchrone pour la spécification du comportement du composant domaine-spécifique, ce qui nous permet d'appeler ce dernier composant domaine-spécifique synchrone.

Ce chapitre montre que l'utilisation des composants domaine-spécifiques pour exécuter un comportement, qui est spécifié suivant la sémantique synchrone, sous différents modèles de calcul n'est pas une tâche facile et nécessite que les concepteurs doivent avoir une bonne culture en ce qui concerne les modèles de calcul afin de pouvoir bien gérer l'hétérogénéité entre la sémantique de spécification (synchrone) et celle du domaine cible (domaine hôte).

Ce chapitre est une étape clé pour la conception des composants domaine-polymorphes. Son but est donc d'utiliser le même comportement synchrone dans différents composants domaine-spécifiques, dont chacun d'eux obéit à un modèle calcul différent.

3.2 Systèmes réactifs, langage synchrone et mise en œuvre

3.2.1 Systèmes réactifs et l'hypothèse synchrone

Systèmes réactifs

La plupart des systèmes temps réel sont des systèmes réactifs [83]. Les systèmes réactifs sont des systèmes qui interagissent continuellement avec leur environnement, mais à un rythme imposé par ce dernier, et qui doivent toujours être en mesure de fournir une réponse immédiate quand l'environnement les sollicite. L'évolution d'un système réactif est une séquence de réactions provoquées par des *stimuli* générés par l'environnement, chaque réaction étant considérée comme instantanée par rapport à l'échelle de temps propre à l'environnement, voir figure 3.1. Le concept de *réactivité* est une représentation idéalisée de systèmes qui, vis-à-vis de leur environnement, doivent réagir de manière réflexe. Les programmes de contrôle de processus industriels (aéronautique, automobile, télécommunication, etc) et les IHM (Interfaces Home-Machine) font partie des systèmes réactifs.

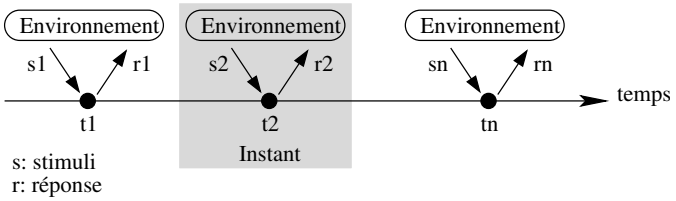


FIG. 3.1: Réactions du système réactif

Concrètement, afin de prendre en compte tous les stimuli et d'y répondre à temps, les systèmes réactifs sont implémentés par des systèmes interactifs suffisamment rapides.

Hypothèse synchrone

Au début, l'implémentation des systèmes réactifs se base sur les approches traditionnelles, à savoir : les automates, les langages de haut niveau associés à des exécutifs temps réel multi-tâches et les langages parallèles synchrones. Cependant, programmer un système en se basant sur les approches traditionnelles pose de sérieux problèmes : par exemple, les automates garantissent un comportement déterministe du système mais sont difficiles à maintenir. En effet, la programmation des systèmes réactifs temps réel doit être une tâche rigoureuse nécessitant des langages pour la spécification et des outils pour la validation. Dans [84], l'auteur confirme que les approches traditionnelles n'apportent pas de solution satisfaisante à la programmation de systèmes réactifs temps réel, et propos de développer des langages dédiés plutôt que d'adapter des langages existants.

Afin de permettre la simplification de l'expression des comportements réactifs, l'hypothèse synchrone conduit à une vision abstraite des interactions. Elle se propose de définir une échelle de temps logique discret. Cette échelle est constituée d'un ensemble d'*instants* dont chacun correspond à une *réaction* du système. Les stimuli ayant déclenché la réaction sont considérés comme simultanés et les réactions se font en *temps nul*, c'est-à-dire, les stimuli ayant provoqué la réaction du système sont *simultanés* avec les réponses produites par le système. L'instantanéité des réactions permet donc d'éviter les interactions concurrentes partielles du système (pour avoir un ordre total des réactions), sources d'indéterminisme. De plus, la communication à l'intérieur du système se fait par diffusion *instantanée*. Aussi, l'hypothèse synchrone considère les stimuli générés par l'environnement et les réponses produites par le système comme des signaux permettant l'abstraction des communications d'une part entre le système et son environnement et d'autre part entre les entités constituant le système.

Notons que l'hypothèse *les réactions se font en temps nul* signifie que toute réaction déclenchée doit terminer avant qu'une autre réaction soit provoquée. De plus, la principale conséquence de l'hypothèse synchrone est la possibilité de composer des systèmes synchrones pour obtenir d'autres systèmes synchrones qui auront ainsi des comportements bien définis et déterministes.

3.2.2 Esterel : Un Formalisme basé sur l'hypothèse synchrone

Plusieurs formalismes, basés sur l'hypothèse synchrone, ont été successivement proposés pour modéliser le comportement des systèmes réactifs. Ces formalismes permettent la validation et la vérification formelle du comportement des systèmes réactifs, et offrent la possibilité :

- D'aider les concepteurs à analyser les comportements des systèmes et d'assurer qu'ils fonctionnent correctement.
- De générer des prototypes pouvant être utilisés pour la réalisation des produits finaux.

Parmi ces formalismes, nous citons : les StateCharts [85][86], Argos [87] [88], Esterel [89] [90], SyncCharts [94], Signal [93] et , Lustre [95] et SCADE [96].

Les langages synchrones comme Lustre ou Signal sont bien adaptés aux systèmes réactifs orientés traitement de données. Cependant, Esterel et SyncCharts sont plutôt utilisés pour des systèmes réactifs pour lesquels le contrôle est primordial.

Dans ce chapitre, nous nous intéressons aux systèmes orientés contrôle, nous avons choisi le langage Esterel, qui est bien adapté à la programmation de ce genre de systèmes, pour exprimer par la suite les comportements synchrones des composants domaine-spécifiques.

Esterel est un langage réactif synchrone impératif dispose d'une sémantique mathématique rigoureuse [91][92]. Il est basé sur des structures de contrôle et des instructions réactives. Les structures de contrôle sont l'itération, la séquence et la composition. Les instructions réactives sont celles qui font référence aux événements auxquels le système décrit doit réagir.

Cette section sera consacrée à la présentation des notions et des instructions nécessaires pour la compréhension de nos programmes Esterel par la suite. Pour plus de détail, le lecteur peut trouver la présentation complète de ce langage dans [89].

Signaux

Un programme Esterel utilise des *signaux* comme moyen de communication avec son environnement et entre les entités le constituant. Les signaux sont diffusés instantanément et sont visibles dans toutes les entités de ce programme.

Un signal est caractérisé par son *état*, qui indique sa présence ou son absence au cours d'un instant ; il est indéterminé jusqu'à ce que son statut soit connu. Il peut être en entrée, dans le cas où il a été émis par l'environnement, et/ou en sortie, dans le cas où il a été émis par le programme Esterel. Il peut être juste un indicateur d'événement, dit *signal pur*, ou porter une information typée, auquel cas il est appelé *signal valué*. Notons que Esterel prédéfinit un signal pur, nommé *tick*, qui est toujours présent et servira comme une horloge d'activation du programme Esterel. Esterel qualifie le temps de *multiforme* afin de permettre l'utilisation des unités variées (mètre, kg, ect.). Les signaux peuvent être utilisés dans des expressions booléennes, dont les opérateurs sont *not*, *and* et *or*, pour tester leur présence.

Modules

L'unité de programmation dans le langage Esterel est le *module*. En effet, un programme Esterel est un module, qui peut être une agrégation de sous-modules. Un module est constitué d'une *interface* et d'un *comportement*. L'interface permet la déclaration des signaux d'entrée et/ou de sortie du module. De plus, elle permet aussi la déclaration des types, des constantes, des fonctions et des procédures. Le comportement représente le corps du module, contenant un ensemble d'instructions impératives et réactives. Un module est vu donc comme une boite noire, qui communique avec son environnement via ses signaux d'entrée et de sortie, voir la figure 3.2.

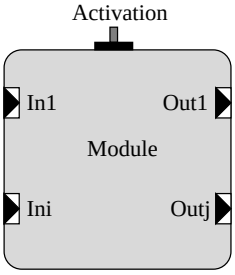


FIG. 3.2: Module Esterel

L'activation d'un module permet l'exécution instantanée de son comportement afin de réagir en un temps nul à la présence et l'absence de ses signaux d'entrée par la production et l'émission des éventuels signaux de sortie, et ensuite, d'effectuer d'éventuelle modification de son état interne.

Types et opérateurs

Les données manipulées par un module Esterel peuvent être de type prédéfini, tel que *integer*, *float*, *double*, *boolean* et *string*, ou de type déclaré (défini par l'utilisateur) dans l'interface du module. Ces données sont les valeurs des variables, des constantes ou des signaux valués. Aussi, le module Esterel peut appliquer des opérateurs prédéfinis sur les données, à savoir les opérateurs booléens, not, and et or, les opérateurs de comparaison, $=$, $\neq$, $<$, $>$, $\leq$ et $\geq$, et les opérateurs arithmétiques, $+$, $-$, $*$ et $/$, ou des opérateurs définis par l'utilisateur et déclarés dans son (le module) interface sous forme de fonctions.

Récapitulatif des instructions

Soit *S* un nom de signal, *s1* et *s2* des instructions, et *E* une exception. Nous donnons une liste d'instructions (les plus utilisées) d'Esterel non exhaustive :

- ***nothing*** : Ne rien faire.
- ***pause*** : Marquer une pause jusqu'au prochain instant.
- ***signal S in s1 end*** : Introduire le signal local *S* et exécuter l'instruction *s1*.
- ***emit S*** : Emettre le signal *S*, cela implique la présence de *S* dans cet instant.
- ***await S*** : Marquer une pause jusqu'à la présence du signal *S*.
- ***s1 ; s2*** : Exécuter *s2* après la fin de *s1*, « *;* » est un opérateur de séquence.
- ***s1 || s2*** : Exécuter *s1* et *s2* jusqu'à ce que les deux terminent, « || » est un opérateur de composition parallèle.
- ***loop s1 end*** : Exécution répétitive de *s1*.
- ***present S then s1 else s2 end*** : Si le signal *S* est présent alors exécuter *s1* sinon exécuter *s2*.
- ***if exp then s1 else s2 end*** : Si l'expression booléenne *exp* est vraie alors exécuter *s1* sinon exécuter *s2*.

- ***suspend s1 when S*** : Exécuter *s1* sauf aux instants où *S* est présent.
- ***abort s1 when S*** : Si *S* est présent alors terminer l'exécution de *s1* et abandonner (avorter) la dans l'instant suivant.
- ***trap E in s1 end*** : Définir une exception locale *E* et exécuter *s1*.

3.2.3 Précédentes mises en oeuvre des modules synchrones

L'hypothèse synchrone a considérablement facilité la spécification des comportements réactifs complexes et a permis une puissance expression. Conceptuellement, cela est très idéal, mais, en pratique, les modules, représentant des systèmes réactifs, écrits en un langage synchrone comme Esterel ne sont pas directement exécutables. Cela revient à l'aspect asynchronisme de l'environnement dans lequel les modules synchrones s'exécutent. L'hypothèse synchrone ne peut donc être respectée. Par exemple, la durée d'exécution d'une réaction n'est jamais nulle à moins qu'on dispose de machines infiniment rapides ou bien qu'on implante le module synchrone sous forme d'un circuit pour se rapprocher de l'instantanéité [100]. Pour faire face à ce problème, il existe plusieurs approches [103][104][105][106] visant à intégrer ou à implanter les modules synchrones. Ces approches ont, soit fait des suppositions sur l'environnement, soit essayé de s'approcher au maximum de l'hypothèse synchrone. Parmi ces approches, nous présenterons le modèle d'objets synchrones de F. Boulanger et la machine d'exécution de H. Boufaïed.

Objets synchrones

Le modèle d'objets synchrones [97][98][99] vise à bénéficier à la fois des avantages de l'approche objet, comme l'abstraction, l'encapsulation, la réutilisation, etc., et des avantages de l'approche synchrone, comme la description formelle, les preuves de correction logique, etc. Il permet donc de faire coopérer les programmes synchrones et asynchrones au sein d'une même application (environnement), et cela, par l'intégration des programmes synchrones dans un langage orienté objet. Cela permet la destruction et la création, d'une façon dynamique, d'objets synchrones, ce qui n'est pas le cas avec les langages synchrones.

L'intégration d'un système réactif dans une application à objets nécessite d'abord la traduction des modules synchrones (formant ce système) en classes synchrones, en utilisant le traducteur Occ++, et ensuite, l'interconnexion explicite des objets synchrones (instances des classes synchrones), en utilisant le langage MDL [1] développés par F. Boulanger. Cette interconnexion forme un réseau d'objets synchrones, représentant le système réactif, à interfacer avec l'environnement asynchrone.

Au sein de l'application, les réseaux d'objets synchrones sont vu comme des zones réactives synchrones, voir la figure 3.3.

Communication et ordonnancement Chaque objet synchrone est doté d'une méthode d'activation, des attributs, correspondant aux signaux d'entrée et de sortie du module synchrone, et des méthodes de connexion. Chaque méthode de connexion est associée à un seul signal d'entrée, prend comme argument un signal de sortie et permet de la déclaration explicite des connexions point à point unidirectionnelles entre les objets synchrones.

Chaque réseau d'objets synchrones a sa propre horloge, qui veut dire son propre temps global. Ce qui permet aux objets synchrones de partager le même instant (instant commun), cela signifie que les valeurs des signaux seront uniques au cours d'un même instant et la communication entre les objets synchrones est bien synchrone. Par contre, pour la communication avec son environnement (objets classiques du langage ou objets synchrones d'un autre réseau, voir figure 3.3), un objet synchrone doit passer par une interface, dite *objet d'interface* défini par F. Boulanger, qui assure la communication

[1] Module Description Language

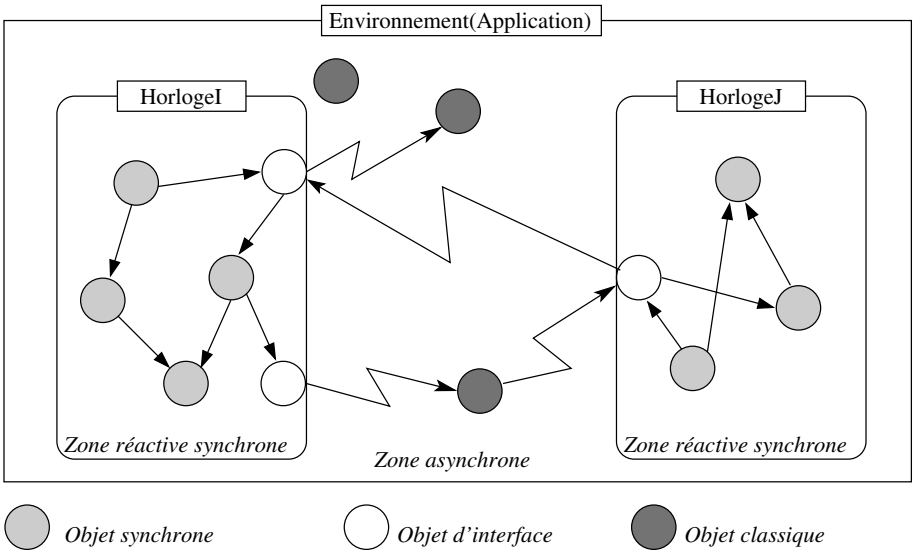


FIG. 3.3: Objets synchrones dans un environnement asynchrone.

asynchrone avec l'environnement. Un objet d'interface permet la traduction des signaux synchrones en événements asynchrones et vice-versa, peut servir d'échantillonneur ou peut, selon la stratégie choisie, déclencher une activation sur l'arrivée d'événements asynchrones.

L'ordre d'activation des objets synchrones d'un même réseau est assuré par un ordonnanceur, appelé *séquenceur synchrone*. Ce dernier active les objets synchrones dans l'ordre établi, à condition que le graphe de dépendance (indiquant l'ordonnancement) entre les objets synchrones soit acyclique. Dans le cas de la présence de cycles, ces derniers doivent contenir au moins un retard, c'est-à-dire qu'ils doivent être non instantanés. Aussi, la topologie du graphe de dépendance peut être modifiée au cours d'un instant, par l'ajout, la destruction, la connexion et la reconnexion des objets. Après la mémorisation des requêtes de modification, l'ordonnanceur vérifie d'abord si ces requêtes ne conduisent pas à des cycles de dépendance, dans ce cas il peut refuser la requête concernée, et ensuite, il effectue les modifications et établit un nouvel ordre d'activation applicable à partir de l'instant suivant.

L'environnement nécessaire aux objets synchrones est assuré par un ensemble de classes, appelé *LibSync*, développé par F. Boulanger. Cet ensemble de classes est vu comme une machine d'exécution d'un réseau d'objets synchrones.

Machine d'exécution

Pour satisfaire l'hypothèse d'instantanéité des réactions, les objets synchrones évoluent en alternance avec leur environnement, c'est-à-dire qu'il est supposé implicitement que l'environnement restera figé (statique) pendant la réaction d'un réseau d'objets synchrones. Cependant, la machine d'exécution de H. Boufaïed [84] prend en compte l'évolution concurrente de l'environnement avec le système réactif (temps de réaction non nul), tout en réalisant une approximation acceptable de l'hypothèse synchrone.

Pour certaines machines d'exécution, le fonctionnement dépend de la cible d'exécution. Par exemple, le fonctionnement de la machine d'exécution *Chorus* développée par le CENT(Centre National d'Etudes des Télécommunication) se base sur les services du système d'exploitation multitâches temps réel pour décrire le comportement d'objets multimédia et gérer la synchronisation de ces objets [102]. La machine d'exécution de H. Boufaïed est complètement indépendante de la cible d'exécution et permet la mise en œuvre détaillée des modules Esterel.

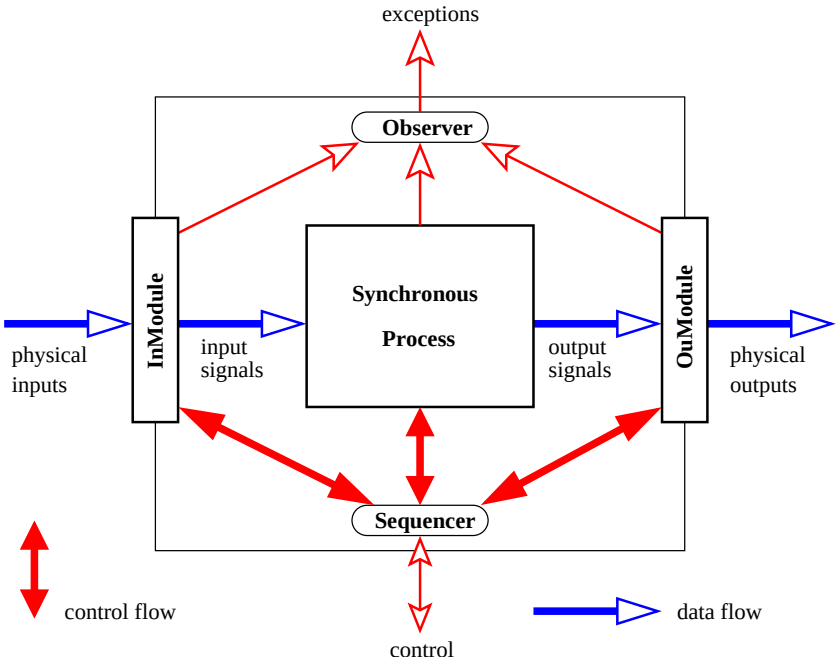


FIG. 3.4: Machine d'exécution.

Architecture La machine d'exécution est vu comme une boite réactive composée d'un module d'entrée (InModule), un processus synchrone (Synchronous Process), un module de sortie (OutModule), un séquenceur (Sequencer) et un observateur (Observer), voir la figure 3.4 prise de [101].

InModule/OutModule Les modules d'entrée et de sortie, InModule et OutModule, ont des structures symétriques et effectuent la même tâche dans des sens inverses. Nous ne présentons que le module d'entrée.

Le module d'entrée est composé d'un ensemble de *handlers* et d'un constructeur d'événements, appelé *Event Builder*. Chaque handler correspond à un signal d'entrée du processus synchrone et sa mission consiste à capter tout signal physique provenant, d'une manière asynchrone, de l'environnement afin de produire un signal logique. Cette production peut être faite soit par échantillonnage, soit par interprétation. Les signaux logiques produits par les handlers sont utilisés, par la suite, par le constructeur d'événements afin de générer le vecteur d'événements d'entrée pour le processus synchrone.

Synchronous Process Le processus synchrone est le résultat de la compilation d'un module Esterel. Il est caractérisé par des signaux d'entrée/sortie et une méthode d'activation permettant le déclenchement de sa réaction.

Sequencer/Observer Le séquenceur et l'observateur sont les deux contrôleurs. Le premier reçoit des signaux du vecteur *control* (voir figure 3.4) pour commander, d'une manière cohérente, les sous-processus composant la machine d'exécution. Ses deux principaux rôles sont : la définition de l'ordonnancement des sous-processus de la machine et la définition de la sémantique des signaux de contrôle (*kill*, *suspend* et *resume*). Tandis que le deuxième veille sur la surveillance du bon fonctionnement de la machine d'exécution en signalant toute anomalie d'exécution sous forme d'exception. Certaines exceptions peuvent être traitées. Par exemple, pour une réaction, le constructeur d'événements génère une exception dans le cas de présence de deux signaux, X et Y, qui ont été déclarés en

exclusion dans le module Esterel, par la primitive *relation X # Y*. Néanmoins, le constructeur d'événements peut prendre une décision pour traiter cette exception (ignorer les deux, retarder l'un des deux pour la prochaine réaction, ect.)

Réaction Chaque exécution de la machine d'exécution caractérise un instant logique et reflète une réaction du module synchrone. Il est clair que la durée des réactions de la machine d'exécution n'est pas nulle, et pour approximer de l'hypothèse d'instantanéité, la machine d'exécution assure l'*atomicité* des réactions. Pour cela, les deux contraintes suivantes sont respectées :

- La machine d'exécution ne doit pas être activée à nouveau si elle est en cours d'exécution. Donc, la réaction de la machine doit être complète.
- Une fois positionnés, les signaux d'entrée ne doivent pas être modifiés pendant la réaction de la machine afin que le processus synchrone aura une perception figée de son environnement.

La machine d'exécution active le processus synchrone selon deux stratégies :

- Soit par activation périodique : A chaque période, le processus synchrone sera activé d'une façon indépendante des occurrences des signaux, sauf qu'il faut s'assurer que la durée d'exécution de la machine doive être inférieure à la période d'activation. Cette stratégie est utilisée dans le cas où le processus synchrone doit réagir continuellement à son environnement.
- Soit par activation déclenchée-événement : Le processus synchrone sera activé à chaque fois qu'un événement soit arrivé. Cette stratégie est mieux adaptée aux événements sporadiques.

La machine d'exécution est le processus synchrone s'exécutent en parallèle pour prendre en compte les événements provenant, d'une façon asynchrone, de l'environnement pendant la réaction du processus synchrone. Les événements apparus lors de la réaction du processus synchrone sont mémorisés jusqu'à la prochaine réaction.

3.2.4 Discussion

Les systèmes réactifs sont souvent des systèmes critiques pour lesquels un dysfonctionnement peut conduire à des conséquences dramatiques. Ils réagissent continuellement aux stimuli de leur environnement par la production des réponses tout en changeant leurs états. L'aspect asynchronisme de l'environnement rend l'analyse et la modélisation de ces systèmes très difficile. Pour cela, l'hypothèse synchrone se propose de rendre la description de ces systèmes plus facile par l'abstraction du temps des réactions, ce qui garantit un ordre total des réactions (qui garantit le déterminisme), et le partage de la même notion d'instant, ce qui assure que toutes les entités du système auront la même vision de l'envirnnement.

Par ailleurs, les langages synchrones, comme Esterel, sont bien adaptés à la description et la conception des systèmes réactifs complexes. De plus, ces langages permettent la vérification et la validation formelle des comportements de ce type de systèmes. Cependant, à cause de l'hypothèse synchrone, qui ne peut être respectée strictement à implémentation (problème d'asynchronisme de l'environnement et la durée non nulle des réactions), ces langages ne sont pas directement exécutables, et la mise en œuvre d'une interface entre le programme synchrone et l'environnement est nécessaire. Les approches : objets synchrones et machine d'exécution rentrent dans ce genre de mise en œuvre et elles consistent à encapsuler le programme synchrone en proposant des interfaces (objets d'interfaces et modules d'entrée/sortie) permettant de gérer l'aspect asynchronisme de l'environnement et garantissant ainsi la bonne interaction entre le programme synchrone et l'environnement.

Cependant, les deux approches (objets synchrones et machine d'exécution) *n'ont fait aucune supposition sur l'hétérogénéité de l'environnement où on a à faire à une variété de sémantiques (des lois physique)*. Autrement dit, dans le cas où l'environnement a sa propre sémantique d'interaction, les deux approches ne garantissent pas le bon fonctionnement du programme synchrone. Pour cela, nous

proposons dans la section suivante la conception des composants domaine-spécifiques synchrones pour montrer comment pouvons nous utiliser les modules synchrones dans la conception des systèmes embarqués hétérogènes, et cela, sans passer ni par l'approche hiérarchiqe ni par l'approche non hiérarchique.

3.3 Conception de composants domaine-spécifiques synchrones

3.3.1 Composants domaine-spécifiques

Définition

Un composant domaine-spécifique est un composant qui est conçu pour fonctionner dans un domaine particulier. Son fonctionnement correct n'est garanti que pour ce domaine. Il peut arriver qu'un composant domaine-spécifique fonctionne par accident dans un autre domaine.

La structure du composant domaine-spécifique se résume par une classe, des paramètres et des ports qui sont spécifiques à la sémantique du domaine. Tandis que son (composant domaine-spécifique) comportement (vis-à-vis de son domaine) est défini par les opérations de manipulation de ses ports, dites opérations de communication, qui sont, elles aussi, spécifiques à son domaine. Donc, la structure et le comportement du composant domaine-spécifique lui permettrent d'interagir correctement dans son domaine. En effet, l'utilisation d'un composant domaine-spécifique dans un domaine hors de celui pour lequel il a été conçu conduit à un fonctionnement non correct.

Exemple

A titre d'exemple, nous donnons deux composants domaine-spécifiques suivant les spécifications de Ptolemy II : Un composant spécifique au domaine DE (Discrete Event), appelé *DEComponent*, et un composant spécifique au domaine SDF (Synchronous Dataflow), appelé *SDFComponent*. Les codes des deux composants sont donnés respectivement par le listing 3.1 et le listing 3.2.

Le composant *DEComponent* implémente deux interfaces, *SequenceActor* et *TimedActor* pour respecter la notion « événements ne sont qu'une séquence de données temporisées totalement ordonnées», et utilise des ports de communication, *DEIOPort*, qui lui permettront de supporter la communication par événement. Par contre, le composant *SDFComponent* n'implémente que l'interface *SequenceActor* pour respecter la notion « flot de données ne sont qu'une séquence de données dont le temps n'a aucune signification » (domaine non temporisé) et utilise des ports de communication, *SDFIOPort*, qui lui permettrent de supporter la communication par flot de données. Cependant, pour dire qu'ils sont des acteurs atomiques, les deux composants domaine-spécifiques (*DEComponent* et *SDFComponent*) héritent de la classe *TypedAtomicActor*.

Listing 3.1: Composant DE-spécifique

```
public class DEComponent extends
  TypedAtomicActor implements
  SequenceActor, TimedActor {

  //DE Communication port
  DEIOPort deOutPort;
  ...//
  //Communication operation
  deInPort.send(data, delay);
  ...//
}
```

Listing 3.2: Composant SDF-spécifique

```
public class SDFComponent extends
  TypedAtomicActor implements
  SequenceActor {

  //SDF Communication port
  SDFIOPort sdfOutPort;
  ...//
  //Communication operation
  sdfOutPort.send(data);
  ...//
}
```

Maintenant que nous avons présenté le composant domaine-spécifique, nous donnons dans la sous section suivante les techniques permettant la spécification de son comportement suivant la sémantique synchrone.

3.3.2 Techniques interfaçage et intégration des modules synchrones

Nous avons choisi le langage Esterel pour spécifier des comportements suivant la sémantique synchrones afin de les utiliser dans la conception des systèmes hétérogènes. Cependant, nous devons faire face au problème d'hétérogénéité des langages car le langage utilisé par la plateforme de conception des systèmes hétérogènes est différent du langage Esterel.

Pour cela, il existe deux techniques permettant de résoudre le problème d'hétérogénéité des langages afin d'utiliser des modules synchrones dans la modélisation et la conception des systèmes :

– *Technique Interfaçage* : Cette technique intervient dans le cas où le langage dans lequel le module synchrone a été codé n'est pas le même que celui avec lequel la plateforme de modélisation et de conception est programmée. En effet, en ajoutant au problème d'hétérogénéité des sémantiques (sémantique synchrone et celle du domaine hôte), le problème d'hétérogénéité des langages se pose aussi.
Pour respecter les exigences du domaine hôte en matière de structure et comportement, cette technique se propose d'utiliser un composant domaine-spécifique (spécifique au domaine hôte), nommé *CDS Représentant*, qui représente le module synchrone dans dudit domaine hôte. Par contre, pour faire face au problème d'hétérogénéité des langages, cette technique consiste à utiliser une interface permettant au *CDS Représentant* de manipuler (commander) le code du module synchrone. Par exemple, sur la figure 3.5, nous avons pris l'environnement d'exécution des programmes C/C++ et la plateforme Ptolemy II, qui est programmée en Java. Dans l'environnement C/C++, nous retrouvons le module synchrone codé en C++, cela peut être réalisé par le compilateur d'Esterel Studio [109] ou par l'interpréteur Occ++ de F. Boulanger, dans ce cas, le module synchrone n'est qu'un objet synchrone. Cependant, dans le domaine hôte D de la plateforme Ptolemy II, nous retrouvons le représentant du module synchrone, *CDS Représentant*. Donc, une fois activé, le composant *CDS Représentant*, à travers l'interface native java (JNI) [110], manipule le module synchrone comme suit : il positionne ses signaux d'entrée, il lance son activation et, finalement, il récupère ses signaux de sortie. Cette technique est utilisée actuellement par certaines industries.

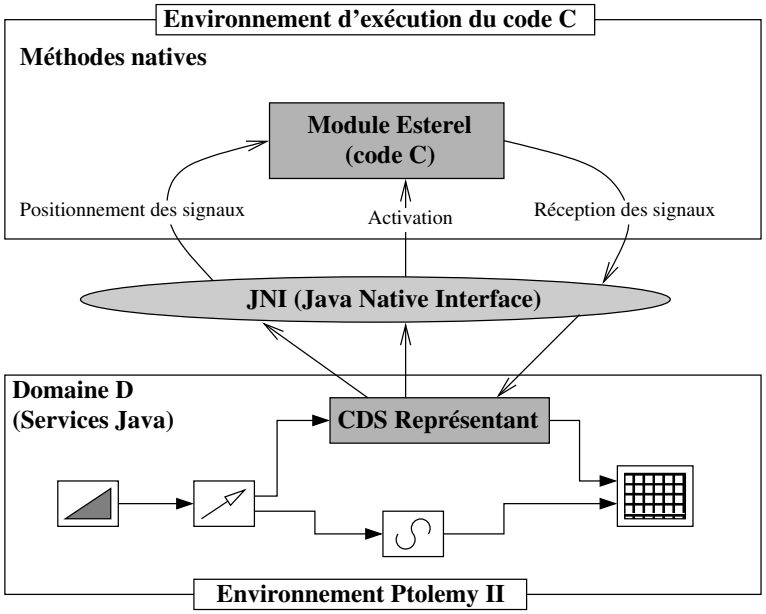


FIG. 3.5: Technique Interfaçage

Avec cette technique, l'hétérogénéité entre la sémantique de spécification (synchrone) et celle du domaine hôte est gérée au niveau du composant *CDS Représentant* et tout doit être prévu lors de la conception de ce dernier.

La technique interfaçage possède des avantages : la possibilité de tirer profit de la spécificité des langages et exploiter au mieux leurs performances, par exemple, on sait bien qu'un programme C est plus performant qu'un programme java en temps d'exécution. Cependant, son inconvénient est la nécessité de gestion des interactions entre deux programmes qui se diffèrent syntaxiquement et sémantiquement, ce qui fait perdre la portabilité et diminuer l'efficacité ;

– *Technique Intégration* : Afin d'éviter les problèmes posés par la technique précédente, la technique *Intégration* consiste, à partir du module synchrone, à générer (par traduction) directement un composant domaine-spécifique homogène, appelé *composant domaine-spécifique synchrone* et noté *CDS Synchrone* (voir figure 3.6). En effet, le composant domaine-spécifique a toujours la structure exigée par le domaine hôte et code un comportement synchrone. Cette technique permet d'éviter le problème d'hétérogénéité du code et présente l'avantage de portabilité, de fiabilité et de manipulation aisée du code représentant le module synchrone. De plus, l'hétérogénéité est gérée lors de la génération du CDS Synchrone et pas dans l'outil de modélisation et de conception.

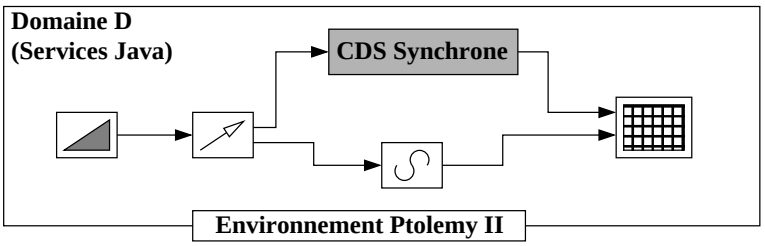


FIG. 3.6: Technique Integration

Dans cette thèse, c'est la *technique Intégration* qui a été retenue. Cela nous a emmené à développer nos propres outils permettant l'intégration des modules synchrones dans Ptolemy II, voir annexe A.

3.3.3 Problématique

Afin d'utiliser les modules synchrones dans la conception des systèmes embarqués hétérogènes, nous avons choisi la technique d'intégration qui consiste à générer à partir d'un module synchrone un composant domaine-spécifique synchrone. Ce dernier a donc un comportement spécifié suivant la sémantique synchrone qui n'est pas la même que celle du domaine hôte, ce qui posera de sérieux problèmes. Pour bien montrer ces problèmes, nous avons pris un module synchrone, nommé *Toggle*, qui nous servira comme exemple d'explication dans le reste de ce chapitre.

Le module synchrone *Toggle* a la mission d'un commutateur. Il communique avec son environnement à travers deux signaux d'entrée, *toggle* et *request*, et deux signaux de sortie, *on* et *off*. Le programme Esterel codant son comportement est donné par le listing 3.3. La présence du signal *toggle* permettra au module *Toggle* de changer son état, tandis que la présence du signal *request* impliquera, suivant son état, l'émission du signal *on* ou signal *off* par le module *Toggle*.

Nous avons vérifié que les signaux *on* et *off* ne seront jamais émis au même instant. Pour cela, nous avons utilisé l'outil XEVE [107], qui est un outil de vérification permettant une exécution symbolique pour déterminer si un signal est potentiellement émis (c'est-à-dire, s'il existe un état dans lequel le signal est présent) ou non. Il existe une autre manière de vérifier les propriétés des modules synchrones, il s'agit de l'écriture d'un module exécutant le module à vérifier en parallèle avec le code de vérification afin de signaler toute erreur rendant les propriétés fausses. Le module de vérification pour *Toggle* est donné par le listing 3.4.

Listing 3.3: Module Toggle

```
module Toggle:
  % This module emits 'on' or 'off'
  % according to its state each time
  % it receives 'request'. Its on/off
  % state changes each time it receives
  % 'toggle'.
  input request, toggle;
  output on, off;

  loop
    abort
      % "Off" initial state
      every immediate request do
        emit off
      end every
    when toggle;
    abort
      % "On" state
      every immediate request do
        emit on
      end every
    when toggle
  end loop
end module
```

Listing 3.4: Module check

```
module check:
  output errorOnAndOff;
  output errorNoAnswerToRequest;
  input request, toggle;

  signal on, off in
    run Toggle;
    ||
    % check that 'on' and 'off'
    % are never emitted in the
    % same instant.
    [
      every immediate on do
        present off then
          emit errorOnAndOff;
        end;
      end every
    ]
    ||
    % Check that each time 'request'
    % is present, either 'on' or
    % 'off' is emitted.
    [
      every immediate request do
        present on or off else
          emit errorNoAnswerToRequest;
        end;
      end every
    ];
  end signal
end module
```

L'utilisation du module *Toggle* dans le domaine SDF par exemple posera deux problèmes :

1. *La non compatibilité des types des données communiquées* : Le domaine SDF utilise des flots de données pour la communication, tandis que le module *Toggle* utilise des signaux logiques. Cela impliquera la mise en œuvre d'une interface spécifique à SDF et au module synchrone pour permettre la production des flots de données à partir des signaux logiques et vice versa ;
2. *La non compatibilité des protocoles de communication* : A chaque activation, le module *Toggle* doit produire sur chacune de ses sorties (*on* et *off*) des données afin de respecter la sémantique du domaine SDF (sachant que dans SDF, un composant doit consommer/produire un nombre fixe de données depuis/sur chacun de ses ports d'entrée/sortie). Cependant, à chaque réaction, le comportement synchrone ne peut produire qu'une donnée sur l'une de ses sorties.

En général, l'utilisation du module synchrone dans différents domaines pose les mêmes problèmes que nous venions de citer. En effet, la transformation des données communiquées (par échantillonnage, interprétation, détection, etc.) et le protocole de communication entre les deux sémantiques (sémantique synchrone et sémantique du domaine hôte) doit être lieu, ce qui résume : l'*adaptation de la sémantique synchrone à celle du domaine hôte est nécessaire*.

Nous définissons donc l'adaptation de la sémantique synchrone à celle du domaine hôte comme suit :

> *Adaptation de la sémantique synchrone à celle du domaine se traduit par conserver la sémantique synchrone et respecter la sémantique du domaine.*

Donc, de vue externe, le CDS Synchrone fonctionne exactement comme tout composant spécifique à son domaine, et de vue interne, doit fournir un environnement permettant à son comportement synchrone de s'exécuter correctement. Par exemple, dans le domaine SDF, le CDS Synchrone doit garantit le bon fonctionnement de la partie synchrone tout en fonctionnant comme un composant spécifique au domaine SDF.

3.3.4 CDS Synchrone

Au niveau abstrait tout CDS Synchrone, obtenu par la technique intégration à partir d'un module synchrone, n'est qu'un composant domaine-spécifique dont le comportement interne est un comportement synchrone. La partie codant donc le module synchrone est appelée *partie synchrone*. De plus, il comporte d'autres parties, appelées *partie pré-réaction* et *partie post-réaction*, voir figure 3.7, permettant d'adapter la sémantique synchrone à la sémantique du domaine. Il a un nombre de ports égal au nombre de signaux du module synchrone, chaque port d'entrée/sortie correspond à un signal d'entrée/sortie.

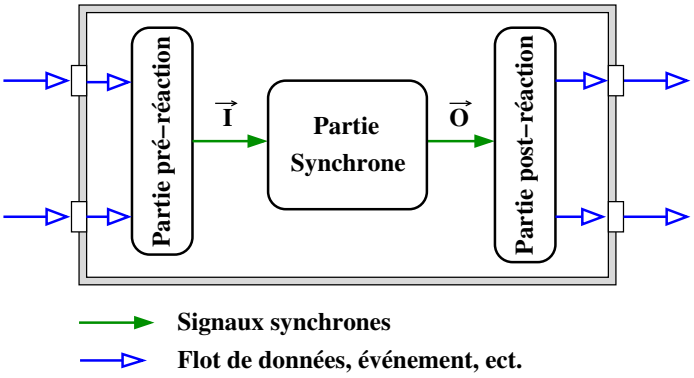


FIG. 3.7: Vu interne du CDS Synchrone

Partie pré-réaction

La mission de la *partie pré-réaction* est de conserver la sémantique de la *partie synchrone* sous les contraintes de la sémantique du domaine. Elle est spécifique au domaine, dans lequel le CDS Synchrone se trouve, et à la *partie synchrone*. Au niveau abstrait, toutes les *parties pré-réactions* des différents CDSs Synchrones réalisent les mêmes tâches, tandis qu'au niveau implémentation, elles ont des comportements différents.

Les tâches effectuées par chaque *partie pré-réaction sont connues d'avance car nous connaissons déjà la sémantique du domaine*. Ces tâches sont :

- La lecture des ports du CDS Synchrone en utilisant les opérations de communication spécifiques au domaine ;
- La production du vecteur de signaux logiques d'entrée, $\vec{I}$, à partir des données lues et qui sont codées suivant la sémantique du domaine. Cette tâche fait partie de la conception du système et peut être effectuée par interprétation, par échantillonnage, etc ;
- Autoriser/non autoriser l'activation de la *partie synchrone*. Dans le cas où le vecteur d'entrée $\vec{I}$ est vide, on passe directement à la *partie post-réaction* ;
- Capturer les exceptions : les exceptions non traitées doivent être signalées tandis que les exceptions causées par une défaillance logique (le non respect des relations Esterel, # et =>, par exemple) peuvent être traitées ;
- D'autres éventuelles tâches. Par exemple, demander au domaine de réitérer le CDS Synchrone une fois que l'itération courante sera terminée.

Partie synchrone

La *partie synchrone* code un comportement synchrone. Une fois autorisée, elle consomme les signaux d'entrée ($\vec{I}$), produit le vecteur de signaux de sortie ($\vec{O}$) et termine par la mise à jour de son état. Cette partie est, même au niveau implémentation, commune à tous les CDSs Synchrones, c'est-à-dire, elle ne comporte aucune opération spécifique à un domaine donné.

Partie post-réaction

Cette partie est, elle aussi, spécifique au domaine et à la *partie synchrone*. Sa mission est plutôt, en fonction du comportement de la *partie synchrone*, de respecter la sémantique du domaine. Les tâches réalisées par cette partie ainsi que leurs comportements sont, eux aussi, connus d'avance car la sémantique du domaine est connue d'avance. Ses tâches sont :

- Coder les signaux de sortie du vecteur $\vec{O}$, produits par la *partie synchrone*, en données ayant le type ou le format exigé par le domaine, tout en respectant au mieux la sémantique synchrone ;
- Ecriture des données, codant les signaux de sortie, sur les ports de sortie correspondant. Les opérations d'écriture (de communication) sont, elles aussi, spécifiques au domaine ;
- Autres éventuelles tâches. En général, ces dernières sont exécutées pour accomplir des faits permettant le respect total de la sémantique du domaine.

3.3.5 Cas d'adaptation étudiés

Vu leur dépendance du domaine, les tâches réalisées par les deux parties, *partie pré-réaction* et *partie post-réaction*, ont été citées d'une manière implicite. Cependant, afin d'expliciter ces tâches, nous avons étudié des cas d'adaptation de la sémantique synchrone à celles de certains domaines connus.

Nous signalons que le nom CDS Synchrone change suivant le nom du domaine dans lequel il sera utilisé. Par exemple, dans le domaine DE, le nom du CDS Synchrone devient C*DE*S Synchrone. Pour simplifier, nous garderons le même nom, CDS Synchrone, mais il sera spécifique au domaine dans lequel il sera utilisé.

Cas du domaine SR (Synchronous Reactive)

Avant de présenter l'adaptation de la sémantique synchrone dans le domaine SR, nous devons faire la distinction entre le domaine SR et le CDS Synchrone.

Le domaine SR a été proposé par Stephen Edwards et actuellement implémenté dans Ptolemy II. Il régit les interactions des entités (dites acteurs) suivant la sémantique synchrone. Les acteurs sont des fonctions monotones calculant des sorties à partir de leurs entrées. Chacune de ces fonctions a une solution qui existe et unique (théorème du point fixe), ce qui garantit le déterminisme. Le domaine SR combine la notion du temps logique (par exemple celle utilisée par Esterel) avec la hiérarchie (en utilisant les acteurs composites) pour permettre la *conception hétérogène des systèmes synchrones*. Dans ce domaine, il n'y a aucune supposition sur les comportements internes des acteurs et peuvent être spécifiés par d'autre langage. Cependant, le CDS Synchrone *cible l'utilisation des comportements synchrones, qui sont déjà spécifiés et compilés par un langage synchrone, dans la conception des systèmes hétérogènes*.

Avec le domaine SR, nous n'avons spécifié aucune tâche d'adaptation de la sémantique, car nous avons à faire à deux sémantiques identiques, le *CDS Synchrone est vu donc comme un objet synchrone dans un réseau synchrone*. Cependant, nous avons pris en compte les deux modes de fonctionnement dans le domaine SR : le mode *strict* et le mode *non-strict* (dit aussi mode relaxé) [77][123]. Avec le mode *strict*, une entité (acteur) a besoin de connaître tous les états de ses entrées pour pouvoir calculer ses sorties, tandis qu'avec le mode *non-strict*, une entité peut calcul au moins une de ses sorties sans que tous les états de ses entrées soient connus. Le deuxième mode permet l'utilisation des *boucles instantanées* et utilise une troisième valeur, dite valeur inconnue, notée $\perp$, qui s'ajoute aux valeurs booléennes, $true$ et $false$.

En effet, les tâches réalisées par la partie *pré-réaction* du CDS Synchrone sont la lecture des ports d'entrée suivie par la construction du vecteur $\vec{I}$. Et comme la *partie synchrone* n'est pas générée

directement à partir du code Esterel, mais plutôt à partir du code intermédiaire *ssc5* (qui est un circuit logique généré par le compilateur Esterel) [111], c'est-à-dire que, la *partie synchrone* code un circuit logique composé de portes logiques (sous forme *and*, *or* et *not*), des fil et des registres (voir annexe A), nous avons donc pris en compte le mode *non-strict* en introduisant la valeur $\perp$ comme troisième valeur booléenne dans les évaluations des portes logiques, ce qui permettra l'*utilisation des modules synchrones dans les boucles instantanées*. Par contre, la *partie post-réaction* n'a qu'une seule tâche à faire, écriture des valeurs du vecteur $\vec{O}$ sur les ports de sortie.

Avec le domaine SR, il est possible donc de supprimer *partie pré-réaction* et *partie post-réaction*, et d'ajouter à la partie synchrone les opérations de communication (lecture/écriture depuis/sur les ports) spécifiques.

Cas du domaine DE (Discrete Event)

Dans le domaine DE, le CDS Synchrone doit se comporter comme un acteur DE, tout en conservant la sémantique de la *partie synchrone*. Donc, il consomme et produit des événements, rappelons qu'un événement est composé d'une donnée et une date.

Une fois que le CDS Synchrone est lancé, la *partie pré-réaction* lit les ports ayant des événements présents et construit par la suite le vecteur $\vec{I}$. De plus, les événements présents à la même date sur le même port ne seront pas tous lus au cours de la même itération du CDS Synchrone car la *partie synchrone* ne peut consommer qu'un seul signal au plus par entrée au cours d'une réaction (propriété synchrone). Donc, pour respecter cette contrainte, la *partie pré-réaction* ne doit lire qu'au plus un événement depuis chaque port d'entrée. Pour le reste des événements, la *partie pré-réaction* doit, par envoi d'un événement pur, demander au domaine DE de réitérer le CDS Synchrone une fois que l'itération courante soit terminée. Cette demande est renouvelable tant qu'il y'a des événements toujours présents. Par contre, pour respecter l'instantanéité de la réaction de la *partie synchrone*, la *partie post-réaction* produira à partir du vecteur $\vec{O}$ des événements ayant comme date le temps courant (qui est la date d'arrivée des événements lus) du domaine, c'est-à-dire avec un retard égal toujours à zéro afin de s'approximer de l'hypothèse synchrone.

Cas du domaine DDE (Distributed Discrete Event)

Le domaine DDE (Distributed Discrete Event) [10][112][113][114][115] incorpore une notion de temps distribuée à un modèle de flot de données. Dans le domaine DDE, les acteurs communiquent par envoi d'événements à travers des files d'attente FIFO (First In First Out) dont chacune est associée à un port d'entrée. Chaque acteur maintient une notion de temps local. La communication des événements entre les acteurs fait progresser les temps de ces derniers. A chaque fois qu'un événement soit consommé, le temps de l'acteur est placé à la date de cet événement. Cependant, chaque événement produit par un acteur doit avoir une date qui doit être supérieure (cas d'envoi d'événements avec retard) ou égale (cas d'envoi d'événement sans retard) à la date courante de cet acteur.

Le temps de réception d'une file d'attente est toujours égal à la date de son plus ancien événement. Par contre, dans le cas où la file est vide, ce temps est égal à la date du dernier événement l'ayant traversée sinon à zéro.

L'opération de lecture fait toujours appel à la contribution de tous les ports d'entrée pour déterminer le port à lire. Ce dernier doit avoir le plus petit temps de réception de sa file d'attente. Dans le cas où cette file est vide, l'acteur bloquera jusqu'à ce qu'un événement soit disponible. De même, une opération d'écriture se bloquera dans le cas où la file du port concerné est pleine. Pour éviter l'interblocage causé par la lecture bloquante, il faut que l'émetteur, dans le cas d'absence des données de sortie, envoie un événement spécial, *nul event*, dit événement de déblocage, servant au déblocage du récepteur.

Le domaine DDE est un domaine orienté processus. Autrement dit, à chaque acteur du domaine DDE est lui associé un thread qui l'exécute d'une façon répétitive, les acteurs, dans DDE, s'évoluent donc en concurrence.

En effet, le CDS Synchrone va donc consommer/produire que des événements, ce qui représente un comportement plus proche de celui du module synchrone, l'adaptation ne pose pas donc de difficultés : la *partie pré-réaction* doit ignorer les événements de déblocage (*nul event*) et la *partie post-réaction* doit prendre, comme date à associer aux données sortantes, la date courante du CDS Synchrone pour s'approximer de l'hypothèse synchrone et envoyer les événements de déblocage à la place des signaux de sortie absents.

Cas du domaine SDF (Synchronous Dataflow)

Rappelons qu'un acteur, dans le domaine SDF, doit consommer un nombre fixe de données depuis chacun de ses ports d'entrée et doit produire un nombre fixe de données sur chacun de ses ports de sortie.

Et comme le CDS Synchrone doit se comporter comme un acteur SDF (il va donc consommer/-produire un nombre fixe de données depuis/sur chacun des ses ports d'entrée/sortie), le nombre de données à consommer/produire pose des problèmes pour l'adaptation.

En effet, la présence d'un nombre de données supérieur à un sur un même port d'entrée bloquera la réaction du CDS Synchrone. Ce blocage est dû aux points suivant :

- La *partie synchrone consomme au plus une donnée par entrée*. On ne peut pas donc consommer plus d'une donnée par port d'entrée au cours d'une même réaction ;
- Contrairement au domaine DE où l'ordonnancement est dynamique, dans le domaine SDF, l'*ordonnancement est statique* (établit avant l'exécution du modèle), ce qui fait qu'une demande de réitération, à la fin de l'itération courant pour consommer les données restantes, ne sera pas possible.

De même, la production d'un nombre de données supérieur à un sur un même port de sortie du CDS Synchrone bloquera l'exécution de ce dernier, car la *partie synchrone produit au plus une donnée par sortie*.

Pour les deux cotés (domaine SDF et la *partie synchrone*), nous avons posé la contrainte suivante : *le nombre de données que le CDS Synchrone doive consommer/produire depuis/sur chacun de ses ports d'entrée/sortie est fixé à un.*

Cette contrainte est nécessaire mais n'est pas suffisante pour l'adaptation car d'une part nous aurons toujours la présence des données sur les ports d'entrée, qui veut dire que la *partie synchrone* aura toujours des signaux d'entrée présents, d'autre part la *partie synchrone* doit toujours produire tous les signaux de sortie. Par exemple, le module *Toggle*, donné précédemment, va toujours, pour la même réaction, consommer les signaux *request* et *toggle*, ce qui fait que son état se changera en alternance, une fois allumé et une fois éteint, et produire les signaux *on* et *off*, ce qui est impossible. Pour résoudre ce problème, nous avons proposé :

- D'ajouter à chaque donnée consommée/produite par le CDS Synchrone un attribut. Ce dernier est un flag qui indique la présence ou l'absence du signal. Cela permet donc, dans le cas de présence, de prendre en considération la donnée lue, sinon elle sera ignorée ;
- La *partie post-réaction* doit, à la place des signaux de sortie absents, produire des données ayant des attributs positionnés à absent.

La proposition d'associer aux données des flags permet d'une part d'expliciter mieux le comportement de la partie synchrone et d'autre part le respect de la sémantique SDF, à condition que les émetteurs/récepteurs des données au/depuis CDS Synchrone doivent la respecter. En effet, nous avons ajouté à la contraite précédente une autre qui est : *l'état des signaux doit apparaître explicitement comme une donnée dans le domaine SDF*.

Cas du domaine PN (Process Networks)

Le domaine PN(réseaux de processus) [10] est, lui aussi (comme le domaine DDE), un domaine orienté processus. Les acteurs (qui sont des processus dans ce cas) communiquent par envoi de messages à travers des canaux unidirectionnels FIFO. Quand un processus essai de lire à partir d'un canal vide, il bloquera jusqu'à ce qu'un message devient disponible. Par contre, les opérations d'écriture ne sont pas bloquantes, ce qui permet d'avoir une communication asynchrone [116][117].

Le modèle de calcul SDF est un cas particulier du modèle de calcul PN, puisque la phase de lecture, qui est bloquante, finira par la récupération d'une donnée par port d'entrée. Donc, un acteur SDF, avec un nombre fixe de donnée à consommer/produire égal à un, peut être utilisé dans le domaine PN. En effet, les mêmes propositions que nous avions faites pour adapter la sémantique synchrone à celle du domaine SDF sont valables pour le domaine PN.

Cas du domaine CSP (Communicating Sequential Process)

Le domaine CSP [10], lui aussi, est un domaine orienté processus. Sa première version a été proposée par Hoare en 1978 [118][119]. Il existe plusieurs versions de ce type de modèle, il peut être utilisé avec ou sans notion de temps [121]. La communication utilise le principe de *rendez-vous* [120], cela implique que les opérations lecture/écriture sont des opérations bloquantes et garantissent une communication synchrone. Donc, si un acteur est prêt à envoyer un message, il bloquera jusqu'à ce que l'acteur récepteur soit prêt à accepter le message. De même, si un acteur est prêt à recevoir un message, il bloquera jusqu'à ce que l'acteur émetteur lui envoiera le message. La différence entre le domaine CSP et le domaine PN est qu'avec le premier (CSP), il n'y a pas des files d'attentes, ce qui cause l'attente du processus émetteur.

Le domaine CSP pose deux problèmes pour l'adaptation de la sémantique synchrone :

- Problème des lectures bloquantes, qui est le même que celui posé par le domaine PN. Cela a la même conséquence que celle avec le domaine SDF : pour toute itération du CDS Synchrone, tous les ports d'entrée auront toujours des données présentes ;
- Problème d'interblocage réel causé par l'ordre d'utilisation des liaisons de connexion des deux acteurs. Pour illustrer ce problème, prenons l'exemple montré par la figure 3.8.

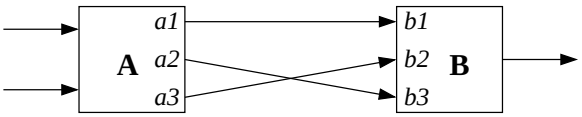


FIG. 3.8: Un exemple de connexion de deux acteurs.

Supposons que l'acteur A utilise ses ports de sortie dans l'ordre a1, a2 puis a3 pour envoyer des messages à l'acteur B, et que ce dernier lira ses ports d'entrée dans l'ordre b1, b2 puis b3. Donc, pour la communication à travers la liaison (a1, b1) aura lieu sans problème, puisque les deux acteurs finissent par se retrouver au rendez-vous. Par contre, les communications à travers les deux autres liaisons (a2, b3) et (a3, b2) n'auront jamais lieu car l'acteur A restera bloqué sur le port a2 attendant jusqu'à ce que l'acteur B soit présent, alors que ce dernier, lui aussi, est bloqué sur le port b2 attendant la présence de l'acteur A.

Comme solution, nous adoptons, pour le premier problème, la même solution que celle proposée dans le cas du domaine SDF. Par contre, pour le deuxième problème, c'est aux concepteurs du système d'utiliser les liaisons dans le bon ordre.

Cas du domaine CT (Continuous Time)

Le domaine CT (Continuous-Time) modélise les systèmes à dynamique continue. Ces derniers sont représentés par des équations différentielles ordinaire (ODEs) comme suit :

$$\begin{aligned} \dot{x} &= f(x,u,t) \\ y &= g(x,u,t) \\ x(t_0) &= x_0 \end{aligned}$$

avec,

- t : représente le temps continu ;
- x : l'état du système, qui est un signal continu ;
- u : le signal d'entrée ;
- y : le signal de sortie du système ;
- $\dot{x}$: la partie dynamique du système ;
- x_0 : L'état initiale du système.

Conceptuellement, le système peut être représenté par le diagramme de la figure 3.9 [122].

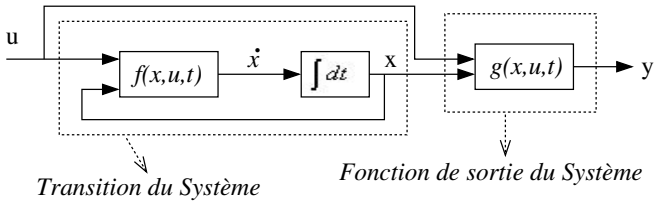


FIG. 3.9: Diagramme conceptuel pour un système à temps continu.

La stratégie de simulation consiste à trouver pour tout instant, t_n, l'état du système, x_n, et appliquer ensuite la fonction de sortie, $g(x,u,t)$, pour calculer la sortie du système. L'idée est donc de résoudre numériquement les ODEs représentant le système. Pour cela, il existe plusieurs algorithmes qui discrétisent le temps continu en une séquence d'instants dans un ordre croissant et calculent ensuite numériquement les valeurs d'état du système à ces instants.

Avec certains algorithmes, le calcul d'état du système nécessite un nombre d'itérations de la partie transition du système. Ce genre d'algorithme est dit *algorithme explicite* et le nouvel état du système sera calculé donc directement à l'aide des valeurs précédentes de l'état du système, ce qui fait qu'au cours des itérations de la partie transition, le système n'est pas stable et les valeurs calculées ne sont que des valeurs intermédiaires. Par exemple, l'algorithme basé sur la méthode *Runge-Kuta* fait partie des algorithmes explicites. Cependant, les *algorithmes implicites* réduisent les ODEs en un ensemble d'équations algébrique et la valeur d'état du système sera calculée donc par une relation implicite faisant intervenir la fonction $f()$. Le choix de l'algorithme dépend largement du système à modéliser.

En effet, l'adaptation de la sémantique synchrone à celle du domaine CT doit prendre en compte les deux points suivant :

1. L'endroit de placement du CDS Synchrone au sein du diagramme conceptuel du système : Est ce que nous plaçons le CDS Synchrone au niveau de la partie transition du système ou au niveau de la partie sortie du système ;
2. Quel type de signal sera utilisé (consommé/produit) par le CDS Synchrone : Dans CT, il existe deux types de signaux, signaux continus et signaux discrets. En effet, nous devons prendre en compte les deux cas.

Comme solution, nous avons proposé :

- Pour le premier problème : Il est clair que le *CDS Synchrone ne sera jamais placé dans la partie transition du système*, car dans cette partie, le *système n'est pas stable*, c'est-à-dire que la valeur d'état du système est incertaine (inconnue), or que nous ne pouvions contrôler (c'est la mission du CDS Synchrone) le système qu'après avoir connu la valeur exacte (stable) de son état ;
- Pour le type du signal : Le CDS Synchrone peut être utilisé comme un composant continu, c'est-à-dire il consomme/produit des signaux continus, comme il peut être utilisé comme un composant discret :
  - Dans le cas où le CDS Synchrone est utilisé comme un composant continu, la persistance des données au niveau des ports (les données sont toujours présentes sur les ports d'entrée après leur lecture) pose le problème de présence des données pour toute itération du CDS Synchrone. La persistance des données est utilisée par le domaine CT pour respecter la notion de signal continu : quelque soit l'instant t, le signal a une valeur connue. Donc, dans le cas où l'état du système sera le même pour l'(es) instant(s) suivant(s) que l'état de l'instant courant, la *partie pré-réaction* aura le choix entre la construction ou non du vecteur d'entrée $\vec{I}$ (pour ne pas lancer la réaction de la *partie synchrone*) car la *partie synchrone* produira toujours les mêmes signaux de sortie pour les même signaux d'entrée (propriété des systèmes synchrones), à condition que l'état interne de celle-ci reste inchanger. Cependant, la *partie post-réaction* doit produire des données indiquant l'absence des signaux de sortie absents afin d'éviter d'avoir la présence (par persistance) des signaux absents. Par exemple, prenons le module *Toggle* et supposons qu'à l'instant t_i son état est à *on*, donc le signal *on* est présent tandis que le signal *off* est absent, ensuite à l'instant t_{i+1} le module change d'état, le signal *on* sera donc présent tandis que le signal *off* sera absent. Or, le fait que les données sont persistantes, la valeur indiquant la présence du signal *on* à l'instant t_i est toujours présente, ce qui fait que les deux signaux *on* et *off* seront présents à l'instant t_{i+1}, ce qui ne reflète pas le comportement correct du module *Toggle*. Aussi, dans le cas où le vecteur $\vec{I}$ n'a pas été construit, la *partie pré-réaction* doit informer la *partie post-réaction* pour que cette dernière ne produira pas des données d'absence (par exemple en utilisant la donnée *no data*), ce qui permet de maintenir les mêmes états des signaux de sortie pour le même état du système ;
  - Dans le cas où le CDS Synchrone est utilisé comme un composant discret, les données ne sont pas persistantes et le CDS Synchrone va consommer/produire que des événements, ce qui est proche de la sémantique synchrone, à condition qu'il faut utiliser des interfaces (par exemple, échantillonneurs (*Samplers*), les bloqueurs, etc.) pour la production des événements d'entrée à partir des signaux continus et vice versa car les ports du CDS Synchrone sont des ports discrets.

Cas du domaine FSM (Finite State Machine)

En général, les états d'une FSM sont soit vides ou raffinés par d'autres domaines. Dans le cas où un état est vide, cela veut dire que le système est en attente de la satisfaction de certaines conditions pour pouvoir changer d'état. Par contre, dans le cas où un état est raffiné par un domaine, le CDS Synchrone peut être utilisé dans le domaine raffinant l'état juste pour générer des événements signalant la satisfaction des conditions de transition (gardes) du système à un autre état. Cela revient donc à adapter la sémantique synchrone à celle du domaine raffinant l'état du FSM.

3.3.6 Variables et opérations du CDS Synchrone

Dans cette section nous donnons au niveau abstrait les variables et les différentes opérations qui sont communes à tous les CDS synchrones et aucune supposition n'est faite sur leur implémentation.

Tout CDS Synchrone a donc un ensemble de variables, noté $CDSSync.X$, composé de deux sous ensembles : le premier sous ensemble contenant ses variables d'interface, qui sont ses ports de communication, et le deuxième contenant ses variables internes :

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

Avec

$$\{CDSSync.P = CDSSyncIn\}, \{CDSSync.Q = CDSSyncOut\} \text{ et}$$
$$\{CDSSync.S = CDSSyncParameters, CDSSyncState\}$$

Ainsi

$$CDSSync.X = \{CDSSyncIn, CDSSyncOut, CDSSyncParameters, CDSSyncState\}$$

Où :

- $CDSSyncIn$ est l'ensemble de ports d'entrée dont chacun est associé à un signal d'entrée de la *partie synchrone* ;
- $CDSSyncOut$ est l'ensemble de ports de sortie dont chacun est associé à un signal de sortie de la *partie synchrone* ;
- $CDSSyncParameters$ est composé de deux sous ensembles. Le premier sous ensemble contient les paramètres du composant domaine-spécifique. Tandis que le deuxième sous ensemble contient les paramètres de la *partie synchrone*, par exemple une constante déclarée au niveau du module synchrone peut être, lors de la génération du CDS Synchrone, représentée par un paramètre ;
- $CDSSyncState$ est l'état de la *partie synchrone*.

Tout au long de son exécution, le CDS Synchrone est contrôlé par son domaine à travers les opérations de flot de contrôle suivantes :

$$CDSSync.Ctrl = \{initialization, preCondition, trigger, postCondition\}.$$

Tandis que lui (CDS Synchrone) exécute :

1. Opérations de rappel (call-back), qui sont des opérations de flot de contrôle :

   $$CDSSync.Clbk = \{finish, \Phi CDSSync.Clbk\}$$

   avec
   - $finish$ est l'opération par laquelle le CDS Synchrone sollicite le domaine pour l'informer sur la fin de ses activités ;
   - $\Phi CDSSync.Clbk$ est un ensemble d'éventuelles opérations. Ces dernières dépendent du domaine pour lequel le CDS Synchrone est spécifique. Par exemple, le cas d'adaptation de la sémantique synchrone à celle du domaine DE, le CDS Synchrone, dans le cas de présence de plusieurs événements au même date et sur le même port, doit consommer qu'un seul événements par $trigger$ et pour le reste des événements doit, à travers l'opération $reTrigger$, solliciter le domaine pour le relancer après la fin de sa réaction courante. Donc, l'opération $reTrigger$ fait partie de $\Phi CDSSync.Clbk$.
2. Opérations de flot de données, voir figure 3.10, qui sont :
   - Opérations de communication, notées $CDSSync.Comm$: Elles sont les opérations de manipulation des ports de communication, tel que $existData$, $read$, $isFull$ et $write$. Leur comportement dépend du domaine dans lequel le CDS Synchrone est utilisé ;
   - Opérations de calcul (computation), notées $CDSSync.Comp$: Elles sont toutes les opérations de la *partie synchrone*. Autrement dit, elles sont les opérations générées à partir d'un module synchrone ;

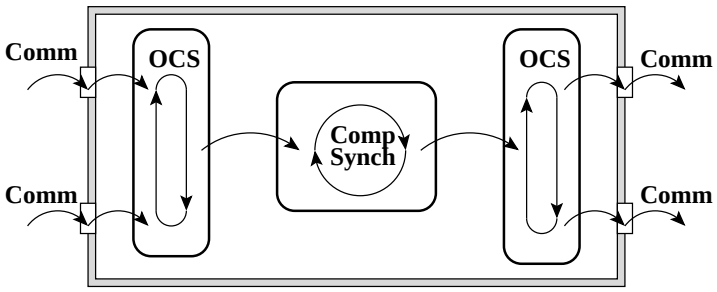


FIG. 3.10: Opérations exécutées par le CDS Synchrone.

– Opérations de conservation de la sémantique synchrone, dites aussi opérations de conservation de la sémantique interne ou encore opérations de conservation de la sémantique de spécification, notées $CDSSync.OCS$: Ce type d'opérations est responsable sur l'offre d'un environnement d'exécution à la *partie synchrone* afin de lui (*partie synchrone*) permettre de se comporter correctement tout en respectant la sémantique du domaine hôte. Par exemple, la construction des signaux logiques (par interprétation, échantillonnage, etc.) à partir des données communiquées et vice versa, les opérations assurant la contrainte « l'état des signaux logiques doit apparaître explicitement comme une donnée dans certains domaine (comme SDF, PN, etc.) », les opérations de configuration (positionnement du nombre de données à consommer/produire depuis/par port SDF à un), etc. font, toutes, partie de ce type d'opérations. Aussi, les opérations de contrôle permettant l'autorisation ou non de la réaction de la *partie synchrone*, une éventuelle capture d'exceptions avec leur signalisation ou leur traitement et la signalisation (cas du domaine CT) de la *partie pré-réaction* à la *partie post-réaction* sur la non construction du vecteur d'entrée $\vec{I}$ font aussi partie des opérations de conservation de la sémantique interne.

Notons que les opérations de communication et les opérations de conservation de la sémantique interne sont réparties entre la *partie pré-réaction* et la *partie post-réaction*, tandis que les opération de contrôle n'appartiennent qu'à la *partie pré-réaction*. Aussi, nous signalons qu'un CDS Synchrone n'aura pas besoin d'opérations, dites *opérations de synchronisation*, permettant d'informer une partie, sauf la *partie pré-réaction*, sur la fin d'exécution de la partie qui la précède car les différentes parties s'exécutent dans un ordre séquentiel.

3.3.7 Exécution du CDS Synchrone

Nous savons maintenant qu'un CDS Synchrone est contrôlé par son domaine à travers les opérations $CDSync.Ctrl$ dont l'ordre de leur lancement est le même que de celui de leur citation dans la sous section 3.3.6. Donc, l'activation de l'opération $initialization$ permet l'initialisation du CDS Synchrone, l'opération $preCondition$ permet le lancement de la *partie pré-réaction* qui doit, avant sa fin, signaler au domaine si le CDS Synchrone est prêt à être lancé ou non, et d'éventuelles sollicitations de type call-back, l'opération $trigger$ permet le lancement de la *partie synchrone* et l'opération $postCondition$ permet le lancement de la *partie post-réaction*. Finalement, le CDS Synchrone signale la fin de ses activités par l'opération $finish$.

En effet, Les parties constituant le CDS Synchrone s'exécutent dans l'ordre indiqué par l'organigramme de la figure 3.11. Donc, c'est la *partie pré-réaction* qui s'exécutera en premier ensuite si le vecteur d'entrée, $\vec{I}$, n'est pas vide, la *partie synchrone* s'exécutera en deuxième et, dans tous les cas, la *partie post-réaction* s'exécutera toujours en dernier. En effet, les trois parties *s'exécutent séquentiellement* (ne sont pas concurrentes) car elles sont corrélées (liens de dépendance).

Le CDS Synchrone, qui est un acteur, s'exécute dans un environnement responsable [62] dont le principe est de ne pas lancer un acteur tant qu'il n'est pas prêt ou l'acteur qui le précède n'a pas

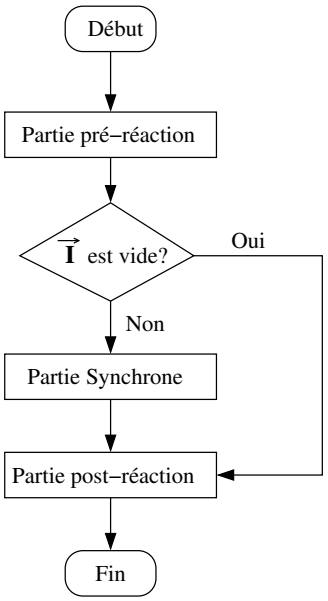


FIG. 3.11: L'ordre d'exécution des différentes parties du CDS Synchrone

encore terminé son exécution. En effet, l'exécution de l'ensemble de parties du CDS Synchrone est atomique, c'est-à-dire qu'une fois lancer par son domaine au moyen de l'opération *trigger*, le CDS Synchrone exécutera son itération jusqu'à la fin sans être interrompu, ce qui garantit l'atomicité des réactions de la *partie synchrone*.

3.4 Composants domaine-spécifiques pour d'autres sémantiques

Nous avons vu que la conception des composants domaine-spécifiques synchrones consiste d'abord à spécifier leurs comportements, qui sont identiques, suivant la sémantique synchrone en utilisant la technique intégration, et ensuite, d'adapter le comportement de chacun d'eux à la sémantique de son domaine en spécifiant les opérations des deux parties *pré-réaction* et *post-réaction*. Ainsi, la spécification des opérations dépend de la sémantique du domaine (dite sémantique externe) et la sémantique de spécification (dite sémantique interne).

Dans le but de généraliser l'adaptation pour d'autres sémantiques, nous donnons dans cette section les règles basées sur les types des opérations cités dans la sous section 3.3.6. Cependant, nous signalons que nous ne pouvons pas adapter n'importe quelle sémantique à celles des différents domaines : *le choix d'une sémantique à adapter doit avoir un sens physique*. Par exemple, le choix d'adapter la sémantique DE, qui est une sémantique temporisée, à celle du SDF, qui est une sémantique non-temporisée où le temps est toujours nul, n'a pas de sens physique car la sémantique DE va fonctionner sous le contrôle (comme esclave, principe de la responsabilité des domaines [81]) de la sémantique SDF et son temps n'avancera jamais. Autrement dit, le temps de la sémantique DE évolue en fonction de celui de la sémantique SDF. Or que cette dernière n'a pas de notion de temps, ce qui n'a pas de sens physique.

Les règles générales pour adaptater une sémantique, notée *Sém*, à celles des différents domaines sont :

– Remplacer la *partie synchrone* par une autre dont la spécification est faite suivant la sémantique *Sém* à conserver. Dans ce cas, le composant domaine-spécifique qui lui est associé sera appelé, *CDS Sém* ;

– Pour chaque domaine, déterminer les opérations de conservation de la sémantique interne (*Sém*) avec la spécification de leurs comportements ;
– Pour chaque domaine, spécifier les éventuelles opérations de contrôle, dites opérations de rappel (call-back).

3.5 Conclusion

Dans ce chapitre, nous avons présenté l'adaptation de la sémantique synchrone à celles des différents domaines afin de permettre l'utilisation des comportements synchrones dans la modélisation et la conception des systèmes hétérogènes. Vu les avantages qu'elle possède, c'est la *technique intégration* que nous avons utilisée pour générer directement des CDSs Synchrones à partir des modules synchrones. Pour la modélisation de ce dernier, nous avons choisi Esterel comme langage de spécification des comportements synchrones.

Chaque CDS Synchrone n'est qu'un composant domaine-spécifique codant un comportement synchrone et doté d'un ensemble d'opérations permettant la conservation de la sémantique synchrone au sein de celle du domaine. Au niveu abstrait, nous avons constaté que chaque CDS Synchrone exécute trois types d'opérations : *opération de communication*, *opération de calcul* correspondant à la partie synchrone et *opération de conservation de la sémantique synchrone*. C'est ce qui nous a permis de dégager les règles générales permettant d'utiliser les composants domaine-spécifiques pour conserver d'autre sémantiques.

Ce chapitre nous a permis donc d'identifier les éléments (structure, des types d'opérations) caractérisant un composant domaine-spécifique dont le comportement est spécifié suivant une sémantique donnée. Au niveau abstrait, ces éléments sont communs à tous les composants domaine-spécifiques, tandis qu'au niveau implémentation, leurs comportements sont spécifiques à la sémantique interne et celle du domaine.

Les éléments identifiés sont des points clés pour la conception du composant domaine-polymorphe. Cependant, ces éléments ne sont pas suffisant pour une telle conception car le modèle du composant domaine-polymorphe que nous allons proposer est un modèle de composant logiciel dont l'implémentation doit tenir de compte des technologies existantes en génie logiciel.

Chapitre 4

Composant Domaine-Polymorphe

4.1 Inroduction

Pour permettre l'utilisation des modules synchrones dans la conception des systèmes hétérogènes, nous avons présenté dans le chapitre précédent la conception des composants domaine-spécifiques synchrones. Chacun de ces derniers est un composant spécifique à son domaine dont le comportement est spécifié suivant la sémantique synchrone.

Nous avons automatisé la conception des composants domaine-spécifiques synchrones où chacun d'eux est généré automatiquement à partir d'une spécification synchrone par la technique intégration. Ainsi l'hétérogénéité entre la sémantique synchrone et celle du domaine hôte (domaine pour lequel le composant domaine-spécifique synchrone est spécifique) est gérée lors de la phase de traduction et non pas dans l'outil de modélisation et de conception des systèmes hétérogènes.

L'utilisation du composant domaine-spécifique, dont le comportement est spécifié suivant une sémantique donnée, donne certes une réponse à notre problématique, c'est-à-dire que le composant domaine-spécifique permet l'hétérogénéité et cela sans passer ni par l'approche hiérarchique ni par l'approche non hiérarchique. Cependant, cette façon de concevoir présente beaucoup d'inconvénients : solution trop lourde et implique que les concepteurs doivent avoir une bonne culture, notamment en ce qui concerne les modèles de calcul afin de bien déterminer et de bien définir les opérations gérants l'hétérogénéité entre la sémantique de spécification et celles des domaines. C'est pourquoi, comme solution, nous proposons le modèle de composant domaine-polymorphe.

Un composant domaine-polymorphe est un composant capable de fonctionner correctement dans différents domaines tout en assurant un fonctionnement interne suivant la sémantique de spécification. Ce composant n'est pas proposé pour bannir l'approche hiérarchique ou l'approche non hiérarchique ou les composants domaine-spécifiques mais plutôt d'apporter d'autres avantages, notamment l'explicitation du passage des données entre la sémantique de spécification du comportement interne et celle du domaine hôte ainsi que la réutilisabilité dans différents domaines.

Dans ce chapitre, après avoir donné les faiblesses engendrées par l'utilisation des composants domaine-spécifiques pour exécuter un comportement, qui est décrit suivant une sémantique de spécification, au sein de différents domaines, nous donnons la définition du composant domaine-polymorphe que nous voulions concevoir ainsi que ces propriétés. Ensuite, étant donné que le composant domaine-polymorphe est un composant logiciel, nous donnons la définition du polymorphisme du point de vue logiciel et discutons par la suite des techniques basées sur des technologies logicielles permettant d'avoir dudit polymorphisme.

4.2 Faiblesses des composants domaine-spécifiques

Vu leur atomicité, les composants domaine-spécifiques présentent plusieurs avantages : taille optimisée, rapidité d'exécution (au lancement, aucune phase d'assemblage n'est nécessaire), etc.

Leur utilisation pour représenter une même entité (par exemple entité de contrôle, voir chapitre 3) dans plusieurs domaines, tout en respectant la sémantique de ces derniers et celle de l'entité, revient à créer pour chaque domaine un composant domaine-spécifique dont le comportement interne est celui de l'entité à représenter tout en spécifiant les parties *pré-réaction* et *post-réaction*. En effet, cette utilisation engendre plusieurs inconvénients, à savoir :

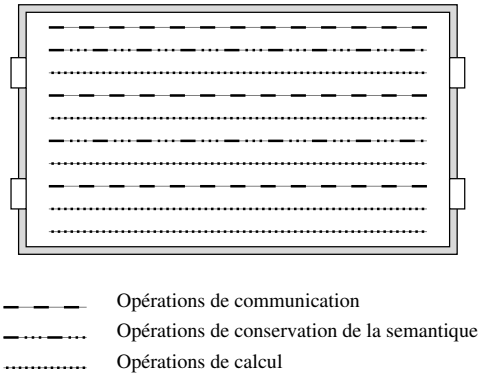


FIG. 4.1: Entrelacement des opérations dans un composant domaine-spécifique.

- *Pas de réutilisabilité :* Une fois conçu, le composant domaine-spécifique est dur dans le sens qu'il n'est réutilisable que pour l'entité qui la représente et dans son domaine. En effet, il n'est pas réutilisable dans d'autres domaines et pour d'autres entités ;
- *Nombre de composants domaine-spécifiques explosif :* A cause du premier inconvénient, le nombre de composants domaine-spécifiques devient de plus en plus important, ce qui rend donc la gestion et la maintenance de ces derniers très lourdes ;
- *Mise à jour très coûteuse :* Dans le cas où l'entité a subi des modifications, cela impliquera la mise à jour de tous les composants domaine-spécifiques qui la représentent. Cet inconvénient est dû à la duplication du code, qui représente l'entité, et qui entrelacé avec le code des composants domaine-spécifiques ;
- *Pas d'évolution :* Dans le cas d'ajout de nouveau domaine, et à cause du premier inconvénient (pas de réutilisabilité), nous serons obligés de créer un nouveau composant domaine-spécifique représentant cette entité dans ce nouveau domaine ;
- *Problème de lisibilité du code :* Il n'y a pas une séparation nette entre les différentes opérations du composant domaine-spécifique représentant l'entité (voir la figure 4.1), ce qui rend la lisibilité du code du composant domaine-spécifique difficile et les mises à jour délicates ;
- *Connaissance des modèles de calcul (domaines) est obligatoire :* La représentation d'une entité dans différents domaines nécessite la connaissance de toutes les caractéristiques des domaines à savoir la structure des composants domaine-spécifiques, opérations de communication, etc. En effet, cela rend la représentation des systèmes dans les domaines faisable que par des concepteurs ayant une bonne culture des modèles de calcul, ce qui rend donc l'utilisation des plateformes de modélisation et de conception des systèmes hétérogènes non accessible par de simples concepteurs. De plus, le fait de se préoccuper, à la fois, des caractéristiques des domaines et du système à représenter, les concepteurs doivent faire beaucoup d'effort. Cela vient de la non séparation entre la préoccupation domaine et celle de l'entité à représenter.

Dans le but de répondre aux inconvénients, posés par l'utilisation des composants domaine-spécifiques pour représenter une même entité dans différents domaines tout en tenant compte de

sa sémantique de spécification et celles des domaines, nous proposons dans la section suivante un modèle de composant, appelé composant domaine-polymorphe.

4.3 Qu'est ce qu'un composant domaine-polymorphe ?

4.3.1 Définition

Un composant domaine-polymorphe est un acteur atomique, dynamique et réutilisable ayant des propriétés lui permettant de s'adapter aux différentes sémantiques des domaines tout en garantissant un comportement interne suivant la sémantique de spécification. En effet, il a la capacité de s'adapter aux sémantiques externes tout en conservant sa sémantique interne.

L'*adaptation à la sémantique du domaine se traduit par la capacité de transformer sa structure et son comportement externe suivant les exigences du domaine hôte toute en conservant la sémantique de spécification de son comportement interne* : Nous voulons dire par la capacité de se transformer au niveau structurelle, la capacité d'avoir des ports et de changer sa classe pour avoir la structure correspondant à celle exigée par le domaine hôte, et par la capacité de se transformer au niveau comportemental externe, la capacité d'avoir et d'utiliser les opérations de communication ainsi que les opérations de configuration spécifiques au domaine hôte. Par contre, nous voulons dire par conserver la sémantique de spécification de son comportement interne, la capacité de fournir un environnement d'exécution à sa partie interne (par exemple la *partie synchrone*) pour lui permettre de s'exécuter correctement et d'une manière transparente la sémantique du domaine hôte.

Donc, comme c'est montré par la figure 4.2, un composant domaine-polymorphe est vu comme un composant composé de trois couches :

- Couche Noyau qui spécifie le comportement interne du composant domaine-polymorphe suivant la sémantique de spécification et ainsi représente une entité à exécuter. Par exemple, elle représente un module réactif synchrone ;
- Couche Conservateur permet la fourniture de l'environnement d'exécution adéquat à la sémantique de la couche Noyau ;
- Couche Frontière permettant d'expliciter le passage des données entre la sémantique du Noyau et la sémantique du domaine hôte. Cette explicitation du passage fait partie de la modélisation et la conception des systèmes ;
- Couche Adaptateur permettant au composant domaine-polymorphe de fonctionner correctement dans le domaine hôte.

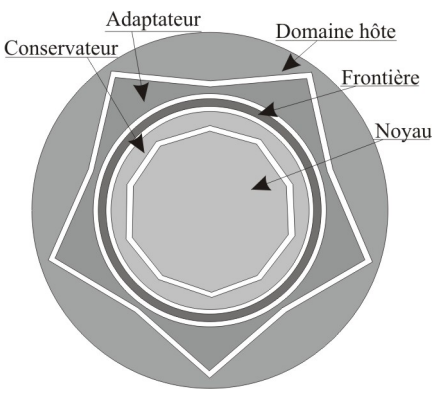


FIG. 4.2: Les couches du composant domaine-polymorphe.

4.3.2 Propriétés du composant domaine-polymorphe

Le composant domaine-polymorphe possède les propriétés suivantes :

1. La capacité de détection du domaine hôte. Cette propriété est cruciale et elle se traduit par la capacité d'interroger le domaine hôte afin de demander son identité. L'identité du domaine permet au composant domaine-polymorphe de savoir quelle structure et quel comportement à présenter. Elle peut être sous forme d'une description, un nom où un message spécial permettant l'identification du domaine ;
2. Garantit la cohérence structurelle : Après avoir détecté le domaine hôte, le composant domaine-polymorphe adapte sa structure aux exigences du domaine détecté, c'est-à-dire, de présenter la classe, les ports de communication et les paramètres répondant aux exigences du domaine. Par exemple, dans le domaine DE, le composant domaine-polymorphe doit avoir la structure qui correspond à celle exigée par le domaine DE ;
3. Garantit la cohérence comportementale : Après avoir présenté la structure répondant aux exigences du domaine hôte (détecté), le composant domaine-polymorphe doit avoir le bon comportement :
   – Par rapport au Noyau : Il doit implémenter et présenter les bonnes opérations permettant au Noyau de s'exécuter correctement. Ces opérations sont des opérations de communication et de contrôle ;
   – Par rapport au domaine hôte : Il doit être capable d'utiliser les bonnes opérations qui lui permettrent d'interagir correctement dans le domaine hôte. Par exemple, s'il se trouve dans le domaine SDF, il doit utiliser les opérations de communication spécifiques aux ports du domaine SDF et non pas celles spécifiques à d'autres ports d'un autre domaine.
4. Garantit la cohérence sémantique : Cette propriété rentre dans l'assurance d'une bonne conservation de la sémantique du Noyau. Par exemple, parmi les exigences pour qu'un Noyau fonctionne correctement dans le domaine DE, il faut que ses données produites doivent sortir vers l'extérieur avec un retard égal à zéro, dans ce cas le composant domaine-polymorphe ne doit pas envoyer les événements représentant les données produites avec un retard supérieur à zéro. Tout simplement, nous voulons dire par la cohérence sémantique, le positionnement des paramètres de configuration et de communication doit être au faveur de la sémantique du Noyau. Pour que la cohérence sémantique soit respectée, il faut d'abord garantir la cohérence structurelle et la cohérence comportementale.

4.3.3 Composant domaine-polymorphe vs. acteur générique

Dans cette sous-section nous présentons les acteurs domaine-polymorphes de Ptolemy II afin de mettre en évidence la différence entre ceux-ci et nos composants domaine-polymorphes.

Dans Ptolemy II, on trouve la notion d'acteur domaine-polymorphe, mais avec un sens différent de celui que nous proposons. Un acteur domaine-polymorphe a une structure générique qui lui permet d'être réutilisé dans différents domaines. C'est pourquoi nous préférons de l'appeler acteur générique au lieu acteur domaine-polymorphe.

Ladite structure générique est la structure mère de tous les acteurs de Ptolemy II y compris les acteurs domaine-spécifiques (composants domaine-spécifiques). En effet, cette structure ne comporte pas des paramètres spécifiques aux domaines pour configurer l'acteur générique, ce qui fait que les domaines attribuent des valeurs par défaut aux paramètres absents lors de l'exécution dudit acteur générique. Par exemple, le domaine DE positionne toujours la date d'envoi des données par les acteurs génériques à la date courante (retard égal toujours à zéro) et il n'y a aucune possibilité d'envoyer des données avec retard. Donc, l'acteur générique acquiert la sémantique des domaines, c'est-à-dire qu'il ne connaisse pas la sémantique du domaine dans lequel il est utilisé et c'est au domaine de fournir

les valeurs par défaut de configuration de son fonctionnement. De plus, avec l'acteur générique, on ne peut pas exécuter n'importe quel comportement. Ce dernier doit être spécifié de telle sorte que la réaction de l'acteur générique soit faisable dans tous les domaines (réaction de l'acteur générique commune pour tous les domaines).

Par contre, un composant domaine-polymorphe est à la fois vu comme acteur générique, car il est réutilisable dans différents domaines, et aussi vu comme acteur domaine-spécifique, car une fois placé dans un domaine, il finira par avoir toutes les caractéristiques d'un composant domaine-spécifique. En effet, contrairement à l'acteur générique, qui acquiert la sémantique du domaine hôte, un composant domaine-polymorphe s'adapte à la sémantique du domaine hôte tout en ayant un fonctionnement interne suivant la sémantique de spécification de sa couche Noyau.

Comme le composant domaine-polymorphe est un composant logiciel, nous signalons que lors de sa conception nous ne concentrerons ni sur un langage de programmation ni sur une technologie génie logicielle particuliers. C'est pourquoi dans la section suivante nous, du point de vue génie logiciel, donnons la définition du polymorphisme, présentons et discutons des technologies que nous avions jugées capables de permettre un tel polymorphisme.

4.4 Techniques pour le polymorphisme

4.4.1 Que signifions nous par polymorphisme ?

Le mot *polymorphisme* est un mot Grec composé de deux sous-mots : *Poly* et *Morphisme*. Le premier mot signifie *plusieurs*, le deuxième mot signifie *formes* et leur concaténation signifiera *plusieurs formes*. En effet, un élément [1] est dit élément polymorphe si et seulement si ce dernier est capable de changer de forme. Cependant, cette définition est très abstraite et ne donnera aucun détail sur la *forme* de l'élément polymorphe, nous pouvons donc voir cette *forme* comme *forme géométrique*.

En effet, la définition du polymorphisme n'est pas unique et elle est fortement sensible au secteur (la chimie, la biologie, la médecine, la physique, l'informatique, etc) dans lequel le mot forme a son propre sens. Par exemple, en chimie, l'eau peut prendre plusieurs formes : liquide, solide et gazeuse. En informatique, le polymorphisme, qui est une caractéristique essentielle de la programmation orientée objets, permet d'avoir, dans la même application, des opérations de même nom, implémentées de façon différente par différentes classes qui héritent d'une même super-classe.

Définition du polymorphisme

Afin de définir d'une manière précise le polymorphisme dans le cadre de cette thèse, nous commençons d'abord par décrire ce qu'est la forme qui va subir les changements.

Nous signifions par le mot *forme* la structure d'un composant qui est décrite par une *classe* comportant des *attributs* et des *méthodes*. En effet, notre définition du polymorphisme est la suivante :

Le polymorphisme est la propriété d'un composant qui lui permet de s'adapter à son environnement afin de fonctionner correctement. Cette propriété se traduit par la capacité de changer sa forme suivant les exigences de son environnement. Le changement de forme peut être au niveau de la structure, dans ce cas, nous disons qu'il s'agit du polymorphisme structurel, et/ou au niveau du comportement, dans ce cas, nous disons qu'il s'agit du polymorphisme comportemental.

[1] nous désignons ici par élément n'importe quoi.

Polymorphisme structurel vs. Polymorphisme comportemental

Le *polymorphisme structurel* correspond à toute modification au niveau de la structure du composant. Cette modification peut être au niveau des attributs et des méthodes, par exemple, la suppression, la modification et/ou l'ajout des attributs, comme peut être au niveau de la signature de la classe elle-même, par exemple, ajout des interfaces. En effet, nous pouvons dire que le polymorphisme structurel touche l'aspect physique du composant. Par contre, le *polymorphisme comportemental* correspond aux modifications qui ne touchent pas la structure du composant, par exemple, la redéfinition d'une méthode ou la modification de l'environnement pour modifier le comportement du composant, et/ou aux modifications qui touchent la structure du composant (polymorphisme structurel).

Les deux types de polymorphisme sont corrélés. La modification de la structure d'un composant aura automatiquement comme conséquence la modification de son comportement, c'est-à-dire, le comportement dudit composant ne sera plus le même après la modification de sa structure. Aussi, la modification du comportement d'un composant peut impliquer d'abord des modifications au niveau de sa structure afin de pouvoir d'apporter par la suite des éventuels modifications ne touchant pas sa structure. Cependant, dans le cas où l'environnement, dans lequel le composant doit fonctionner correctement, exige des modifications qui ne touchent pas la structure, le polymorphisme comportemental est indépendant du polymorphisme structurel, par exemple, une simple redéfinition de méthode.

Pour des raisons de clarté, nous signalons que le polymorphisme comportemental peut être un *polymorphisme comportemental dur*, dans ce cas les modifications toucheront la structure du composant, comme peut être un *polymorphisme comportemental simple*, dans ce cas les modifications ne toucheront pas la structure du composant.

La définition donnée ci-dessus montre que le composant polymorphe que nous visions est un composant qui s'auto-change (s'auto-modifie). Autrement dit, les changements de la forme s'effectuent par le composant lui-même. Cependant, sur le plan concret, le respect de cette définition est complètement lié à ce qui, en matière de *changer de comportement* et/ou de *structure*, existe comme technologies au niveau du génie logiciel.

4.4.2 Techniques de l'approche objet pour le polymorphisme

L'apparition de l'approche objet remonte aux années soixante [132] ayant comme but d'apporter une réponse à la monolicité des programmes basés sur l'approche procédurale. Son principe est de diviser un problème en sous problèmes pour réduire la complexité et accroître la réutilisabilité. Elle est à la base de beaucoup de langages à savoir *smalltalk* [125], *C++* [126][127][128][129], *eiffel* [130], *Java* [131], etc.

L'approche objet a introduit de nouvelles techniques, à savoir la composition, l'héritage, le polymorphisme [2], la liaison dynamique, la délégation, etc. Chacune de ces techniques a ses propres caractéristiques permettant d'une façon ou d'une autre d'avoir du polymorphisme. En effet, parmi l'ensemble de techniques qui nous intéressent, nous avons choisi les techniques : la composition, l'héritage et la délégation, et à travers lesquelles nous montrerons comment peut on avoir du polymorphisme.

La *composition* permet d'avoir un composant polymorphe à partir d'assemblage des éléments logiciels. Cependant, cette technique, lors de la programmation du composant polymorphe, oblige la déclaration de toutes les entités logicielles prévues pour l'assemblage, ce qui explique donc que la structure du composant, une fois définie, est statique (un assemblage statique). En effet, la composition ne permet que le polymorphisme comportemental simple. Par exemple, par le remplacement

[2]Polymorphisme dans le sens de l'approche objet et non pas dans le sens de notre polymorphisme tel qu'il a été défini avant.

d'une classe [3] par une autre à condition que la classe remplaçante a le même type que celui de la classe remplacée mais leurs comportements peuvent être différents. Par contre, pour ce qui concerne le polymorphisme structurel, la composition ne permet pas un tel polymorphisme car cela revient au fait que tout soit prévu à la phase programmation.

L'*héritage* permet d'étendre une structure abstraite (appelée super-classe ou classe mère) afin de modifier un comportement donné, voir figure 4.3(a). En effet, il est possible donc d'avoir du polymorphisme par la spécification d'une structure abstraite et fixe pour le composant polymorphe, et en suite de modifier son comportement par héritage. Donc, le composant polymorphe n'est jamais complètement fermé car nous pouvons changer son comportement par de nouveaux composants répondants aux nouvelles exigences, et cela, par redéfinition des méthodes de la classe mère. Ainsi, cette technique, elle aussi, exige que tout soit prévu pendant la phase de programmation. En effet, l'héritage ne permet que le polymorphisme comportemental simple basé sur la substitution des instances des sousclasses de la classe mère du composant polymorphe. Le choix de la classe pour la substitution dépendra de son comportement, celle jugée capable de répondre mieux aux exigences de l'environnement sera choisie. Par contre, cette technique ne permet pas le polymorphisme structurel car tous les composants polymorphes issus de l'héritage sont manipulés à travers la même structure abstraite.

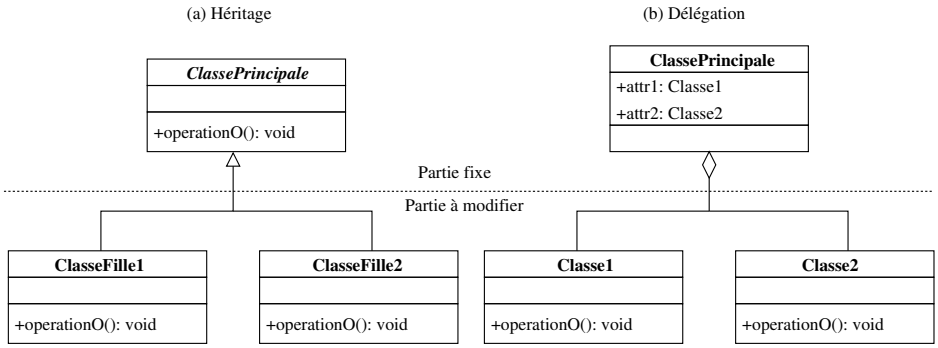


FIG. 4.3: Modification du comportement par (a)héritage et (b)délégation

La *délégation* est une relation de type « a un » entre les entités. Elle est basée sur l'agrégation, voir figure 4.3(b). Contrairement à l'héritage où les propriétés (attributs et opérations) sont propagées automatiquement, avec la délégation la propagation doit être réalisée manuellement. Elle permet donc à une entité de déléguer le travail à une autre entité [124]. Cette technique offre un degré de réutilisabilité aussi élevé que celui offert par l'héritage. Donc, en se basant que sur la délégation, il est possible d'avoir du polymorphisme en délégant les tâches à réaliser aux objets jugés capables de répondre bien aux exigences de l'environnement. Cependant, le choix de l'objet a qui la tâche sera déléguée peut être statique (lors de l'écriture du code du composant polymorphe) comme peut être dynamique (en remplaçant la valeur de l'attribut cible de délégation par une nouvelle occurrence). Cependant, cette technique exige que tout soit prévu lors de l'écriture du code source du composant polymorphe.

Discussion

Dans cette section, nous avons vu qu'il est possible d'avoir du polymorphisme, qui est limité au polymorphisme comportemental simple, basé sur les techniques de l'approche objet, l'assemblage, l'héritage ou la délégation. Cependant, tout doit être planifié à l'avance et exactement lors de la phase de programmation du composant. Ainsi, toute évolution des besoins et/ou de l'environnement impliquera une intervention externe au niveau du code source du composant polymorphe afin de prendre en compte cette évolution.

[3] Une classe est une description abstraite des données et du comportement d'objets.

Par ailleurs, nous signalons que le polymorphisme basé sur les techniques de l'approche objet peut rencontrer des problèmes. Par exemple, le problème de l'anomalie d'héritage [144] ne permet pas de composer facilement plusieurs comportements de manière cohérente.

4.4.3 Technique des conteneurs pour le polymorphisme

Les modèles de composant à base de conteneurs ont été proposés comme une réponse aux problèmes du monde industriels. Ils adoptent le principe du paradigme « séparation des préoccupations ».

Partant du vieux principe *diviser pour régner*, la séparation des préoccupations a comme objectif de dissocier la programmation des différentes préoccupations composantes un programme afin d'avoir une meilleure modularité et une bonne réutilisabilité [149][133]. Ses avantages sont :

- Facilite l'implémentation du programme qui est décrit avec des préoccupations réellement séparées [134] ;
- Facilite l'étude du programme séparé en différentes préoccupations car le code ne contient plus un mélange de préoccupations [135] ;
- Dans le cas où le couplage entre les préoccupations est faible ou nul, ça sera facile de modifier et de réutiliser chacune des préoccupations indépendamment des autres [136].

En effet, la séparation des préoccupations est considérée comme l'une des approches les plus prometteuses dans le génie logiciel. Son principe est que n'importe quel programme ayant un service à rendre sera vu comme un ensemble de préoccupations. Ce dernier comporte deux types de préoccupations :

- Préoccupation fonctionnelle, dite aussi préoccupation de base : Elle représente la raison pour laquelle le programme a été écrit, c'est-à-dire, le métier de ce programme ;
- Préoccupation non fonctionnelle : Elle regroupe les préoccupations secondaires permettant de réaliser la préoccupation fonctionnelle. Autrement dit, elle correspond aux services permettant de réaliser le métier (préoccupation de base) du programme.

Donc, le principal objectif des modèles de composant à base de conteneurs est de décharger les programmeurs du monde industriel de certaines préoccupations non-fonctionnelles, dites aussi services, tel que la persistance, les transactions, la distribution, etc., et cela, afin, d'une part, de bien gérer ces services d'une façon transparente aux programmeurs, et d'autre part, de permettre à ces derniers de concentrer que sur leur préoccupation.

Plus particulièrement, les modèles de composant à base de conteneurs s'adressent aux applications basées composants dans le monde industriel. En effet, nous donnerons d'abord une brève présentation des composants industriels et des conteneurs afin d'aborder dans la section 4.4.3 comment ces modèles permettent d'avoir du polymorphisme.

Composant industriel

Clemens Szyperski définit dans [141] un composant comme étant une unité de composition avec des interfaces contractualisées et un contexte de dépendance exprimé explicitement. Il ajoute ensuite qu'un composant peut être employé indépendamment et qu'il est sujet à composition par une tierce personne [4].

Cette définition est abstraite. Elle peut désigner un simple composant comme elle peut désigner un composant complexe. Cependant, tout composant se caractérise par l'*autonomie*, pour le déployer

[4] *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third party.*

et l'exécuter indépendamment des autres composants, l'*auto-description*, pour connaître ses caractéristiques dynamiques, la *configuration*, pour le paramétrer afin d'ajuster son comportement dans différents environnements, et l'*assemblage*, pour le connecter avec d'autres composants.

Un composant industriel représente un processus (transfert bancaire, gestion d'ordres, etc.) ou une entité (compte bancaire, client, etc.) utilisé couramment dans l'activité de différents domaines de l'industrie (banques, finances, etc.) (voir figure 4.4). Ses objectifs sont la réduction de la complexité de développement des logiciels et la réutilisation des codes existants afin d'augmenter la qualité des logiciels.

Tout composant logiciel a un cycle de vie composé de cinq étapes :

1. La spécification du type du composant consiste à donner au composant sa définition abstraite dans laquelle on spécifie les interfaces, les propriétés configurables et les modes d'interaction (synchrone, asynchrone, etc.) avec son environnement ;

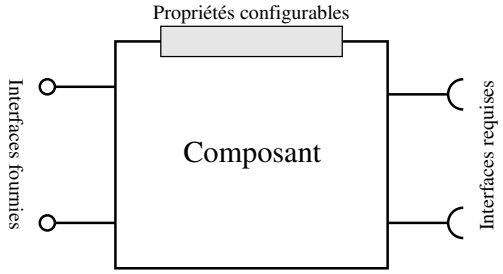


FIG. 4.4: Composant industriel

   Les interfaces permettrent à un composant industriel d'interagir avec son environnement et avec d'autres composants. Il y'a deux types d'interfaces :
   – Les interfaces fournies permettant à d'autres composants de solliciter les services fournis par le composant industriel. Ces interfaces sont un ensemble de signatures d'opérations dont l'implémentation est réalisée par le composant industriel ;
   – Les interfaces requises permettant au composant industriel d'exprimer ses besoins auprès de ceux qui les réalisent (environnement ou d'autres composants).

   Tandis que les propriétés configurables permettrent le paramétrage et le réglage du composant industriel afin d'avoir le comportement souhaité permettant de répondre aux exigences de l'environnement accueillant le composant industriel.
2. L'implantation du composant correspondant à la mise en œuvre de son type ;
3. Le paquetage et la diffusion consistent à archiver et puis regrouper les composants dans une même unité, appelée paquetage, afin de faciliter la diffusion et l'échange des composants entre développeurs, ce qui favorise la réutilisabilité et l'intégration de composants ;
4. Le déploiement des composants correspond à l'extraction du composant de son paquetage, de l'instancier, de vérifier sa compatibilité avec son environnement et d'éventuellement le configurer. Généralement, cette étape se base sur des descripteurs des composants ;
5. L'exploitation du composant consiste à exploiter le composant au sein de son application, c'est l'exécution. Dans cette étape, le composant interagit avec son environnement est remplit sa fonction.

Conteneur

L'idée du conteneur (en anglais *Container*) a été proposée afin de permettre aux développeurs de se concentrer que sur les préoccupations purement métier (fonctionnelles) de leurs applications sans

se préoccuper des services techniques (préoccupations non fonctionnelles) tels que, la persistance, la connexion à une base de données, les transactions, la sécurité, etc.

Un conteneur est donc un environnement d'exécution (dit aussi structure d'accueil) permettant le déploiement des composants industriels. Ses rôles sont la gérance du cycle de vie des composants industriels à savoir, la création, l'accès, l'activation, passivation, la destruction, etc. et la fourniture, d'une manière transparente, un certain nombre de services. En effet, les conteneurs ont permis d'extérioriser les préoccupations techniques (non fonctionnelles) des composants industriels et de déléguer leurs gestions au système de contrôle des composants, ce qui permet d'avoir un bon déploiement et une réutilisabilité maximale des composants industriels. Par exemple, un composant compte bancaire ne se préoccupe plus de la comptabilité de son compte et peut être redéployé pour tout compte bancaire même si l'ensemble de comptes ne sont pas comptabilisés de la même manière. Le paramétrage des services proposés est effectué au moment du déploiement des composants industriels et d'une manière déclarative, à travers le *descripteur de déploiement* qui doit être fourni par le développeur du composant pour décrire quelle classe est l'implémentation ainsi que les interfaces du composant industriel à déployer.

En effet, l'avantage des modèles de composant à base de conteneurs par rapport à l'approche objet est que l'implantation des services (préoccupations non fonctionnelles) requis n'est pas au niveau des composants industriels, ce qui n'est pas le cas avec l'approche objet où les services requis sont sollicités à travers des références sur des objets qui seront enfouis dans le code.

A titre d'exemple, les modèles de composant à base de conteneurs les plus répondus dans le monde industriel sont le modèle EJB (Entreprise Java Beans) [137] de Sun MicroSystem et le modèle CCM (CORBA Component Model) [138] de l'OMG. Par contre, les conteneurs de ces deux modèles ne sont pas extensibles. Il est donc impossible d'ajouter et/ou de modifier les services techniques mais leurs paramétrisations est possible. Cette limite a trouvé une réponse par la proposition de nouveau modèle, le modèle de composant à base de conteneur extensible [142].

Discussion

Les modèles de composants à base de conteneurs offrent un polymorphisme niveau applicatif exprimée par le paramétrage déclaratif des services techniques (préoccupations non fonctionnelles). Ainsi, les composants, qui s'exécutent au sein des conteneurs, déclarent juste les services fournis et les services requis, ensuite c'est aux conteneurs de les adapter à l'environnement. Donc, il est possible d'avoir du polymorphisme par le changement des implémentations des services techniques proposés par les conteneurs, qui se fait d'une manière transparente aux composants. En effet, l'idée est de varier l'environnement au lieu des composants. Cependant, ce polymorphisme est un polymorphisme comportemental simple, puisque les conteneurs n'ont aucune influence sur la structure des composants.

La bonne idée que nous tenions des conteneurs est bien *la séparation nette entre l'environnement et la préoccupation des développeurs (composants industriels)*, car cela permet une bonne réutilisabilité et laisse les développeurs se concentrer que sur leur préoccupation.

4.4.4 Réflexion pour le polymorphisme

Concepts et définitions

Transposée du domaine philosophique au domaine informatique dans le début des années 80 [158], la réflexion, d'après Brian Smith, est :

la capacité d'une entité à s'auto-représenter et plus généralement à se manipuler elle-même, de la même manière qu'elle représente et manipule son principal sujet. [5] [159].

En bref, la réflexion est la capacité d'un système de raisonner et d'agir sur lui-même afin qu'il puisse contrôler son propre fonctionnement et de l'adapter aux évolutions des besoins ou de l'environnement dans lequel il se trouve. Elle est introduite dans plusieurs domaines à savoir l'intelligence artificielle, la programmation fonctionnelle, la programmation logique et la programmation par objets [169]. Par exemple, Pattie Maes, avec ses travaux, a fortement contribué à l'introduire dans les langages à objets [170][171].

La réflexion représente beaucoup d'intérêt pour l'implémentation des systèmes réflexifs, car ces derniers ont la capacité à représenter un système (ses fonctionnalités et son implémentation), dans le système lui-même. Jacques Ferber mentionne plusieurs propriétés intéressantes des systèmes réflexifs [172] :

- Description de l'implémentation : allocation des entités, ramassage de miettes (en anglais *garbage collection*), extension incrémentale et ré-implémentation de la totalité du système ;
- Superviser les entités du langage : inspection, analyse et modification dynamique du code avec le langage même ;
- Interfacer et débuguer : implémentation des outils de développement avec le langage.
- Auto-réorganiser : capacités d'apprentissage du comportement du système, pour augmenter la consistance et l'efficacité ;

L'introduction de la réflexion dans la programmation par objets a permis l'apparition de plusieurs concepts à savoir :

Introspection vs. intercession Il y'a deux formes de réflexion : *l'introspection et l'intercession*. Bobrow et al. [173] ont définit ces deux formes de réflexion comme suit :

- L'introspection est la capacité d'un programme d'observer et donc de raisonner sur son propre état ;
- Par contre, l'intercession est la capacité d'un programme de modifier son propre état d'exécution, ou d'altérer sa propre interprétation ou signification ;

L'introspection et l'intercession, selon Bobrow et al., permettent à un programme de manipuler en tant que données les entités qui le représentent pendant sa propre exécution. En suite, il ajoutent que ces deux aspects (l'introspection et l'intercession) nécessitent un mécanisme pour encoder l'état d'exécution ; fournir un tel encodage est appelé *réification*.

Réflexion structurelle vs. comportementale Deux types de réflexion à distinguer : *la réflexion structurelle* et *la réflexion comportementale* [174]. Jacques Malenfant définit ces deux types de réflexion comme suit :

- La *réflexion structurelle* implique la réification complète à la fois du programme en cours d'exécution et ainsi que de ses types de données abstraites ;
- La *réflexion comportementale* implique la réification complète de sa sémantique (son processeur) de même qu'une réification complète des données utilisées pour exécuter le programme courant.

Cette distinction a été faite principalement parce qu'il est beaucoup plus facile d'implanter la réflexion structurelle que la réflexion comportementale [175].

En effet, la réflexion structurelle (resp. la réflexion comportementale) se traduit par la capacité d'un système à représenter et manipuler les structures de données (resp. la manière de réaliser ses services) qui le constituent (resp. qu'il offre).

[5] An entity's integral ability to represent, operate on, and otherwise deal with itself in the same way that it represents, operates or and deals with its primary subject matter

Méta-Objet Il existe plusieurs définitions du terme méta-objet. Parmi ces dernières, c'est la définition donnée par Gregor Kiczales qui nous intéresse :

Les éléments de base du langage de programmation — classes, méthodes et fonctions génériques — sont rendus manipulables en tant qu'objets. Comme ces objets représentent des fragments de programme, ils sont nommés méta-objets [176].

Cette définition a été reprise par plusieurs auteurs comme Ira Forman et al. [177] et Shigeru Chiba et al dans OpenC++ [153].

Lien méta Comme tous les systèmes, les systèmes réflexifs, eux aussi, ont des services à réaliser suivant des mécanismes d'exécution. Ainsi, avec les systèmes réflexifs, les services réalisés et les mécanismes d'exécution sont clairement séparés, puisque les deux appartiennent à des niveaux clairement séparés : le niveau de base et le méta-niveau. Les services d'un système sont décrits par des objets dans le niveau de base et les mécanismes d'exécution sont décrits par des méta-objets dans le méta-niveau [160][161][178][163]. Ainsi, la relation qui lie les objets aux méta-objets est appelée *lien méta*.

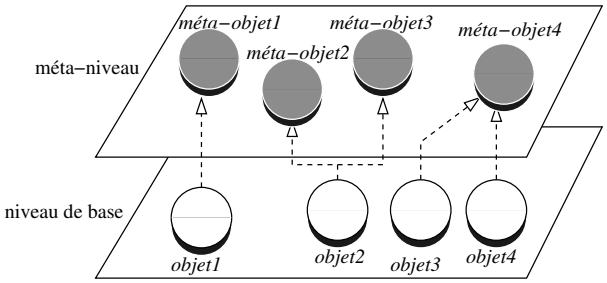


FIG. 4.5: Arités entre les objets et les méta-objets.

Comme c'est montré par la figure 4.5, l'arité du lien méta peut varier d'un système à l'autre. En effet, un objet peut donc être contrôlé par un ou plusieurs méta-objets et un méta-objet, lui aussi, peut contrôler un ou plusieurs objets.

Métaclasse Une métaclasse est une classe dont l'instance est aussi une classe. Par exemple, la classe Class du langage java. Elle sert donc à définir différentes sortes de classes en lui attribuant différentes propriétés de classe. Par exemple, être abstraite ou avoir plusieurs superclasses sont deux propriétés de classe [179]. Une métaclasse peut être manipulée directement par les développeur, ce qui est le cas dans Clos [180], ClassTalk [181], SOM [182], NéoClassTalk [183] ou MetaclassTalk [143]. Cela permet la définition des propriétés de classes réutilisables.

MOP Les protocoles de méta-objets (les MOPs) sont des interfaces au langage qui offrent aux utilisateurs la possibilité de modifier incrémentalement le comportement et l'implémentation du langage, de manière analogue à la possibilité d'écrire des programmes avec le langage. L'approche MOP est basée sur l'idée qu'on peut et on doit «ouvrir les langages», pour permettre aux utilisateurs d'ajuster la conception et l'implémentation à leurs besoins particuliers [176].

Les MOPs se basent sur certaines propriétés, parmi lesquelles : la réification, l'introspection, l'intercession et la réflexivité [184]. Il existe deux catégories de MOPs : les MOPs explicites et les MOPs implicites [185]. Les MOPs explicites sont les protocoles qui apparaissent explicitement dans le code du niveau de base. Autrement dit, le niveau de base a connaissance du méta-niveau. Par exemple, le MOP de Java fait partie des MOPs explicites, la classe des méthodes *java.lang.reflect.Method* définit une méthode *invoke*, qui permet d'invoquer une méthode réifiée sur un objet, n'est exécutée que si

elle apparaît explicitement dans le code du niveau de base. Par contre, les MOPs implcites sont les protocoles qui n'apparaissent pas dans le code du niveau de base, c'est-à-dire ils sont utilisés d'une manière transparente au niveau de base, cette transparence se traduit par les appels des méthodes constituants les MOPs implicites sont implicites et automatique. Autrement dit, le niveau de base n'a pas connaissance du méta-niveau. Par exemple, le MOP du langage Moostarp [186] fait partie des MOPs implicites, dans ce langage, c'est le méta-objet de l'objet receveur d'un message qui prend la main et contrôle l'envoi puis l'application de la méthode adéquate.

Techniques réflexives

Etendre un langage donné pour le rendre réflexif n'est pas une tâche facile. Cette extension se traduit par la définition d'un MOP permettant l'introduction et l'utilisation des méta-objets. Cependant, le moment d'existence et d'utilisation des méta-objets dépendra de la phase de la vie du programme écrit en langage à étendre. Cependant, il existe différentes techniques permettant d'implémentation de la réflexion dont la classification dépend de ce moment et aussi des caractéristiques du langage à rendre réflexif.

Comme nos concepts et nos expérimentations ont été programmés en langage java, nous donnons, ici, les techniques d'implémentation de la réflexion dans le langage java. Donc, dans le cas du modèle d'exécution des programmes java, il est connu que ce langage est semi-réflexif, autrement dit, il ne permet que la réflexion structurelle (c'est-à-dire l'introspection). Ainsi, la plupart des techniques d'extension de la réflexion dans java visent d'introduire la réflexion comportementale.

Il y'a quatre catégories de techniques d'extension de la réflexion dans java. Elle dépendent des phases de la vie d'un programme Java : la compilation, le chargement des classes dans la machine virtuelle et l'exécution. Les noms de ces catégories sont : *techniques transformation du code source*, *techniques transformation du code byte*, *techniques basées proxy* et *techniques modification de la Machine Virtuelle Java (JVM)*.

1. *techniques transformation du code source* : Ces techniques consistent à transformer le code source du programme à la phase de compilation afin de permettre le passage du niveau de base au méta-niveau. Ce passage peut être réalisé soit par la fusion des deux niveaux (niveau de base avec le méta-niveau), dans ce cas on dit que le passage est réalisé statiquement, soit par l'interprétation des traitements du niveau de base afin d'exécuter les traitements du méta-niveau, c'est-à-dire on n'aplatit pas la tour du méta-niveau et le passage d'un niveau à l'autre sera réalisé dynamiquement. Par exemple, OpenJava [154] fait partie des extensions réflexives de java basées sur la transformation du code source ;
2. *techniques transformation du code byte* : Ces techniques consistent à transformer le code byte de la classe compilée au cours du chargement des classes dans la machine virtuelle. De même pour ce genre de techniques, le passage du niveau de base au méta-niveau peut être réalisé statiquement ou dynamiquement. La transformation du code byte est souvent réalisée à travers des chargeurs de classes spécifiques. Par exemple, Javassist [157] est basé sur les techniques de transformation du code byte ;
3. *techniques basées proxy* : L'idée de ces techniques est de mettre un objet intermédiaire entre le programme et l'environnement. Le rôle de l'objet intermédiaire, appelé *proxy* ou *wrapper*, est de capturer (intercepter) les appels de méthodes provenant de l'environnement vers le niveau de base afin de les (appels) dérouter vers le méta-niveau. L'introduction de l'objet intermédiaire peut se faire au moment de la compilation, au moment de chargements des classes ou encore au moment de l'exécution, précisément lors de la création des objets. Par exemple, l'introduction du proxy se fait à la compilation avec OpenJava [154], au chargement des classes avec Kava [187] et au création des objets avec ReflectiveJava [188], ProActive [189], RAM [190] et Reflex [191] ;

4. *techniques modification de la Machine Virtuelle* : Afin d'introduire des intercepteurs, ces techniques consistent à modifier la machine virtuelle. Avec ces techniques, le passage du niveau de base au méta-niveau est réalisé statiquement. Le problème avec ces techniques, c'est qu'elle rendrent java non portable. Par exemple, MetaXa [192], Guaraná [193], Correlate [194] et Iguana/J [195] font partie de la catégorie d'extension réflexive de java basée sur les techniques de modification de la machine virtuelle.

Discussion

Dans la littérature, la réflexion est citée toujours comme le meilleur moyen permettant la flexibilité, qui dit la flexibilité dit bien l'adaptabilité. En effet, une application écrite en un langage réflexif est plus flexible que celle écrite en un langage non réflexif, ce qui lui permettra de s'auto-modifier. Cependant, la réflexion est un concept difficile à mettre en œuvre qui peut impliquer la présence de certaines contraintes dépendrant des caractéristiques du langage à rendre réflexif. Par exemple, l'extension de la réflexivité du langage java par modification de la machine virtuelle impliquera la perte de « portabilité ».

Nous avons présenté par catégorie les techniques permettant d'étendre la réflexion du langage java et chacune d'elles à ses avantages et ses inconvénients par rapport aux autres, ce qui influe sur le polymorphisme structurel et le polymorphisme comportemental :

1. Avec les langages dont la mise en œuvre de la réflexion est basée sur la transformation du code source ou la modification de la machine virtuelle, il est possible d'avoir du polymorphisme comportemental et du polymorphisme structurel par interception des messages (appels de méthodes, accès aux attributs, etc.) envoyés par l'environnement au composant polymorphe. Cependant, le polymorphisme structurel n'est que partiel, car dans le cas d'évolution des besoins ou de l'environnement, il est possible que ce dernier envoi un message que le niveau méta ne pourra pas l'intercepter. Par exemple, appel d'une méthode non déclarée par le composant polymorphe, ce qui implique donc que cette méthode n'est pas réifiée dans le niveau méta. Aussi, avec ce genre de langages, il n'est pas possible de modifier la signature de la classe du composant polymorphe sans intervention externe (par exemple, il n'est pas possible de modifier *public class C extends B {. . . }* en *public class C extends H implements I {. . . }*) ;
2. Avec les langages dont la mise en œuvre de la réflexion est basée sur la transformation du code byte, il est possible, lors du chargement du composant polymorphe, de modifier complètement sa structure, par l'ajout, la suppression et la modification de sa signature, ses méthodes et/ou ses attributs. Par contre, ce genre de manipulation est trop lourd, compliqué et en plus nécessite l'utilisation d'un chargeur de classes spécifique qui doit être modifié dans le cas d'évolution des besoins et/ou de l'environnement. Aussi, les environnements, dans lesquels le composant polymorphe est utilisé, doivent passer par ce chargeur de classes afin d'apporter les modifications sur le composant polymorphe et puis de le charger ;
3. Avec la technique des proxy, il est possible d'avoir un composant polymorphe. L'idée est donc de créer pour chaque environnement un composant spécifique représentant le composant polymorphe. Ensuite, dans l'objet proxy, il faut définir l'interprétation de tous les types de messages provenant des environnements. Les messages regroupent les messages d'instanciation et les messages d'invocation des méthodes des composants domaine-spécifiques. Donc, l'objet proxy jouera le rôle de niveau de base tandis que les composants domaine-spécifiques joueront le rôle de méta-niveau. Cependant, cette solution présente trois inconvénients : (1) Afin d'instancier le bon composant parmi ceux représentant le composant polymorphe, l'environnement doit toujours passer par l'objet proxy et pour respecter cette contrainte, la modification de l'environnement est obligatoire. (2) Dans le cas où le nombre d'environnements est important, le nombre de composants sera aussi important, ce qui rend la gestion des messages et les composants très lourde, ce qui représente une autre préoccupation. (3) L'évolution des besoins,

des environnements et/ou la modification des composants spécifiques touchant les types de ces derniers impliquera(ont) la modification de l'objet proxy.

4.4.5 Programmation orientée aspect pour le polymorphisme

Les carences de la programmation orientée objet

Avec la programmation orientée objet, le développement d'un logiciel ne se limite pas au développement d'un programme qui réalise une simple fonction. Des aspects tels que la distribution, la synchronisation ou encore la concurrence doivent être prises en considération. Les définitions de ces derniers se trouvent souvent entrelacées (mélangées) dans le code du programme. Cet entrelacement est à la source de la difficulté de la compréhension, de la maintenance et de la réutilisabilité des programmes [133]. Pour illustrer ces problèmes, nous reprenons l'exemple d'une librairie électronique donné par [143] et montré par la figure 4.6.

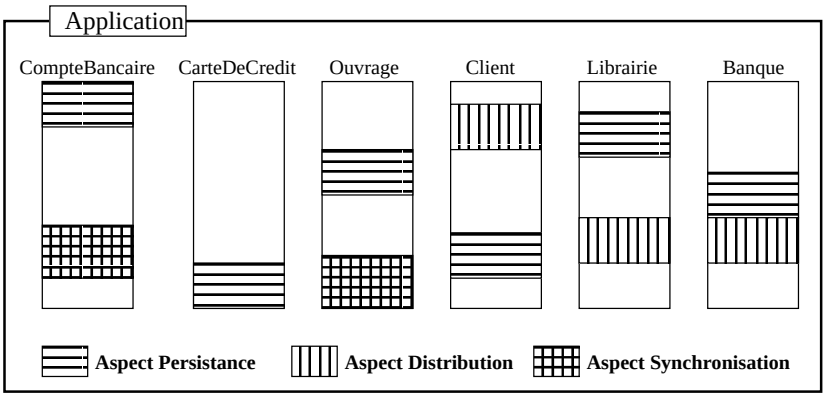


FIG. 4.6: Exemple d'aspects d'une librairie électronique

Cette librairie est accessible via un réseau et à travers lequel les clients peuvent consulter et éventuellement commander des livres. Les clients peuvent se connecter au même temps et les commandes peuvent être traitées en parallèle. L'application basée objet de cette librairie utilise les classes *Librairie*, *Client*, *Ouvrage*, *Commande*, *CarteDeCredit*, *CompteBancaire* et *Banque*.

Il est possible d'identifier au moins trois aspects au niveau de cette application, à savoir : la distribution, la persistance et la synchronisation. L'aspect distribution réside dans le fait que la banque, la librairie et les clients ne se trouvent pas dans la même zone, c'est-à-dire ils se trouvent sur des zones distantes. En conséquence, les instances de la classe librairie doivent gérer les communications distantes qui font donc partie de l'aspect distribution. Les objets comme, par exemple, les ouvrages mis en vente et les commandes en cours doivent survivre à l'arrêt de l'application. Pour ce faire, ces objets doivent être sauvegardés sur un support de sauvegarde et la modification de leurs états doit être répercutée sur leur sauvegarde. Donc, après une écriture dans une variable d'instance, la sauvegarde sur le support doit être actualisée. En effet, la sauvegarde représente une partie de l'aspect persistance. Quand à l'aspect synchronisation, il réside dans le fait qu'il faut gérer les accès concurrents aux instances des classes telles que les ouvrages et les comptes bancaires.

L'implémentation de la classe *CompteBancaire*, représentée dans la figure 4.7, montre l'entrelacement du code correspondant aux aspects synchronisation et persistance avec le code des métiers réalisés par la classe (les métiers sont débiter et créditer). En effet, le mélange des lignes de code représentant les aspects avec celles de l'application limite la réutilisabilité du code et rend difficile la distinction du code qui réalise l'aspect du code métier. En plus, dans le cas où on définit des sous-classes pour modifier, par exemple, le protocole de synchronisation pose aussi le problème d'anomalie d'héritage [144] (vu dans la section 4.4.2). Ce dernier vient du fait que la modification du protocole de

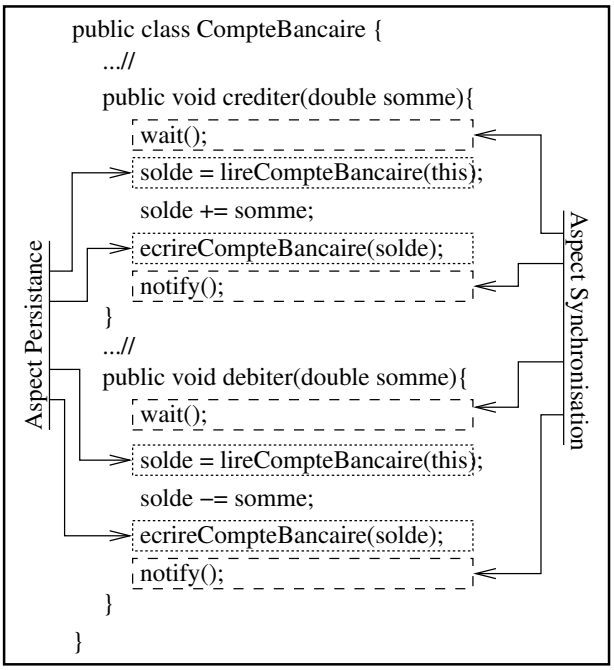


FIG. 4.7: Entrelacement des aspects avec le code fonctionnel de la classe CompteBancaire

synchronisation dans la sousclasse puisse obliger les développeurs de modifier quelques ou la totalité des méthodes de la classe mère. Ce qui affaiblit la réutilisabilité offerte habituellement par l'héritage.

Présentation et concepts

La programmation orientée aspect (AOP) [139][140] est apparue il y a quelques années au Xerox PARC en Californie et va dans le sens du principe de la séparation des préoccupations. Elle introduit un autre type d'organisation de la modularité des applications pour résoudre les problèmes posés par la programmation modulaire classique. Cette organisation consiste à appliquer le concept *inverser le sens de la dépendance* entre ce qui est fonctionnel et ce qui est non-fonctionnel [145][146][147]. L'inversion du sens de dépendance permet de faire totalement disparaître du code de l'application les dépendances relatives aux aspects.

Un aspect représente donc une préoccupation sous forme d'une abstraction d'un comportement ou d'un rôle, dont la particularité est de s'appliquer à un ensemble de classes, voir d'objets. Ainsi, un aspect correspond à la définition de structures et/ou comportements qui se supposent à la définition de l'application métier dite aussi aspect de base [148].

Le développement d'une application à base d'aspect consiste à définir :

- L'aspect fonctionnel, dit aspect de base, qui correspond à la tâche réalisée par l'application. Il définit le « Quoi » de l'application. Par exemple, l'aspect fonctionnel de la classe CompteBancaire correspond aux tâches débiter et créditer ;
- Les aspects non fonctionnels, dits aspects techniques, définissant les mécanismes qui régissent l'exécution de l'application. Ils définissent le « Comment » de l'application. Par exemple, la classe CompteBancaire comporte les aspect non fonctionnels synchronisation et persistance.

L'application de cette séparation sur la classe CompteBancaire a permis d'avoir des modules complètement séparés : le premier module est l'aspect fonctionnel et les deux autres modules correspondrent aux aspects non fonctionnels (synchronisation et persistance), voir figure 4.8. Cela a fait complètement disparaître le mélange du code non fonctionnel avec le code fonctionnel.

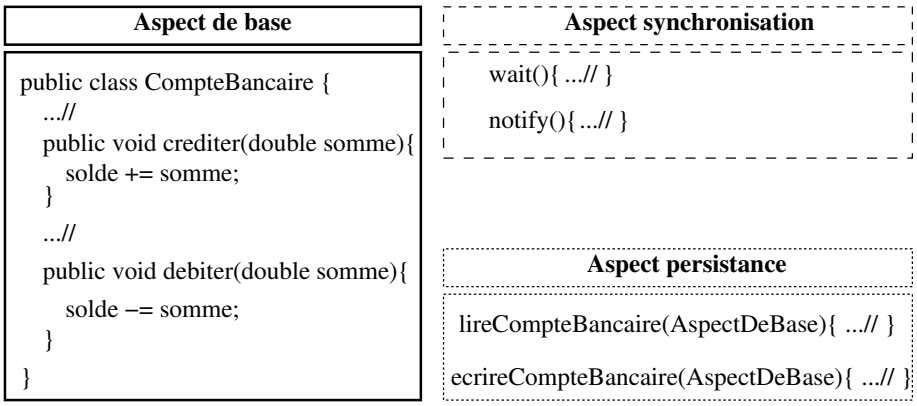


FIG. 4.8: Les aspects de la classe CompteBancaire

Comme, d'une part, les aspects sont des modules à part entière, et d'autre part, l'application subissant la séparation des aspects doit retrouver sa forme monolithique à l'exécution, il faut pouvoir composer les aspects (fonctionnel et non fonctionnels) à nouveau, d'où le besoin d'un mécanisme de composition. Ce dernier s'appele le *tissage* (en anglais weaving) (voir figure 4.9).

Le principe du tissage est d'établir une jonction entre l'aspect de base et les aspects non fonctionnels. Cette jonction s'établie au niveau des points d'entrées du flot d'exécution de l'aspect de base. Ces points sont appelés *points de jonction* [139]. Un point de jonction peut correspondre à une invocation de méthode, une affectation, etc. Par exemple, les invocations des méthodes crediter et debiter serviront comme des points de jonction.

Le mécanisme de tissage des aspects non fonctionnels avec l'aspect de base nécessite de connaître les points de jonction. En effet, les développeurs des applications à base d'aspects doivent définir les points de jonction utilisés par l'outil de tissage afin de composer chaque aspect non fonctionnel avec l'aspect fonctionnel. La définition des points de jonction est appelée *configuration des aspects* [148]. De plus, le mécanisme de tissage des aspects doit permettre une superposition harmonieuse des aspects non fonctionnels dans le cas où ces derniers ne sont pas orthogonaux. Cela permet de respecter la sémantique des applications à base d'aspects.

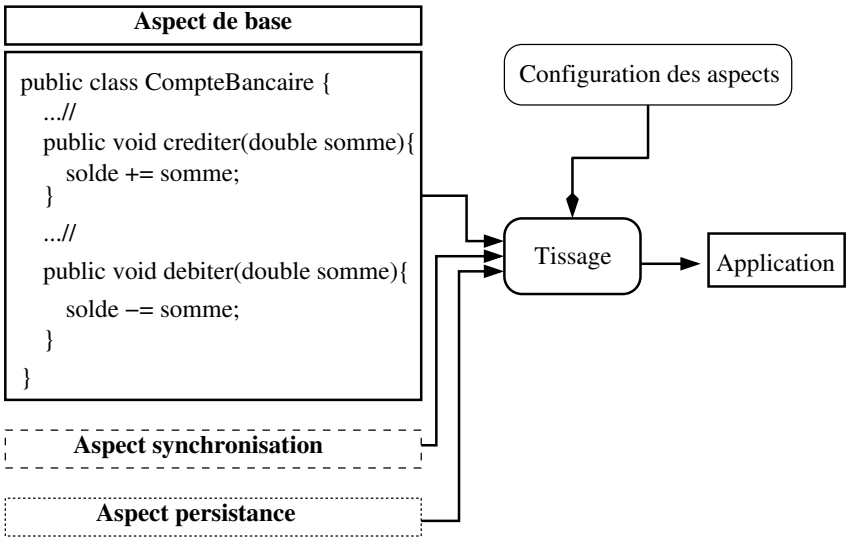


FIG. 4.9: Composition des aspects (tissage)

Le problème d'aspects non orthogonaux vient du fait que la composition d'un aspect de base

avec des aspects non fonctionnels peut utiliser plusieurs points de jonction et, inversement, un point de jonction peut servir pour la composition de différents aspects non fonctionnels. Ce qui résume le partage de points de jonction qui indique la corrélation des aspects non fonctionnels (conflit).

La résolution du problème de non orthogonalité des aspects dépend fortement de l'approche de mise en œuvre de la programmation orientée aspect. Certaines mises en œuvre fournissent aux développeurs de nouvelles constructions pour indiquer l'ordre d'exécution des aspects non fonctionnel, ce qui leurs permet d'établir l'ordonnancement des aspects non fonctionnels [150][151][152].

Diverses approches de mise en œuvre de la programmation orientée aspect

La programmation orientée aspect est une voie très prometteuse dans le génie logiciel. Elle a donné l'apparition de plusieurs travaux ciblant sa mise en œuvre, ainsi permettre de pouvoir bénéficier de ses avantages. Néanmoins, malgré l'avancement de ces travaux, la mise en œuvre de la programmation orientée aspect pose toujours des problèmes au niveau de la représentation et la composition des aspects.

En partant du principe que tout résultat d'un traitement informatique induit du couple < programme, interprète >. Dans [148], l'auteur montre que l'action sur le programme, ou sur l'interprète ou sur les deux peut modifier ce résultat. Cette action se traduit par la transformation du programme et/ou l'interprète. Mais, cette dernière n'est pas une tâche aisée. Suivant le langage utilisé et les outils disponibles, la transformation du programme peut s'avérer plus adaptée que celle de l'interprète et vice versa.

Selon les cas possibles de transformation, les différentes approches de mise en œuvre de la programmation orientée aspect sont classées en trois catégories :

Approche par transformation de programme Cette catégorie de mise en œuvre se base uniquement sur la modification du programme dans le couple < *programme, interprète* >. L'idée est donc de construire un nouveau programme monolithique à partir de l'aspect fonctionnel et les aspects non fonctionnels par l'inclusion des traitements de ces derniers au niveau des points de jonction. Cette construction peut être statique, c'est-à-dire avant ou pendant la compilation, comme peut être dynamique, c'est-à-dire au lancement ou pendant l'exécution.

Cette approche de transformation est à la base de différents travaux [153][139][154][151][155]. Par exemple, afin de rendre la programmation orientée aspect possible avec le langage java, le langage AspectJ [155] étend ce dernier par l'ajout de quelques nouvelles constructions syntaxiques, pour déclarer les aspects et les points de jonction, ainsi qu'un tisseur, nommé *ajc*, pour composer l'aspect de base et les aspects non fonctionnels. La composition se fait pendant la phase de compilation et elle est sous forme de règles de transformation. Mais, avec le langage AspectJ, toute modification au niveau de l'aspect de base et/ou des aspects non fonctionnels impliquera la recomposition de nouveau des aspects de l'application. A l'inverse d'AspectJ, JAC (Java Aspect Component) [156] ne requit aucune extension du langage java. Il est basé sur la définition d'un encapsulateur (en anglais *wrapper*) de façon programmée. La particularité de JAC est de permettre la manipulation (ajout, retrait ou re-ordonnancement) des aspects d'une façon dynamique. Le fonctionnement de JAC est basé sur *Javassist* [157] qui permet d'instrumenter le code byte des objets comportant l'aspect de base lors de la phase de chargement des classes dans la machine virtuelle java.

Approche par transformation de l'interprète Contrairement à l'approche précédente, l'approche par transformation d'interprète se base uniquement sur la variation de l'interprète dans le couple < *programme, interprète* >. L'idée est donc de construire un nouvel interprète s'appuyant sur la réflexion [159] [160] à partir des aspects non fonctionnels.

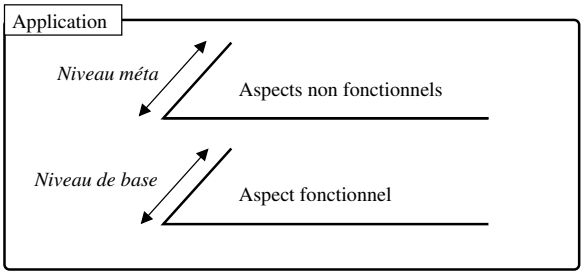


FIG. 4.10: Séparation des aspects en deux niveaux(base et méta)

La réflexion permet donc de séparer l'aspect fonctionnel de l'application des aspects non fonctionnels par la structuration de l'application en deux niveaux, un niveau de base et un niveau méta (voir figure 4.10).

Le niveau de base, dans lequel résident plusieurs objets, décrit l'aspect de base de l'application et le niveau méta, dans lequel, lui aussi, résident plusieurs objets appelés méta-objets, décrit les aspects non fonctionnels de l'application. Le passage d'un objet du niveau de base au niveau méta se fait par l'intermédiaire des liens appelés *liens méta*.

L'approche par transformation de l'interprète est à la base de plusieurs formalismes [162] [163] [164] [165] [166] [167] [168].

Approche hybride A l'inverse des deux approches de mise en œuvre présentées dans les deux sections précédentes, l'approche hybride transforme aussi bien le programme que l'interprète, pour introduire les aspects non fonctionnels.

L'intérêt de l'approche hybride réside, d'une part, dans la possibilité de choisir l'une des deux transformations s'avérant plus adaptée pour certains aspects particuliers, d'autre part, dans la possibilité de choisir entre ces deux transformations suivant ce que l'on veuille privilégier, l'efficacité ou la flexibilité d'une application. L'introduction des aspects non fonctionnels par transformation de programme permet d'avoir une application plus efficace, par contre, celle par transformation de l'interprète permet d'avoir une application plus flexible. Par exemple, pour optimiser une application contenant des boucles, il est plus facile de rendre cette dernière efficace en introduisant l'aspect optimisation par transformation de l'aspect de base (fusionner les boucles) que par transformation de l'interprète. Par contre, c'est l'inverse pour l'aspect synchronisation [148].

Javassist [157] est l'un des rares représentants de cette approche. Il introduit des APIs et des capacités réflexives permettant ainsi de transformer le programme et l'interprète. Cependant, son utilisation n'est possible qu'avec les programmes java.

Discussion

La programmation orientée aspect a apporté une réponse au problème de réutilisabilité posé par les applications ayant le code fonctionnel entrelacé avec le code non fonctionnel. Contrairement aux techniques introduites par l'approche objet et les modèles de composants à base de conteneurs, la programmation orientée aspect permet la modification et/ou le court-circuit des méthodes et des instructions élémentaires de l'application. Par exemple, avec le langage AspectJ, il est possible d'insérer un aspect permettant de modifier la valeur d'une variable avant, après ou à la place d'une instruction d'affectation désignée comme un point de jonction.

La programmation orientée aspect permet d'avoir du polymorphisme comportemental simple. Cependant, elle n'assure que partiellement le polymorphisme structurel car le problème de modification

de la signature d'une classe persiste toujours. Par contre, elle permet d'avoir le polymorphisme comportemental dur dans le cas où ce dernier ne nécessitera pas des modifications préalables au niveau de la signature de la classe.

Nous signalons aussi que l'inconvénient des approches de mise en œuvre basées sur la transformation du programme est que toute modification, que ce soit au niveau de l'aspect de base ou au niveau des aspects non fonctionnels, impliquera la recompilation de nouveau de l'application. Par contre avec les approches de mise en œuvre basées sur la transformation de l'interprète, il est possible d'ajouter, de supprimer ou de changer à l'exécution (dynamiquement) des aspects non fonctionnels de l'application, ce qui permet de répondre aux évolutions des besoins ou de l'environnement sans modifier l'aspect fonctionnel.

4.5 Conclusion

Dans ce chapitre, nous avons d'abord présenté les faiblesses des composants domaine-spécifiques lors de leur utilisation pour conserver une sémantique dans différents domaines. Ensuite, nous avons présenté le modèle de composant domaine-polymorphe que nous voulions concevoir.

Comme le composant domaine-polymorphe est un composant logiciel, dont sa conception doit tenir de compte des technologies logicielles, nous avons défini le terme polymorphisme dans le cadre software et, avons présenté et discuté les techniques des technologies jugées capables, même partiel, d'avoir dudit polymorphisme.

Les technologies présentées dans ce chapitre donnent des réponses différentes aux questions suivantes :

- Qui va effectuer le polymorphisme ? Cette question permet l'identification du responsable (au pire des cas c'est le programmeur et au meilleur des cas c'est le composant polymorphe lui-même !) sur la réalisation du polymorphisme ;
- Comment effectuer le polymorphisme ? Cette question permet de spécifier les mécanismes permettant la réalisation du polymorphisme. Par exemple, par l'héritage, par la transformation du code source, etc. ;
- Quant effectuer le polymorphisme ? Cette question permet la détermination du moment de déclenchement du changement.

Cependant, les réponses apportées par ces technologies dépendront des points suivants :

- Les caractéristiques du langage de programmation en lequel le composant polymorphe a été programmé. Par exemple, la réflexivité est une caractéristique des langages réflexifs, etc ;
- Les exigences de l'environnement dans lequel le composant polymorphe va être utilisé. Par exemple, un environnement peut exiger sur le composant polymorphe de respecter une structure et/ou un comportement bien déterminé(e)(s) ;
- L'approche sur laquelle la conception du composant polymorphe a été basée. Par exemple, une conception basée sur l'approche aspect donne des réponses différentes de celles données par une conception basée que sur l'approche objet.

En effet, pour ne pas être lié qu'à une seule technologie, le modèle de composant domaine-polymorphe, que nous allons proposer dans le chapitre suivant, doit prendre en compte les différentes technologies et être ouvert, c'est-à-dire de garantir que son implémentation bénéficiera de la totalité des caractéristiques de la technologie utilisée. Ainsi, la satisfaction de toutes les technologies sera garantie.

Chapitre 5

Conception de Composant Domaine-Polymorphe

Dans ce chapitre, nous donnons toutes les étapes de notre approche de conception de conception de composant domaine-polymorphe [196]. Cette approche fixe d'abord les objectifs et les contraintes, qui sont nos besoins. Ensuite, à partir de la structure abstraite commune aux composants domaine-spécifiques ayant les comportements spécifiés suivant une sémantique donnée, notre approche propose des modèles de composants intermédiaires jusqu'à parvenir au modèle de composant domaine-polymorphe.

Les modèles de composants intermédiaires sont le modèle de composant domaine-spécifique modulaire et composable, et le modèle de composant domaine-spécifique flexible.

Après avoir évalué le modèle du composant domaine-polymorphe, nous donnons les améliorations apportées à ce modèle afin de réduire le nombre ses composants. Finalement, nous présentons les opérations et le modèle d'exécution du modèle amélioré ainsi que son méta modèle.

5.1 Vers un modèle de composant domaine-polymorphe

Notre approche part des objectifs et des contraintes, qui sont nos besoins, de la structure abstraite commune aux composants domaine-spécifiques, qui est le moyen existant, et des techniques pour le polymorphisme présentées dans le chapitre précédent, qui sont les possibilités technologiques, pour proposer d'abord le modèle de composant domaine-spécifique modulaire et composable, qui sera suivi par la proposition du modèle de composant domaine-spécifique flexible et nous terminerons par la proposition du modèle de composant domaine-polymorphe.

5.1.1 Objectifs et contraintes

Nos objectifs sont :

- La réutilisabilité : le composant domaine-polymorphe est réutilisable sous plusieurs modèles de calcul, même dans le cas où un modèle de calcul n'existait pas au moment de sa conception, ce qui implique que la conception de la couche Noyau se fait une fois pour toute ;
- La conception de la couche Noyau ne nécessite que la connaissance des caractéristique de la sémantique de spécification ;
- Permettre un passage explicite entre les deux modèles de calcul (les deux sémantiques) car cela fait partie de la conception des systèmes. En effet, les concepteurs auront le choix entre différentes possibilités pour générer des données internes à partir des données externes et vice versa. Ces possibilités peuvent être l'échantillonnage, l'interprétation, la détection, une simple conversion, etc.

Cependant, nous devons respecter les contraintes suivantes :

– Le composant domaine-polymorphe doit être portable : Aucune modification n'est exigée au niveau des domaines pour que le composant domaine-polymorphe puisse fonctionner correctement ;
– L'implémentation du composant domaine-polymorphe doit être possible en n'importe quel langage car nous avons vu dans le chapitre 4 que les caractéristiques des langages de programmation (comme la réflexion, etc.) influent sur le polymorphisme. Donc, nous tenons compte des différentes caractéristiques des langages.

5.1.2 Composant domaine-spécifique modulaire et composable

A cause de sa structure, qui est fixe, et l'entrelacement de ses différentes opérations, un composant domaine-spécifique n'est pas réutilisable dans d'autre domaine et même pour d'autre sémantique à conserver. En effet, le but de cette étape est d'introduire la flexibilité aux composants domaine-spécifiques.

Nous partons du principe *séparer pour régner* que nous le déclinons ici *séparer pour avoir la flexibilité* afin d'introduire la flexibilité sur le composant domaine-spécifique. Ce principe est réalisé par l'application du paradigme *séparation des préoccupations* sur les composants domaine-spécifiques conservant une sémantique donnée.

Application de la séparation des préoccupations

Dans le chapitre 3, nous avons vu que tout composant domaine-spécifique conservant une sémantique donnée possède un ensemble de types d'opérations. Chaque type d'opérations est une préoccupation. En effet, cet ensemble n'est qu'un ensemble de préoccupations, voir figure 5.1.

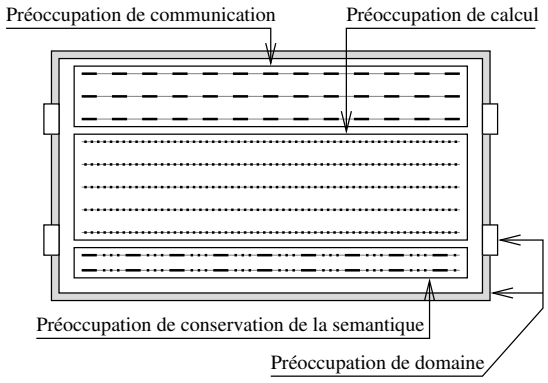


FIG. 5.1: Préoccupations possédées par un composant domaine-spécifique conservant une sémantique donnée.

L'ensemble de préoccupations se divise en deux parties principales de préoccupations :

1. Partie fonctionnelle ou de base : Elle correspond à la *préoccupation de calcul* qui représente l'entité dont la sémantique doit être conservée par le composant domaine-spécifique (par exemple, la *partie synchrone*). Cette partie justifie l'existence du composant domaine-spécifique ;
2. Partie non fonctionnelle : Elle assure l'exécution de la préoccupation de calcul (partie fonctionnelle) suivant la sémantique de spécification sous le modèle de calcul implémenté par le domaine hôte. Elle est composée d'un ensemble de préoccupations non fonctionnelles. Ces

dernières fournissent tous les services nécessaires et répondent aux besoins de la partie fonctionnelle pour que son (la partie fonctionnelle) exécution s'effectue correctement avec le respect total de sa sémantique (sémantique de spécification, dite aussi sémantique interne) ainsi que celle (dite aussi sémantique externe) du domaine hôte. Elle est composée de :

– La *préoccupation domaine* : Elle correspond à la structure (classe et ports de communication) exigée par le domaine. Elle désigne le type de composant domaine-spécifique. Par exemple, un composant conçu spécifiquement pour fonctionner dans le domaine DE, il doit avoir une structure correspondant à celle exigée par le domaine DE ;
– La *préoccupation de conservation de la sémantique interne* : Elle correspond aux opérations permettant la conservation de la sémantique interne du composant domaine-spécifique (par exemple, la *partie synchrone*) tout en tenant de compte de la sémantique du domaine hôte ;
– La *préoccupation de communication* : Elle correspond à l'ensemble d'opérations de communication qui sont spécifiques et permettrent la manipulation des ports du composant domaine-spécifique, et cela, pour lire/écrire depuis/sur les ports.

Architecture

Nous séparons puis modélisons chaque préoccupation par un composant et l'interconnexion de l'ensemble de composants permet d'avoir une nouvelle architecture de composant, qui est l'architecture du composant domaine-spécifique modulaire et composable, voir la figure 5.2.

Les liens (représentés par des flèches) reliant les différents composants du composant domaine-spécifique modulaire et composable sont fixes et connus d'avance, c'est-à-dire que toutes les interconnexions doivent être établies avant l'exécution.

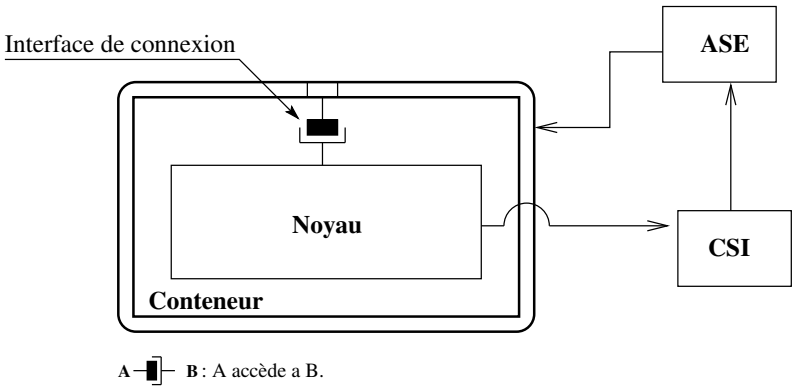


FIG. 5.2: Architecture du composant domaine-spécifique modulaire et composable.

Les composants du composant domaine-spécifique modulaire et composable sont le composant Conteneur, le composant Noyau, le composant ASE (Adaptation à la Sémantique Externe) et le composant CSI (Conservation de la Sémantique Interne), qui modélisent, respectivement, la préoccupation domaine, la préoccupation de calcul, la préoccupation de communication et la préoccupation de conservation de la sémantique interne. Ces composants (sauf le composant Noyau) sont spécifiques à un seul domaine. Par contre, le composant CSI est non seulement spécifique à un seul domaine mais aussi à une seule sémantique à conserver. Ces composants sont détaillés comme suit :

Composant Conteneur L'idée de définir un composant Conteneur est inspirée des conteneurs CCM (CORBA Component Model) de l'OMG et EJB (Enterprise Java Beans) de Sun Microsystem, dans le sens de cacher la complexité des environnements (dans notre cas la complexité des domaines).

Le composant Conteneur assure au composant domaine-spécifique modulaire et composable d'avoir la structure (classe et ports) exacte qui répond aux exigences du domaine. En effet, séparer la structure du composant domaine-spécifique de la partie de calcul décharge les concepteurs des systèmes de la préoccupation domaine. Il sert à encapsuler les composants Noyaux, et ainsi, de déclencher leurs réactions à travers leurs interfaces de connexion et de leurs permettre donc d'interagir avec l'environnement (domaine) d'une manière transparente. Donc, le composant Conteneur est vu comme une structure d'accueil qui représente un espace d'exécution des composants Noyaux. Aussi, le composant Conteneur présente une interface pour permettre aux composants Noyau, ASE et CSI de le référencer, et cela, pour des raisons d'assemblage et d'exécution. Parmi les bonnes propriétés du composant Conteneur c'est qu'il a la capacité de création des ports de communication pendant l'exécution (dynamiquement), ce qui donne l'avantage d'utiliser qu'un seul composant Conteneur par domaine pour exécuter tous les composants Noyaux. Donc, à travers l'interface de connexion, il demande au composant Noyau les informations sur les ports à créer. Ces informations sont les noms des ports, types des ports (qui correspondent aux types des données communiquées), et natures des ports (qui correspondront au sens de direction des données communiquées : en sortie, en entrée ou en entrée/sortie).

La création des bons ports de communication spécifiques au domaine est assurée par le composant Conteneur. En effet, la *propriété de ne pas avoir des ports qu'à l'exécution* est cruciale et *permet au composant Conteneur d'être indépendant et de pouvoir encapsuler n'importe quel composant Noyau*.

Composants ASE et CSI Les composants ASE et CSI sont des composants dont l'utilisation n'est limitée que pour un seul domaine, celui pour lequel ils ont été conçus. Ils sont les intermédiaires entre le composant Conteneur et le composant Noyau. Ils coopèrent ensemble pour assurer le bon comportement des deux composants, Conteneur et Noyau, tout en assurant le bon passage entre les deux sémantiques. Le composant ASE est utilisé directement par le composant CSI, et cela, à travers la liaison statique qui les relie.

ASE Le composant ASE possède le savoir d'utilisation des ports de communication du composant Conteneur. Il définit deux méthodes de communication paramétrables, une pour la lecture depuis les ports d'entrée et l'autre pour l'écriture sur les ports de sortie. Ces méthodes utilisent les méthodes de communication spécifiques aux ports du composant Conteneur et elles sont, à travers l'interface du composant ASE, invoquées par les composants CIS.

Par exemple, pour un composant ASE spécifique au domaine DE, nous définissons les deux méthodes *write* et *read* comme le montre le listing 5.1.

Listing 5.1: Méthodes write et read du composant ASE spécifique au domaine DE

```
// Méthode write
public void write (NameOfPort, Data, Delay){
  (RefContainer.getPort(NameOfPort)).send(Data, Delay);
}

...//

public Data read (NameOfPort) {
  return (RefContainer.getPort(NameOfPort)).get();
}
```

Le composant ASE assure la dernière étape de la phase de communication du composant Noyau avec le domaine hôte. Pour cela, il possède une référence sur le composant Conteneur qui lui permet, en utilisant le nom comme moyen d'identification, de localiser le port désiré.

Le nombre de composants ASEs nécessaire pour tous les domaines est toujours le même et égal aux nombre de ces derniers à raison d'un composant par domaine. En effet, une fois conçu, le composant ASE sera toujours réutilisable pour tout type de sémantique à conserver car il est indépendant

de la sémantique interne. Le choix de faire sortir la préoccupation de communication à part et de la modéliser par un composant ASE revient au fait que les méthodes de communication spécifiques aux ports sont toujours les mêmes et leur utilisation ne se varie pas mais avec des valeurs de paramètres différentes.

Le composant ASE permet donc :

- De cacher la complexité d'utilisation des ports de communication qui se traduit par la non préoccupation de comment localiser et utiliser les ports de communication, en proposant des méthodes de plus en plus génériques et paramétrables au composant CSI ;
- De libérer le composant Conteneur : Il était possible de ne pas introduire le composant ASE et de définir ses méthodes dans le composant Conteneur, mais nous n'avons pas voulu que l'utilisateur (qui conçoit les composants CSI et Noyau) manipule le composant Conteneur, ce qui ne cache pas vraiment la complexité du domaine hôte.

CSI Le composant CSI propose un ensemble d'opérations au composant Noyau pour lui permettre de s'exécuter correctement d'une façon transparente au domaine. A part les opérations de communication et de génération[1] des données d'entrée en fonction des données de sortie et vice versa, qui sont, au niveau abstrait, communes à tous les composants CSI, le nombre d'opérations proposées par chacun de ces derniers ainsi que leurs comportements dépendront de la sémantique à conserver ainsi que la sémantique du domaine. Les opérations de communication proposées par un composant CSI sont plutôt spécifiques à la sémantique du Noyau mais leurs comportement dépendent de la sémantique du domaine, c'est à dire, que le composant Noyau utilise les opérations de communication suivant sa sémantique tandis que leurs comportements dépendent de la sémantique du domaine.

Le composant CSI utilise les ports de communication du composant Conteneur indirectement en passant par les opérations de communication du composant ASE. Nous pouvons considérer le composant CSI comme un composant à double facette jouant l'intermédiaire entre les deux sémantiques. Le composant CSI est complètement dédié à la sémantique interne (sémantique du composant Noyau) en fonction de la sémantique externe, ce qui nous oblige d'avoir un composant CSI pour chaque sémantique interne à conserver et pour chaque domaine.

Au-delà de la communication, le composant CSI assure :

- La configuration, si nécessaire, du composant Conteneur et les ports de communication qui font partie de la conservation de la sémantique interne. Par exemple, pour qu'une sémantique donnée puisse être conservée dans le domaine SDF, il faut fixer le taux de flot de données sur tous les ports à un, dans ce cas, le composant CSI doit fixer le taux de données sur tous les ports du composant Conteneur à un ;
- L'accomplissement du bon fonctionnement du composant domaine-spécifique modulaire et composable, et cela, en fournissant les instructions nécessaires. Par exemple, dans le cas où un composant Noyau synchrone, qui est encapsulé par un composant Conteneur spécifique au domaine SDF, n'a pas produit des données pour un port donnée, le composant CSI doit produire et envoyer des données d'absence sur le port concerné ;
- Assure la génération des données internes depuis les données externes et vice versa, ce qui permet aux concepteurs d'expliciter le passage des données entre les deux sémantiques.
  La manière par laquelle les données seront générées est complètement laissée aux chois des concepteurs car cela fait partie de la conception des systèmes. En effet, la génération des données peut être par échantillonnage, interprétation, etc.
  Cette fonctionnalité est introduite à cause de la différence entre les deux sémantiques au niveau du protocole de communication et les formats des données communiquées.

[1] La génération des données peut être faite par échantillonnage, interprétation, détection, etc.

Composant Noyau Le composant Noyau modélise la préoccupation fonctionnelle (partie de calcul). Il représente une entité obéissant à la sémantique de sa spécification. Par exemple, il peut représenter un module synchrone. Tout composant Noyau doit implémenter une interface générique prédéfinie pour permettre aux composants Conteneurs de récupérer les informations sur les ports de communication à créer et de déclencher son exécution.

Dans tous les domaines, le composant Noyau interagit avec son extérieur à travers les ports fournis par le composant Conteneur, et cela, en utilisant les méthodes de communication , qui sont fournies par le composant CSI et correspondent à sa (Noyau) sémantique. Par contre, avec le composant domaine-spécifique modulaire et composable, le choix du composant CSI correspondant à la sémantique du composant Noyau et à celle du domaine hôte se fait manuellement.

Dans cette étape de notre approche, le composant domaine-spécifique modulaire et composable a comme avantage la permission de réutiliser dans son domaine ses composants (Conteneur, ASE et CSI) pour exécuter n'importe quel composant Noyau. Pour cela, il suffit de concevoir les composants CSIs pour les différentes sémantiques à conserver et de choisir par la suite, lors d'établissement des liens d'interconnexion, le bon composant CSI correspondant à la sémantique du composant Noyau à exécuter.

5.1.3 Composant domaine-spécifique flexible

Le composant domaine-spécifique modulaire et composable nous a permis d'avoir une flexibilité permettant la réutilisabilité. Cependant, l'aspect statique de ses liens d'interconnexion entre ses différents composants pose le problème *tout doit être prévu à l'avance*. En effet, l'établissement des liens se fait avant l'exécution et nécessite une intervention externe (qui peut être un programmeur, un logiciel, etc.). Plus précisément, le problème suscité est posé par deux liens : le lien connectant le composant Conteneur au composant Noyau et celui connectant le composant Noyau au composant CSI, qui doivent être rétablis à chaque fois que nous déplacions le composant Noyau d'un composant Conteneur à un autre afin de l'exécuter dans différents domaines. Ces deux liens doivent être donc dynamiques. Par contre, les liens entre les composants CSI et ASE, et entre Conteneur et ASE sont statiques, ils sont établis une fois pour toute et ne posent aucun problème.

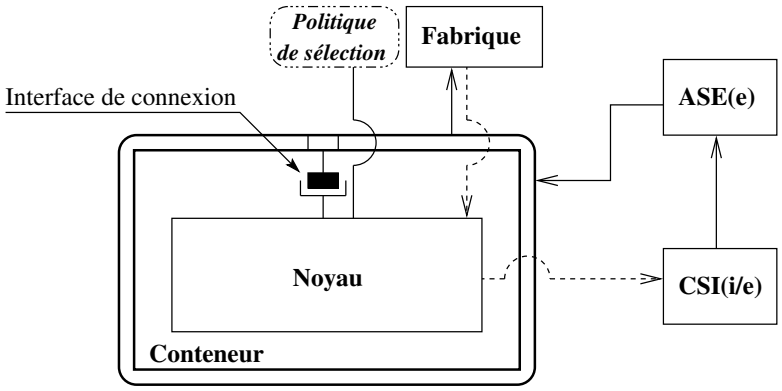
Comme réponse à ce problème, nous proposons dans cette étape un nouveau composant, appelé composant domaine-spécifique flexible.

Architecture

L'architecture du composant domaine-spécifique flexible est obtenue par l'ajout des éléments à l'architecture du composant domaine-spécifique modulaire et composable, voir figure 5.3. Ces éléments sont :

– *La politique de sélection :* Elle est ajoutée au composant Noyau en lui permettant de trouver, parmi l'ensemble de composants CSIs dédiés aux différentes sémantiques à conserver, le bon composant CSI correspondant à sa sémantique et à celle du domaine hôte.
Comme le composant Noyau a maintenant la capacité de récupérer le bon composant CSI, il est possible donc d'établir dynamiquement (à l'exécution) le lien (représenté par une flèche discrète sur la figure 5.3) entre les deux composants (Noyau et CSI). Ainsi, la réutilisabilité du composant Noyau vis-à-vis des différents composants CSIs se fait d'une façon dynamique, c'est-à-dire que nous n'avons plus besoin d'établir manuellement ce lien d'interconnexion. En effet, l'ajout de cette politique est justifié. Par contre, nous l'avons nommée *politique* pour dire que son implémentation dépend des caractéristiques du langage utilisé dans l'implémentation du composant domaine-spécifique flexible ;

– *Le composant Fabrique :* Sa tâche est, à partir d'une identité, de trouver n'importe quel composant Noyau pour tout composant Conteneur, et ensuite, d'établir dynamiquement le lien (représenté par une flèche discrète sur la figure 5.3) entre eux (Conteneur et Noyau). Les raisons justifiant l'ajout du composant Fabrique ne sont que des raisons de réutilisabilité.



ASE(e): ASE spécifique à la sémantique externe e.
CSI(i/e): CSI spécifique à la sémantique interne i et à la sémantique externe e.

FIG. 5.3: Architecture du composant domaine-spécifique flexible.

Comme les liens entre les composants Noyau et CSI, et entre les composants Conteneur et Noyau doivent être établis dynamiquement, le composant domaine-spécifique flexible passera forcément par une phase d'assemblage.

Relations de dépendances fonctionnelles entre l'ensemble de composants Elles sont les liens d'interconnexion des composants du composant domaine-spécifique flexible. Elles sont représentées sur la figure 5.3 par des flèches continues pour les liens statiques et par des flèches discrètes pour les liens dynamiques.

Conceptuellement, les relations de dépendances sont des liens de communication et à travers lesquels les composants du composant domaine-spécifique flexible coopèrent ensemble pour accomplir le fonctionnement de ce dernier. Concrètement, ces relations, tout dépend des caractéristiques du langage de programmation utilisé et des techniques d'implémentation choisies (voir chapitre 4), ne sont que des entités logiciel permettant l'interconnexion de l'ensemble de composants et qui influent le comportement de ces derniers. Donc, elles peuvent être des objets, des interfaces, des aspects, etc.

Nous avons représenté ces relations par des flèches dont le sens signifie que les composants de départ ont des références sur ceux de l'arrivée. La flèche discrète signifie que le composant d'arrivée ne sera connu pour le composant de départ qu'à l'exécution et plus précisément à la phase d'assemblage. Donc, pour établir un lien dynamique, le composant de départ doit avoir la capacité de trouver d'abord celui de l'arrivée.

Assemblage

Le diagramme UML de séquence de la figure 5.4 décrit la phase d'assemblage réalisée par l'ensemble de composants du composant domaine-spécifique flexible dont le rôle de chacun d'eux est détaillé comme suit :

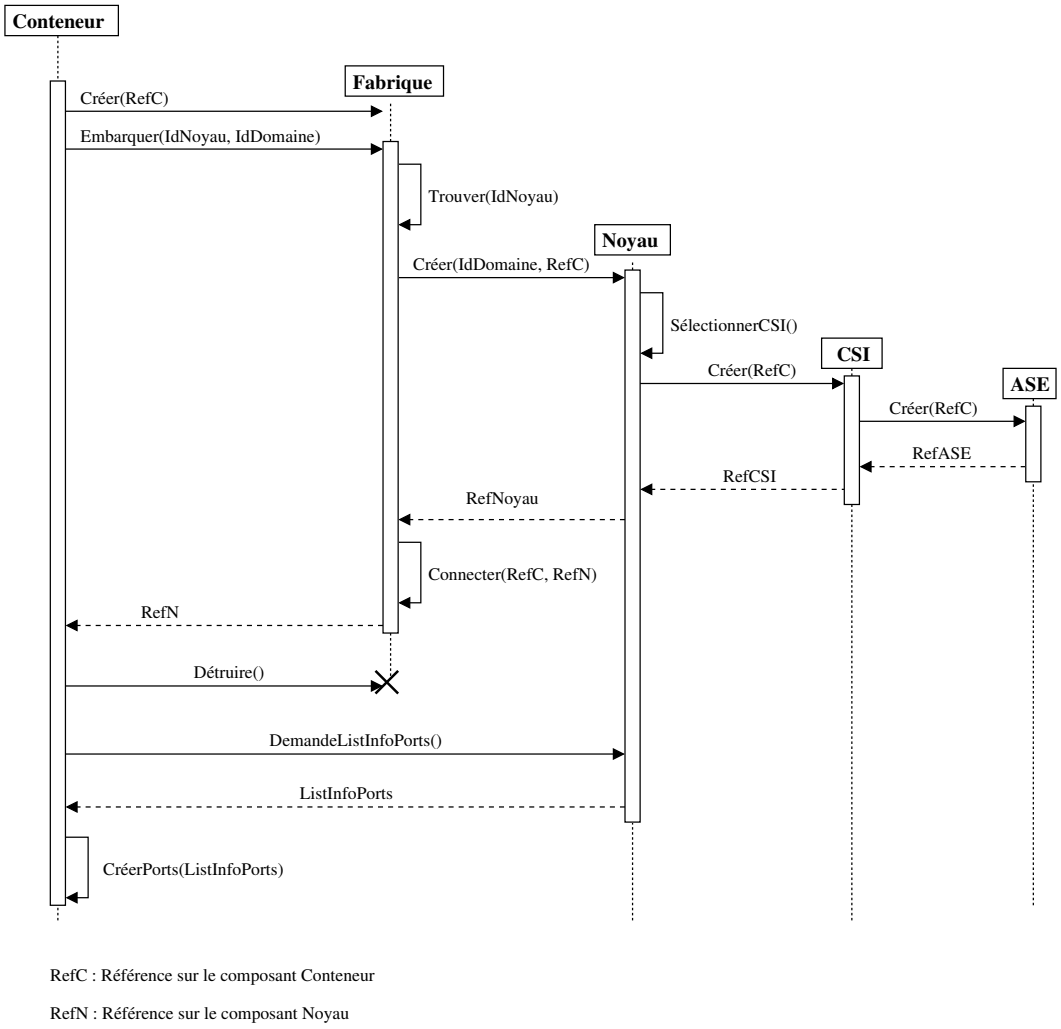


FIG. 5.4: Diagramme de séquence : Phase d'assemblage du composant domaine-spécifique flexible.

Le rôle du composant Conteneur Le composant Conteneur est le composant responsable sur le déclenchement de la phase d'assemblage. Donc, une fois que l'identité du composant Noyau, IdNoyau, soit reçue, le composant Conteneur passe sa référence, RefC, au composant Fabrique, et cela, après l'avoir créé. Ensuite, il appelle la méthode d'embarquement du composant Fabrique, en lui passant l'identité [2] du composant Noyau, IdNoyau, et celle du domaine, IdDomaine, pour recevoir par la suite une référence, RefN, sur le composant Noyau. A ce niveau, la mission du composant Fabrique est accomplie et provoquera la destruction de ce dernier par le composant Conteneur. La référence RefN permet au composant Conteneur de demander au composant Noyau les informations nécessaires sur les ports de communication à créer et de l'exécuter.

Le rôle du composant Fabrique Une fois que sa méthode d'embarquement soit invoquée par le composant Conteneur, le rôle du composant Fabrique est d'abord de trouver le composant Noyau, en utilisant son identité, IdNoyau, afin de l'instancier et de lui passer par la suite l'identité du domaine, IdDomaine, et la référence sur le composant Conteneur, RefC. Et comme tâche finale, il établit la connexion entre les deux composants, Conteneur et Noyau, et cela, à travers leurs références, RefC et RefNoyau. Le composant Fabrique trouve le composant Noyau suivant sa politique qui est identique à celle utilisée par le composant Noyau, politique de sélection. La concrétisation de ces deux

[2]Nous avons bien dit *identité* et non pas *référence*. Cela signifie que le composant identité n'est pas encore connu et avant de le référencer il faut d'abord le trouver et puis l'instancier.

politiques dépend des caractéristiques du langage de programmation et les approches utilisées lors de leur implémentation.

Le rôle du composant Noyau Une fois instancié par le composant Fabrique, le composant Noyau est le seul responsable sur la sélection du composant CSI convenant à sa sémantique et à celle du domaine dans lequel il va s'exécuter. Pour cela, le composant Noyau s'appuie sur sa politique de sélection, en utilisant l'identité du domaine, IdDomaine, pour trouver le bon composant CSI. Finalement, le composant Noyau doit retourner sa référence, RefN, au composant Fabrique, et cela, après avoir instancié et référencié le composant CSI.

Le rôle du composant CSI Comme, d'une part, le composant CSI est spécifique à une seule sémantique à conserver et un seul domaine, et d'autre part, le composant ASE est spécifique qu'à un seul domaine et est unique, le rôle du composant CSI est limité juste à l'instanciation du composant ASE pour le référencier.

Le rôle du composant ASE De même, comme le composant Conteneur est spécifique à un seul domaine, le composant ASE connaît donc d'avance le composant Conteneur et son rôle est limité juste au référencement du composant Conteneur.

Tâche d'assemblage partagée Comme chaque composant a des relations de dépendances avec d'autres composants, notamment pour ceux ayant des liens dynamiques, nous avons choisi d'affecter à chaque composant la responsabilité de trouver, cas d'un lien dynamique, le composant à référencer. Cela donne l'avantage d'ajouter de nouveaux composants ayant de nouvelle politique de sélection sans se préoccuper ni de comment retrouver les composants à référencer ni de comment les liens seront établis. En effet, cet avantage écarte toute sorte de modification des autres composants, ce qui permet donc d'avoir une bonne réutilisabilité et évolutivité des composants. Par exemple, le composant Noyau, avec sa politique de sélection, est le seul qui sait quel bon composant CSI qui lui faut et comment le trouver. Cependant, il était possible de dédier à un seul composant (par exemple le composant Fabrique) la totalité de la tâche d'assemblage du composant domaine-spécifique flexible. Malheureusement, cette solution est très lourde et oblige au composant d'assemblage de manipuler toutes les références des composants ainsi que le schéma de leur interconnexion (un tel composant doit référencer un tel composant). En effet, le but est de rendre chacun des composants responsable sur l'instanciation et/ou le référencement des composants ayant des liens avec lui.

Le partage de la tâche d'assemblage entre l'ensemble de composants, constituant le composant domaine-spécifique flexible, donne les avantages suivants :

- Bonne réutilisabilité ;
- Bonne évolutivité ;
- Simplicité ;
- Bonne modularité.

5.1.4 Composant domaine-polymorphe

Le modèle de composant domaine-spécifique flexible nous a permis d'avoir l'avantage de réutiliser ses composants d'une manière dynamique. Pour cela, il suffit juste de fournir au composant Conteneur l'identité du composant Noyau à exécuter sans se préoccuper de comment le reste de l'ensemble de composants sera trouvé et assemblé. Cependant, cette réutilisabilité dynamique n'est complètement possible que dans un même domaine, c'est-à-dire, pour faire exécuter le composant Noyau dans différents domaines, nous nous sommes obligés de fournir son identité à tous les composants Conteneurs car le composant domaine-spécifique n'est réutilisable que dans son domaine.

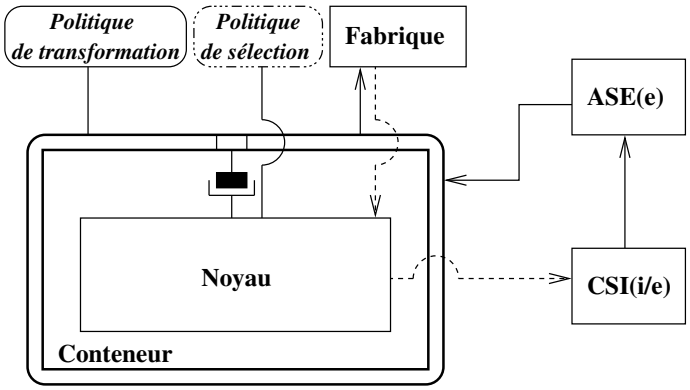
A ce niveau, nous sommes arrivés à un polymorphisme intra domaine et la dernière étape de notre approche va montrer comment nous nous sommes passés du modèle de composant domaine-spécifique flexible au modèle de composant domaine-polymorphe.

Architecture

En observant l'ensemble de composants domaine-spécifiques flexibles, dont chacun d'eux est unique et spécifique à un seule domaine, nous remarquons qu'ils ont qu'un seul composant en commun [3], c'est le composant Noyau. Ce dernier est réutilisable dans tous les domaines et a la capacité de retrouver le composant CSI dédié à sa sémantique et à celle du domaine dans lequel il sera exécuté.

Pour être réutilisable dans différents domaines, le composant domaine-spécifique flexible doit changer sa structure suivant les exigences des domaines. Et comme, les composants CSI et ASE ne font pas parti de la structure exigée par le domaine hôte mais plutôt font parti du comportement (manipulation, configuration), le seul composant responsable sur la structure du composant domaine-spécifique flexible répondant aux exigences du domaine est bien donc le composant Conteneur. En effet, la non réutilisabilité du composant domaine-spécifique flexible dans d'autres domaines est causée par le composant Conteneur.

Donc, pour qu'un composant domaine-spécifique flexible puisse être réutilisable dans différents domaines, il lui suffit juste de pouvoir, une fois qu'il a identifié le domaine, remplacer le composant Conteneur initial par celui convenant au domaine identifié. Et comme le composant Conteneur est le point de lancement du composant domaine-spécifique flexible, son remplacement, à notre connaissance, tout en respectant les objectifs cités dans la section 5.1.1, est impossible par contre sa transformation est possible. Pour cela, nous proposons d'ajouter une *politique de transformation* au composant Conteneur, ce qui nous permet d'obtenir une nouvelle architecture, qui est l'architecture du composant domaine-polymorphe, voir figure 5.5.



ASE(e): ASE spécifique à la sémantique externe e.
CSI(i/e): CSI spécifique à la sémantique interne i et à la sémantique externe e.

FIG. 5.5: Architecture du composant domaine-polymorphe.

La politique de transformation permet donc au composant domaine-polymorphe :

- L'identification du domaine hôte : Cette fonctionnalité passe par l'interrogation du domaine hôte afin de connaître sa sémantique ;
- La transformation de la structure du composant Conteneur : A partir d'une structure initiale, cette fonctionnalité permet de changer la structure initiale du composant Conteneur en celle répondant aux exigences du domaine identifié. Cependant, nous signalons que *l'implémentation de cette politique est fortement dépendante des techniques, citées dans le chapitre 4, permettant*

[3] Dans cette comparaison, nous n'avons pas pris en compte le composant Fabrique.

d'avoir du polymorphisme, ainsi que les structures des composants exigées par la plateforme de modélisation et de conception dans laquelle nous voulions intégrer le composant domaine-polymorphe.

Assemblage du composant domaine-polymorphe

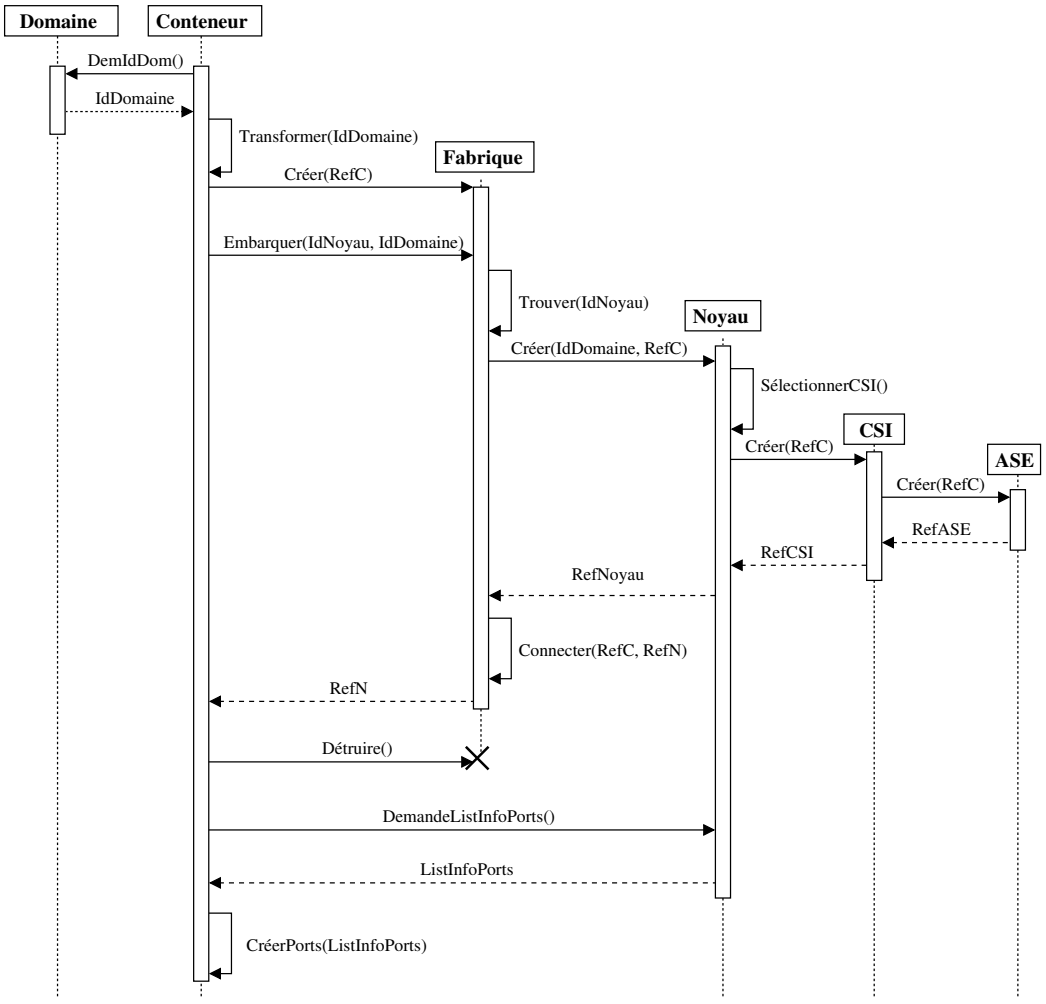


FIG. 5.6: Diagramme UML de séquence montrant la phase d'assemblage du composant domaine-polymorphe.

La phase d'assemblage du composant domaine-polymorphe, montrée sur la figure 5.6, se diffère par rapport à celle du composant domaine-spécifique flexible juste au niveau du composant Conteneur. En effet, en utilisant sa politique de transformation, le composant Conteneur doit d'abord interroger le domaine dans lequel il se trouve en lui demandant son identité pour modifier par la suite sa structure en celle exigée par le domaine identifié. Une fois que la transformation soit faite, le reste de la phase d'assemblage est complètement identique à celle du composant domaine-spécifique flexible. Donc, la politique de transformation permet de modifier le composant domaine-polymorphe en un composant domaine-spécifique flexible.

5.2 Opérations du composant domaine-polymorphe

L'exécution de la phase d'assemblage du composant domaine-polymorphe et celle du composant domaine-spécifique flexible leur permet d'avoir la même forme finale et complète, montrée par la figure 5.7. Cette forme est spécifique au domaine dans lequel le composant domaine-polymorphe est utilisé et à la sémantique interne. En effet, après la phase d'assemblage, le composant domaine-polymorphe, noté CDP, s'exécutant dans un domaine D_i et ayant comme sémantique interne la sémantique S_j, comporte l'ensemble de composants suivant :

$$CDP.Composant = \{Conteneur_i, Noyau_j, ASE_i, CSI_{ij}\}$$

Avec le composant $Conteneur_i$ a une structure (classe + ports de communication) spécifique au domaine D_i, le composant $Noyau_j$ spécifié suivant la sémantique S_j, ASE_i correspond au composant ASE assurant le bon comportement exigé par le domaine D_i et CSI_{ij} correspond au composant CSI permettant au composant $Noyau_j$ de s'exécuter suivant la sémantique S_j tout en respectant la sémantique du domaine D_i.

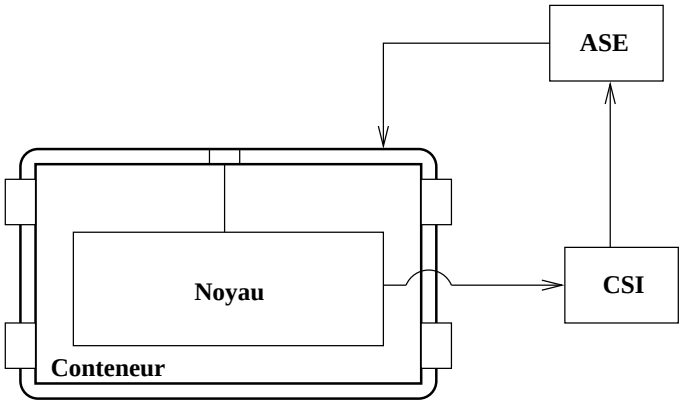


FIG. 5.7: Composant domaine-polymorphe après la phase d'assemblage.

A l'exécution, le composant domaine-polymorphe dispose d'un ensemble d'opérations de flot de données et de flot de contrôle (voir la figure 5.8, noté $CDP.Oper$, qui est l'union des opérations de ses différents composants :

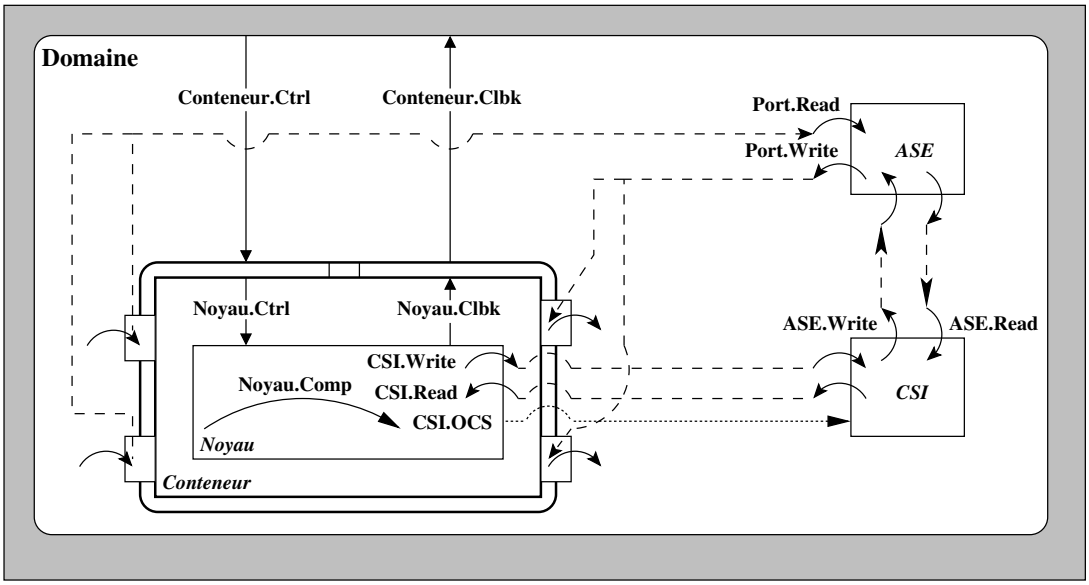
$$CDP.Oper = Conteneur.Oper \bigcup Noyau.Oper \bigcup ASE.Oper \bigcup CSI.Oper$$

Opérations de flot de données

Opérations de flot de données du composant ASE Les opérations de flot de données exécutées par le composant ASE ne sont que l'ensemble d'opérations de communication, notées $ASE.Comm$. Conformément aux primitives abstraites présentées dans la sous sction 2.6.5, ce dernier est composé de deux sous ensembles :

- le sous ensemble $Port.Read$ d'opérations de lecture comportant les deux opérations $existData$ et $read$;
- le sous ensemble $Port.Write$ d'opérations d'écriture comportant les deux opérations $isFull$ et $write$.

Les comportements exacts des opérations $Port.Read$ et $Port.Write$ sont déterminés par le domaine.



OCS : Opérations de Conservation de la Sémantique.

FIG. 5.8: Opérations du composant domaine-polymorphe.

Opérations de flot de données du composant CSI L'ensemble d'opérations de flot de données exécuté par le composant CSI sont des opérations de communication, noté $CSI.Comm$, composé de deux sous-ensembles :

- le sous-ensemble $ASE.Read$ des opérations de lecture qui permettent la récupération des données lues par le composant ASE. Ce sous-ensemble comporte les trois opérations : $hasData$, $getData$ et $transformerDataIn$. Cette dernière permet la transformation des données lues en données ayant un format respectant la sémantique interne (celle du composant Noyau) ;
- le sous ensemble $ASE.Write$ d'opérations d'écriture permettant d'envoyer les données venant du composant Noyau au composant ASE. Cet ensemble, lui aussi, comporte trois opérations : $hasPlace$, $sendData$ et $transformerDataOut$. Cette dernière permet la transformation des données provenant du composant Noyau en données ayant un format respectant la sémantique externe (celle du domaine).

Les opérations de transformation $transformerDataIn$ et $transformerDataOut$ peuvent être effectuées par interprétation, par échantillonnage, etc. Elles font partie de la modélisation et la conception du système.

Opérations de flot de données du composant Noyau Les opérations de flot de données exécutées par le composant Noyau sont :

- L'ensemble $Noyau.Comp$ d'opérations de traitement permettant au composant Noyau de calculer les nouvelles données (son nouvel état) et les sorties à partir des anciennes données (ancien état) et des d'entrée. Cet ensemble comporte l'opération $computeBehaviorCore$;
- L'ensemble $Noyau.Sem$ d'opérations de conservation de la sémantique interne permet au composant Noyau de fonctionner correctement, tout en respectant la sémantique du domaine. Il comporte les opérations BoR (Begin of Reaction), EoR (End of Reaction) et d'éventuelles opérations noté $\Phi Noyau.Sem$. L'opération BoR (resp. EoR) est invoquée par le composant Noyau avant (resp. après) sa réaction afin de permettre au composant CSI d'effectuer d'éventuel traitement nécessaire aux deux sémantiques (interne et externe). Tandis que les opérations $\Phi Noyau.Sem$ varies en nombre comme en comportement d'un domaine à un autre et suivant la sémantique du composant Noyau. Par exemple, l'opération $Ready$ permet au composant

Noyau synchrone de s'informer auprès du composant CSI s'il est autorisé à réagir ou non à l'opération $trigger$;
- L'ensemble $Noyau.Comm$ d'opérations de communication qui comporte les deux sous ensembles :
  - Le sous ensemble $CSI.Read$ d'opérations permettant la récupération des données transformées venant de l'extérieur depuis le composant CSI. Ce sous-ensemble comporte les deux opérations, $hasInputData$ et $receiveData$;
  - Le sous ensemble $CSI.Write$ d'opérations permettant l'envoie des données produites par le composant Noyau au composant CSI. Il comporte deux opérations, $hasFreePlace$ et $emitData$.

Opérations du flot de contrôle

Le composant domaine-polymorphe est contrôlé par le domaine à travers un ensemble d'opérations, noté $Conteneur.Ctrl$. Conformément aux primitives abstraites données dans la sous section 2.6.5, cet ensemble comporte les opérations $initialization$, $preCondition$, $trigger$ ainsi que $postCondition$. Aussi, l'ensemble d'opération de rappel (call-back), noté $Conteneur.Clbk$, utilisé par le composant domaine-polymorphe pour solliciter les services du domaine sont les opérations $finish$ et $\Phi Conteneur.Clbk$. L'opération $finish$ permet au composant domaine-polymorphe de signaler au domaine la fin de ses activités tandis que les opérations $\Phi Conteneur.Clbk$ lui permettent de solliciter d'autres services auprès du domaine. Par exemple, l'opération $reTrigger$, permettant la demande d'un relancement du composant domaine-polymorphe, fait partie de l'ensemble $\Phi Conteneur.Clbk$.

Le composant Conteneur utilise l'ensemble $Noyau.Ctrl$ d'opérations de contrôle pour contrôler l'exécution du composant Noyau. Cet ensemble comporte les opérations $initCore$, $preReaction$, $reaction$ et $postReaction$. Cependant, le composant Noyau utilise un ensemble $Noyau.Clbk$ d'opérations pour solliciter le composant Conteneur afin de lui signaler la fin de ses activités et de solliciter d'éventuels services. L'ensemble $Noyau.Clbk$ comporte donc l'opération $finishReaction$ et d'éventuelles opérations $\Phi Noyau.Clbk$.

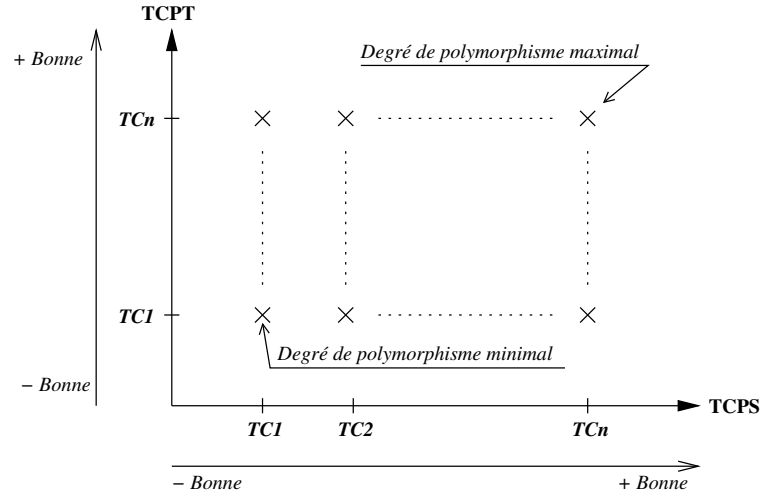
Nous résumons donc l'ensemble d'opérations d'exécution du composant domaine-polymorphe dans le tableau 5.1.

Opérations de flot de données			Opérations de flot de contrôle
Op. OCS	Op. de calcul	Op. de comminication	
		existData	
		read	initialization
		isFull	preCondition
		write	trigger
		hasData	postCondition
BoR		getData	finish
EoR	computeBehaviorCore	transformerDataIn	ΦConteneur.Clbk
$\Phi Noyau.Sem$		hasPlace	initCore
		transformerDataOut	preReaction
		sendData	reaction
		hasInputData	postReaction
		receiveData	finishReaction
		hasFreePlace	ΦNoyau.Clbk
		emitData	

TAB. 5.1: Opérations de flot de données et de flot de contrôle du CDP

5.3 Différents degrés du polymorphisme

Dans la sous section 5.1.1 (consacrée aux objectifs et contraintes), nous avons bien dit que le composant domaine-polymorphe doit être programmable en n'importe quel langage orienté objet, au pire des cas, en utilisant que les concepts de base de l'approche objet. En effet, les éléments de l'architecture du composant domaine-polymorphe où leur concrétisation influe sur le degré du polymorphisme ne sont que la politique de transformation et celle de sélection.



TCPS : Techniques de Concrétisation de la Politique de Sélection.
TCn : Techniques de Concrétisation numéro n.
TCPT : Techniques de Concrétisation de la Politique de Transformation.

FIG. 5.9: Degré du polymorphisme lié à la concrétisation de la politique de transformation et celle de sélection.

Comme le montre la figure 5.9, le degré du polymorphisme du composant domaine-polymorphe dépend donc des techniques utilisées pour la concrétisation de la politique de sélection et celle de transformation. Autant que la concrétisation des deux politiques est de plus en plus bonne autant que le degré du polymorphisme sera de plus en plus élevé. En effet, nous disons qu'une concrétisation est plus bonne qu'une autre si elle n'implique pas un nombre d'intervention externe, au niveau du composant domaine-polymorphe, plus grand que celui impliqué par l'autre concrétisation. L'intervention externe, qui est assurée par un élément externe (par exemple le programmeur, un autre programme, etc.), s'explique par d'apporter des modifications au niveau du composant domaine-polymorphe afin de prendre en compte des nouvelles situations (cas d'ajout de nouveau de domaines, etc.), et cela, dans le but d'assurer le bon fonctionnement du composant domaine-polymorphe. Par exemple, avec la politique de sélection, l'ajout d'un nouveau domaine impliquera l'ajout de nouveaux composants CSIs spécifiques à lui et aux sémantiques à conserver. Ainsi, la question qui se pose est « dans le cas d'utilisation du composant domaine-polymorphe dans ce nouveau domaine, est ce que la politique de sélection sera capable de trouver le nouveau composant CSI, qui ne lui est pas connu auparavant, ou non ? ». Donc, si elle sera capable de le trouver, nous disons que sa concrétisation est plus bonne, sinon nous serons obligés d'intervenir au niveau de son code pour que le composant Noyau puisse trouver le nouvel composant CSI, et dans ce cas, nous disions que la concrétisation est moins bonne.

5.3.1 Degré lié à la concrétisation de la politique de transformation

Nous avons dit que le rôle de la politique de transformation est de permettre au composant Conteneur de se changer pour avoir la structure qui correspond à celle exigée par le domaine dans lequel il est utilisé. Concrètement, cela se traduit par le pouvoir du composant Conteneur de modifier sa structure initiale, qui revient donc au modification de sa classe (son type).

Changement de type du composant Conteneur

Le problème posé par la concrétisation de la politique de transformation est au niveau du changement de la classe du composant Conteneur que nous expliquerons ici. Nous savons bien qu'une classe se diffère par rapport à une autre classe au niveau de sa signature et de son contenu. La signature d'une classe peut introduire les classes mères et/ou les interfaces quelle implémente. Le contenu d'une classe est représenté par l'ensemble de ses attributs et de ses méthodes. Donc pour changer une classe, c'est-à-dire changer de type, cela nécessite la modification de sa signature et/ou son contenu, et pour avoir le bon degré de polymorphisme, ces modifications doivent être faites par la classe elle-même.

Le changement de la signature d'une classe par soit même, en utilisant un langage orienté objet ayant les types statiques [4] est impossible. De même, l'utilisation des langages orientés objet ayant les types dynamiques [5] ne fait que repousser le problème à l'exécution, c'est-à-dire, qu'une fois la classe est instanciée, elle ne peut pas changer sa signature.

Par contre, le changement du contenu d'une classe est permis avec certains langages. Par exemple avec le langage Python [198], il est possible d'ajouter des attributs pendant l'exécution à un objet, la modification du comportement des méthodes [6]. Aussi, avec certains langages réflexifs, il est possible d'utiliser des méta-objets, qui forment un niveau méta, dont chacun peut être un objet de contrôle ou un objet de réification d'une entité du niveau de base (Composant Conteneur), et cela dans le but de dérouter tous les appels, destinés au composant Conteneur, vers le niveau méta, ce qui permet de choisir le méta-objet correspondant au domaine, ainsi d'effectuer le bon traitement.

5.3.2 Degré lié à la concrétisation de la politique de sélection

Contrairement à la concrétisation de la politique de transformation, celle de la politique de sélection pose moins de problèmes et, avec certains langages et sans aucune difficultés, elle peut atteindre son degré maximal qui se traduit par la capacité de prendre en considération les évolutions dans le future (cas d'ajout de nouveaux domaines).

Il existe plusieurs façons de concrétisation de la politique de sélection dont chacune permet d'avoir une concrétisation de moins en moins ou de plus en plus bonne. A titre illustratif, nous proposons ici deux façons de concrétisation de la politique de sélection. La première façon se base sur l'utilisation des design patterns [197] et permet d'avoir la moins bonne concrétisation. La deuxième se base sur l'utilisation de chargeur de classes et permet d'avoir la meilleure concrétisation.

Solution basée sur l'utilisation des design patterns : La moins bonne

Cette solution représente la solution la moins bonne et elle est conseillée pour les langages de programmation ne permettant que les concepts de base de l'approche objet (abstraction, héritage, etc.). Dans ce cas, nous conseillons d'utiliser les design patterns afin de bénéficier de leurs avantages.

En effet, nous avons choisis d'utiliser le pattern Factory, dont le principe est de masquer la véritable nature des objets, qui se base sur sa classe d'usinage, nommée CSIFactory, pour sélectionner et instancier le bon composant CSI afin de le retourner au composant Noyau. Voici les étapes de la concrétisation de la politique de sélection, en utilisant ce pattern :

[4] Un type statique est un type et un seul, qui ne variera pas au cours de l'exécution. Ce type est connu lors de la compilation, il s'agit de celui joignant le nom de la variable là où elle est déclarée.

[5] Une variable de type dynamique est une variable dont le type n'est connu qu'à l'exécution, exactement au moment où elle prendra une valeur (dépend du type de sa valeur). Variable dynamique dite aussi variable sans type.

[6] Le comportement d'une méthode est décrit par sa signature et son corps.

1. D'abord, nous définissons une classe abstraite, qui désigne un objet retourné par une méthode, nommée getCSI, de la classe CSIFactory. Cette classe, nommée CSIInterface, n'est que l'interface implémentée par les composants CSIs ;
2. Ensuite, nous spécifions dans la méthode getCSI de la classe CSIFactory comment le composant CSI correspondant à la sémantique interne et celle du domaine sera sélectionné. Pour une telle sélection, nous avons utilisé les instructions conditionnelles basées sur les noms des domaines.

Le listing 5.2 montre le code Java de cette solution. Cependant, pour cette concrétisation, nous avons pris :

– n domaines dont le nom de chacun est la concaténation de la lettre D avec son indice ;
– n composants CSI, qui sont nécessaires à l'utilisation d'un composant Noyau, qui obéit à la sémantique S, dans les n domaines. Le nom de chaque composant CSI est la concaténation de la chaîne de caractères CSI avec le nom du domaine correspond.

Listing 5.2: Concrétisation de la politique de sélection basée sur l'utilisation du pattern Factory

```
/* Interface implementee par l'ensemble de composants CSIs */
public interface CSIInterface {
  // Operations de communication
  //...

  // Operations OCS definies par les composants CSIs
  //...
}


/* Core : composant Noyau */
public class Core {
  /** Constructeur */
  public Core(String IdDomain, ...) {
    /* Instancier la classe CSIFactory et recuperer
       une reference sur le bon composant CSI
    */
    _csiFactory = new CSIFactory();
    _csiInterface = _csiFactory.getCSI(IdDomain);
    //...
  }
  //...

  // Variables
  private CSIFactory _csiFactory;
  private CSIInterface _csiInterface;
}


/* CSIFactory retournant des objets de types CSIs */
public class CSIFactory {
  /* Retourne l'instance du composant CSI selectionne */
  public CSIInterface getCSI(String IdDomain) {

    /* Cas du domaine D1 */
    if(IdDomain.equals("D1")
      return new CSID1(...);

    /* Cas du domaine D2 */
    elseif(IdDomain.equals("D2")

    //...

    /* Cas du domaine Dn */
```

```
    else(IdDomain.equals("Dn")
    return new CSIDn(...);
  }
}
```

Solution basée sur l'utilisation d'un chargeur de classes : La meilleure

Cette solution est la meilleure façon de concrétisation de la politique de sélection. Elle nécessite que le langage de programmation utilisé doit vérifier les deux conditions suivantes :

1. Il doit permettre la concrétisation des classes sous la forme d'objets manipulables par le programme (pour notre cas, manipulables par le composant Noyau) (introspection voir le paragraphe 4.4.4 du chapitre 4). La classe *Class* en Java est un bon exemple ;
2. Il doit fournir un chargeur de classes permet de charger les classes dont on aura besoin. Par exemple, le langage Java fournit la classe *ClassLoader* comme un chargeur de classes.

Les étapes de concrétisation de la politique de sélection, en utilisant un chargeur de classes, sont donc :

1. Comme les noms des sémantiques (sémantiques des domaines et celles à conserver) sont uniques, nous proposons de nommer chaque composant CSI, qui est spécifique à une sémantique d'un domaine et une sémantique interne, par un nom résultant de la concaténation du nom du domaine et celui de la sémantique interne ;
2. Comme le composant Noyau peut connaître le nom du domaine, dans lequel le composant domaine-polymorphe est utilisé, et le nom de sa sémantique, l'identification du nom du composant CSI, qui lui (Noyau) correspond, ne lui (Noyau) posera donc aucun problème, il suffit de concaténer les deux noms, le nom du domaine et celui de sa sémantique ;
3. Finalement et après avoir chargé le bon composant CSI, le composant Noyau doit l'instancier, en utilisant une méthode fournie par la méta-classe (pour Java, il s'agit de la méthode *newInstance(...)* de la classe *Class*).

A titre illustratif, le listing 5.3 donne le code de concrétisation de la politique de sélection en utilisant un chargeur de classe. Cependant, nous prenons les données suivantes :

- Le nom de l'interface implémentée par les composants CSIs est toujours CSIInterface ;
- Peut n'importe le nombre de domaines dont le nom de chacun d'eux est celui de sa sémantique ;
- Le nom de la sémantique à conserver est Sem.

Listing 5.3: Concrétisation de la politique de sélection basée sur l'utilisation d'un chargeur de classes

```
/** Core : composant Noyau */
public class Core {
  /** Constructeur */
  public Core(String IdDomain, ...) {

    //Charger et instancier le bon composant CIS
    try{
      //Instancier le chargeur de classes
      PolymorphClassLoader cl = new PolymorphClassLoader();
      // Charger le composant CSI
      Class cc = cl.loadClass(IdDomain.concat("Sem"));
      // Instancier le composant CSI
      _csiInterface = (_csiInterface)(cc.newInstance());
    }catch(ClassNotFoundException e)
      { System.err.println("Error:" + e);}
    catch(InstantiationException e)
```

```
        { System.err.println("Error:" + e);}
      catch(IllegalAccessException e)
        { System.err.println("Error:" + e);}
      //...

    }

    //...
    // Variables
    private CSIInterface _csiInterface;
}
```

5.3.3 Synthèse

Les solutions pour la concrétisation des deux politiques que nous venions de proposer ont bien montré que, concrètement, le degré du polymorphisme dépend fortement des caractéristiques du langage et des approches utilisés (comme nous l'avons détaillé au dans la section Techniques pour le polymorphisme du chapitre 4. Par exemple, pour la concrétisation de la politique de transformation, nous remarquons qu'elle a une forte influence sur le degré du polymorphisme. Cette influence est due au problème de modification de la classe par soit même, notamment la modification de sa signature. En effet, nous pouvons dire :

1. Dans le cas d'utilisation d'un langage réflexif ayant les types dynamiques et que les domaines ne fassent pas la vérification des types après l'instanciation du composant conteneur, c'est-à-dire, que le respect des exigences des domaines ne dépend que du contenu (attributs et méthodes) de la classe représentant le composant Conteneur et peut n'importe sa signature, la concrétisation de la politique de transformation, dans ce cas, est plus bonne, mais nous signalons que cette solution présente pas mal d'inconvénients :
   - Programmation (dynamique) non souple et demande d'être très attentionné ;
   - A cause des types dynamiques, il peut y avoir, pendant l'exécution, des bugs difficile à détecter.
   - Exécution plus lente.

   Ces inconvénients deviennent de plus en plus durs à chaque fois qu'on cherhce à maximiser le degré du polymorphisme. Donc, c'est aux programmeurs d'assurer le compromis entre « De plus en plus le degré du polymorphisme est maximal, et, de plus en plus les inconvénients sont plus durs » ;
2. Dans le cas d'utilisation d'un langage ayant les types statiques, la transformation du composant Conteneur est impossible. Donc la concrétisation de la politique de transformation est moins bonne et nécessite une intervention au niveau composant domaine-polymorphe pour changer le composant Conteneur par celui qui convient au domaine utilisé. En effet, dans ce cas, le composant domaine-polymorphe n'est que l'implémentation de l'architecture du composant domaine-spécifique flexible.

Par contre, la concrétisation de la politique de sélection, dans le pire des cas, n'influe que légèrement le degré du polymorphisme :

1. Avec la première solution (la moins bonne), une intervention n'est nécessaire que dans le cas d'ajout de nouveau domaine, mais juste au niveau du pattern Factory et non pas au niveau du composant domaine-polymorphe ;
2. Avec la deuxième solution (la meilleure), aucune intervention n'est nécessaire et l'évolution du nombre de domaines est prise en considération.

5.4 Evaluations

Dans cette section, nous donnons des évaluations d'utilisation des trois composants : composant domaine-spécifique (noté CDS), composant domaine-spécifique flexible (noté CDSF) et composant domaine-polymorphe (noté CDP) dans la modélisation et la conception des systèmes embarqués hétérogènes, et cela, afin de montrer les avantages d'utilisation de nos composants, CDSF et CDP, par rapport à l'utilisation des composants CDS.

Pour cela, nous avons pris un système *Syst* composé de n entités et complètement hétérogène, c'est-à-dire que chacune de ses entités obéit à un modèle de calcul différent des autres, et de le concevoir dans d domaines, en utilisant les CDS, les CDSF et puis les CDP.

Ensuite, les évaluations d'utilisation des trois composants sont faites par rapport aux cas possibles suivants : Par rapport à la représentation du système *Syst* dans les d domaines, par rapport aux modifications du système *Syst* et par rapport à l'évolution du nombre de domaines. Nous signalons que les différentes évaluations sont faites au niveau conceptuel (abstrait) et non pas au niveau implémentation, c'est-à-dire qu'aucune hypothèse n'est faite sur l'implémentation.

5.4.1 Par rapport à la représentation des systèmes

Cas d'utilisation du composant CDS

L'utilisation des composants CDS pour la représentation du système *Syst* dans d domaines revient à représenter chacune de ses n entités par un composant CDS, et cela, dans chacun des d domaines. En effet, le nombre de composants CDS représentants le sytème *Syst* est proportionnel au nombre de domaines, c'est à dire en variant $\triangle d$ de domaines implique un variant $\triangle n$ de nombre de composants CDS. Nous résumons cela comme suit :

$\forall n \in N^+$, nombre d'entités du système *Syst*, et $\forall d \in N^+$, nombre de domaines, la représentation du système *Syst* dans les d domaines nécessite l'utilisation de $n * d$ composants CDS.

Cas d'utilisation du composant CDSF ou le composant CDP

L'utilisation de l'un de nos deux composants, CDSF ou CDP, pour représenter le système *Syst* dans les d domaines revient à représenter chacune de ses entités par un composant Noyau. Donc, quelque soit le nombre de domaines, dans lesquels nous voulons représenter le système *Syst*, nous utilisons que n composants Noyau, mais nous supposons que les composants suivants existent déjà :

- d composants ASE et $d * n$ composants CSI ;
- d composants Conteneurs pour les d domaines, dans le cas d'utilisation du composant CDSF ;
- Un composant Conteneur avec sa politique de transformation pour les d domaines, dans le cas d'utilisation du composant CDP.

Donc, nous résumons cela comme suit :

$\forall n \in N^+$, il faut n composants Noyau pour représenter le système *Syst*, et cela $\forall d \in N^+$.

5.4.2 Par rapport à la modification des systèmes déjà représentés

Après avoir représenté le système *Syst* dans les d domaines, des modifications peuvent avoir lieu afin d'apporter des améliorations au niveau du fonctionnement du système *Syst*. En effet, cela revient à modifier les n entités qui le composent, ce qui implique la modification des composants qui le représentent. Pour cela, nous montrons dans cette sous section les conséquences de modification de n' ($1 \leq n' \leq n$) entités du système *Syst* sur le nombre de composants (représentant le système *Syst*) à modifier.

Cas d'utilisation des composants CDS

Dans tel cas, le nombre de composants CDS à modifier est proportionnel au nombre de domaines. Nous résumons cela comme suit :

$\forall n' \in N^+$, nombre d'entités modifiées, et $\forall d \in N^+$, nombre de domaines, le nombre de composants CDS à modifier est toujours égal à $n' * d$.

Cas d'utilisation du composant CDSF ou le composant CDP

Dans ce cas, la modification de n' entités du système *Syst* a comme conséquence la modification des n' composants Noyau qui les représentent, et cela, quelque soit le nombre de domaines utilisés. Nous résumons cela comme suit :

$\forall n' \in N^+$ et $\forall d \in N^+$, le nombre de composants Noyaux à modifier est toujours égal à n'.

5.4.3 Par rapport à l'évolution du nombre de domaines

Maintenant, supposons qu'après avoir représenté le système *Syst* dans les d domaines, un nombre d' de domaines vient d'être ajouté et nous voulons bien sûr que le système *Syst*, déjà conçu, soit représenté dans les d' nouveaux domaines. Pour cela, nous montrons ici, pour chacun des composants utilisé, les conséquences d'évolution du nombre de domaines sur le nombre de composants à utiliser.

Cas d'utilisation des composants CDS

Dans le cas où le système *Syst* est représenté par les composants CDS, la seule solution consiste à représenter de nouveau chacune des n entités du système *Syst* par un composant CDS, et cela, dans chacun des d' domaines. Le nombre de composants CDS pour représenter le système *Syst* est proportionnel au nombre de nouveaux domaines ajoutés. Nous résumons cela comme suit :

$\forall n \in N^+$, nombre d'entités du *Syst* et $\forall d' \in N^+$, nombre de nouveaux domaines ajoutés, il faut ajouter $n * d'$ composants CDS pour représenter le *Syst* dans les d' nouveaux domaines.

Cas d'utilisation du composant CDSF

Puisque les entités du système *Syst* sont représentées par des composants Noyau, qui sont réutilisables dans tous les domaines, il suffit juste d'ajouter pour chacun des d' nouveaux domaines les composants suivants : un composant Conteneur, un composant ASE, et n composants CSIs (car un composant CSI est spécifique à une sémantique interne et une sémantique d'un domaine). En effet, l'ajout de d' nouveaux domaines impliquera l'ajout de d' composnts Conteneur, d' composants ASE et $n * d'$ composants CSI.

Cas d'utilisation du composant CDP

De même, dans le cas d'utilisation du composant CDP, l'ajout de d' nouveaux domaines n'impliquera que l'ajout de d' composants ASE et $n * d'$ composants CSI. Par contre, pour que le composant Conteneur soit réutilisable dans ces nouveaux domaines, il faudra mettre à jour sa politique de transformation.

5.4.4 Synthèse

Nous avons résumé les évaluations (présentées précédemment) d'utilisation des composant CDS, CDSF et CDP dans le tableau 5.2. Ainsi, nous remarquons que :

- L'utilisation des composants CDS nécessite toujours un nombre important de ces derniers, qui est toujours proportionnel au nombre de domaines, et cela, quelque soit les cas suscités (représentation, modification, évolution du nombre de domaines) ;
- L'utilisation des composants CDSF et CDP nécessite le même nombre de composants Noyaux, ASE et CSI, par contre, pas le même nombre de composants Conteneurs. Dans le cas d'utilisation des composants CDFS, il faut créer pour chaque domaine son composant Conteneur, et cela, quelque soit la sémantique à conserver, tandis que dans le cas d'utilisation des composants CDP, il suffit de créer un seul composant Conteneur pour tous les domaines sauf qu'il faut mettre à jour sa politique de transformation à chaque évolution du nombre de domaines afin de les prendre en compte.

Ces évaluations nous permettant donc d'ajouter à la section 5.3.3 que le composant CDSF est très proche au CDP en nombre de composants utilisés. La différence est juste au niveau du nombre de composants Conteneur. En effet, les composants CDSF sont très utiles dans le cas où l'implémentation du composant CDP est compliquée ou voir impossible, notamment au niveau de la concrétisation de la politique de transformation. Dans un tel cas, nous conseillons d'utiliser le composant CDSF comme un composant domaine-polymorphe, en acceptant une légère perte de réutilisabilité mais garantissant une simple implémentation qui sera aussi aisée à maintenir par rapport à celle du CDP.

En général, dans le cas où nous voyons les composants Conteneur, ASE et CSI comme un environnement d'exécution des composants Noyaux, nous pouvons dire que nous avons assurer aux utilisateurs une réutilisabilité totale, et cela, quelque soit le type de composant utilisé, CDSF ou CDP.

	Composant CDS	Composant CDSF	Composant CDP
La représentation de n entités du *Syst* nécessite	$n * d$ composants CDS	n composants Noyaux, avec les composants Conteneurs, ASE et CSI existent déjà	n composants Noyaux, avec les composants Conteneurs, ASE et CSI existent déjà
La modification de n' entités du *Syst* nécessite la modification de	$n' * d$ composants CDS	n' composants Noyaux	n' composants Noyaux
L'ajout de d' domaines nécessite d'ajouter	$n * d'$ composants CDS	d' composants Conteneur, d' composants ASE et d' composants CSI	d' composants ASE et d' composants CSI avec la mise à jour de la politique de transformation

TAB. 5.2: Evaluations d'utilisation des composants CDS, CDSF et CDP dans la modélisation et la conception des systèmes.

5.5 Modèle de composant domaine-polymorphe amélioré

Le modèle de composant domaine-polymorphe que nous venions de proposer nous a permis d'aboutir à notre but principal : *La proposition d'un modèle de composant réutilisable et ayant la capacité de conserver une sémantique interne tout en respectant la sémantique du domaine hôte*. Cependant, ce modèle présente des limites et nécessite d'être amélioré.

Dans cette section, nous donnons d'abord les limites posées par la version actuelle du composant domaine-polymorphe, et ensuite, nous présentons la version améliorée de ce dernier.

5.5.1 Limites du modèle actuel de CDP

Les limites du modèle de CDP proposé viennent des composants CSI. Ces derniers offrent un environnement d'exécution au composant Noyau en fonction de la sémantique du domaine. En effet, tout composant CSI n'est spécifique qu'à une seule sémantique interne et une seule sémantique externe, ce qui implique que tout composant CSI est dépendant de deux sémantiques. Cette dépendance a les conséquences suivantes :

Nombre d'opérations OCS varié :

Les composants CSI nécessaires pour l'exécution d'un même composant Noyau dans différents domaines ne proposent pas le même nombre d'opérations OCS. Ce nombre dépend de la sémantique du domaine. En effet, les interfaces proposées par les composants CSI au composant Noyau ne sont pas les mêmes. Cela nous oblige, pour chaque sémantique interne, d'élaborer une interface commune à ses composants CSI, ce qui n'est pas une tâche facile.

Nombre de composants CSI explosif :

Dans le cas où nous supposons qu'une sémantique peut être utilisée en externe comme peut être utilisée en interne, le nombre de composants CSI nécessaires est le nombre de sémantiques à la puissance deux. Par exemple, si on a n sémantiques, et afin de permettre l'utilisation de ces sémantiques en interne comme en externe, c'est-à-dire que chacune de ces sémantique peut être utilisée par un composant Noyau comme peut être utilisée par un domaine, nous devons concevoir n^2 composants CSI.

De plus, dans le cas d'ajout d'une nouvelle sémantique, S, le nombre de composants CSI s'augmente de $(n+1)+n$, avec $n+1$ est le nombre de composants CSI pour utiliser la nouvelle sémantique en interne dans soi-même et dans les n sémantiques déjà existées, et n est le nombre de composants CSI pour utiliser les n sémantiques en interne dans la nouvelle sémantique.

En effet, la conception et la gestion d'un tel nombre de composants CSI deviennent de plus en plus lourdes.

Notre but dans cette section est donc de réduire le nombre de composants CSI dédiés à une sémantique S (utilisée en interne ou en externe) à un, c'est-à-dire que nous utilisons qu'un seul composant à la place des composants CSI dédiés à cette sémantique et d'aporter d'autres améliorations possibles au composant domaine-polymorphe actuel.

5.5.2 Architecture

Nous savons maintenant que, pour la même sémantique interne, un composant CSI ne peut être réutilisé dans d'autres domaines à part celui pour lequel a été conçu.

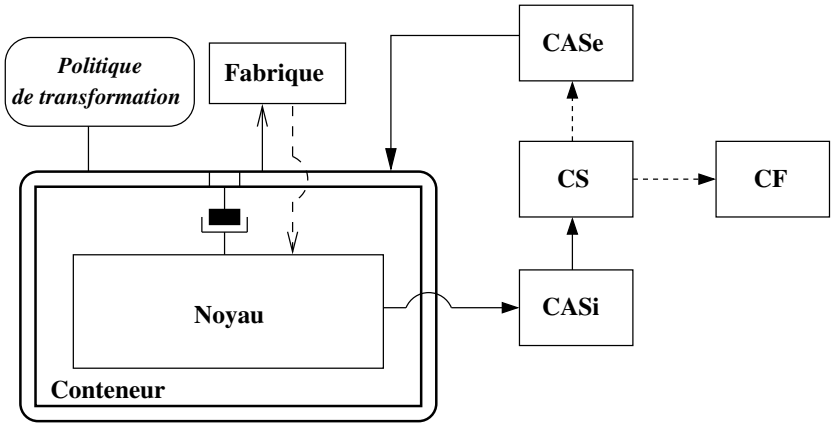
Au niveau abstrait, nous avons constaté que tous les composants CSI possèdent les mêmes préoccupations, qui sont hétérogènes (des préoccupations spécifiques à la sémantique interne et la sémantique externe) et entrelacées entre elles. Ces préoccupations sont :

1. La préoccupation de la sémantique interne : Elle regroupe toutes les opérations ou les tâches dédiées à la sémantique interne ;
2. La préoccupation de passage entre les deux sémantiques : Elle regroupe toutes les opérations ou les tâches concernant la transformation (par interprétation, par échantillonnage, etc.) des données venant/allant de l'extérieur. Cette préoccupation est vue comme la frontière entre les deux sémantiques ;
3. La préoccupation de la sémantique externe : Elle regroupe toutes les opérations ou tâches spécifiques à la sémantique externe. Elle veille sur le respect de la sémantique externe.

En effet, pour avoir une séparation nette entre la sémantique externe et la sémantique interne, nous avons, en appliquant de nouveau le principe de la séparation des préoccupations, transformé le composant CSI comme suit :

1. Nous avons extériorisé la préoccupation de passage entre les sémantiques et nous l'avons modélisée par un composant, appelé Composant Frontière (CF) ;
2. Nous avons maintenu la préoccupation de la sémantique interne tel quelle est. Par contre, puisque la sémantique interne peut être, elle-même, utilisée en externe, nous avons modifié la préoccupation de la sémantique externe (du composant CSI) en une préoccupation dédiée à la sémantique interne du composant CSI et lui avons ajouté le rôle de son (sémantique interne) composant ASE.

   En effet, le composant CSI n'est dédié (ou spécifique) qu'à une seule sémantique. Donc, le composant CSI peut être utilisé en interne, dans ce cas il aura le rôle d'un conservateur de la sémantique, comme peut être utilisé en externe, dans ce cas il aura le rôle d'un adaptateur de la sémantique, c'est pourquoi nous l'avons renommé composant Conservateur Adaptateur de la Sémantique (CAS). Dans le cas où le composant CAS est utilisé en interne, il sera noté CASi (CAS interne), et dans le cas où il est utilisé en externe, il sera noté CASe (CAS externe).



CASi : composant Conservateur Adaptateur de la Sémantique en interne.
CASe : composant Conservateur Adaptateur de la Sémantique en externe.
CS : composant Connecteur des Sémantiques.
CF : Composant Frontière.

FIG. 5.10: Architecture optimisée du composant domaine-polymorphe.

La figure 5.10 montre l'architecture améliorée du composant domaine-polymorphe obtenue à partir de l'architecture précédente, et cela, en remplaçant chaque ensemble de composants CSI, dédié à la même sémantique interne, par un composant CAS spécifique à cette dernière (appelé donc CASi) et chaque composant ASE par un composant CAS spécifique à la sémantique externe (appelé donc CASe). Aussi, nous avons introduit un nouveau composant, appelé CS (Connecteur des Sémantiques), permettant de connecter toute paire de composants, CASi et CASe et utilisant le composant CF comme paramètre afin d'expliciter le passage entre les sémantiques interne et externe.

Pour ce qui concerne les liens de connexion entre les composants de la nouvelle architecture, nous remarquons que le lien entre le composant Noyau et son composant CASi est statique, ce qui explique la suppression de la politique de sélection du composant Noyau. En effet, le composant Noyau n'utilise qu'un seul composant CASi, et cela, quelque soit le domaine utilisé. De même, les liens entre les composants CASi et le composant CS sont, eux aussi, des liens statiques. Par contre, les liens entre le composant CS, d'un coté, et, les composants CASe et le composant CF, d'un autre coté, sont des liens dynamiques car le domaine utilisé n'est pas connu d'avance.

Assemblage

Comme la nouvelle architecture du composant domaine-polymorphe se diffère par rapport à l'ancienne, cela implique systématiquement la modification de la phase d'assemblage. La figure 5.11 montre les différentes interactions entre les composants de l'architecture améliorée pour permettre au composant domaine-polymorphe de retrouver sa forme intégrale répondant aux exigences de la sémantique interne et celle du domaine. Le scénario d'assemblage est comme suit :

Après avoir identifié le domaine et effectué la phase de transformation, le composant Conteneur lance le composant Fabrique, qui sera détruit par la suite, et lui fournira l'identité du domaine et celle du composant Noyau à charger, et cela, afin de permettre au composant Conteneur de se connecter au composant Noyau et de lui créer les ports dont il a besoin. Une chargé et instancié par le composant Fabrique, le composant Noyau instancie directement le composant CASi (CAS interne) et récupère sa référencer. De même, comme les liens entre les composants CAS internes et le composant CS sont statiques, cela permet au composant CASi d'instancier directement le composant CS et de le référencer. Par contre, puisque les liens entre le composant CS, d'un conté, et, les composants CASe (CAS externes) et les composants CF, d'un autre coté, sont dynamiques, le composant CS doit d'abord sélectionner le composant CASe correspondant à la sémantique externe et récupérer le composant CF correspondant aux deux sémantiques, puis les instancier. Pour la sélection du bon composant CASe, le composant CS se base sur sa politique de sélection, qui est basée sur le même principe de celle donnée dans l'ancienne architecture. Par contre, nous avons choisis de passer le composant CF comme un paramètre au composant CS. Dans la section 5.5.2, nous avons expliqué pourquoi un tel choix a été pris.

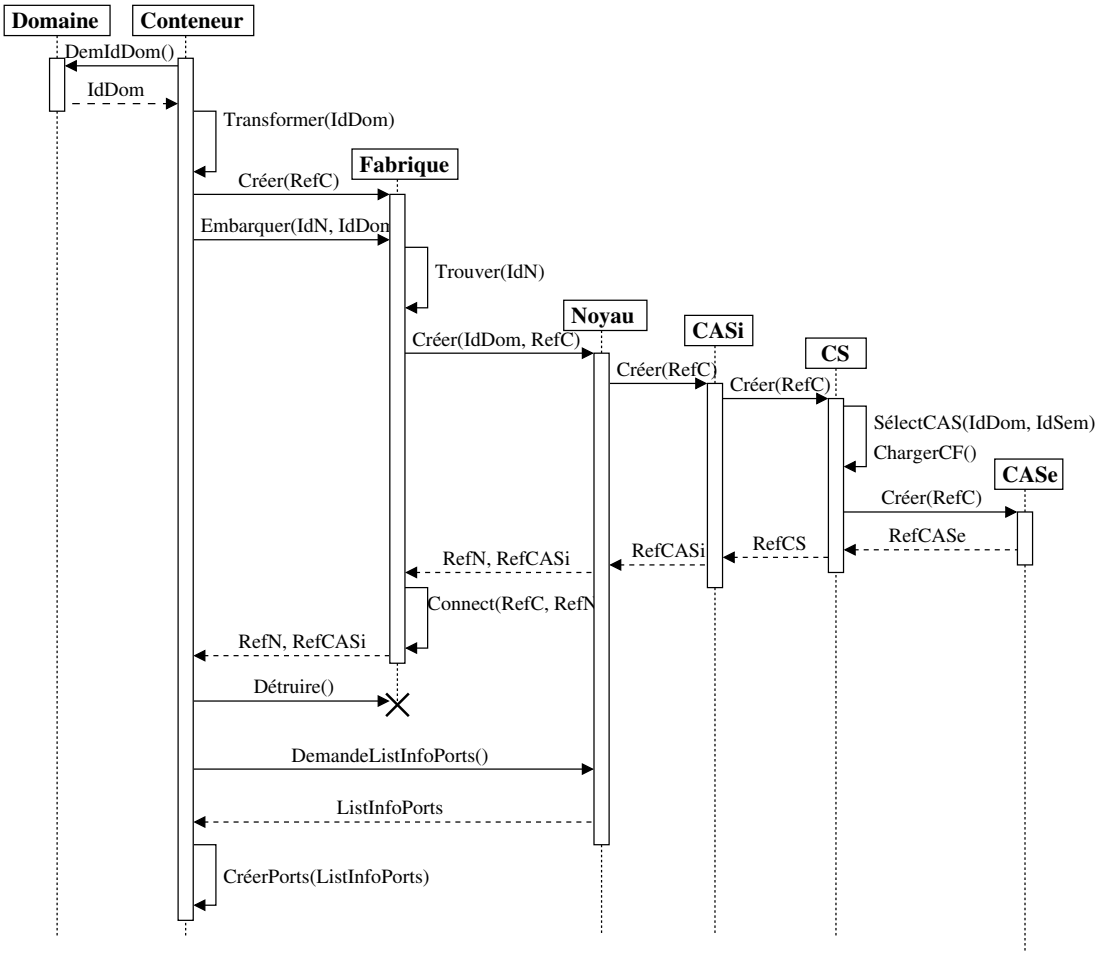
Composants CAS, CF et CS

Composant Conservateur Adaptateur de la Sémantique (CAS) Le composant CAS a deux rôles, le premier rôle est la conservation de la sémantique du composant Noyau et le deuxième rôle est le respect de la sémantique externe. C'est son emplacement dans l'architecture qui détermine son rôle à jouer. Nous détaillons les deux rôles comme suit :

CASi : Conservateur de la sémantique interne Dans le cas où le composant CAS est placé en interne, appelé CASi (CAS interne), son rôle est le même que celui assuré par les composants CSI sauf qu'avec le composant CAS, contrairement à l'utilisation des composants CSI où à chaque changement de domaine impliquera le changement de composant CSI, ne se préoccupe pas de la sémantique externe et est réutilisable dans tous les domaines.

CASe : Adaptateur à la sémantiqe externe Par contre, dans le cas où le composant CAS est placé en externe, appelé CASe (Cas externe), il assure les mêmes fonctionnalités que celles du composant ASE. De plus, dans le cas nécessaire, il réalise des tâches permettant le respect total de la sémantique externe. Par exemple, prenons le cas où la sémantique interne est une sémantique SR (réactive synchrone) et la sémantique externe est une sémantique SDF (flot de données synchrone), le composant CASe (qui est spécifique à la sémantique SDF) doit produire les données d'absence à la place de celles non produites par le composant Noyau afin de respecter les taux de production des données exigés par le domaine SDF.

Composant Frontières (CF) Le composant CF peut être spécifique à une sémantique donnée, S, et dans ce cas, il est responsable sur la production des données, ayant le format exigé par S, depuis les données, ayant d'autres formats exigés par les autres sémantiques, et vice versa, et cela pour les deux cas : S en interne ou en externe, c'est-à-dire qu'il assure le passage des données, qui non pas le



RefC : Référence sur le composant Conteneur
RefCS : Référence sur le composant CS.
RefN : Référence sur le composant Noyau
RefCASe : Référence sur CAS externe.
RefCASi : Référence sur CAS interne.
IdDom : Identité du domaine.
IdSem : Identité Sémantique interne.
IdN : Identité du Noyau.

FIG. 5.11: Diagramme UML de séquence montrant la phase d'assemblage du composant domaine-polymorphe amélioré.

même format, entre la sémantique S et les autres sémantiques, comme peut être spécifique à plusieurs sémantiques, c'est-à-dire qu'il assure le passage des données entre différentes sémantiques.

L'extériorisation de la préoccupation de passage entre les deux sémantiques, qui fait partie de la conception des systèmes, sous forme d'un composant présente l'avantage de permettre aux concepteurs des systèmes d'expliciter le passage des données entre les sémantiques sans altérer les autres composants, ce qui augmente la modularité et la maintenabilité des composants. De plus, cela permet aux concepteurs d'utiliser plusieurs composants CF afin de leurs attribuer différents rôles à savoir interpréteur, convertisseur, détecteur, etc. Cette variété de rôles, que le composant CF puisse assurer, justifie notre choix de prendre le composant CF comme un paramètre du composant CDP, ce qui permet de faciliter l'introduction des traitement souhaités au niveau de la frontière entre la sémantiques externe et la sémantique interne.

Composant Connecteur des Sémantiques (CS) Le composant CS (Connecteur des Sémantiques) permet de relier les composants CASi, CF et CASe. Son principal rôle est d'assurer l'exécution des opérations fournies par les composants CASe et CF dans le bon ordre afin de permettre une circulation correcte des données entre les composants CASi, CF et CASe. Il est la meilleure façon permettant

l'utilisation du composant CF avec le reste du composant CDP. Son avantage est donc d'éviter la duplication du même code dans les composants CAS, ce qui permet en effet d'avoir une bonne réutilisabilité et une bonne modularité. Le composant CS utilise toujours les mêmes interfaces implémentées par l'ensemble de composants CASe et le composant et CF.

5.5.3 Variables du CDP après assemblage

A l'exécution, tout composant domaine-polymorphe a un ensemble de variables, noté $CDP.X$, composé de deux sous ensembles : le premier sous ensemble contenant ses variables d'interface, qui sont ses ports de communication et ses paramètres, et le deuxième sous ensemble contenant ses variables internes. Donc,

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

Avec

$\{CDP.P = CDPIn\}$, $\{CDP.Q = CDPOut\}$ et $\{CDP.S = CDPParameters, CDPState\}$

Ainsi

$$CDP.X = \{CDPIn, CDPOut, CDPParameters, CDPState\}$$

Où :

- $CDPIn$ est l'ensemble de ports d'entrée dont chacun est associé à une entrée du composant Noyau ;
- $CDPOut$ est l'ensemble de ports de sortie dont chacun est associé à une sortie du composant Noyau ;
- $CDPParameters$ est composé de quatre sous ensembles permettant la configuration du composant domaine-polymorphe : le premier sous ensemble contient les paramètres du composant Noyau, le deuxième sous ensemble contient les paramètres du composant CASe, le troisième sous ensemble contient les paramètres du composant CASi et finalement le dernier sous ensemble contient les paramètres du composant Conteneur ;
- $CDPState$ est l'état du composant domaine-polymorphe.

5.5.4 Opérations

Comme l'architecture du composant domaine-polymorphe a changé, cela implique systématiquement que les opérations d'exécution du composant domaine-polymorphe données dans la section 5.2 ne sont plus valables.

Nous remarquons que le changement du modèle de composant domaine-polymorphe n'est qu'au niveau des composants interfaçant le composant Noyau et le composant Conteneur, ce qui implique donc la révision des opérations d'interaction entre les composants CASi, CS, CASe et CF.

Les composants CASi, CS, CF et CASe utilisent trois types d'opérations (voir la figure 5.12) à savoir : les opérations de communication pour permettre le transfert des données de l'intérieur vers l'extérieur et vice versa, les opérations de transformation des données, dites opérations de passage, permettant la transformation des données ayant format externe aux données ayant format interne et vice versa et les opérations de synchronisation entre les deux sémantiques. Nous avons introduit ces dernières à cause de la non dépendance entre les deux sémantiques afin de permettre aux composants CS, CF et CASe d'effectuer au bon moment des initialisations et d'éventuel traitement nécessaire au respect des sémantiques.

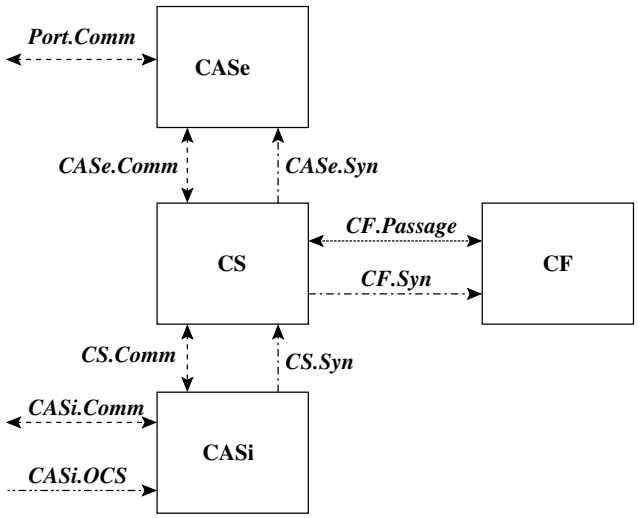


FIG. 5.12: Opérations d'interactions entre les composants CASi, CS, CF et CASe.

Opérations exécutées par CASi

Le composant CASi exécute un ensemble d'opérations, noté $CASi.Oper$, fourni par le composant CS et comporte deux sous ensembles d'opérations. Le premier sous ensemble, $CS.Comm$, regroupe les opérations de communication et le deuxième sous ensemble, noté $CS.Syn$, regroupe les opérations de synchronisation. Les opérations de communications permettent l'envoi (resp. la récupération) des données vers (resp. venant) l'intérieur (resp. de l'extérieur) et les opérations de synchronisation sont des opérations permettant, à travers le composant CS, l'initialisation des composants CASe et CF et de les mettre au courant avant le début et après la fin de chaque réaction interne, ce qui permet au composant CASe d'effectuer d'éventuel traitement nécessaire au respect de la sémantique externe. Par exemple, un composant CASe spécifique (dédié) à la sémantique SDF ne peut savoir quels sont les ports de sortie pour lesquels le composant Noyau n'a pas produit des données jusqu'à ce qu'il soit, par une opération de synchronisation (EoR : End of Reaction), informé sur la fin de la réaction interne.

Nous résumons donc comme suit :

$$CASi.Oper = CS.Comm \cup CS.Syn$$

avec

$$CS.Comm = \{hasData, getData, hasPlace, sendData\}$$

et

$$CS.Syn = \{InitCS, BoRCS, EoRCS\}$$

Où

- $hasData$: Demande au composant CS s'il y'a des données disponibles ou non.
- $getData$: Permet la récupération des données depuis le composant CS.
- $hasPlace$: Demande au composant CS si l'opération sendData peut être effectuée avec succès.
- $sendData$: Permet l'envoi des données au composant CS.
- $InitCS$: Initialise le composant CS.
- $BoRCS$: Met au courant le composant CS avant la réaction interne.
- $EoRCS$: Met au courant le composant CS après la réaction interne.

Opérations exécutées par CS

Le composant CS exécute un ensemble d'opérations, noté $CS.Oper$, fourni par les composants CASe et CF. Les opérations fournies par le composant CASe sont les opérations de communication et les opérations de synchronisation. Les opérations fournies par le composant CF sont les opérations de synchronisation et les opérations de passage.

L'exécution d'une opération de synchronisation par le composant CASi implique systématiquement l'exécution des opérations de synchronisation (fournies par CASE et CF) correspondantes par le composant CS. Par exemple, l'exécution de l'opération $InitCS$ par le composant CASi impliquera l'exécution systématique des deux opérations $InitCASe$ et $InitCF$ par le composant CS.

Les opérations de passage sont responsables sur la transformation des données échangées entre les deux sémantiques. Elles permettrent ainsi au composant CS de fournir aux composants CASi et CASe des données ayant les formats respectant leurs sémantiques. Pour cela, nous avons définit quatre opérations de passage dont la spécification des comportements est laissée aux concepteurs car cela fait partie de la conception des systèmes. Ces opérations sont détaillées comme suit :

- $DefaultOutputData$: Dans le cas où la production des données sur tous les ports de sortie du CDP est exigée par la sémantique externe et que le composant Noyau n'a pas produit des données pour certains ports de sortie, cette opération permet donc au composant CS de fournir au composant CASe la donnée de sortie à envoyer par défaut. Par exemple, cas d'un CASe spécifique à la sémantique SDF. Cette donnée est fournie par le concepteur ;
- $DefaultInputData$: De même, cette opération permet au composant CS de fournir au composant CASi la donnée d'entrée à utiliser par défaut dans le cas où la présence des données sur tous les ports d'entrée est exigée par la sémantique interne alors que l'environnement n'a pas fourni des données sur quelques ports d'entrée et nous voulions quand même lancer la réaction du composant Noyau. Par exemple, cas d'un CASi spécifique à la sémantique SDF. Cette donnée, elle aussi, est fournie par le concepteur ;
- $InputDataFromOutputData$: Cette opération est utilisée par le composant CS juste après l'invocation de la méthode de lecture définie par le composant CASe et avant le retour d'appel de la méthode de lecture définie par le composant CS afin de retourner au composant CASi les données ayant le format exigé par sa (CASi) sémantique ;
- $OutputDataFromInputData$: Cette opération a le même rôle que la précédente mais dans le sens inverse.

Nous résumons comme suit :

$$CS.Oper = CASe.Comm \cup CASe.Syn \cup CF.Passage \cup CF.Syn$$

avec

$$CASe.Comm = \{hasCASeData, getCASeData, hasCASePlace, sendCASeData\},$$

$$CASe.Syn = \{InitCASe, BoRCASe, EoRCASe\}$$

$$CF.Syn = \{InitCF, BoRCF, EoRCF\}$$

et

$$CF.Passage = \{DefaultOutputData, DefaultInputData, InputDataFromOutputData, OutputDataFromInputData\}$$

Où

- $hasCASeData$: Demande au composant CASe s'il y'a des données disponibles ou non.
- $getCASeData$: Permet la récupération des données depuis le composant CASe.
- $hasCASePlace$: Demande au composant CASe si l'opération sendCASeData peut être effectuée avec succès.
- $sendCASeData$: Permet l'envoi des données au composant CASe.
- $InitCASe$: Initialise le composant CASe.
- $BoRCASe$: Met au courant le composant CASe avant la réaction interne.
- $EoRCASe$: Met au courant le composant CASe après la réaction interne.
- $InitCF$: Initialise le composant CF.
- $BoRCF$: Met au courant le composant CF avant la réaction interne.
- $EoRCF$: Met au courant le composant CF après la réaction interne.

Opérations exécutées par CASe

Les opérations exécutées par le composant CASe sont les mêmes opérations de communications du composant ASE de l'architecture du composant domaine-polymorphe prcédente.

Réorganisation du flot de contrôle

L'architecture améliorée du composant domaine-polymorphe nous a permis d'avoir une séparation nette entre la sémantique interne et la sémantique externe, ce qui a donné l'avantage d'une importante réduction du nombre de composants utilisés et d'extériorisation la frontière entre les deux sémantiques ainsi les concepteurs peuvent expliciter le passage des données sans modifier le reste de composants du CDP. Cependant, les opérations du flot de contrôle données précédemment posent de sérieux problèmes par rapport au modèle amélioré du composant domaine-polymorphe, à savoir :

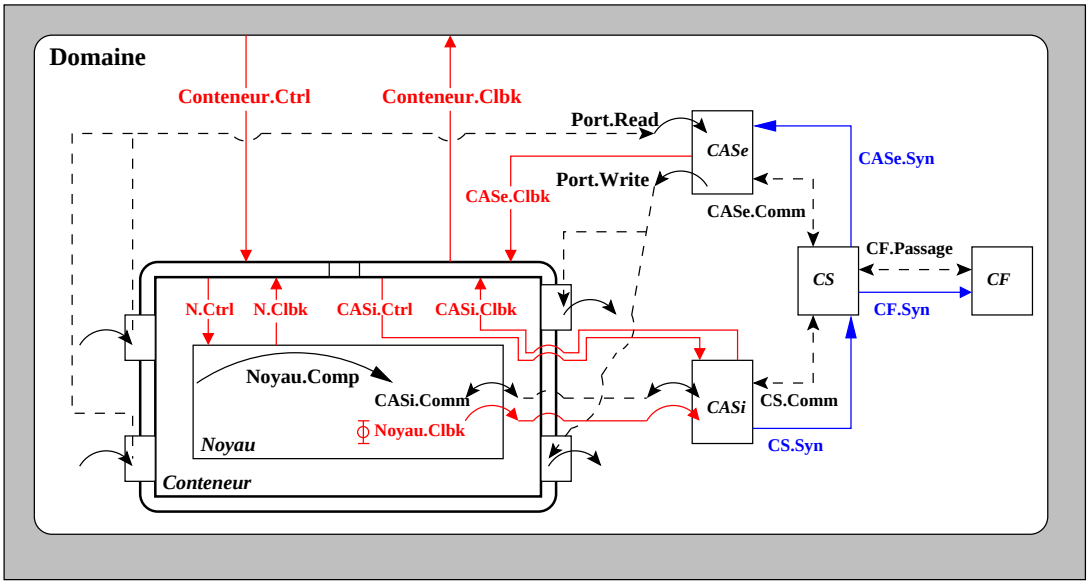
- Le contrôle du composant CASi est assuré par le composant Noyau à travers les opérations BoR et EoR appartenant à $Noyau.Sem$, ce qui explique la non séparation entre les opérations de flot de contrôle et les opérations de flot de données au niveau du composant Noyau, ce qui peut conduire à des mauvaises conséquences ;
- Les opérations $\Phi Noyau.Clbk$ de rappel (call-back) permettant au composant Noyau de solliciter des services auprès du composant Conteneur (donc auprès de la sémantique externe) ne sont pas contrôlées par le composant CASi. Cela peut conduire à des incohérences entre les deux sémantiques ;
- Pour solliciter un service auprès de la sémantique externe, le composant CSI doit passer par le composant Noyau à travers les opérations $\Phi Noyau.Sem$. Par exemple, à la fin d'une réaction, le composant Noyau consulte le composant CSI pour voir s'il veut solliciter un $reTrigger$ ou non. Cela rend les opérations de contrôle et les opérations de call-back accessibles aux concepteurs, ce qui implique la non séparation entre celles-ci (de contrôle et de call-back) et les opérations de flot de données.

Comme solution, nous avons proposé de :

- Rendre le contrôle du composant CASi sous la responsabilité du composant Conteneur. En effet, nous avons déplacé les opérations $\Phi Noyau.Sem$ du composant Noyau au composant Conteneur, ensuite, nous les avons regroupées en deux sous ensembles d'opérations :

  1. Le sous-ensemble $CASi.Ctrl$ permettant au composant Conteneur, à travers l'opération $InitCASi$, l'initialisation du composant CASi et, à travers les opérations BoR et EoR, de le mettre au courant avant et après chaque réaction du composant Noyau. Nous résumons donc comme suit :

$$CASi.Ctrl = \{InitCASi, BoR, EoR\}$$



N.Ctrl/N.Clbk : Noyau.Ctrl/Noyau.Clbk. X.Syn : X.Synchronization.

FIG. 5.13: Opérations exécutées par le composant domaine-polymorphe amélioré.

2. Le sous ensemble $CASi.Clbk$ qui permet au composant CASi, à travers les opérations $Ready$ et $NotReady$, d'informer le composant Conteneur si les conditions sont satisfaites ou non pour faire réagir le composant Noyau et, grâce aux opérations $\Phi CASi.Clbk$, de solliciter d'éventuels services auprès du composant Conteneur. Le nombre et le comportement des opérations $\Phi CASi.Clbk$ dépendent de la sémantique externe. Nous résumons donc comme suit :

$$CASi.Clbk = \{Ready, NotReady\} \bigcup \Phi CASi.Clbk$$

– Rendre les opérations $\Phi Noyau.Clbk$ de rappel (call-back) sous la responsabilité du composant CASi. En effet, le composant Noyau sollicitera toujours des services auprès du composant CASi. Donc, l'ensemble d'opérations $Noyau.Clbk$ ne contient qu'une seule opération, $finishReaction$;
– Rendre l'exécution de l'ensemble d'opérations appartenant à $\Phi Noyau.Sem$ et permettant le respect de la sémantique externe sous la responsabilité du composant CASe. En effet, une fois informé à travers une opération de synchronisation, le composant CASe peut solliciter d'éventuels services auprès du composant Conteneur (donc auprès de la sémantique externe qui est sa sémantique) afin de respecter sa sémantique. Cet ensemble d'opérations, noté $CASe.Clbk$, dépend de la sémantique externe.

La figure 5.13 montre les différentes opérations d'exécution du composant domaine-polymorphe amélioré. Nous résumons ces opérations dans le tableau 5.3

Op. flot de données			Op. de synch.	Op. flot de contrôle
Op. de calcul	Op. de comm.	Op. de passage		
				initialization
	existData			preCondition
	read			trigger
	isFull			postCondition
	write		InitCS	finish
	hasCASeData	DefaultInputData	BoRCS	ΦConteneur.Clbk
	getCASeData	DefaultOutputData	EoRCS	initCore
computeBehaviorCore	hasCASePlace	InputDataFromOutputData	InitCASe	preReaction
	sendCASeData	OutputDataFromInputData	BoRCASe	reaction
	hasData		EoRCASe	postReaction
	getData		InitCF	finishReaction
	hasPlace		BoRCF	ΦNoyau.Clbk
	sendData		EoRCF	InitCASi
	hasInputData			BoR
	receiveData			EoR
	hasFreePlace			Ready
	emitData			NotReady
				ΦCASi.Clbk
				CASe.Clbk

TAB. 5.3: Opérations d'exécution du CDP

5.5.5 Polymorphisme de messages

L'exécution des opérations de communication par les composants CASi, CS et CASe permet d'acheminer les données provenant du composant Noyau jusqu'à l'extérieur et vice versa. Cependant, les deux sémantiques (interne et externe) sont différentes et chacune d'elles a son propre format des données, qui peut être un événement, flot de données, etc. Par exemple, la sémantique DE utilise la notion d'événement, qui a le format $(data, delay)$, et la sémantique SDF utilise le flot de données bruts. Cette variété de formats des données pose le problème de typage lors d'échange des données entre les composants CASi, CS, CF et CASe (cf. figure 5.14), ce qui aura donc comme conséquence la non possibilité de réutiliser les composants CS et CF.

Pour répondre à ce problème nous avons fixé les objectifs suivants :

- Permettre à chacun des composants CASi et CASe de toujours travailler sur (manipuler) le même format de données correspondant à sa sémantique, et cela, d'une manière transparente par rapport à l'autre sémantique ;
- Utiliser le même type de flot de données afin de permettre aux composants CASi, CS, CF et CASe d'échanger toujours le même type de messages entre eux. Cela permet surtout au composant CS de faire circuler des données entre les deux sémantiques (interne et externe) sans se préoccuper du problème de typage, les formats des données sont complètement cachés ;
- Donner aux concepteurs des systèmes plus d'information sur les données échangées entre les deux sémantiques afin de leurs permettre, au niveau du composant CF, de spécifier les bons traitements permettant ainsi de bien expliciter le passage.

Comme solution, nous avons proposé l'encapsulation des données échangées entre les composants CASi, CS, CF et CASe dans des messages, ce qui permet de cacher les formats des données. Cette solution permet d'avoir le *polymorphisme de messages*.

Le principe du polymorphisme de messages est le même que celui de données (Token) proposé par l'équipe de Ptolemy II. Donc, un message est une classe encapsulant un format de données.

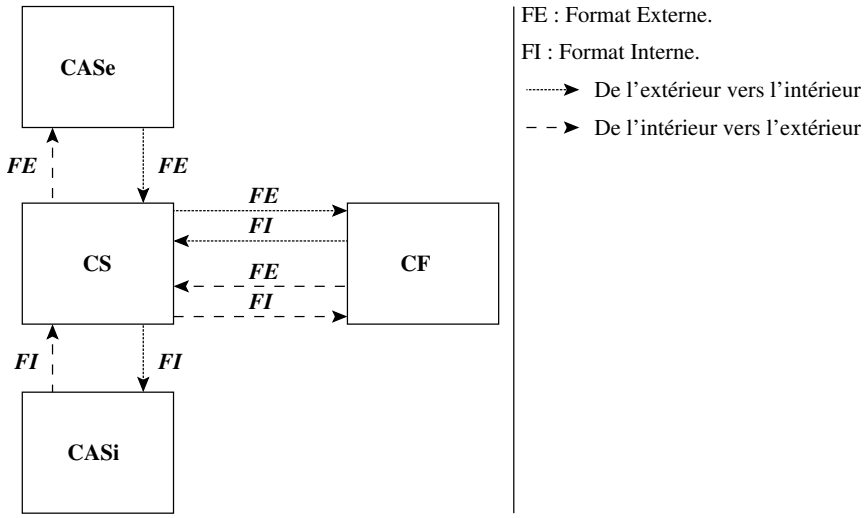


FIG. 5.14: Formats des données échangés entre les composants CASi, CS, CF et CASe.

Cette classe permet ainsi de cacher tout format de donées aux composants CASi, CS, CF et CASe, et fournit une interface pour la lecture et/ou l'écriture des données. La classe message peut être redéfinie en utilisant l'héritage afin d'introduire de nouveaux formats de données.

Dans la figure 5.15 nous avons donné le diagramme UML de classes montrant comment le polymorphisme de messages peut être obtenu en se basant sur l'héritage. Par exemple, la classe abstraite $Message$, qui est le type des données échangées entre les composants CASi, CS, CF et CASe, est la classe qui représente le message de base permettant d'encapsuler une donnée brute. En effet, l'introduction de tout nouveau type de message nécessite la définition d'une nouvelle classe s'héritant de la classe $Message$. Selon les domaines que nous avons vus, nous sommes arrivés à définir les messages suivants, qui s'héritent de la classe $Message$: $DataMessage$ pour les sémantiques à flot de données non temporisé, $DatedMessage$ pour les sémantiques temporisées et $SynchronousMessage$ pour la sémantique synchrone. Aussi, par exemple dans le cas d'une sémantique interne DE, il est possible de définir un message contenant une liste de messages dans chacun d'eux encapsule un événement, et cela, afin de prendre en considération les rafales d'événements. Il est clair que le polymorphisme de messages apporte non seulement une réponse au problème de typage mais aussi représente beaucoup d'intérêts : Permettre aux concepteurs d'avoir toutes les informations concernant les données échangées (date d'apparition, retard, etc.) et de leurs permettre aussi d'introduire leurs propres informations au profil de ces données. Par exemple, prenons un composant domaine-polymorphe dont le composant Noyau obéit à la sémantique DE et s'exécutant dans une sémantique SDF. En effet, le type de message utilisé pour le format des données externe est $DataMessage$ et pour celui des données interne est $DatedMessage$. Cela permet à un utilisateur, dans $DatedMessage$, de spécifier la date (relative) d'occurrence de l'événement par rapport au temps courant de la sémantique DE, c'est-à-dire, la possibilité de permettre aux concepteurs de retarder la livraison de la donnée, provenant de l'extérieure, au composant Noyau, ce qui n'est pas possible actuellement avec les plates formes de modélisation et de conception des systèmes hétérogènes où dans telle situation, le retard est toujours égale à la valeur par défaut, qui est zéro.

5.6 Méta modèle de composants domaine-polymorphes

La figure 5.16 montre le méta modèle pour les composants domaine-polymorphes. Les composants sont représentés par des classes et les liens d'interconnexion sont représentés par des interfaces. Sans rentrer dans la spécification de leurs contenus, ces classes et interfaces sont présentées comme suit :

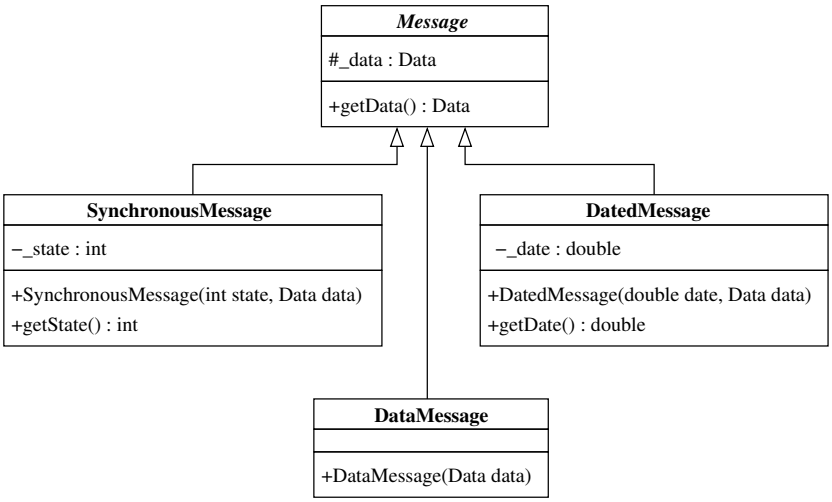


FIG. 5.15: Diagramme UML de classe montrant le polymorphisme de message.

- $Container$: Classe représentant le composant Conteneur.
- $Core$: Classe représentant le composant Noyau.
- CAS : Classe représentant le composant CAS. Elle est dupliquée, une pour représenter le composant CASi et l'autre pour représenter le composant CASe.
- $BorderComponent$: Classe représentant le composant CF.
- $ConnectorSemantics$: Classe représentant le composant CS.
- $Execution$: Interface implémentée par la classe $Core$ et permettant au composant Conteneur d'exécuter le composant Noyau.
- $Control$: Interface implémentée par la classe CAS et permettant au composant Conteneur d'exécuter les opérations de contrôle du composant CASi.
- $ConnectICAS$: Interface implémentée par la classe CAS et permettant au composant Noyau d'exécuter les opérations de communication ainsi que les opérations de rappel (call-back)du composant CASi.
- $ConnectCS$: Interface implémentée par la classe $ConnectorSemantics$ et permettant au composant CASi d'exécuter les opérations de communication et de synchronisation du composant CS.
- $ConnectBC$: Interface implémentée par la classe $BorderComponent$ et permettant au composant CS d'exécuter les opérations de passage et de synchronisation du composant CF.
- $ConnectECAS$: Interface implémentée par la classe CAS et permettant au composant CS d'exécuter les opérations de communication et de synchronisation du composant CASe.
- $ConnectContainer$: Interface implémentée par la classe $Container$ et permettant au composant CASe d'exécuter les opérations de communication et de rappel du composant Conteneur.

5.7 Modèle d'exécution du composant domaine-polymorphe

Maintenant, que nous avions donné l'architecture, les opérations d'exécution et le méta modèle du composant domaine-polymorphe, nous présentons dans cette section le modèle d'exécution du composant domaine-polymorphe.

L'exécution du composant domaine-polymorphe passe par deux phases. La première phase permet son initialisation et la deuxième phase consiste à le réitérer plusieurs fois jusqu'à la fin d'exécution du système duquel fait partie.

Le déroulement d'une itération du composant domaine-polymorphe est décrit par la figure 5.17. Il est donné à titre illustratif et aucune supposition n'est faite ni sur les deux modèles de calcul

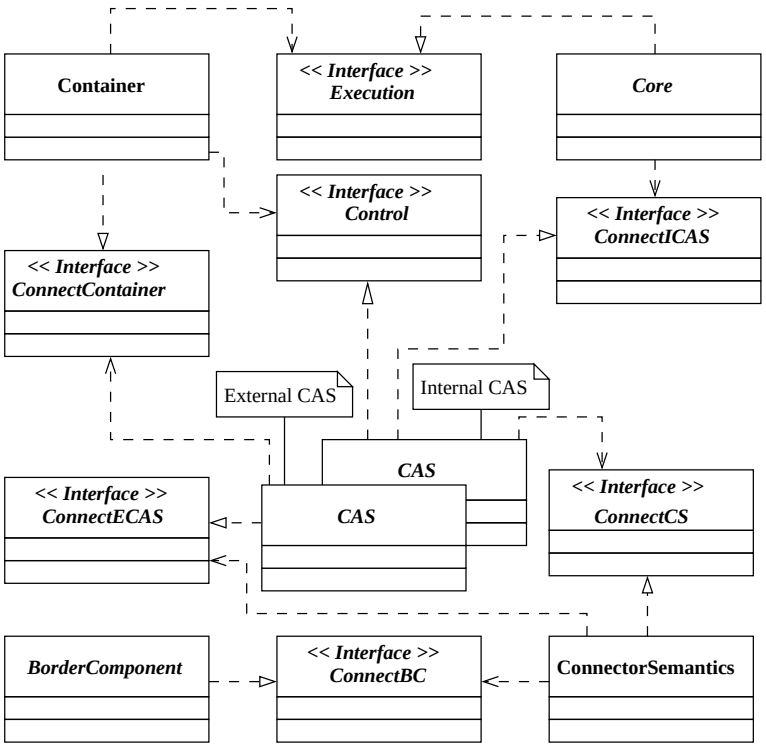


FIG. 5.16: Méta modèle de composants domaine-polymorphes.

(interne et externe) utilisés ni sur l'implémentation. Par contre, puisque notre étude est faite sous un environnement responsable, les composants domaine-polymorphes, une fois déclenchés par leurs domaines, sont toujours prêts à réagir.

Voici les étapes d'exécution d'une itération du composant domaine-polymorphe :

1 : Le domaine exécute $preCondition$.
2 : Avant d'exécuter toute opération du Noyau, le Conteneur met au courant CASi en exécutant BoR.
3 : CASi exécute $BoRCS$.
4 : CS exécute successivement $BoRCASe$ et $BoRCF$.
5 : CASi exécute les opérations de lecture du $CS.Comm$.
6 : CS exécute les opérations de lecture du $CASe.Comm$.
7 : CASe exécute les opérations de lecture du $Port.Comm$
8 : Après avoir récupérer les données sous format externe, CS execute $CF.Passage$.
9 : Après avoir eu les données lues sous format interne, CASi répond par $Ready$ ou $NotReady$.
10 : Si $NotReady$ alors le Conteneur exécute EoR et on passe à l'étape 12.
11 : Si $Ready$:
  11.1 : Le Conteneur exécute $preReaction$.
  11.2 : Le domaine exécute $trigger$.
  11.3 : Le Conteneur exécute $reaction$.
  11.4 : Le Noyau lit les données par $hasInputData$ et $receiveData$.
  11.5 : Le Noyau exécute $computeBehaviorCore$.
  11.6 : Le Noyau envoie ses données par $hasFreePlace$ et $sendData$.
  11.7 : Le domaine exécute $postCondition$.
  11.8 : Le Conteneur exécute $postReaction$.
  11.9 : Le Conteneur exécute EoR.
  11.10 : CASi exécute les opérations d'écriture du $CS.Comm$.
  11.11 : CS exécute $CF.Passage$.

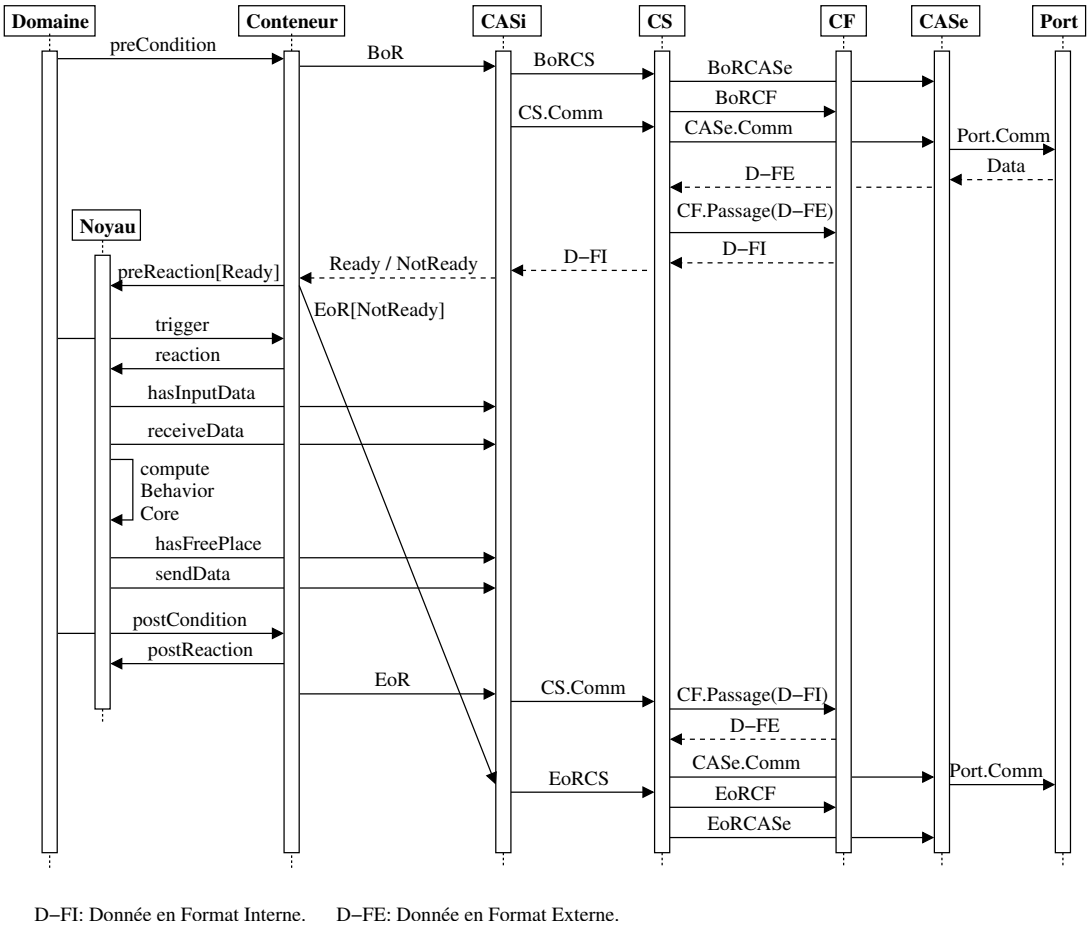


FIG. 5.17: Diagramme UML de séquence décrivant une itération complète du CDP.

11.12 : CS exécute les opérations d'envoi du $CASe.Comm$.
11.13 : CASe exécute les opérations d'écriture des données du $Port.Comm$.
12 : CASi exécute $EoRCS$.
13 : CS exécute $EoRCF$.
14 : CS exécute $EoRCASe$.

5.8 Conclusion

Nous avons présenté dans ce chapitre la conception du modèle de composant domaine-polymorphe. Ce dernier permet l'adaptation automatique du comportement de son composant Noyau obéissant à une sémantique dans différents domaines.

Le modèle de composant domaine-polymorphe que nous avions proposé est donné à un niveau abstrait et il comporte un composant Conteneur et deux composants CAS (Conservateur Adaptateur de la Sémantique), dont chacun d'eux est spécifique à une seule sémantique.

Le composant Conteneur permet au composant domaine-polymorphe d'avoir la structure exigée par le domaine dans lequel il est utilisé. Le premier CAS est utilisé en interne, noté CASi, et permet au composant Noyau de fonctionner correctement. Le deuxième composant CAS est utilisé en externe, noté CASe, manipule les ports de communication du composant Conteneur et exécute d'éventuel traitement permettant au composant domaine-polymorphe de fonctionner correctement dans le domaine hôte.

Les deux composants CASi et CASe sont interconnectés à travers le composant CS (Connecteur des Sémantiques) qui utilise, lui aussi, un composant CF (Composant Frontière). Ce dernier permet aux concepteurs des systèmes d'expliciter le passage des données entre la sémantique du domaine hôte (sémantique externe) et celle auquelle le composant Noyau obéit (sémantique interne). Cette explicitation du passage fait partie de la conception des systèmes.

Pour être réutilisable dans différents domaines, le composant domaine-polymorphe est doté d'une politique de transformation lui permettant de changer sa structure et son comportement suivant les exigences du domaine hôte.

Le degré de polymorphisme du composant domaine-polymorphe dépend des techniques, qui sont liées aux technologies existantes en génie logiciel, utilisées dans la concrétisation de la politique de transformation, voir chapitre 4. en effet, tant que la transformation du composant domaine-polymorphe n'implique pas une intervention externe, qui peut être un utilisateur ou un programme, le degré de polymorphisme sera plus élevé sinon sera moins élevé.

Par contre, d'après le chapitre 4, il est difficile, pour ne pas dire impossible, d'avoir un degré de polymorphisme à cent pour cent. En effet, dans ce cas nous proposons d'utiliser le composant domaine-spécifique flexible qui présentent les mêmes avantages que le composant domaine-polymorphe, sauf que ce dernier utilise un seul composant Conteneur alors que lui nécessite la création d'un composant Conteneur pour chaque domaine.

Dans le chapitre suivant, nous descendons au niveau de spécification afin de montrer comment le composant CAS peut être spécifié pour une sémantique et ainsi l'utilisation des composants domaine-polymorphes dans le mélange des modèles de calcul.

Chapitre 6

Conception Hétérogène à base de Composants Domaine-Polymorphes

6.1 Introduction

Dans le chapitre précédent, nous avons présenté les étapes de conception du composant domaine-polymorphe en définissant tous ses composants ainsi que le rôle de chacun d'eux. Ensuite, nous avons, à un niveau abstrait, défini toutes les opérations d'exécution du composant domaine-polymorphe. Ces opérations sont classées en trois types : les opérations de contrôle, les opérations de synchronisation et les opérations de flot de données qui comportent les opérations de communication, de calcul et de passage.

Par sa définition, le composant domaine-polymorphe est un composant ayant une façade externe (composée des composants Conteneur et CASe) obéissant à la sémantique du domaine hôte et un comportement interne (représenté par le composant Noyau) obéissant à une sémantique interne (implémentée dans le composant CASi), cela fait de lui un composant hétérogène. Cette caractéristique permet donc l'introduction d'une nouvelle approche d'hétérogénéité que nous l'avons appelée *approche hétérogène atomique*.

L'utilisation des composants domaine-polymorphes dans la conception des systèmes embarqués hétérogènes nous oblige de descendre au niveau plus bas (niveau de spécification) car le traitement effectué par chaque composant CAS (qu'il soit en interne ou en externe) ainsi que ses opérations dépendent de la sémantique pour laquelle il est spécifique.

En effet, dans ce chapitre, nous avons montré comment spécifier le comportement du composant CAS (pour les cas en interne et en externe) pour les modèles de calcul. Pour cela, nous avons choisi les modèles de calcul SR, DE et SDF.

La spécification du composant CAS pour un modèle de calcul revient à définir sa structure interne et le comportement de ses opérations d'interaction avec les autres composants du composant domaine-polymorphe. Ensuite, nous avons présenté des cas de mélange de modèles de calcul (en anglais : mixing of model of computation) à base de composant domaine-polymorphe afin d'étudier les interactions entre les sémantiques et montrer les avantages de l'approche hétérogène atomique par rapport aux autres approches hétérogènes.

6.2 Composant domaine-polymorphe : Une approche hétérogène atomique

De son architecture, le composant domaine-polymorphe exécute un composant Noyau suivant la sémantique de sa spécification tout en fonctionnant suivant la sémantique du domaine hôte, ce qui fait de lui un *composant hétérogène*.

En effet, le composant domaine-polymorphe permet d'exécuter son comportement interne spécifié suivant une sémantique, qui est implantée à son intérieur, au sein d'une autre sémantique, et cela, sans passer ni par l'approche hiérarchique ni par l'approche non hiérarchique ni par l'utilisation des composants domaine-spécifiques.

Comme le montre la figure 6.1, grâce à l'hétérogénéité du composant domaine-polymorphe, il est actuellement possible de faire communiquer plusieurs composants obéissants aux sémantiques D_i, D_j et D_k au sein d'une même sémantique, qui est différente, sans faire recours ni aux niveaux hiérarchiques ni aux composants HICs. Cependant, la question qui se pose est *Quel est le type d'hétérogénéité du composant domaine-polymorphe ?*

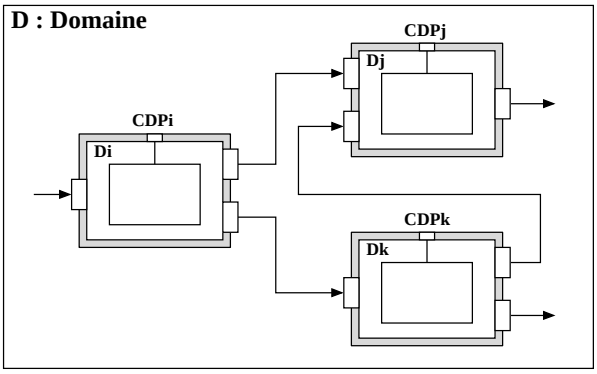


FIG. 6.1: Hétérogénéité atomique.

Pour répondre à cette question, nous savons que, d'une part, l'hétérogénéité hiérarchique consiste à un changement de niveau hiérarchique à chaque changement de modèle de calcul, et d'autre part, l'hétérogénéité non hiérarchique consiste à un changement de sous ensemble homogène à chaque changement de modèle de calcul. Or, le composant domaine-polymorphe permet un changement de modèle de calcul sans changer ni niveau hiérarchique ni de sous ensemble homogène. En effet, la réponse que nous donnions à la question posée ci-dessus est : comme le composant domaine-polymorphe est, à la fois, hétérogène et atomique, le type de son hétérogénéité est une *hétérogénéité atomique*.

6.3 Conception hétérogène atomique

Nous savons maintenant que le composant domaine-polymorphe permette une conception des systèmes embarqués hétérogènes sans passer ni par l'approche hiérarchique ni par l'approche non hiérarchique, c'est donc la conception hétérogène atomique des systèmes.

Nous montrons dans cette section comment utiliser les composants domaine-polymorphes dans la conception des systèmes, en particulier pour le mélange des modèles de calcul.

Pour des raisons illustrative, nous prenons un exemple de système hétérogène simple, voir figure 6.2. Ce système est composé de trois modules A, B et C, avec A et C obéissent au modèle de calcul $D1$ et B obéit au modèle de calcul $D2$. Pour concevoir ce système suivant l'approche hétérogène atomique, nous devons choisir les modèles de calcul à utiliser comme $D1$ et $D2$.

Le choix et le mélange des modèles de calcul dépendent du système à concevoir. Et comme le système donné par la figure 6.2 est un système anonyme, c'est-à-dire que nous ne connaissons pas les modèles de calcul $D1$ et $D2$, un choix et un mélange aléatoires des modèles de calcul peuvent conduire à une conception du modèle non cohérente et ainsi n'aura aucun sens physique car les caractéristiques du modèle de calcul englobé peuvent être induites par les caractéristiques du modèle

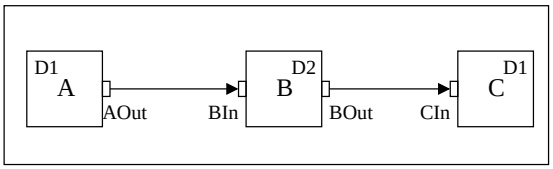


FIG. 6.2: Exemple d'un système hétérogène.

de calcul englobant, c'est-à-dire, le modèle de calcul englobé ne fonctionnera pas avec la prise en compte de toutes ses caractéristiques dans le cas où ces dernières ne seront pas toutes contenues par celles du modèle de calcul englobant. Par exemple, le modèle de calcul DE perd la notion de temps dans le cas où il est englobé par le modèle de calcul SDF. Il est donc impossible de produire des événements avec des retards, ce qui fait perdre à DE l'une de ses caractéristiques.

En effet, pour des raisons de cohérence, nous avons choisi les modèles de calcul utilisés par l'équipe de Ptolemy II dans des cas de mélange de modèles de calcul. Ces modèles de calcul sont : DE (Discrete Event), SDF (Synchronous DataFlow) et SR (Synchronous Dataflow).

Dans ce qui suit, nous commençons d'abord par la spécification des composants CASs pour les modèles de calcul choisis et ensuite nous présentons les cas de mélange de ces derniers.

6.3.1 Spécification des composants CASs

Comme tout composant CAS peut être en interne comme en externe (conservateur, adaptateur), nous spécifions les deux parties CASi et CASe de chacun des CASs pour les modèles de calcul DE, SDF et DE.

Nous signifions ici par la phrase *spécification du composant CAS pour le modèle de calcul S* : l'implémentation du modèle de calcul S au niveau de la partie CASi pour permettre au composant Noyau de fonctionner suivant la sémantique S et la spécification au niveau de la partie CASe le traitement nécessaire permettant la bonne interaction du composant domaine-polymorphe au sein du domaine hôte implémentant le modèle de calcul S. Ainsi, le composant CAS sera appelé *CAS S*

La spécification des composants CASs pour les modèles de calcul n'est pas une tâche facile et demande de connaître toutes les caractéristiques desdits modèles de calcul. Cependant, le fait que le composant domaine-polymorphe est un composant atomique, nous n'avons pas besoin de spécifier un ordonnanceur (en anglais Scheduler), ce qui est différent pour le composant composite (acteur composite) implémentant un modèle de calcul dont l'ordonnanceur doit être spécifié afin d'établir l'ordre d'exécution de ses acteurs internes.

Spécification du composant CAS SR

Partie de conservation de la sémantique SR : CASi SR La spécification de CASi SR pour le modèle de calcul SR doit prendre en compte les deux modes de fonctionnement du composant Noyau : Le composant Noyau peut être un composant Strict ou NonStrict. Avec le mode Strict, le composant Noyau exige que toutes ses entrées soient connues (présentes ou absentes) avant sa réaction. Par cotre, avec le mode NonStrict, qui n'a de sens qu'avec un domaine hôte SR, le composant Noyau peut réagir même s'il a quelques entrées qui sont inconnues et le CDP sera réitéré au cours du même instant jusqu'à sa convergence.

La figure 6.3 montre l'architecture de la partie interne (CASi SR) du composant CAS SR. CASi SR utilise des tampons d'entré et des tampons de sortie dont chacun ayant deux champs. Le premier

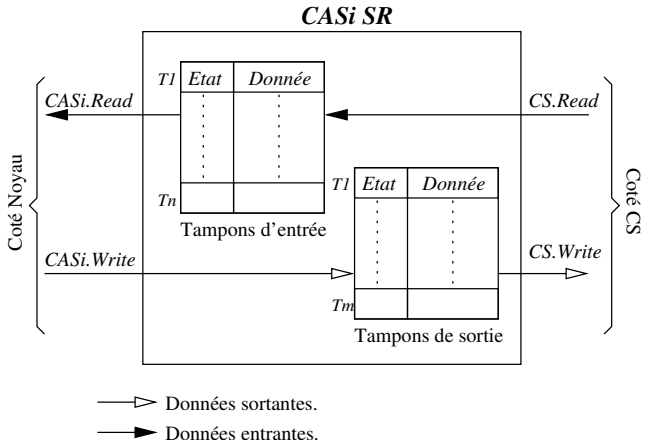


FIG. 6.3: Architecture du CASi SR.

champ indique l'état du deuxième champs, qui peut contenir les valeurs *Inconnu, Présent ou Absent* tandis que le deuxième champ contient la donnée communiquée.

Au court du même instant, CASi SR passe par trois étapes : Avant, au cours et après la réaction du composant Noyau. Les traitements effectués par CASi SR lors des trois étapes sont détaillés comme suit :

Avant la réaction : Une fois l'opération $CASi.BoR \in CASi.Ctrl$ est déclenchée, CASi SR effectue le traitement donné par l'organigramme de la figure 6.4. D'abord, il exécute l'opération de synchronisation $BoRCS$. Ensuite :

Cas mode NonStrict : Avec ce mode, CASi SR effectue les tâches suivantes :

- CASi SR vérifie si le domaine hôte est bien SR sinon il génère une exception.
- Si domaine hôte est bien SR, CASi SR court-circuite les tampons d'E/S s'ils ne les sont pas déjà.
- CASi SR répond toujours *Ready* afin de permettre l'exécution du composant Noyau qui n'est plus sous son contrôle.

Avec le mode NonStrict, le respect du principe de la monotonie (le composant domaine-polymorphe ne peut être réitéré au cours du même instant que si au moins l'état d'un de ses ports a changé d'inconnu à connu) est assuré par le domaine hôte SR. En effet, CASi SR n'effectue aucune tâche de contrôle de convergence.

Comme les tampons sont court-circuités, nous recommandons aux concepteurs des systèmes d'attribuer au composant CF le rôle d'un relais, c'est-à-dire de faire passer les données de l'intérieur à l'extérieur ou vice versa sans modifier ni l'état ni le contenu du SynchronousMessage venant du CASe SR car le non respect de cette recommandation conduit à un mauvais fonctionnement du composant domaine-polymorphe. Avec le mode NonStrict, le composant Noyau s'interagit avec son extérieur comme s'il n'est pas encapsulé dans le composant Conteneur ;

Cas mode Strict : Avec ce mode, CASi SR effectue les tâches suivantes :

- Réinitialisation des tampons d'E/S à *Absent*.
- Lecture des données depuis l'extérieur.
- Si CASi SR reçoit SynchronousMessage alors il place les data dans les tampons correspond, met à jour leurs états et répond ensuite *Ready* sinon il répond *NotReady*.

Au cours de la réaction : suivant le mode de fonctionnement, CASi SR réagit aux opérations $CASi.Read$ et $CASi.Write$ déclenchées par le composant Noyau comme suit :

- *Cas mode NonStrict* : CASi SR répond aux déclenchement des opérations $CASi.Read$ par la récupération des SynchronousMessage depuis les ports à travers les composants CS et CASe SR, puis la décapsulation des données qui sont délivrées ensuite au composant Noyau. De

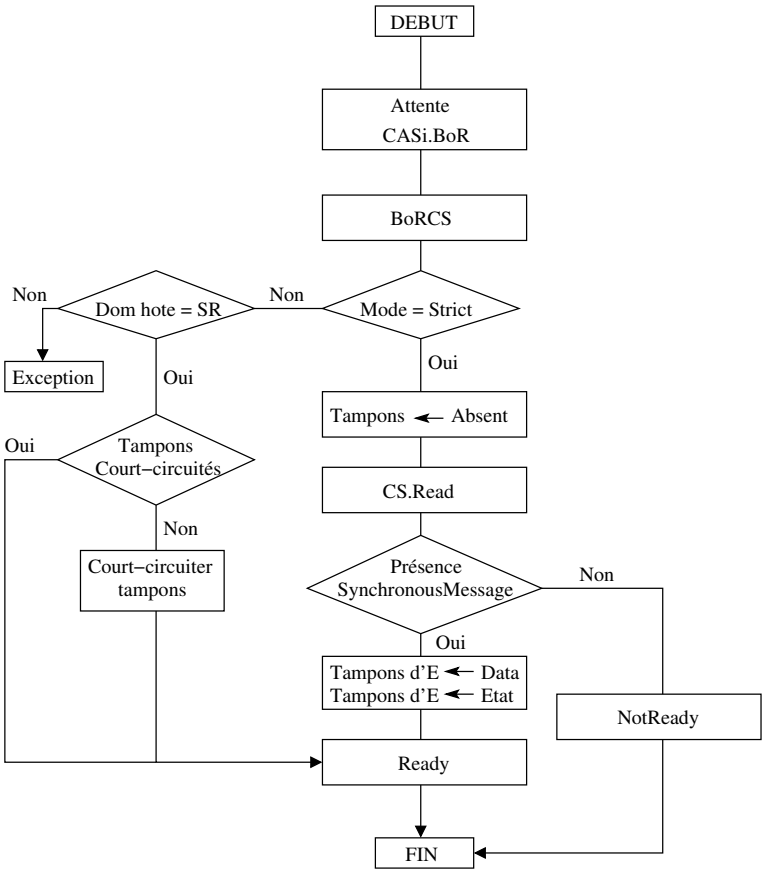


FIG. 6.4: Traitement exécuté par CASi SR avant réaction.

même pour les opérations $CASi.Write$, CASi SR encapsule les données venant du Noyau dans des SynchronousMessage puis les dépose sur les ports à travers les composants CS et CASe SR ;

– *Cas mode Strict* : CASi SR répond aux déclenchements des opérations $CASi.Read$ par transmission des données des tampons d'entrée ayant les champs états *Present* au composant Noyau. Par contre, il répond aux déclenchements des opérations $CASi.Write$ par le placement des données venant du Noyau dans les tampons de sortie tout en positionnant leurs états à *Present*. CASi SR ne permet la modification des tampons d'E/S qu'une seule fois au cours du même instant sinon il génère une exception.

Après la réaction : Une fois informé sur la fin de réaction du composant Noyau, CASi SR effectue les tâches suivantes :

– *Cas mode NonStrict* : Exécute l'opération de synchronisation $EoRCS$;

– *Cas mode Strict* : Encapsule d'abord les données des tampons de sortie ayant des états à *Present* dans des SynchronousMessage puis les envoie par l'opération $CS.Write$. Finalement, exécute l'opération de synchronisation $EoRCS$.

Partie d'adaptation à la sémantique SR : CASe SR La partie CASe SR ne pose pas des difficultés d'implémentation. Donc, le composant CASe SR réagit aux opérations de lecture, $CASe.Read$, d'abord par la lecture des ports, en utilisant $Port.Read$, puis il encapsule les éventuelles données lues dans des SynchronousMessages. Et, aux opérations d'écriture, $CASe.Write$, par la récupération des données encapsulées dans des SynchronousMessages venant de l'intérieur et puis, suivant leurs états, il les envoie vers l'extérieur par les opération d'écriture $Port.Write$. Notons qu'ici, le composant CASe SR n'effectue aucune opération de contrôle pour empêcher de changer l'état d'un même port plus d'une fois au cours du même instant. Cela est assuré par le domaine SR.

Spécification du composant CAS DE

Partie de conservation de la sémantique DE : CASi DE Une itération suivant le modèle de calcul DE consiste à traiter tous les événements ayant l'estampille de temps égale au temps courant. En effet, c'est à la base de cette définition que nous allons définir la structures du CASi DE et l'algorithme permettant à un composant Noyau DE de s'exécuter correctement.

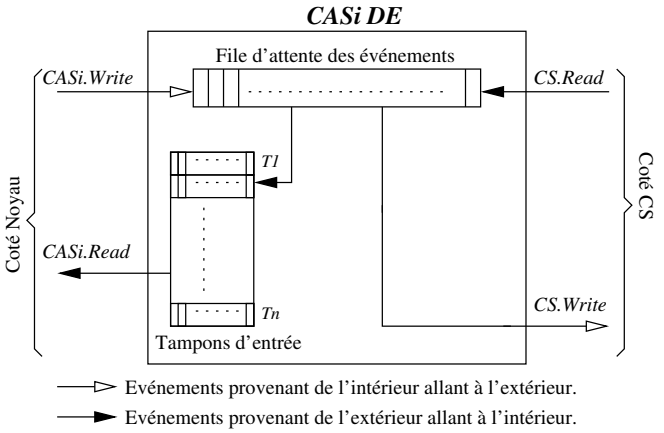


FIG. 6.5: Architecture du CASi DE.

La figure 6.5 montre la structure interne du composant CASi DE. Donc, le composant CASi DE reçoit des événements (entrant) du composant Noyau et du composant CS (événements entrants), et envoie des événements (sortant) à ces derniers. Cependant, pour la gestion des événements, nous avons défini une fille d'attente des événements et des tampons d'entrée, qui ont tous des tailles illimitées. En effet, tout événement (que ce soit entrant ou sortant) doit transiter par la file d'attente dans laquelle il sera stocké jusqu'à ce que son estampille de temps soit égale au temps courant pour être transféré par la suite aux tampons d'entrée ou vers le composant CS.

Avant de donner l'algorithme d'implémentation du modèle de calcul DE dans CASi DE, nous devons savoir que :

- Un événement est stocké dans la file d'attente avec les informations : *Direction* (sortant ou entrant), *Provenance/Destination* (mentionnant le nom du port), *estampille de temps* et la *donnée* ;
- Un événement venant du composant CS est encapsuler dans un message sous format (data, timestamp), avec data représente la donnée et timestamp représente la date relative de livraison de la donnée data au composant Noyau. Donc, la date absolue de livraison d'un événement est la somme de la date courante local et la date relative. Cet exigence est faite pour éviter d'avoir des événements dans le passé. Aussi, il est possible de recevoir une liste d'événements depuis le composant CS allant au même tampon d'entrée ;
- Un événement destiné au composant CS est encapsulé dans un message sous format (data, date), avec data représente une donnée et date représente la date absolue à laquelle la donnée data a quitté le composant CASi. Il est possible d'envoyer une liste d'événements au composant CS provenant du même tampon de sortie ;
- Un événement provenant du composant Noyau est sous format (data, timestamp), avec data représente la donnée produite par le composant Noyau et timestamp représente la date d'envoi relative de la donnée data à l'extérieur (au composant CS) ;
- Par contre, il n'y a que la donnée data qui sera délivrée au composant Noyau.

Une itération DE passe passe par trois étapes exécutées par CASi DE. Ces étapes sont : Avant, au cours et après la réaction du composant Noyau dont voici le détail :

Avant la réaction : Comme c'est montré par la figure 6.6, CASi DE attend le déclenchement de $CASi.BoR$ pour effectuer les tâches suivantes :
- Avancer le temps local : le temps courant local sera toujours positionné au temps courant externe car nous avons toujours le temps courant extérieur est inférieur ou égal à l'estampille de temps du prochain événement (entrant ou sortant) à traiter. Cela est dû à l'attribution des dates relatives aux données venant du composant CS et non pas absolues, ce qui évite d'avoir des événements dans le passé ;
- Récupérer d'éventuels DatedMessages à travers l'opération $CS.Read$;
- Décapsuler les données ainsi que leurs estampilles de temps. Créer les événements Events contenant les données et les estampilles de temps reçues en ajoutant à chacun d'eux les informations : *direction* = *entrant* et provenance = nom du port lu. Ensuite, enfiler Events ;
- Déposer les événements dans les tampons d'entrée : Si la file d'attente contient des événements ayant les dates égales au temps courant du CASi DE alors retirer ceux ayant direction égale à entrant, placer les dans les tampons d'entrée correspondant à provenance et répond par $Ready$. Si la file d'attente ne contient aucun événement ayant la date égale au temps courant, le composant CASi DE répond par $NotReady$;

Au cours de la réaction : Le composant Noyau consomme les données disponibles sur les tampons d'entrée à travers $CASi.Read$ et envoie les événements, qui les a produits, à travers $CASi.Write$. Quand le composant CASi DE reçoit les événements du composant Noyau, il crée les Events correspondants et les enfile dans la file d'attente des événements, et cela, après avoir ajouté à chacun d'eux les informations : *direction* = *sortant*, *destination* = *nom du port* de sortie correspond, *estampille de temps* et la *donnée* ;

Après la réaction : Comme c'est montré par la figure 6.7, CASi DE attend $CASi.EoR$ (fin de réaction du composant Noyau) pour effectuer les tâches suivantes :
- Envoyer les événements de sortie : Retirer de la file d'attente les événements ayant le champ direction égale sortant et l'estampille de temps égale au temps courant, encapsuler les données de ces événements et la date courante dans des DatedMessages. Envoyer ces derniers à travers $CS.Write$;
- Dans le cas où il existe des tampons d'entrée non vides (c'est-à-dire que le composant Noyau n'a pas consommé tous les événements d'entrée de la date courante) alors CASi DE demande, à travers $reTrigger \in CASi.Clbk$, au composant Conteneur de solliciter le domaine hôte de réitérer le composant domaine-polymorphe pour la date courante (rappelons qu'une itération DE peut se dérouler en plusieurs réitérations du composant domaine-polymorphe au même date). Ensuite, CASi DE exécute $EoRCS$ avant la fin de l'itération DE.

  Par contre, dans le cas où tous les tampons d'entrée sont vides, CASi DE procède comme suit :
  - Si la file d'attente est vide alors l'itération DE est terminée, CASi DE exécute $EoRCS$ et on arrête l'exécution ;
  - Si la file d'attente n'est pas vide alors CASi DE, à travers l'opération $reTriggerAt \in CASi.Clbk$, demande au composant Conteneur de programmer auprès du domaine hôte un lancement ($trigger$) à la date correspond à l'estampille de temps du prochain événement à traiter, ce qui fait qu'au prochain $trigger$, nous aurons forcément le temps global du domaine hôte égal à la date du prochain événement à traiter. Finalement, il exécute $EoRCS$ et l'itération DE se termine.

Important :
- Il est possible qu'un composant Noyau puisse demander de ne pas être lancé prochainement, dans tel cas, le composant CASi écarte ses événements entrant pour éviter le problème d'existence des événements dans le passé ;
- Dans le cas où un composant Noyau, après l'avoir lancé plusieurs fois, n'a pas consommé ses données d'entrée, nous devons fixer un nombre de fois de ses relancements afin d'éviter de

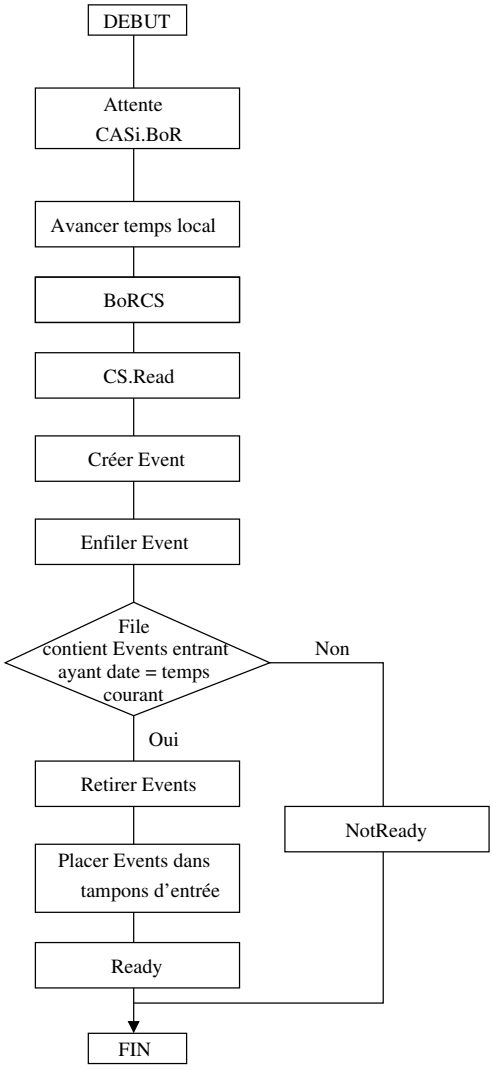


FIG. 6.6: Phase avant réaction exécutée par CASi DE.

boucler infiniment, ce qui empêche l'évolution du système.

Partie d'adaptation à la sémantique DE : CASe DE Par contre, dans le cas où le composant CAS est en externe (CASe), c'est-à-dire que le domaine hôte est un domaine DE, le respect de la sémantique du modèle de calcul DE n'est pas compliqué. En effet, le composant CASe a deux tâches principales :

- Lecture des ports d'entrée et l'encapsulation des données lues dans des DatedMessages ayant la date courante comme estampille de temps. Cette tâche est déclenchée par $CASe.Read$ Dans le cas de présence de plusieurs événements au même date et sur un même port, CASe DE prend ce cas en considération en lisant tous les événements et les encapsulant dans un message de type liste de messages dont chacun encapsule un événement ;
- L'écriture des données reçues du composant CS sous forme de DatedMessage sur les ports de sortie. Les données seront envoyées avec des retards égaux à leurs estampilles de temps indiquées dans les DatedMessages. Dans le cas où les estampilles de temps ne sont pas indiquées, les données ne seront pas retardées. De même, le cas de rafale d'événements est pris en compte par le composant CASe.

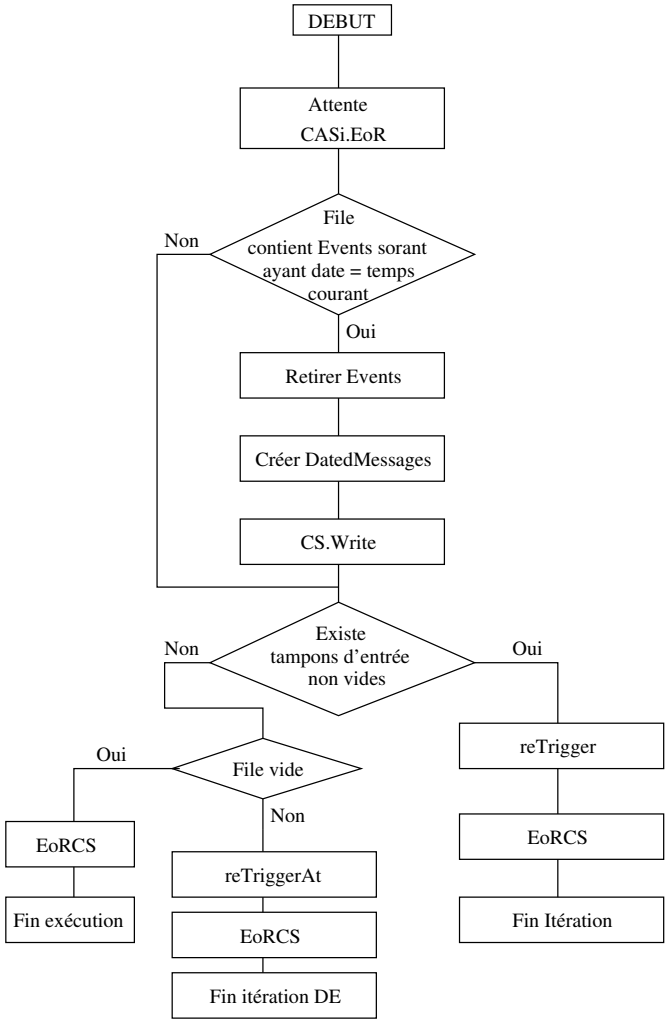


FIG. 6.7: Phase après réaction exécutée par CASi DE.

Spécification du composant CAS SDF

Partie de conservation de la sémantique SDF : CASi SDF Nous rappelons qu'avec le modèle de calcul SDF, un acteur doit consommer depuis chacune de ses entrées un nombre fixe de données et produire sur chacune de ses sortie un nombre fixe de données. Ce qui fait que l'implémentation de ce modèle de calcul dans CASi SDF est plus simple que celles de SR dans CASi SR et de DE dans CASi DE sur tout qu'il n'y a pas d'ordonnancement à faire. Donc, une itération SDF à l'intérieur du composant domaine-polymorphe implique que le composant Noyau doit consommer un nombre fixe de données et produire un nombre fixe de données.

De même, une itération SDF se fait en trois étapes exécutées par CASi SDF : avant, au cours et après la réaction du composant Noyau. Pour gérer le flot de données, CASi SDF comporte des tampons d'entrée et des tampons de sortie ayant des tailles illimitées, voir la figure 6.8.

Les tâches effectuées par CASi SDF tout au long des trois étapes sont données comme suit :

Avant la réaction : CASi SDF attend le déclenchement de l'opération $CASi.BoR$ pour effectuer les tâches suivantes :
- Récupérer des données encapsulées dans des DataMessages depuis le composant CS à travers $CS.Read$. Ici, il est possible, pour le même tampon d'entrée, de recevoir des rafales de données. Chacune des données reçues sera déposée dans le tampon d'entrée correspondant à sa provenance ;

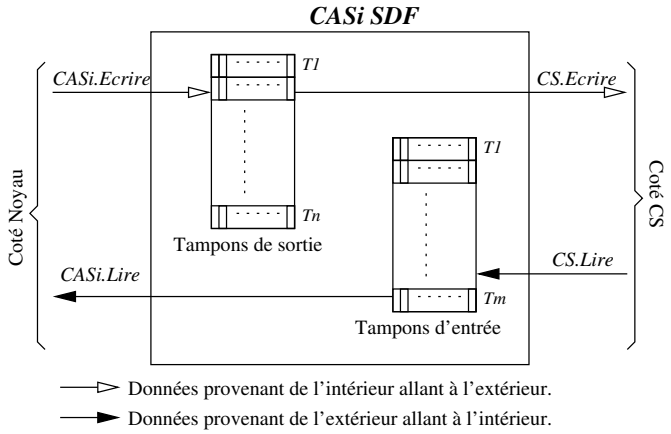


FIG. 6.8: Architecture du CASi SDF.

– Si le nombre de données présentes sur chaque tampon d'entrée est supérieur ou égal au nombre de données à consommer par le composant Noyau sur chacune de ses entrées alors CASi SDF répond par $Ready$ sinon il répond par $NotReady$ afin de permettre au composant Conteneur de lancer ou non la réaction du composant Noyau. $CASi.BoR$.

Au cours de la réaction : CASi SDF reçoit les données produites par le composant Noyau et les dépose dans les tampons de sortie correspondant ;

Après la réaction : CASi SDF attend le déclenchement de $CASi.EoR$ par le composant Conteneur. Ensuite, dans le cas où le composant Noyau a produit sur chaque tampon de sortie un nombre de données égal au nombre de données exigé, CASi SDF encapsule les données produites dans des DataMessages et les envoie à travers $CS.Write$. Dans le cas contraire, CASi SDF génère une exception.

Important La sémantique SDF conservée par le composant domaine-polymorphe est toujours embarquée dans une autre sémantique qui ne produira pas forcément le nombre de données exigé sur chaque port d'entrée. Dans telle situation, il sera impossible de lancer la réaction du composant Noyau. En effet, afin de permettre le lancement de la réaction de ce dernier, nous avons introduit la notion de *SDF relaxé par rapport au composant Noyau* : CASi SDF peut être *configuré en mode SDF relaxé*. Avec ce mode de fonctionnement, le composant CS, à travers l'opération $DefaultInputData$, récupère la donnée par défaut depuis le composant CF et la passera à CASi SDF. Ladite donnée par défaut sera utilisée par CASi SDF comme *donnée de bourrage* à la place des données manquantes afin d'avoir le nombre exigé de données à consommer sur chaque tampon d'entrée.

Partie d'adaptation à la sémantique SDF : CASe SDF Pour respecter la sémantique SDF, CASe SDF doit s'assurer que le nombre de données produites sur chaque port de sortie soit respecté. En effet, dans le cas où le nombre de données produites sur un port de sortie est inférieur à celui exigé, CASe SDF utilise la donnée par défaut afin de compléter dudit nombre de données pour atteindre le nombre de données exigé. Ainsi, la sémantique SDF sera respectée. La aussi, la donnée par défaut sert comme une donnée de bourrage et elle est récupérée par le composant CS depuis le composant CF, et cela, à travers l'opération $DefaultOutputData$.

Aussi, CASe SDF assure l'encapsulation/récupération des données dans/depuis les DataMessages.

6.3.2 Cas de mélange des modèles de calcul

Nous avons effectué deux conceptions différentes du système donné par la figure 6.2, voir figure 6.9. La première conception utilise les composants domaine-polymorphes pour faire face à l'hétérogénéité entre $D1$ et $D2$ du système. Donc, les modules A et C sont représentés par des acteurs atomiques et le module B par un composant domaine-polymorphe et les trois sont interconnectés et fonctionnent sous le modèle de calcul $D1$. En effet, le composant domaine-polymorphe utilise un composant CAS interne spécifique à $D2$ (CASi D2) et un composant CAS externe spécifique à $D1$ (CASe D1) pour fonctionner correctement sous $D1$ tout en exécutant son Noyau (représentant le module B) suivant le modèle de calcul $D2$. Par contre, la deuxième conception utilise l'approche hiérarchique pour faire coexister les modèles de calcul $D1$ et $D2$ au sein du même modèle. Donc, les modules A et C sont représentés par des acteurs atomiques tandis que le module B est représenté par un acteur atomique encapsulé dans un acteur composite CC. Ce dernier implémente le modèle de calcul $D2$ afin de faire fonctionner correctement son acteur atomique.

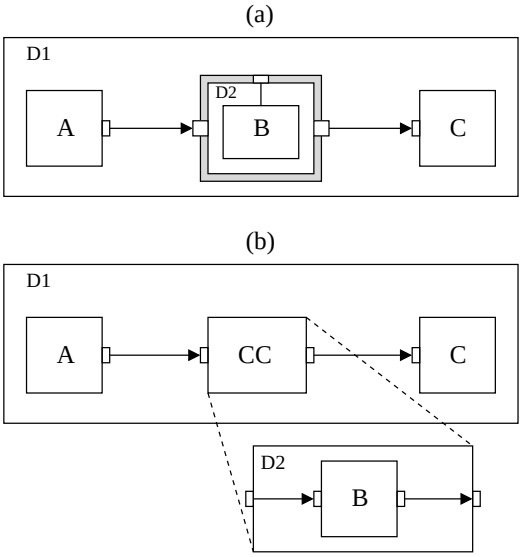


FIG. 6.9: Conception du système de la figure 6.2 (a) à base de CDP (b) suivant l'approche hiérarchique.

Afin d'étudier les interactions entre les deux sémantiques ($D1$ et $D2$) de la conception hétérogène à base de composant domaine-polymorphe par rapport à la conception hétérogène hiérarchique, nous avons considéré les cas suivants : $D1 = DE$ et $D2 = SDF$, $D1 = DE$ et $D2 = SR$, $D1 = SR$ et $D2 = SDF$, et $D1 = SR$ et $D2 = DE$.

SDF dans DE

Dans le cas où nous prenons $D1 = DE$ et $D2 = SDF$, le modèle de calcul SDF est à l'intérieur du modèle de calcul DE.

En effet, avec la conception hétérogène hiérarchique, l'acteur composite CC apparaît comme un block DE sans retard (retard toujours égal à zéro) car la sémantique SDF n'a pas de notion de temps et c'est la valeur zéro qui est, au niveau du passage entre les deux sémantiques, toujours associée aux données produites par l'acteur B comme estampille de temps par défaut. De plus, dans le cas où l'acteur B est configuré pour consommer un nombre de données supérieur à un depuis son port d'entrée, il sera impossible de le lancer tant qu'il n'y aura pas ce nombre d'événements présents sur le port d'entrée de l'acteur composite CC.

Par contre, avec la conception hétérogène à base de composant domaine-polymorphe, le composant domaine-polymorphe est bien vu comme un acteur atomique DE car il peut produire des événements avec retard. La valeur de ce dernier est laissée au choix du concepteur du système. De plus, dans le cas où le composant Noyau est configuré pour consommer un nombre de données supérieur à un, il est, avec le mode de fonctionnement *SDF relaxé*, possible de lancer sa réaction même s'il n'y a pas ce nombre d'événements sur le port d'entrée du composant domaine-polymorphe, en complétant par es données de bourrage fournies par le concepteur du système.

SR dans DE

La sémantique du modèle de calcul SR est presque identique à celle du modèle de calcul DE, sauf que la première a une notion de temps logique tandis que la deuxième a une notion de temps discret. En effet, avec la conception hétérogène hiérarchique, l'acteur composite CC apparaît comme un block DE sans retard (retard toujours égal à zéro) car la sémantique SR a une notion de temps logique. Donc, les données produites par l'acteur B seront toujours envoyées par l'acteur composite CC avec un retard égal toujours à zéro. De plus, la présence d'au moins d'un événement sur le port d'entrée de l'acteur composite CC impliquera systématiquement l'activation de l'acteur B.

Par contre, comme le cas précédent, avec la conception hétérogène à base de composants domaine-polymorphes, le composant domaine-polymorphe est bien vu comme un acteur atomique DE car il peut produire des événements avec retard. La valeur de ce dernier est laissée au choix du concepteur du système. De plus, la présence des événements sur le port d'entrée du composant domaine-polymorphe n'impliquera pas systématiquement le lancement du composant Noyau car cela dépend de la traduction des événements entrants au niveau du composant CF. Dans le cas où il n'y aura pas des données générées depuis desdits événements entrants, le composant CASi SR n'autorisera pas le lancement du composant Noyau.

SDF dans SR

Ce cas est identique au cas SDF dans DE sauf qu'il n'y a pas de notion de retard.

SR dans SDF

Dans tel cas, avec la conception hétérogène hiérarchique, l'acteur B doit toujours consommer des données depuis chacun de ses ports d'entrée car l'acteur composite CC ne peut être lancé tant qu'il n'a pas le nombre exigé de données sur son port d'entrée. Aussi, l'acteur composite CC doit toujours consommer une seule donnée depuis son port d'entrée et produire une donnée sur son port de sortie car l'acteur B (qui est un acteur SR) ne peut consommer plus d'une donnée par itération. De plus, le système peut rentrer dans l'*état d'interblocage* dans le cas où l'acteur B ne produira pas une donnée sur son port de sortie. Dans une telle situation, l'utilisation du modèle SR à l'intérieur du modèle de calcul SDF en utilisant l'approche hiérarchique devient impossible.

Par contre, avec la conception hétérogène basée composant domaine-polymorphe, le composant Noyau n'est pas obligé de toujours consommer des données à chaque activation du composant domaine-polymorphe, cela dépend de l'interprétation des données au niveau du composant CF. Aussi, le composant domaine-polymorphe peut être configuré pour consommer plus d'une donnée à la fois depuis son port d'entrée à condition que le concepteur du système doit, au niveau du composant CF, s'assurer qu'il génère au plus une donnée pour le tampons d'entrée du CASi SR depuis les données consommées. De plus, l'interblocage ne parviendra pas avec la conception hétérogène à base de composant domaine-polymorphes, car dans le cas où le composant Noyau ne produira pas de données de sortie, le composant CASe SDF utilisera les données de bourrage afin de respecter la sémantique SDF.

6.4 Conclusion

Dans ce chapitre, nous avons montré que le composant domaine-polymorphe est un composant hétérogène, ce qui nous a permis d'introduire une nouvelle approche d'hétérogénéité pour la conception des systèmes hétérogènes. Cette approche n'est que l'approche hétérogène atomique. Ensuite, nous avons donné la spécification du composant CAS (pour les cas en interne et en externe) pour les modèles de calcul SR, DE et SDF. Et finalement, nous avons, à travers des cas de mélange des modèles de calcul, montré les avantages d'utilisation de l'approche hétérogène atomique par rapport à l'utilisation de l'approche hétérogène hiérarchique.

Deuxième partie

Intégration, Validation et Simulation dans Ptolemy II

Chapitre 7

Intégration de l'Approche de Conception de Composant Domaine-Polymorphe dans PTOLEMY II

7.1 Introduction

Dans le chapitre 5, nous avons présenté notre approche de conception de composant domaine-polymorphe. Cette approche part de l'architecture abstraite (architecture initiale) commune aux composants domaine-spécifiques, dont les comportements internes sont spécifiés suivant une sémantique, jusqu'à l'architecture finale du composant domaine-polymorphe, et cela, après plusieurs étapes de conception. Notre approche de conception a permis la définition des architectures de composants intermédiaires, notamment le composant domaine-spécifique flexible.

Puisque la conception, par sa définition, consiste à aller jusqu'à l'implémentation (la réalisation), nous donnons dans ce chapitre l'intégration de notre approche de conception de composant domaine-polymorphe dans PTOLEMY II qui a été choisi dans le chapitre 2 comme plateforme pour l'intégration et la validation de nos concepts.

PTOLEMY II est entièrement programmé en Java, qui utilise un typage statique qui ne nous permet pas d'implémenter la politique de transformation du composant domaine-polymorphe. Or, nous avons dit dans la section « Evaluations »du chapitre 5 que lorsque la politique de transformation ne peut pas être implémentée, le composant domaine-spécifique flexible doit être pris comme composant domaine-polymorphe. C'est pourquoi nous avons choisi d'intégrer le composant domaine-spécifique flexible dans PTOLEMY II.

Dans ce chapitre, nous commençons d'abord par la présentation de PTOLEMY II. Ensuite, nous montrons comment les classes du composant domaine-polymorphe ont été intégrées dans l'architecture de PTOLEMY II. Et finalement, pour faciliter l'utilisation des composants domaine-polymorphes en mode graphique, nous donnons l'intégration des composants domaine-polymorphes dans l'outil Vergil [8][9][10], qui est l'interface graphique utilisateur (GUI) pour PTOLEMY II.

7.2 Aperçu sur PTOLEMY II

PTOLEMY II est une plateforme développée à l'université de Berkeley et basée sur la méthodologie acteur. Il permet la modélisation, la conception et la simulation des systèmes embarqués hétérogènes. PTOLEMY II est programmé en Java. Tous les composants dans PTOLEMY II sont représentés par des classes Java. Parmi ces composants, nous citons les acteurs, les ports, les relations, etc.

Comme c'est montré sur la figure 7.1, les modèles dans PTOLEMY II sont des graphes où les nœuds représentent des entités et les arcs sont des relations. Les entités sont des acteurs et les relations sont des liaisons de communication entre lesdits acteurs.

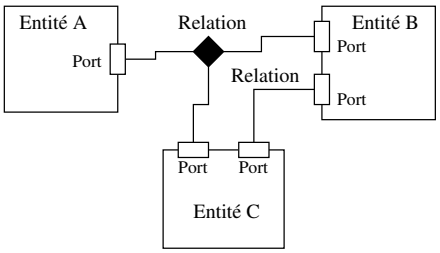


FIG. 7.1: Représentation graphique des modèles dans PTOLEMY II.

PTOLEMY II offre une infrastructure unifiée [8] permettant de supporter un nombre illimité de modèles de calcul, dont les implémentations sont appelées *domaines*.

7.2.1 Architecture

L'architecture globale de PTOLEMY II se compose d'un ensemble de packages qui fournissent l'appui générique pour tous les modèles de calcul et d'un ensemble de packages qui fournissent un appui plus spécifique pour des modèles de calcul particuliers. Tous les packages de PTOLEMY II sont regroupés sous un seul package nommé *ptolemy*. Voici quelques packages de PTOLEMY II :

ptolemy.kernel Ce package est un package générique qui fournit l'architecture logicielle pour les modèles sous PTOLEMY II. Il est vu comme la syntaxe abstraite des conceptions sous PTOLEMY II. Il définit les classes entités, ports et relations permettant de construire des graphes groupés. Aussi, il comporte d'autres sous packages. Le sous package *ptolemy.kernel.util* fournit un ensemble de classes utiles ne dépendant pas du package *ptolemy.kernel*.

ptolemy.actor Ce package regroupe les classes définissant des entités exécutables qui reçoivent et émettent des données à travers des ports. Il inclut la classe de base *Director* qui est étendue dans les sous packages du package *ptolemy.domains* pour contrôler l'exécution d'un modèle. Aussi, ce package regroupe d'autres sous packages. Par exemple, le sous package *ptolmy.actor.sched* inclut les classes permettant à la classe *Director* d'établir l'ordonnancement des acteurs de son modèle et le sous package *ptolemy.actor.corba* inclut les classes représentant des acteurs et l'infrastructure pour les modèles distribués utilisant CORBA.

ptolemy.domains Ce package comporte un ensemble de sous packages dont chacun est l'implémentation d'un modèle de calcul est appelé ainsi domaine. Généralement, ces sous packages contiennent un sous package *kernel*, qui définit des classes étendant celles dans le package *ptolemy.kernel* ou le package *ptolemy.actor*, et d'éventuel sous package *lib*, qui inclut les acteurs domaine-spécifiques. Les domaines définis dans PTOLEMY II sont Synchronous/Reactive (SR), Synchonous Dataflow (SDF), Discrete-Events (DE), Distributed Discrete Events (DDE), Discret Time (DT), Continuous Time (CT), Finite State Machines (FSM), Process Network (PN), Communicating Séquential Processes (CSP), Timed Multitasking (TM) et Giotto.

ptolemy.data Ce package inclut les classes permettant l'encapsulation et la manipulation des données communiquées entre les acteurs dans un modèle PTOLEMY II. La classe *Token* est la classe clé qui définit un ensemble de méthodes polymorphes permettant d'effectuer des opérations comme l'addition, la soustraction, etc. sur les tokens sans se préoccuper du type des données. Ce package, lui aussi, comporte un ensemble de sous packages. Par exemple, le sous package *ptolemy.data.type* inclut les classes et les interfaces définissant le système de type.

7.2.2 Acteur

La conception de PTOLEMY II est basée sur la méthodologie orientée acteur [62] [141] [199] [64]. Par conséquent, un acteur a une interface composée de ports d'entrée et de ports de sortie et des paramètres. Certains acteurs sont conçus pour être polymorphes de domaine (que nous avons appelés acteurs génériques afin de mettre en évidence leur différence par rapport aux composants domaine-polymorphes). Les acteurs génériques peuvent donc opérer dans divers domaines en accueillant les sémantiques de ces derniers avec les paramètres par défaut 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, la plupart des acteurs dans la bibliothèque d'acteurs héritent de ces classes de base.

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.

Interface

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. Voici la syntaxe de sa déclaration :

```
Public Type_du_Port Nom_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 :

```
Nom_du_Port = new Type_du_Port(this, "Nom_du_Port", boolean, boolean)
```

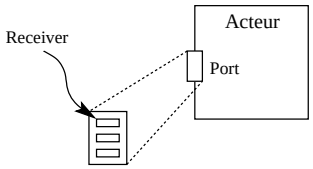


FIG. 7.2: Receivers dans PTOLEMY II.

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.

Comme c'est montré par la figure 7.2, les ports contient des receivers qui implémentent le mécanisme de communication. Chaque receiver est dédié à un canal de communication.

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 créé par la classe *IOPort* du package *ptolemy.actor* 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.

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

```
Public Parameter Nom_du_Parametre
```

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

```
Nom_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 [8], 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.

Opérations de communication

PTOLEMY II définit un ensemble d'opérations de communication permettant aux acteurs d'envoyer et de recevoir des données à travers leurs ports. Ces opérations sont traduites par les méthodes suivantes :

1. Nom_du_Port.send(Numero_du_canal, token) : C'est une méthode d'écriture permettant à un acteur d'envoyer un token à travers un canal d'un port.

2. Token token = Nom_du_Port.get(Numero_du_canal) : C'est une méthode de lecture permettant à un acteur de récupérer un token depuis un canal d'un port.
3. Boolean reponse_booleen = Nom_du_Port.hasToken(Numero_du_canal) : C'est une méthode de test de présence des tokens sur un canal d'un port d'entrée. Elle permet à un acteur de savoir s'il peut utiliser la méthode *Nom_du_port.get()* ou non.
4. Boolean reponse_booleen = Nom_du_Port.hasRoom(Numero_du_canal) : C'est une méthode de test d'existence de l'espace en réception d'un port de destination permettant à un acteur de savoir s'il pourra exécuter la méthode *Nom_du_Port.send(Numero_du_canal, token)* avec succès ou non.

Opérations de flot de contrôle

Dans Ptolemy II, les acteurs sont des entités exécutables. En effet, PTOLEMY II utilise des méthodes traduisant les opérations de flot de contrôle, qui ont été présentées dans la section méthodologie de conception orientée acteur du chapitre 2, comme mécanisme d'exécution des acteurs. Ces méthodes sont les méthodes d'initialisation *preinitialize* et *initialize*, les méthodes d'itération *prefire, fire* et *postfire* et la méthode de fin d'exécution *wrapup*.

L'ensemble de méthodes d'exécution est définit dans l'interface *Executable* du package *ptolemy.actor* et tous les acteurs doivent l'implémenter. Par contre, pour les acteurs l'implémentation de ces méthodes est faite via l'interface *Actor* qui, elle aussi, fait partie du package *ptolemy.actor*.

7.2.3 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, qui sont présentés dans la sous section suivante. 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, les 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.

7.2.4 Acteur 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.

Comme c'est montré par la figure 7.3, A0 est un acteur composite supérieur (top-level) ayant le directeur local D0 et contenant les acteurs A1, A2 et A3. A2 est, lui aussi, un acteur composite opaque car il a un directeur local, D1, tandis que D0 est son directeur exécutif. Par contre, l'acteur A3 est un acteur composite transparent car il n'a pas de directeur local, ainsi son port d'entrée, p31, est un port transparent. Cependant, l'acteur composite opaque supérieur n'a pas de directeur exécutif mais un manager. Ce dernier est présenté dans la sous section suivante.

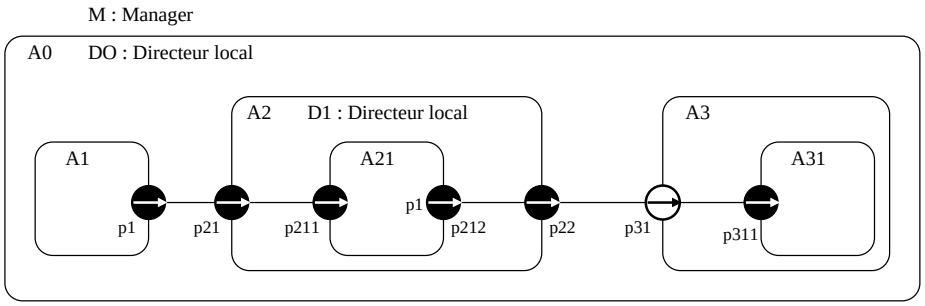


FIG. 7.3: Exemple de modèle PTOLEMY II montrant directeur, manager et acteurs atomique, composite opaque et transparent.

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 un acteur 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.

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.

7.2.5 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.

7.2.6 Exécution d'un acteur atomique

Le composant domaine-polymorphe est un acteur atomique. C'est pourquoi nous présentons dans cette sous section comment PTOLEMY II exécute un acteur. Cela nous permettra par la suite de bien expliquer comment nous avons intégré la phase d'exécution du composant domaine-polymorphe.

L'exécution de tout acteur dans un modèle PTOLEMY II passe par trois étapes : initialisation, plusieurs itérations et termine par l'affichage des résultats. L'étape d'initialisation est décomposé en deux méthodes *preinitialize()* et *initialize()*, l'étape itération comporte les méthodes, dites méthodes d'action, *prefire()*, *fire()* et *postfire()*, et l'étape d'affiche des résultats comporte la méthode *wrapup()*, voir figure 7.4. Les méthodes d'initialisation ne sont exécutées qu'une seule fois. Au cours d'une itération, la méthode *prefire()* est appelée une seule fois et si elle rend la valeur booléenne vraie, la méthode *fire()* sera appelée au moins une fois et la méthode *postfire()* une fois. Toutes ces méthodes, dites méthodes d'exécution, sont implémentées par l'acteur atomique. En effet, l'exécution de l'acteur atomique consiste à invoquer ses méthodes d'exécution par le directeur, dans le cas où il réside dans un acteur composite opaque, sinon par l'acteur composite, lui même, dans le cas où ce dernier est non opaque.

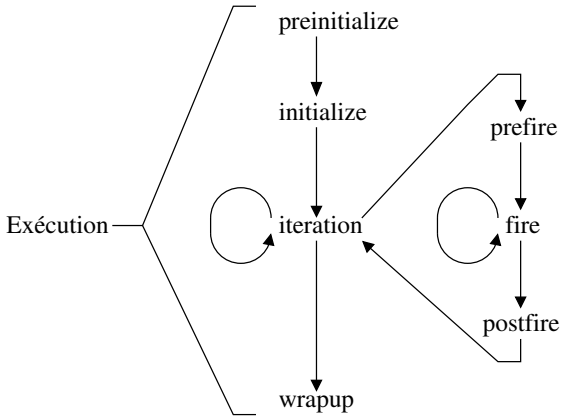


FIG. 7.4: Cycle d'exécution d'un acteur atomique dans PTOLEMY II.

Les tâches des méthodes d'exécution sont présentées comme suit :

preinitialize() : Création des receivers, validation des attributs de l'acteur et de ses ports.

initialize() : Production d'éventuels événements de sortie ou des événements programmés avant l'étape d'itération de l'acteur.

prefire() : Renvoie un booléen pour indiquer si l'acteur est prêt à être lancé ou non. Elle permet la vérification des conditions d'acceptation du lancement de l'acteur. Elle peut servir aussi à exécuter des opérations qui se produiront exactement une seule fois par itération.

fire() : Lecture des paramètres, des ports d'entrée. Exécution du traitement pour lequel l'acteur est créé. Production des données de sortie. Et comme elle peut être invoquée plusieurs fois au cours d'une même itération, cette méthode ne devrait pas changer l'état persistant de l'acteur.

postfire() : Mettre à jour de l'état de l'acteur. Rendre une valeur booléenne pour indiquer au directeur si l'acteur souhaite de se réitérer ou non. Si la valeur retournée est vraie, l'exécution de l'acteur est complète et le directeur ne le réitère pas prochainement.

wrapup() : Affichage des résultats. Elle est appelée une seule fois et à la fin de l'exécution de l'acteur ou lorsqu'une exception se produit.

Comme PTOLEMY II comporte des domaines temporisés et des domaines non temporisés, un acteur peut avoir un comportement qui dépend du temps courant du modèle. Pour cela, l'acteur doit

implémenter l'interface *TimedActor*, qui ne déclare aucune méthode. Un acteur implémentant cette interface fait signaler au directeur qu'il dépend du temps. Cependant, cette interface n'a aucune signification dans les domaines non temporisés. Un acteur peut accéder au temps courant du modèle à travers la méthode suivante :

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

Un acteur peut solliciter son directeur pour demander une invocation dans un temps futur en utilisant la méthode *fireAt()*, *fireAtRelativeTime()* et *fireAtCurrentTime()*. Les deux premières méthodes prennent comme argument l'acteur demandeur et une date tandis que la troisième méthode ne prend qu'un sel argument, qui est l'acteur demandeur. En effet, un acteur peut s'auto-programmer pour être lancé à une date dans le futur en plaçant un *événement pur* [1]

7.3 Intégration du composant domaine-polymorphe dans PTOLEMY II

Rappelons que d'une part PTOLEMY II est une plateforme entièrement programmée en Java et d'autre part la politique de transformation du composant domaine-polymorphe permet à ce dernier de changer sa structure ainsi que son comportement suivant les exigences du domaine dans lequel il est utilisé. Cependant, comme le langage Java est un langage à typage statique, la mise œuvre de la politique de transformation est impossible.

Cependant, nous avons montré dans la section évaluations du chapitre 5 que les avantages de l'utilisation du composant domaine-spécifique flexible dans la conception des systèmes se diffèrent légèrement par rapport à ceux de l'utilisation du composant domaine-polymorphe. Cette différence est au niveau du composant Conteneur. Donc, l'utilisation du premier composant nécessite la création d'un composant Conteneur pour chaque domaine.

Le composant domaine-spécifique flexible doit être considéré comme un composant domaine-polymorphe dans le cas où l'implémentation de la politique de transformation est impossible. En effet, c'est le composant domaine-spécifique flexible que nous avons implémenté.

7.3.1 Architecture de l'implémentation

L'implémentation du composant domaine-polymorphe est organisée en un ensemble de sous packages. Tous ces derniers sont regroupés dans un seul package, *ptolemy.polymorphcom*, voir la figure 7.5 page 143. Le rôle de chaque package est détaillé comme suit :

polymorphcom.containers : Ce package contient les classes définissant les composants Conteneurs où chacune d'elles est spécifique à un domaine. Afin de savoir pour quel domaine une classe conteneur est spécifique, le nom de cette dernière est composé du nom de son domaine et de la chaîne de caractères "Container".

polymorphcom.cas : Ce package contient des sous packages où chacun d'eux est dédié à un composant CAS. Il y'a autant de sous packages CASs que de domaines. Un sous package CAS contient donc la classe définissant le composant CAS.

polymorphcom.message : Ce package contient les classes permettant l'encapsulation de différents formats de données utilisées par les domaines.

polymorphcom.kernel : Ce package regroupe les interfaces contenant les méthodes traduisant les opérations d'exécution du composant domaine-polymorphe et les classes définissant les autres composants du composant domaine-polymorphe.

[1] Un événement pur est un événement qui ne comporte que la date.

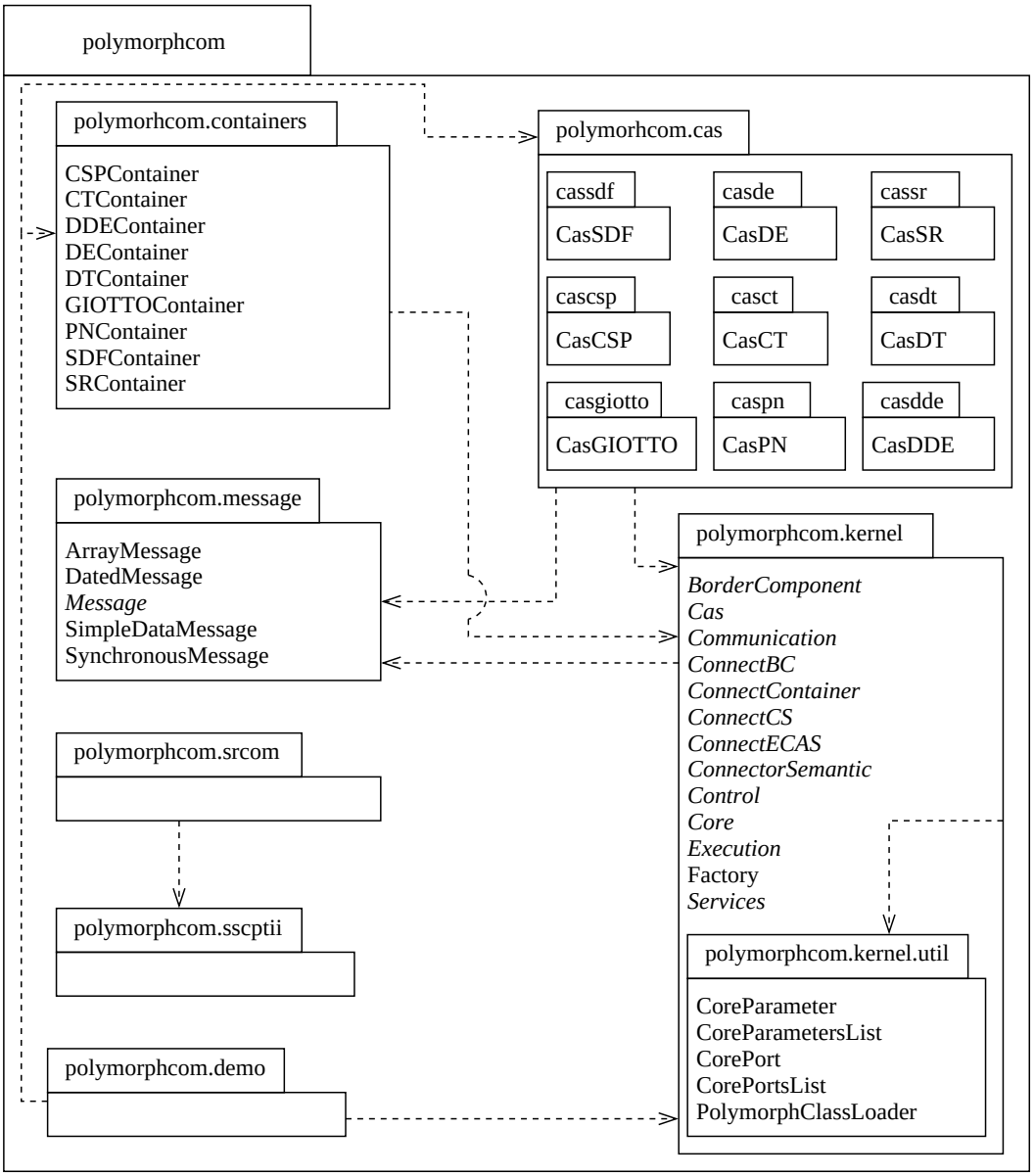


FIG. 7.5: Diagramme UML de package montrant l'architecture du composant domaine-polymorphe.

polymorphcom.srcom et polymorphcom.sscptii : Ces deux packages regroupent les classes du traducteur des modules Esterel en composants domaine-spécifiques synchrone ou en Noyau synchrone, voir annexe A. Le premier comporte les classes du traducteur tandis que le deuxième comporte les classe de son interface graphique.

polymorphcom.demo : Ce package regroupe les classes de l'exemple de validation de nos concepts.

Maintenant que nous avons présenté l'architecture d'implémentation du composant domaine-polymorphe dans PTOLEMY II et avant de décrire les classes de base, nous donnons dans la sous section suivante les appellations des terminologies de la partie théorique adoptées dans PTOLEMY II.

7.3.2 Appellations adoptées dans PTOLEMY II

Appellations des classes et interfaces

Comme le montre le diagramme UML de packages de la figure 7.5, nous avons gardé dans PTOLEMY II les mêmes appellations des classes et des interfaces du méta modèle du composant domaine-polymorphe donné par la figure 5.16 sauf pour la classe *Container* et l'interface *ConnectICAS*.

Puisque nous avons pris le composant domaine-spécifique flexible comme composant domaine-polymorphe, nous avons défini pour chaque domaine une classe *Container* dont l'appellation est la concaténation du nom du domaine avec la chaîne de caractères "Container". Par contre, pour séparer l'aspect communication de celui des services de rappel (call-back), nous avons décomposé l'interface *ConnectICAS* en deux interfaces, *Communication* et *Services*.

Appellations des opérations

Parmi les opérations d'exécution du composant domaine-polymorphe, il y'en a celles qui sont issues de la méthodologie acteur et celles qui sont introduites et définies par nous même.

Pour les opérations issues de la méthodologie acteur, qui sont définies par PTOLEMY II, nous utilisons les appellations proposées par PTOLEMY II. Par contre, pour les opérations que nous avons définies, nous utilisons dans PTOLEMY II presque les mêmes appellations données dans la partie théorique.

Puisque la phase d'initialisation dans PTOLEMY II est décomposée en deux méthodes, *preinitialize()* et *initialize()*, les opérations d'initialisation des composants Container, CAS, CS et BC présentées dans le chapitre 5 seront également décomposées en deux méthodes. De plus, au niveau d'implémentation, certaines opérations de contrôle ne seront pas explicitement intégrées dans PTOLEMY II et seront plutôt exprimées par des variables booléennes retournées par les méthodes d'exécution. L'opération *finish* est exprimée par des variables booléennes retournées par les méthodes *prefire()* et *postfire()*. En effet, nous avons exprimé aussi l'opération *finishReaction* par des variables booléennes retournées par les méthodes *pretreatment()* et *posttreatment()* et les opérations *Ready* et *NotReady* par une variable booléenne, retournée par la méthode beginOfReaction() du composant CAS interne, dont les valeurs vraie et faux désignent respectivement *Ready* et *NotReady*. Donc, ces opérations ne seront pas représentées.

Voici certaines appellations des opérations d'exécution du composant domaine-polymorphe utilisées dans PTOLEMY II :

Opérations de flot de contrôle

Appellations théoriques	$\longrightarrow$	***Appellations dans P*TOLEMY *II***
initialization	$\longrightarrow$	preinitialize & initialize

preCondition	⟶	prefire
trigger	⟶	fire
postCondition	⟶	postfire
InitCore	⟶	preconfiguration & configuration
preReaction	⟶	pretreatment
reaction	⟶	treatment
postReaction	⟶	posttreatment
InitCASi	⟶	preconfiguration & configuration
BoR	⟶	beginOfReaction
EoR	⟶	endOfReaction

Opérations de synchronisation

Appellations théoriques	⟶	*Appellations dans PTOLEMY II*
InitCS	⟶	preconfiguration & configuration
BoRCS	⟶	beginOfReactionCS
EoRCS	⟶	endOfReactionCS
InitCASe	⟶	preconfigurationECAS & configurationECAS
BoRCASe	⟶	beginOfReactionECAS
EoRCAS	⟶	endOfReactionECAS
InitCF	⟶	preconfiguration & configuration
BoRCF	⟶	beginOfReactionCF
EoRCF	⟶	endOfReactionCF

Opérations de communication

Appellations théoriques	⟶	*Appellations dans PTOLEMY II*
existData	⟶	hasToken
read	⟶	get
isFull	⟶	hasRoom
write	⟶	send

Pour le reste des opérations, nous avons gardé les mêmes appellations pour la plupart d'elles.

7.3.3 Description des classes de base

Classe Container

Comme nous avons implémenté pour chaque domaine une classe Container, nous ne pouvons pas présenter l'ensemble de ces classes. Cependant, et à titre illustratif, nous présentons la classe Container *DEContainer* du domaine DE (Discrete Event) dont le diagramme est donné sur la figure 7.6 page 146.

Nous signalons que toutes les classes conteneurs héritent, d'une façon directe ou indirect, de la classe *TypedAtomicActor*. En effet, les Containers sont des acteurs atomiques.

Comme c'est montré par la figure 7.6, la classe *DEContainer* hérite de la classe $DEActor$ afin d'avoir les spécificités d'un acteur spécifique au domaine DE et implémente l'interface *ConectContainer* pour la référencier.

Les attributs et les méthodes (méthodes d'exécution des acteurs non incluses) de la classe *DEContainer* sont présentés comme suit :

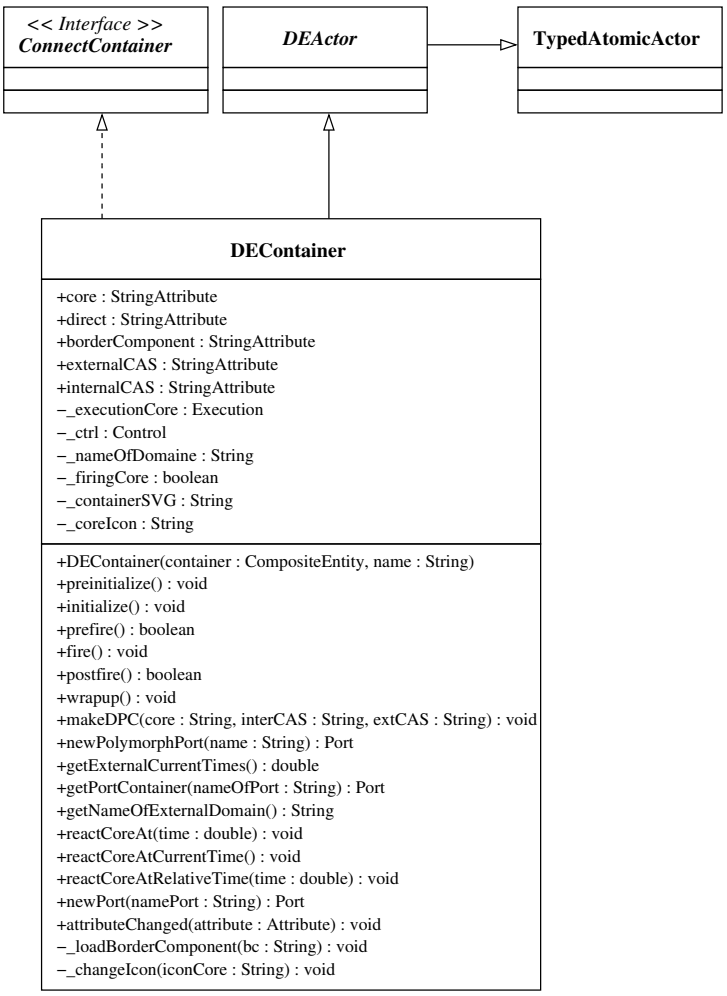


FIG. 7.6: La classe DEContainer.

Attributs

+core	:	Paramètre indiquant le composant Noyau à exécuter
+direct	:	Paramètre pour court-circuiter les composants CASi, CS, CF et CASe
+borderComponent	:	Paramètre indiquant le composant CF à utiliser
+externalCAS	:	Paramètre indiquant le composant CAS externe à utiliser
+internalCAS	:	Paramètre indiquant le composant CAS interne à utiliser
-_executionCore	:	Référence servant à exécuter le composant Noyau
-_ctrl	:	Référence sur le composant CAS interne servant d'exécuter $CASi.Ctrl$
-_nameOfDomaine	:	Nom du domaine hôte
-_firingCore	:	Indique si la réaction du composant Noyau doit être activée ou non
-_containerSVG	:	Décrit en format SVG l'effigie du Conteneur dans Vergil
-_coreIcon	:	Décrit en format SVG l'effigie du CDP (Container avec Noyau) dans Vergil

Méthodes

+DEContainer	:	Constructeur
+makeDPC	:	Le lanceur de la phase d'assemblage du CDP
+newPolymorphPort	:	Permet de créer puis de retourner un port
+getExternalCurrentTimes	:	Retourne le temps courant du domaine hôte
+getPortContainer	:	Permet de localiser un port et retourne sa référence
+getNameOfExternalDomain	:	Retourne le nom du domaine hôte
+reactCoreAt	:	Permet de programmer une future réaction du Noyau
+reactCoreAtCurrentTime	:	Permet de demander un lancement au temps courant
+reactCoreAtRelativeTime	:	Permet de programmer une future réaction du Noyau mais relativement au temps courant
+newPort	:	Crée les ports du CDP et qui sont spécifiques au domaine hôte
+attributeChanged	:	Permet de prendre en compte les changements des paramètres du Conteneur
-_changeIcon	:	Permet de changer l'effigie du CDP dans Vergil en fonction du chargement/déchargement du Noyau dans/du Conteneur

Classe Cas

La classe *Cas* est une classe abstraite, dont le diagramme est donné sur la figure 7.7 page 148. Elle implémente l'interface *ConnectECAS* pour permettre son utilisation en externe et les interfaces *Communication*, *Services* et *Control* pour pouvoir l'utiliser en interne. Elle contient des attributs et des méthodes dont la partie interne (CASi) et la partie externe (CASe) auront besoin. En effet, toute classe CAS à spécifier pour une sémantique donnée doit obligatoirement spécialiser la classe *Cas*.

Attributs

#containerCas	:	Indique le Conteneur d'appartenance
#connectorSemantics	:	Référence sur le composant ConnectorSemantics
#portsList	:	Indique les ports du CDP ainsi que leurs caractéristiques (en entrée, en sortie, etc.)
#reactionCore	:	Indique si le Noyau a demandé une future réaction ou non
#semanticsOfCas	:	Indique le nom de la sémantique du composant CAS

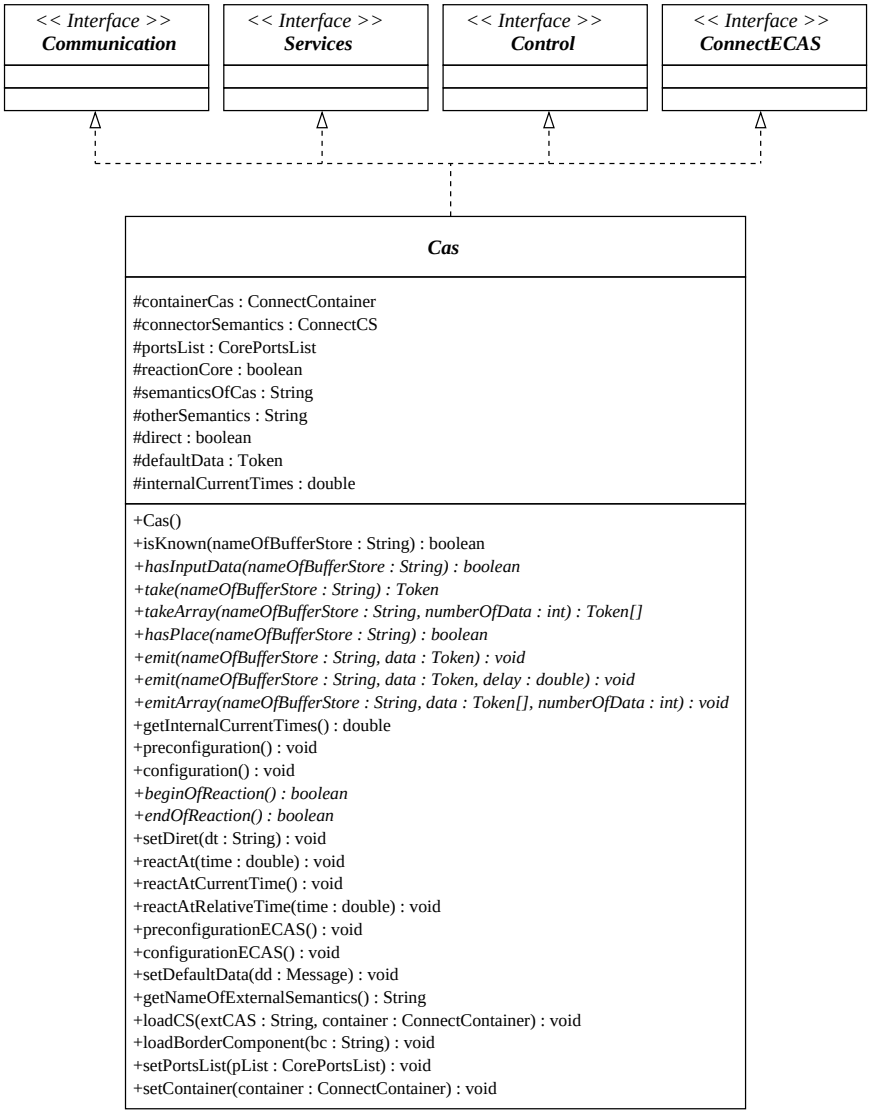


FIG. 7.7: La classe abstraite Cas.

#otherSemantics	:	Indique le nom de la sémantique du composant CAS de l'autre coté
#direct	:	Indique si le composant CAS est court-circuité ou non
#defaultData	:	Indique la donnée de bourrage
#internalCurrentTimes	:	Indique le temps courant de la sémantique interne

Méthodes

+Cas	:	Constructeur
+isKnown	:	Permet de savoir si l'état d'une entrée (buffer) est connue ou non (utilisé pour SR)
+hasInputData	:	Permet de savoir si une entrée a une donnée disponible
+take	:	Pour récupérer une(des) donnée(s) depuis une entrée
+takeArray	:	Pour récupérer une suite de données depuis une entrée
+hasPlace	:	Permet de savoir si une sortie est libre ou non
+emit	:	Permet d'envoyer une donnée avec ou sans retard sur une sortie
+emitArray	:	Permet d'envoyer une suite de données sur une sortie

+getInternalCurrentTimes	:	Pour récupérer le temps courant du modèle de calcul interne
+preconfiguration	:	Pour l'initialisation
+configuration	:	Pour l'initialisation
+beginOfReaction	:	Pour mettre au courant le CAS (quand il est utilisé en interne) avant la réaction du Noyau
+endOfReaction	:	Pour mettre au courant le CAS (quand il est utilisé en interne) après la réaction du Noyau
+setDirect	:	Pour positionner l'attribut en mode court-circuit ou mode non court-circuit
+reactAt	:	Permet au Noyau de programmer une future réaction
+reactAtCurrentTimes	:	Permet au Noyau de demander le lancement de sa réaction au temps courant
+reactAtRelativeTimes	:	Permet au Noyau de programmer une future réaction mais à une date relative au temps courant
+preconfigurationECAS	:	Permet l'initialisation du composant CAS externe
+configurationECAS	:	Permet l'initialisation du composant CAS externe
+setDefaultData	:	Pour positionner la valeur de l'attribut defaultData
+getNameOfExternalSemantics	:	Retourne le nom du domaine hôte
+loadCS	:	Permet l'instanciation puis le référencement du composant *ConnectorSemantics*
+loadBorderComponent	:	Même tâche que la méthode précédente mais pour le composant *BorderComponent*
+setPortsList	:	Pour diffuser aux composants la liste des ports à créer pour le Noyau.
+setContainer	:	Permet au composant Cas de référencier le composant Conteneur

Classe Core

La classe *Core* est la classe mère de tous les composants Noyaux. Son diagramme est donné sur le figure 7.8 page 151. Elle implémente l'interface *Execution*.

Ses attributs et ses méthodes sont présentés comme suit :

Attributs

#_com	:	Référence sur le composant Cas
#services	:	Référence sur le composant Cas sert à invoquer les méthodes traduisant les opérations $\Phi Noyau.Clbk$
#_containerCore	:	Référence sur le Conteneur dans lequel le composant Noyau est encapsulé
#portsList	:	Liste des ports du CDP

Méthodes

+Core	:	Constructeur
+preconfiguration	:	Permet l'initialisation du Noyau
+configuration	:	Permet l'initialisation du Noyau
+pretreatment	:	Retourne un booléen pour accepter ou non d'être lancé
+treatment	:	C'est la réaction du Noyau
+posttreatment	:	Retourne un booléen pour accepter ou non d'être lancé
+putFinalData	:	Pour rendre et/ou afficher les résultats

+getCoreIcon	:	Retourne l'effigie du CDP (Conteneur + Noyau) sous format SVG
+getCoreInputOutputPorts	:	Retourne les noms et les propriétés des ports à créer pour le Noyau
+broadcastPortsList	:	Diffuser la liste des ports aux autre composants du CDP
+createParameters	:	Créer les paramètres demandés par le Noyau
+loadInternalExternalCASs	:	Pour instancier les composants CAS interne et CAS externe
+loadBorderComponent	:	Pour lancer l'instanciation du composant BorderComponent
+setContainer	:	Positionner l'attribut _containerCore
+getControl	:	Retourne la référence du composant CAS interne

Classe ConnectorSemantics

La classe *ConnectorSemantics* implémente l'interface *ConnectCS* afin de définir les méthodes utilisées par la classe Cas en mode interne. Son diagramme est donnée sur la figure 7.9 page 151.

Attributs

_borderComponent	:	Référence sur le composant BorderComponent
_eCas	:	Référence sur le composant CAS externe
_nameOfInternalSemantics	:	Indique le nom de la sémantique interne

Méthodes

+ConnectorSemantics	:	Constructeur
+isPortKnown	:	Test si l'état d'une entrée est connue ou non
+hasPortData	:	Test la présence des données d'entrée
+getDataPort	:	Retourne les données d'entrée
+takeArray	:	Retourne une suite de données d'entrée
+hasPortPlace	:	Test la disponibilité de place sur un port de sortie
+emitDataPort	:	Envoyer des données de sortie
+emitArray	:	Envoyer une suite de données de sortie
+sendMessage	:	Pour envoyer des messages
+getMessage	:	pour récupérer des messages
+preconfiguration	:	Permet l'initialisation du composant ConnectorSemantics
+configuration	:	Permet l'initialisation du composant ConnectorSemantics
+beginOfReactionCS	:	Met au courant le composant ConnectorSemantics avant la réaction du Noyau
+endOfReactionCS	:	Met au courant le composant ConnectorSemantics après la réaction du Noyau
+loadExternalCAS	:	Permet l'instanciation du composant CAS externe
+loadBorderComponent	:	Permet l'instanciation du composant BorderComponent
+setPortsList	:	Permet de diffuser la liste des ports du CDP aux composant CAS externe et BC

Classe BorderComponent (CF)

La classe *BorderComponent* est une classe abstraite qui implémente l'interface *ConnectBC*. Son diagramme est donné sur la figure 7.10 page 153. Cette classe définit des attributs permettant aux

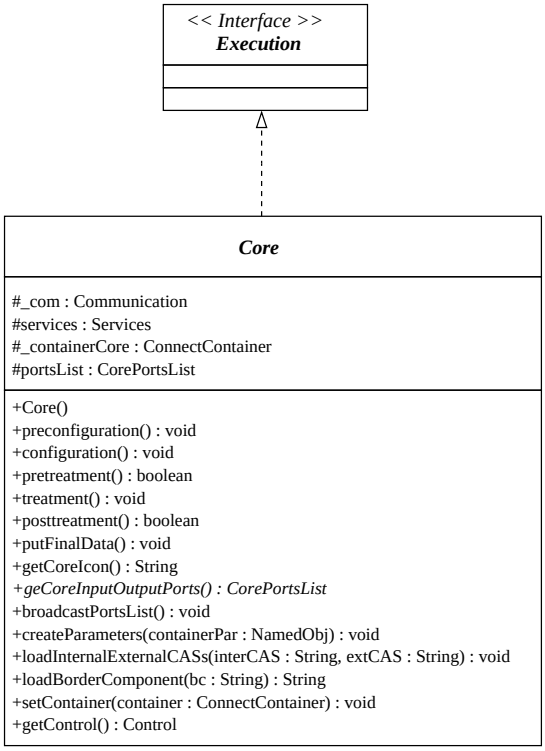


FIG. 7.8: La classe abstraite Core.

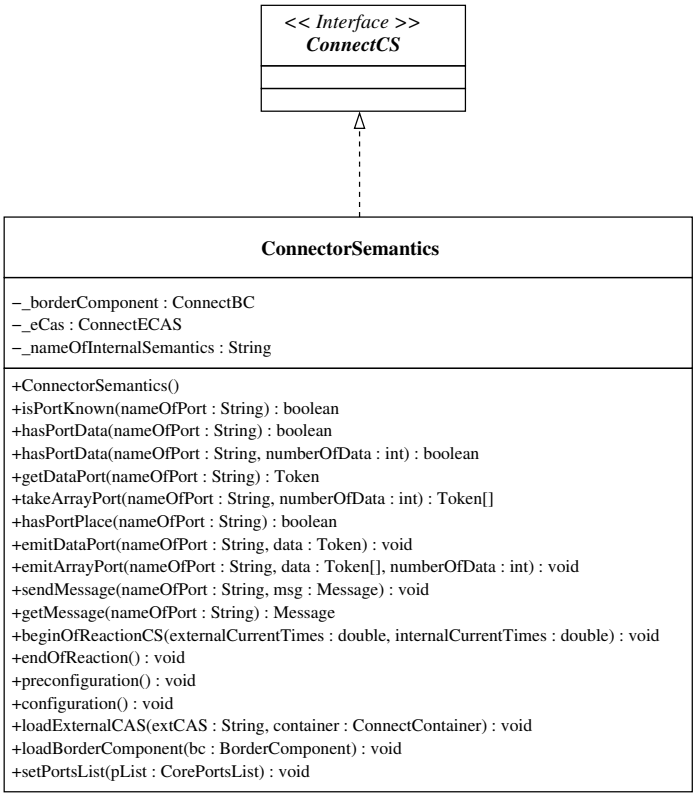


FIG. 7.9: La classe ConnectorSemantics.

concepteurs des systèmes d'avoir des informations sur les noms des sémantiques (interne et externe) et les temps courants des deux sémantiques afin de mieux expliciter le passage des données. Les comportements des méthodes de passage ne sont pas définis, c'est aux concepteurs des systèmes de les définir car cela fait partie de la conception des systèmes.

Attributs

#nameOfExternalSemantics	:	Indique le nom de la sémantique externe
#nameOfInternalSemantics	:	Indique le nom de la sémantique interne
#eCas	:	Référence sur le composant CAS externe
#externalCurrentTimes	:	Indique le temps courant de la sémantique externe
#internalCurrentTimes	:	Indique le temps courant de la sémantique interne
#portsList	:	Indique la liste des ports du CDP

Méthodes

+BorderComponent	:	Constructeur
+getDefaultInternalData	:	Permet de fournir à CAS interne la donnée de bourrage
+getDefaultExternalData	:	Permet de fournir à CAS externe la donnée de bourrage
+getInternalDataFromExternalData	:	Produit des données interne depuis les données externe
+getExternalDataFromInternalData	:	Produit des données externe depuis les données interne
+isInputPort	:	Indique si un port est un port d'entrée
+isOutputPort	:	Indique si un port est un port de sortie
+isInputOutputPort	:	Indique si un port est un port d'entrée et de sortie
+setNamesOfSemantics	:	Fournit au BorderComponent les noms des sémantiques (interne et externe)
+preconfiguration	:	Permet l'initialisation du BorderComponent
+configuration	:	Permet l'initialisation du BorderComponent
+beginOfReactionBC	:	Met au courant le BorderComponent avant la réaction du Noyau
+endOfReactionBC	:	Met au courant le BorderComponent après la réaction du Noyau
+setPortsList	:	Fournit au BorderComponent la liste des ports du CDP

7.3.4 Intégration de la phase d'exécution du CDP dans l'itération

Nous avons vu que l'exécution d'un acteur dans PTOLEMY II se résume par le lancement de ses méthodes d'itération *prefire()*, *fire()* et *postfire()* par son directeur.

En effet, une fois sa méthode *prefire()* est lancé par le directeur, le Conteneur met au courant le composant CASi avant la réaction du Noyau à travers la méthode *beginOfReaction()*. Cette dernière lui rend une valeur booléen, qui exprime l'opération *Ready/NotReady*, indiquant l'autorisation ou non de la réaction du Noyau. Si la valeur est vraie, le Conteneur lance la méthode *pretreatment()* du Noyau et signale au directeur qu'il accepte d'être lancé sinon il refuse d'être lancé, voir le code de la méthode *prefire()* sur le listing 7.1.

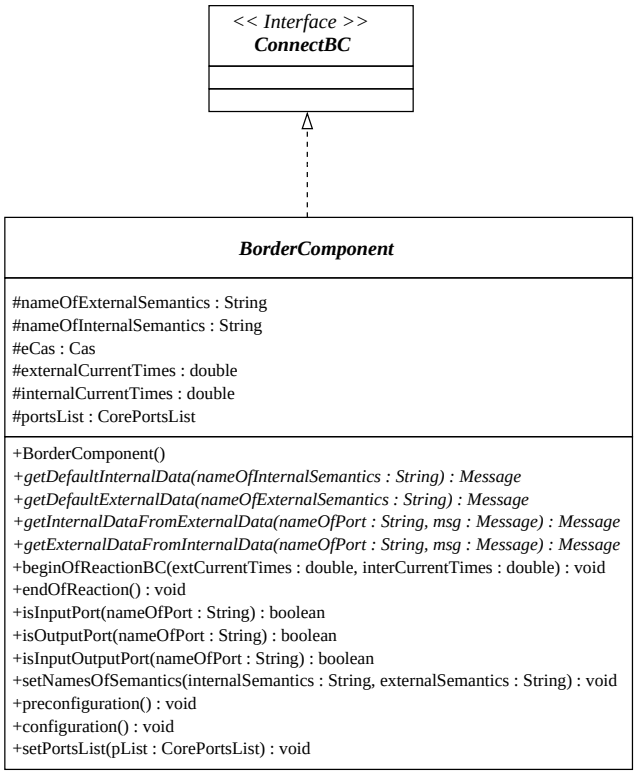


FIG. 7.10: La classe abstraite BorderComponent.

Listing 7.1: Méthode prefire du Conteneur

```
public boolean prefire() throws IllegalActionException {
  boolean react = true;
  _firingCore = _ctrl.beginOfReaction();
  if(_firingCore)
    react = _executionCore.pretreatment();
  return (super.prefire() && react);
}
```

Dans la méthode *fire()*, le composant Container ne lancera la méthode *treatment()* du composant Noyau que lorsque ce que dernier soit autorisé à être lancé, voir le code de la méthode *fire()* sur le listing 7.2.

Listing 7.2: Méthode fire du Conteneur

```
public void fire() throws IllegalActionException {
  if(_firingCore)
    _executionCore.treatment();
}
```

De même, dans la méthode *postfire()*, le Conteneur lance la méthode *posttreatment()* du Noyau et par la suite met au courant le composant CAS interne sur la fin de la réaction du Noyau, voir le code relatif à la méthode *postfire()* donnée par le listing 7.3.

Listing 7.3: Méthode postfire du Conteneur

```
public boolean postfire() throws IllegalActionException {
  boolean react = true;
  if(_firingCore)
    react = _executionCore.posttreatment();
  _ctrl.endOfReaction();
  return (super.postfire() && react);
}
```

7.4 Intégration du composant domaine-polymorphes dans Vergil

La construction des modèles dans PTOLEMY II peut se faire de deux façons. La première façon consiste à écrire tout un programme Java commençant par l'instanciation des acteurs et leurs paramétrisations jusqu'à leur interconnexion et la mise en place du Manager et les Directors. Pour cela, PTOLEMY II fournit des classes permettant une telle construction. Cette façon de faire est trop lourde sur tout lorsqu'il s'agit d'un modèle comportant un nombre important d'acteurs, et pour exécuter les modèles, le concepteur doit incorporer sa construction dans une application ou une applette Java. Ainsi, toute modification du modèle construit impliquera la recompilation de l'application ou l'applette. C'est pourquoi l'équipe de PTOLEMY II a proposé la deuxième façon de construction des modèles. Cette façon est basée sur l'outil Vergil [8][9][10].

Dans cette section nous donnons d'abord un bref aperçu sur Vergil et ensuite nous montrons comment nous avons intégré le modèle de composant domaine-polymorphe dans Vergil.

7.4.1 Aperçu sur Vergil

Vergil [8][9][10] est un framework permettant la construction des modèles à base de blocks. Autrement dit, il permet l'assemblage des composants logiciels. Il est l'interface graphique utilisateur (GUI) pour PTOLEMY II. La figure 7.11 montre un simple modèle PTOLEMY II dans la fenêtre Graph Editor, qui est un parmi plusieurs éditeurs dans Vergil. Cet éditeur comporte trois zones. La première zone se située à gauche affichant une arborescence d'acteurs, qui représente une librairie d'acteurs prédéfinis et peuvent être utilisés dans la construction des modèles. La deuxième zone se située à droite et est la scène de construction des modèles. Et, la dernière zone se située à gauche en bas servant de naviguer les modèles dont la taille dépasse celle de la zone de construction.

Pour construire un modèle à l'aide de Vergil, il suffit de cliquer puis déplacer les acteurs de la librairie vers la scène de conception, et pour interconnecter deux ports il suffit de cliquer sur le port source et puis déplacer sans lâcher le bouton de la souris vers le port cible, ainsi une liaison entre les deux ports apparaîtra. Aussi, en cliquant sur le bouton droit de la souris, Vergil permet la mise à jour des paramètres des acteurs, des ports, etc.

Vergil mémorise les modèles construits dans des fichier MoML (Modeling Markup Language) [8] [9] [10], qui est un langage basé sur XML [200] respectant une DTD (Document Type Declaration) définie par l'équipe de PTOLEMY II. Donc, dans un fichier MoML, on trouve la description de toutes les entités (acteurs, ports, liens, etc.) avec leurs paramètres ainsi que leurs valeurs d'initialisation. En effet, la reconstruction des modèles se fait automatiquement par Vergil en se basant sur les informations fournies par les fichiers MoML. De plus, Vergil met à jour le fichier MoML d'un modèle au fure et à mesure de la modification du modèle par le concepteur.

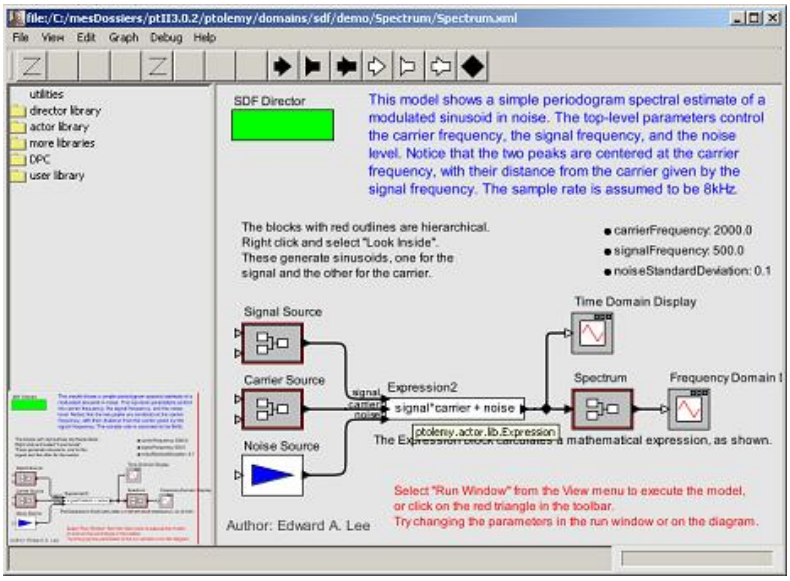


FIG. 7.11: Exemple de modèle dans la fenêtre de conception de PTOLEMY II sous Vergil.

7.4.2 Intégration des composants domaine-polymorphes dans la librairie d'acteurs de Vergil

Pour rendre possible l'utilisation d'un acteur dans la construction des modèles sous Vergil, il faut que cet acteur apparaisse dans la librairie d'acteurs se trouvant dans la zone gauche de Graph Editor.

Pour localiser et puis charger les acteurs de la librairie, Vergil se base sur le fichier MoML *defaultFullConfiguration.xml* dans lequel il trouve les chemins (path) et les noms des classes des acteurs ainsi que les autres entités. Une fois chargés par Vergil dans Graph Editor, les acteurs de cette librairie sont regroupés dans des répertoires suivant leurs fonctionnalités et chacun d'eux est représenté par une icône, qui est son effigie.

Toute entité (acteur, port, etc.) dans PTOLEMY II a un attribut prédéfini, appelé *_iconDescription*, qui décrit son icône en SVG [2] [201]. En effet, c'est à partir de cette description que Vergil puisse dessiner l'effigie d'une entité dans Graph Editor.

Pour permettre l'utilisation des composants domaine-polymorphes sous Vergil, nous avons juste inséré dans *defaultFullConfiguration.xml* des balises XML indiquant le chemin du fichier MoML, containers.xml, qui à son tour indique les chemins et les noms des classes conteneurs, voir le code donné par le listing 7.4. Donc, une fois chargés par Vergil, les conteneurs sont regroupés dans le sous répertoire containers du répertoire DPC. Ce dernier est placé dans la librairie d'acteurs. Ainsi, nous avons rendu leur utilisation très aisée. Pour cela, il suffit juste de cliquer sur un conteneur, le glisser vers la zone de construction des modèles, voir figure 7.12, et finalement éditer ses paramètres.

[2] SVG (Scalable Vector Graphic) permet la description des dessins vectoriels en format XML

Listing 7.4: Balises XML montrant l'intégration des conteneurs dans Vergil

```
<!— All components used by the Domain–Polymorph Component —>
<class name="DPC" extends="ptolemy.moml.EntityLibrary">
  <configure>
    <?moml
      <group>
        <!— Containers —>
        <input source="/polymorphcom/containers/containers.xml"/>
      </group>
    ?>
  </configure>
</class>
```

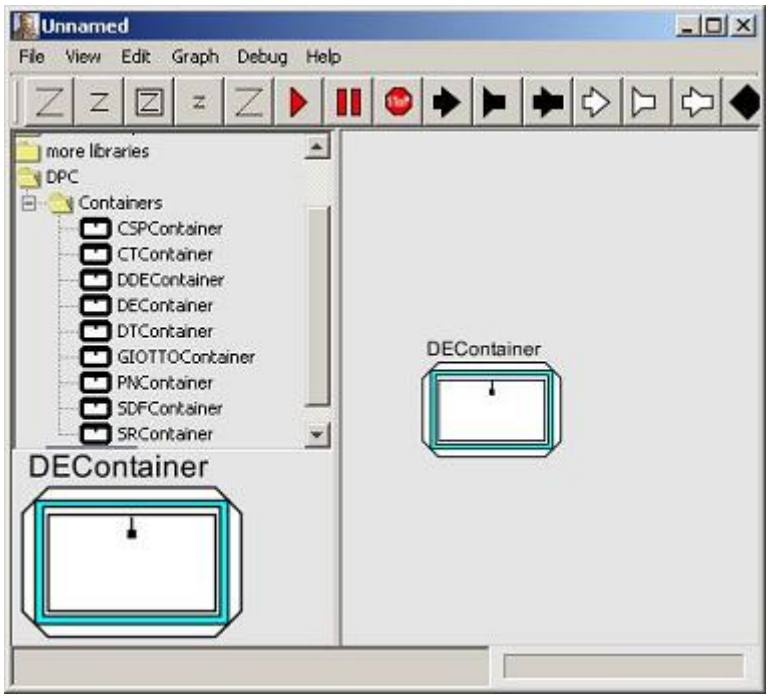


FIG. 7.12: Les Containers dans Vergil.

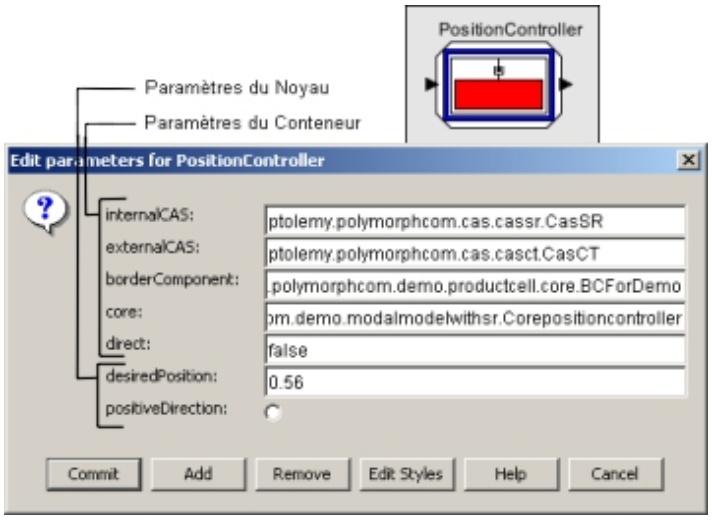


FIG. 7.13: Edition des paramètres d'un composant domaine-polymorphe dans Vergil.

La figure 7.13 montre un exemple d'utilisation des composants domaine-polymorphes. Cet exemple montre un composant domaine-polymorphe, PositionController, dont le composant Noyau a été spécifié suivant la sémantique synchrone, issu de la traduction d'un module Esterel, et ayant la tâche de contrôler si un mobile a atteint une position donnée ou non. Donc, pour avoir le composant PositionController, nous avons choisi un conteneur et ensuite édité ses paramètres afin de lui passer les chemins et les noms des classes CasSR, CasCT, BCForDemo et Corepositioncontroller correspondent respectivement aux composants CAS interne, CAS externe, BorderComponent et Core. L'édition des paramètres du conteneur déclenchera l'exécution de la méthode *attributeChanged()*, c'est ainsi dans cette méthode que nous avons choisi le déclenchement de la phase d'assemblage afin de prendre en compte tout changement des valeurs des paramètres.

Après la phase d'assemblage où nous voyons l'apparition de la forme complète du composant domaine-polymorphe, les paramètres du composant Noyau sont créés et deviennent visible sur la fenêtre d'édition des paramètres, voir la figure 7.13.

7.5 Conclusion

Dans ce chapitre, nous avons intégré notre approche de conception de composants domaine-polymorphes dans la plateforme PTOLEMY II.

Par contre, la difficulté rencontrée dans cette intégration est l'impossibilité d'implémenter la politique de transformation et qui est causée par le typage statique du langage Java avec lequel la plateforme PTOLEMY II est programmée. Cela nous a obligé de prendre le composant domaine-spécifique flexible comme composant domaine-polymorphe à intégrer dans PTOLEMY II.

En effet, nous avons d'abord présenté PTOLEMY II pour éclaircir comment les acteurs sont exécutés, ce qui nous a permis par la suite de montrer aisément comment notre approche de conception de composant domaine-polymorphe a été intégrée dans PTOLEMY II.

Finalement, comme Vergil est un outil offrant une interface graphique permettant la construction et la simulation des modèles PTOLEMY II avec une grande facilité et rapidité, nous avons aussi intégré les composants domaine-polymorphes dans celui-ci. Ainsi, son (CDP) utilisation est très aisée et n'est basée que sur des actions du genre cliquer, glisser et éditer des paramètres.

Chapitre 8

Exemple de Conception à base de Composants Domaine-Polymorphes

8.1 Introduction

Dans le but de valider par simulation nos concepts, nous présentons dans ce chapitre la conception et la simulation basées composants domaine-polymorphes de l'atelier de production automatisé proposé par FZI[1].

L'atelier de production automatisé est un système hétérogène qui mélange le contrôle et la dynamique continue, et comporte des contrôleurs et des capteurs. Les contrôleurs sont responsables sur son bon fonctionnement et les capteurs ont le rôle d'informer les contrôleurs sur l'état de l'atelier.

Afin d'utiliser des comportements synchrones dans la conception du modèle de l'atelier sous Ptolemy II, nous avons d'abord spécifié les contrôleurs et les capteurs suivant la sémantique synchrone, en utilisant le langage Esterel, et ensuite, à partir des modules Esterel codants les contrôleurs et les capteurs, nous avons généré par notre traducteur les composants Noyaux synchrones pour les composants domaine-polymorphes.

Dans le modèle Ptolemy II de l'atelier, les composants domaine-polymorphes représentants les contrôleurs sont utilisés sous le modèle de calcul DE tandis que les composants domaine-polymorphes représentants les capteurs sont utilisés sous le modèle de calcul CT.

Finalement, nous avons validé les composants domaine-polymorphes par la simulation du fonctionnement de l'atelier en trois dimensions.

8.2 Atelier de production automatisé

L'atelier de production [203] donné par la figure 8.1 présente un problème industriel réel où les exigences de sûreté jouent un rôle signifiant. Cet atelier est composé de six dispositifs : deux convoyeurs (FB et DB), une table rotatrive élévatrice (ERT), une grue (TC), une presse (Press) et un robot (Robot) équipé de deux bras (Arms).

La tâche de chaque dispositif est plus détaillée dans la section 8.3. Cependant, nous expliquons brièvement le fonctionnement de l'atelier. Donc, une fois la pièce est déposée sur le convoyeur d'alimentation (Feed Belt), ce dernier la déplace vers la table qui, par la suite, la déplace à une position permettant au robot de la récupérer par son premier bras (Arm1). Ensuite, le robot dépose la pièce

[1] Forschungszentrum Informatik de Karlsruhe

dans la presse pour la presser. Une fois la pièce est préssée, la presse la déplace à une position permettant au robot de la récupérer par son deuxième bras (Arm2). Finalement, le robot dépose la pièce sur le convoyeur de dépot (Deposit Belt). Ce dernier déplace la pièce à la position permettant à la grue (Crane) de le décharger. Pour avoir un comportement cyclique, la grue réintroduit de nouveau la pièce pressée dans le cycle d'usinage, en la déposant sur le convoyeur d'alimentation.

Pour un bon fonctionnement, l'atelier est doté d'une partie de contrôleur permettant de commander les différents dispositifs et d'enchaîner les différentes opérations sans violer les contraintes de sûreté.

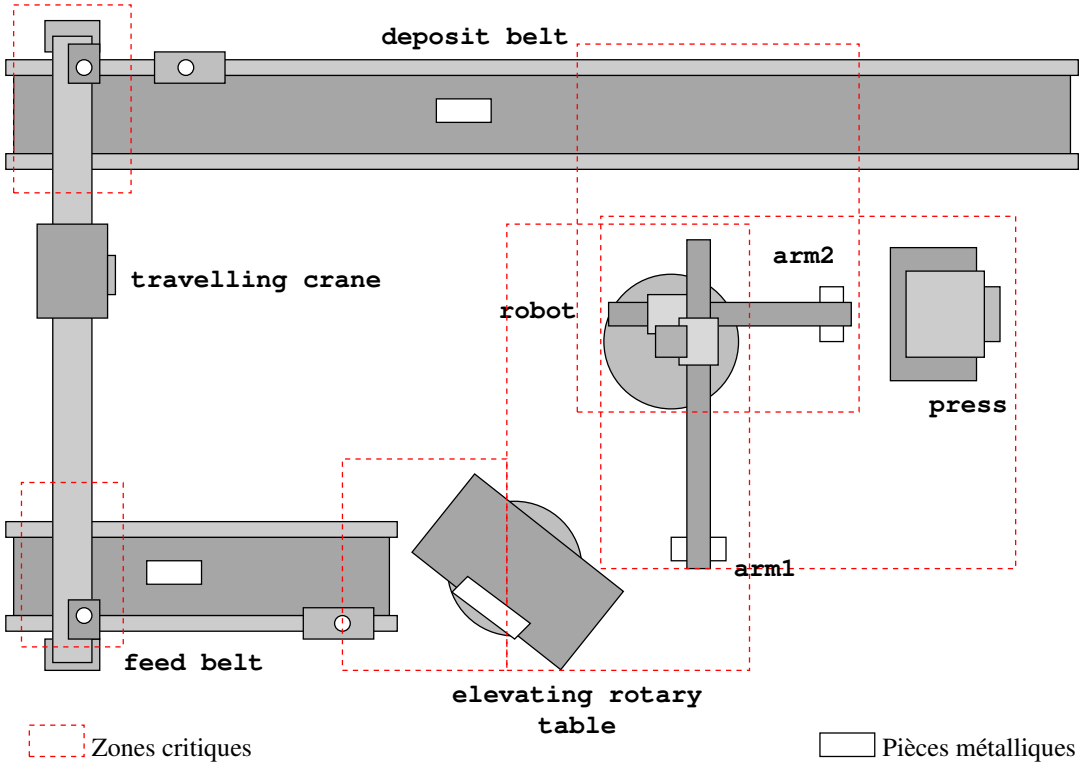


FIG. 8.1: Vue globale de l'atelier avec le découpage en zones critiques.

Tout au long de son cycle d'usinage, une pièce passe toujours par deux types de zones : *Zones Critiques* et *Zones Non Critiques*. Dans une Zone Non Critique, les problèmes de sûreté ne se posent pas et la pièce circule normal, ce qui explique l'absence des capteurs dans ces zones. Par contre, dans une Zone Critique, les propriétés de sûreté doivent être fermement respectées ainsi tout lancement d'action par la partie contrôle doit être précédé par la satisfaction de toutes les propriétés de sûreté concernées. Dans les Zones Critiques, il doit y'avoir une synchronisation entre les dispositifs pour faire passer une pièce d'un dispositif à un autre sans commettre des dégâts.

Sur la figure 8.1, nous avons encadré les zones critiques par des rectangles rouge à trait discret. En effet, nous comptons six zones :

Zone TC-FB : Elle se situe entre la grue et le convoyeur d'alimentation ;
Zone FB-ERT : Elle se situe entre le convoyeur d'alimentation et la table ;
Zone ERT-Robot : Elle se situe entre la table et le robot ;
Zone Robot-Press : Elle se situe entre le robot et la presse ;
Zone Robot-DB : Elle se situe entre le robot et le convoyeur de dépôt ;
Zone DB-TC : Elle se situe entre le convoyeur de dépôt et la grue.

L'atelier de production est vu comme un système hétérogène comportant le contrôle et la dynamique continue. Ainsi, sa conception impliquera l'utilisation de plusieurs modèles de calcul.

Avant de donner la conception et la simulation de l'atelier sous Ptolemy II, nous devons d'abord spécifier en Esterel les contrôleurs et les capteurs de l'atelier. C'est ce que nous présenterons dans les deux sections qui suivent.

8.3 Spécification des contrôleurs de l'atelier suivant la sémantique synchrone

8.3.1 Identification des contrôleurs

Nous avons décomposé la partie contrôle de l'atelier en sept contrôleurs dont les noms sont : FBController, ERTController, RobotController, PressController, DBController, TCController et ProductCellController. Chacun des six premiers contrôleurs est associé à un dispositifs afin d'assurer le bon déroulement de la tâche effectuée par dudit dispositif. Tandis que le dernier est vu comme un Scheduler. La responsabilité de ce dernier est d'établir l'ordre total de fonctionnement de l'atelier (cycle d'usinage) et d'assurer la synchronisation entre les six contrôleurs des dispositifs. Nous avons choisi cette décomposition pour permettre la réutilisabilité de chacun des dispositifs indépendamment des autres ainsi tout réaménagement de l'atelier (c'est-à-dire la modification du cycle d'usinage) n'impliquera pas la mise à jour des contrôleurs des dispositifs.

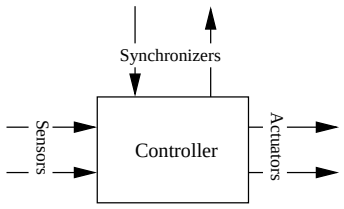


FIG. 8.2: Forme d'un contrôleur

Chacun des contrôleurs des six dispositifs est vu comme une boite noire qui a des signaux d'entrée et des signaux de sortie, voir figure 8.2. L'ensemble de signaux comporte trois types de signaux : les signaux capteurs (sensors), les signaux déclencheurs (actuators) et les signaux synchroniseurs (synchronizers). Les capteurs permettent au contrôleur d'avoir des informations sur l'état du dispositif qui le contrôle, les déclencheurs permettent au contrôleur de commander le dispositif et finalement les synchroniseurs permettent au contrôleur de communiquer avec le contrôleur principal de l'atelier pour transférer une pièce en toute sûreté au dispositif voisin.

Nous avons pris le langage Esterel pour spécifier les comportements des contrôleurs suivant la sémantique synchrone.

Pour la spécification des comportements des contrôleurs des dispositifs, nous nous basons sur les descriptions des tâches des dispositifs données par [204], les détails et les appellations utilisées par [205]. Par contre, pour la spécification du comportement du contrôleur ProductCellController, nous nous sommes basés sur le cycle d'usinage effectué par l'atelier.

8.3.2 Spécification des contrôleurs

Contrôleur du convoyeur d'alimentation

«...The task of the feed belt consists in transporting metal blanks to the elevating rotary table. The belt is powered by an electric motor, which can be started up or stopped by the control program. A photoelectric cell is installed at the end of the belt ; it indicates whether a blank has entered or left

the final part of the belt. . . . the photoelectric cells switch on when a plate intercepts the light ray. Just after the plate has completely passed through it, the light barrier switches off. At this precise moment, the plate . . . has just left the belt to land on the elevating rotary table–provided of course that the latter machine is correctly positioned . . . the feed belt may only convey a blank through its light barrier, if the table is in loading position . . . do not put blanks on the table, if it is already loaded . . . [204]»

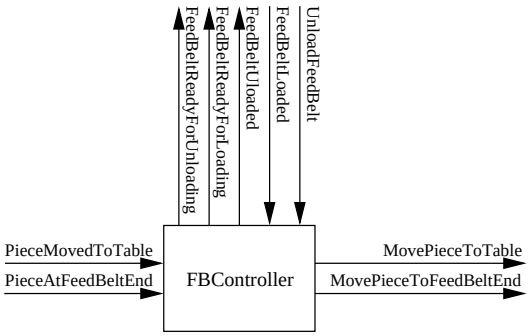


FIG. 8.3: Contrôleur du convoyeur d'alimentation.

D'après la description de la tâche du convoyeur d'alimentation FB, la mission du contrôleur FBController (cf. figure 8.3) est d'assurer l'acheminement des pièces métalliques déposées sur le convoyeur FB jusqu'à la table ERT. En effet, une fois informé par le contrôleur ProductCellController à travers le signal FeedBeltLoaded que la pièce est déposée sur le convoyeur FB, le contrôleur FBController, à travers le signal MovePieceToFeedBeltEnd, met en marche le convoyeur FB afin d'acheminer la pièce au bout final du convoyeur FB. L'arrivée de la pièce au bout final du convoyeur FB est détectée par la cellule photoélectrique et nous supposons que cela provoquera l'arrêt automatique du convoyeur FB et permettra au capteur PieceAtFeedBeltEnd d'informer le contrôleur FBController qui à son tour informe le contrôleur ProductCellController à travers le signal FeedBeltReadyForUnloading que le convoyeur FB est prêt pour le déchargement. A cette instant, si le signal UnloadFeedBelt a été déjà envoyé (présent) par ProductCellController, le contrôleur FBController transfert la pièce à la table ERT en mettant de nouveau en marche le convoyeur FB par l'envoi du signal MovePieceToTable, sinon il se mettra en attente jusqu'à la présence du signal UnloadFeedBelt. Ensuite, une fois la pièce est transférée à la table ERT, le contrôleur FBController reçoit le capteur PieceMovedToTable et informe par la suite le contrôleur ProductCellController qu'il est déchargé et prêt au chargement à travers les signaux FeedBeltUnloaded et FeedBeltReadyForLoading. Finalement, le contrôleur FBController revient à l'attente du signal *FeedBeltLoaded* pour répéter sa tâche de contrôle au profile de la pièce suivante.

En effet, la tâche du contrôleur FBController est spécifiée en Esterel comme suit :

```
1  module FBController:
2    % The Controller of the FB
3
4    % input signals
5    input UnloadFeedBelt;       % from ProductCellController
6    input FeedBeltLoaded;       % from ProductCellController
7    input PieceMovedToTable;    % from FB
8    input PieceAtFeedBeltEnd;   % from FB
9
10   % output signals
11   output FeedBeltReadyForLoading;     % to ProductCellController
12   output FeedBeltReadyForUnloading;   % to ProductCellController
13   output FeedBeltUnloaded;            % to ProductCellController
14   output MovePieceToTable;            % to FB
15   output MovePieceToFeedBeltEnd;      % to FB
16
17   loop
18     present FeedBeltLoaded then
19       emit MovePieceToFeedBeltEnd
20     end present;
```

```
21
22      present PieceAtFeedBeltEnd then
23        emit FeedBeltReadyForUnloading
24      end present;
25
26      present UnloadFeedBelt then
27        emit MovePieceToTable
28      end present;
29
30      present PieceMovedToTable then
31        [
32          emit FeedBeltReadyForLoading
33        ||
34          emit FeedBeltUnloaded
35        ]
36      end present;
37      await tick;
38    end loop
39  end module
```

Contrôleur de la table

«... The task of the elevating rotary table is to rotate the blanks by about 45 degrees and to lift them to a level where they can be picked up by the first robot arm. The vertical movement is necessary because the robot arm is located at a different level than the feed belt and because it cannot perform vertical translations. The rotation of the table is also required, because the arm's gripper is not rotary and is therefore unable to place the metal plates into the press in a straight position by itself. [204]»

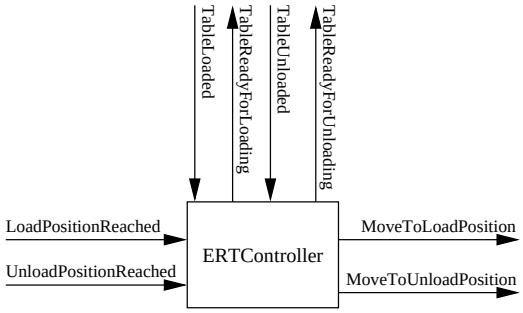


FIG. 8.4: Contrôleur de la table élevatrice rotatrice (ERT).

La tâche du contrôleur ERTController (cf. figure 8.4) est de faire amener la pièce métallique de la position où elle a été chargée par le convoyeur FB à une position permettant au bras Arm1 du Robot de la récupérer, c'est-à-dire que la table se déplace d'une façon répétitive de la position de chargement à la position de déchargement et vice versa. Donc, après la réception du signal TableLoaded venant du contrôleur ProductCellController, le contrôleur ERTController met en marche la table par l'envoi du signal MoveToUnloadPosition pour la déplacer à la position de déchargement. Pour cela, nous supposons que l'atteinte de la position de déchargement provoquera l'arrêt automatique et immédiat de la table, cela permettra à cette dernière d'envoyer le signal UnloadPositionReached au contrôleur ERTController qui, à son tour, envoie le signal *TableReadyForUnloading* au contrôleur ProductCellController pour l'informer que la table est prête au déchargement. A ce niveau, le contrôleur ERTController se met en attente jusqu'à la présence du signal TableUnloaded de ProductCellController pour pouvoir par la suite de mettre en marche la table par l'envoi du signal MoveToLoadPosition afin qu'elle se déplace à la position de chargement. Une fois arrivée à ladite position de déchargement, nous supposons que la table s'arrête automatiquement et provoquera l'envoi du signal LoadPositionReached au contrôleur ERTController. Finalement, ce dernier envoie le signal TableReadyForLoading au contrôleur ProductCellController pour l'informer que la table est prête au le chargement de la pièce métallique suivante. Ainsi, ce processus se répète de nouveau.

Nous avons spécifiée le comportement du contrôleur ERTController en Esterel comme suit :

```
1  module ERTController:
2    % The Controller of the ERT
3
4    % input signals
5    input TableUnloaded;           % from ProductCellController
6    input TableLoaded;             % from ProductCellController
7    input LoadPositionReached;     % from ERT
8    input UnloadPositionReached;   % from ERT
9
10   % output signals
11   output TableReadyForUnloading;  % to ProductCellController
12   output TableReadyForLoading;    % to ProductCellController
13   output MoveToLoadPosition;      % to ERT
14   output MoveToUnloadPosition;    % to ERT
15
16   loop
17     present TableLoaded then
18       emit MoveToUnloadPosition
19     end present;
20
21     present UnloadPositionReached then
22       emit TableReadyForUnloading
23     end present;
24
25     present TableUnloaded then
26       emit MoveToLoadPosition
27     end present;
28
29     present LoadPositionReached then
30       emit TableReadyForLoading
31     end present;
32
33     await tick
34
35   end loop
36
37 end module
```

Contrôleur du robot

«...The robot comprises two orthogonal arms. For technical reasons, the arms are set at two different levels. Each arm can retract or extend horizontally. Both arms rotate jointly. Mobility on the horizontal plane is necessary, since elevating rotary table, press, and deposit belt are all placed at different distances from the robot's turning center. The end of each robot arm is fitted with an electromagnet that allows the arm to pick up metal plates. The robot's task consists in : taking metal blanks from the elevating rotary table to the press ; transporting forged plates from the press to the deposit belt. [204]»

D'après la description ci-dessus, la tâche du robot consiste à répéter le cycle suivant :

1. Décharger la table élévatrice rotatrice (ERT) et la presse ;
2. Effectue une rotation d'un angle α° pour aller à la position de chargement ;
3. Charger la presse et le convoyeur de dépôt ;
4. Effectuer une rotation d'un angle $-\alpha^{\circ}$ pour revenir à la position de déchargement.

Le rôle du contrôleur RobotController (cf. figure 8.5) est d'assurer l'exécution correcte de la tâche suscitée par le robot d'une façon répétitive. En effet, au début, le robot est à la position de déchargement avec le premier bras Arm1 orienté vers la table et le deuxième bras Arm2 orienté vers la presse et les deux bras sont en état rétracté. Donc, une fois le signal LoadRobot est envoyé par le contrôleur ProductCellController, le contrôleur RobotController envoie parallèlement les signaux Arm1Extend et Arm2Extend au robot pour étendre les bras du robot jusqu'à ce que Arm1 (resp.

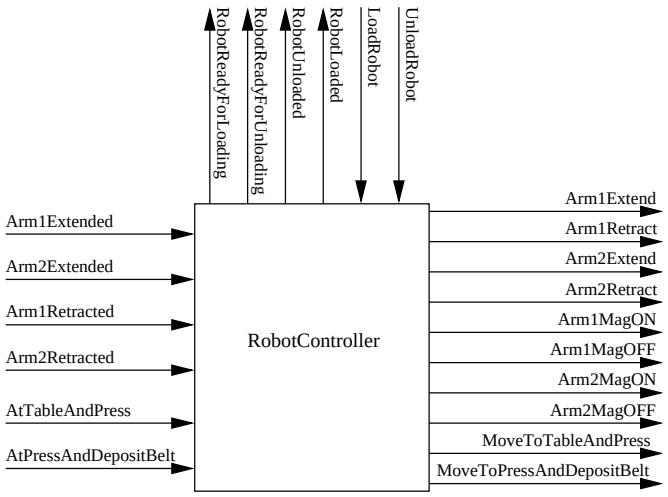


FIG. 8.5: Contrôleur du robot.

Arm2) soit au dessus de la pièce posée sur la table (resp. presse). Une fois que les deux bras soient complètement étendus, le contrôleur RobotController est informé par les signaux Arm1Extended et Arm2Extended, ce qui lui permet de magnétiser les têtes des bras du robot par l'envoi des signaux Arm1MagON et Arm2MagON ainsi de récupérer les pièces posées sur la table et la presse. Ensuite, après avoir envoyé le signal RobotLoaded au contrôleur ProductCellController pour l'informer que la table et la presse ont été déchargée (libres), le contrôleur RobotController envoie les signaux Arm1Retract et Arm2Retract pour que le robot rétracte ses bras. La présence des signaux Arm1Retracted et Arm2Retracted permet au contrôleur RobotController d'envoyer le signal MoveToPressAndDepositBelt au robot pour que ce dernier effectue une rotation lui permettant d'atteindre la position de déchargement dans laquelle le bras Arm1 (resp. Arm2) sera orienté vers la presse (resp. le convoyeur DB). L'atteinte de la position de déchargement provoquera la réception du signal AtPressAndDepositBelt par le contrôleur RobotController qui, à son tour, informe le contrôleur ProductCellController qu'il est prêt pour le déchargement, en lui envoyant le signal RobotReadyForUnloading et se met par la suite en attente du signal UnloadRobot. Une fois ce dernier est reçu, le contrôleur RobotController répète la tâche d'extension des bras du robot de la même façon citée précédemment et ensuite il envoie les signaux Arm1MagOFF et Arm2MagOFF pour que les bras du robot relâchent les pièces. De même, après le dépôt des pièces, le contrôleur RobotController demande au robot de rétracter ses bras de la même façon suscitée. Ensuite, le contrôleur RobotController envoie le signal RobotUnloaded au contrôleur ProductCellController pour l'informer que la presse et le convoyeur de dépôt sont chargés, et le signal MoveToTableAndPress au robot pour qu'il effectue une rotation au sens inverse de la première rotation jusqu'à l'atteinte de la position de chargement dans laquelle le bras Arm1 (resp. Arm2) sera en face la table (resp. presse). L'arrivée à la position de chargement provoquera la présence du signal AtTableAndPress, ce qui permettra au contrôleur RobotController d'envoyer au contrôleur ProductCellController le signal RobotReadyForLoading. Ainsi, le contrôleur RobotController se mettra de nouveau en attente afin de répéter sa tâche.

Qui veut dire que le signal *DepositBeltReadyForLoading* est présent. Donc, le signal *AtTableAndPress* est présent, les signaux *Arm1Retract* et *Arm2Retract* sont présents. Le signal *PressReadyForUnloading* est présent. Le signal *TableReadyForUnloading* est absent.

Dans la spécification du contrôleur RobotController en Esterel, nous avons programmé certaines tâches pour que leur exécution soient séquentielle malgré qu'elles sont parallèles car elles posent des problèmes de conception au niveau de Ptolemy II qui a des limites à ce niveau.

Le code Esterel spécifiant le contrôleur RobotController est le suivant :

```
1  module RobotController:
2    % The Controller of the Robot
3
4    % input signals
5    input LoadRobot;                % from ProductCellController
6    input UnloadRobot;              % from ProductCellController
7    input Arm1Extended;             % from Robot
8    input Arm2Extended;             % from Robot
9    input Arm1Retracted;            % from Robot
10   input Arm2Retracted;            % from Robot
11   input AtTableAndPress;          % from Robot
12   input AtPressAndDepositBelt;    % from Robot
13
14   % output signals
15   output RobotReadyForUnloading;     % to ProductCellController
16   output RobotReadyForLoading;       % to ProductCellController
17   output RobotLoaded;                % to ProductCellController
18   output RobotUnloaded;              % to ProductCellController
19   output Arm1Extend;                 % from Robot
20   output Arm1Retract;                % from Robot
21   output Arm2Extend;                 % from Robot
22   output Arm2Retract;                % from Robot
23   output Arm1MagON;                  % from Robot
24   output Arm1MagOFF;                 % from Robot
25   output Arm2MagON;                  % from Robot
26   output Arm2MagOFF;                 % from Robot
27   output MoveToTableAndPress;        % from Robot
28   output MoveToPressAndDepositBelt;  % from Robot
29
30
31   loop
32   [
33     % load robot
34     present LoadRobot then
35       emit Arm1Extend;
36       await Arm1Extended;
37       emit Arm1MagON;
38       emit Arm1Retract;
39       await Arm1Retracted;
40       emit Arm2Extend;
41       await Arm2Extended;
42       emit Arm2MagON;
43       emit Arm2Retract;
44       await Arm2Retracted;
45       [
46         emit RobotLoaded;
47       ||
48         emit MoveToPressAndDepositBelt;
49       ]
50     end present;
51   ||
52     % Robot ready for unloading
53     present AtPressAndDepositBelt then
54       emit RobotReadyForUnloading
55     end present;
56     await tick
57   ||
58     % unload Robot
59     present UnloadRobot then
60       emit Arm1Extend;
61       await Arm1Extended;
62       emit Arm1MagOFF;
63       emit Arm1Retract;
64       await Arm1Retracted;
65       emit Arm2Extend;
66       await Arm2Extended;
67       emit Arm2MagOFF;
68       emit Arm2Retract;
69       await Arm2Retracted
70       [
71         emit RobotUnloaded
72       ||
73         emit MoveToTableAndPress
74       ];
75     end present;
```

```
76 ||   ||
77 ||     % Robot ready for loading
78 ||     present AtTableAndPress then
79 ||       emit RobotReadyForLoading;
80 ||     end present;
81 ||     await tick
82 ||   ]
83 ||  end loop
84 || end module
```

Contrôleur de la presse

«... The task for the press is to forge metal blanks. The press consists of two horizontal plates, with the lower plate being movable along a vertical axis. The press operates by pressing the lower plate against the upper plate. Because the robot arms are placed on different horizontal planes, the press has three positions. In the lower position, the press is unloaded by arm 2, while in the middle position it is loaded by arm 1. The operation of the press is coordinated with the robot arms as follows : 1. Open the press in its lower position and wait until arm 2 has retrieved the metal plate and left the press, 2. Move the lower plate to the middle position and wait until arm 1 has loaded and left the press, 3. Close the press, i.e. forge the metal plate. This processing sequence is carried out cyclically. [204]»

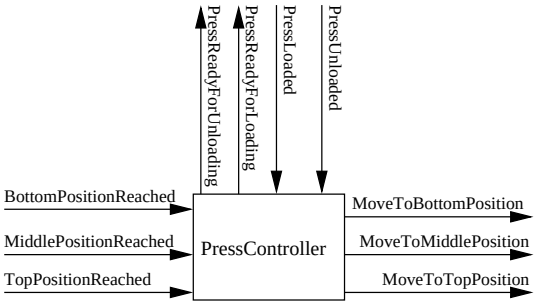


FIG. 8.6: Contrôleur de la presse.

Selon la description de la tâche de la presse, le contrôleur PressController (cf. figure 8.6) doit assurer que la plaque bas de la presse se déplace dans l'ordre chargement, pressement (forging) et déchargement, et cela, d'une façon répétitive.

Au début, la plaque bas est à la position de chargement (position du milieu) et le contrôleur PressController attend la présence du signal PressLoaded envoyé par le contrôleur ProductCellController. Une fois que le signal PressLoaded soit présent (presse chargée), le contrôleur PressController envoie le signal MoveToTopPosition à la presse pour qu'elle déplace sa plaque bas à la position de pressement (position du haut). L'atteinte de cette position impliquera automatiquement le pressement de la pièce et dans ce cas la presse envoie le signal TopPositionReached au contrôleur PressController. Ensuite, ce dernier envoie le signal MoveToBottomPosition pour déplacer la plaque bas de la presse à la position de déchargement laquelle une fois atteinte, la presse informe le contrôleur PressController par l'envoi du signal BottomPositionReached, ce qui exprime le prêt de la presse au déchargement. A ce niveau, le contrôleur PressController envoie le signal PressReadyForUnloading. La présence du signal PressUnloaded permet au contrôleur PressController d'envoyer le signal MoveToMiddlePosition à la presse pour qu'elle déplace la plaque bas à la position de chargement (position du milieu). Une fois cette position est atteinte, la presse informe le contrôleur PressController par l'envoi du signal MiddlePositionReached que la plaque est à la position de chargement. Finalement, le contrôleur PressController envoi le signal PressReadyForLoading au contrôleur ProductCellController et se met de nouveau en attente du signal PressLoaded pour répéter de nouveau sa tâche.

Nous avons spécifié le comportement du contrôleur PressController en Esterel comme suit :

```
1  module PressController:
2    % The Controller of the Press
3
4    % input signals
5    input PressUnloaded;           % from ProductCellController
6    input PressLoaded;             % from ProductCellController
7    input TopPositionReached;      % from Press
8    input MiddlePositionReached;   % from Press
9    input BottomPositionReached;   % from Press
10
11   % output signals
12   output PressReadyForUnloading;  % to ProductCellController
13   output PressReadyForLoading;    % to ProductCellController
14   output MoveToTopPosition;       % to Press
15   output MoveToMiddlePosition;    % to Press
16   output MoveToBottomPosition;    % to Press
17
18   loop
19     present PressUnloaded then
20       emit MoveToMiddlePosition;
21     end present;
22
23     present MiddlePositionReached then
24       emit PressReadyForLoading;
25     end present;
26
27     present PressLoaded then
28       emit MoveToTopPosition;
29     end present;
30
31     present TopPositionReached then
32       emit MoveToBottomPosition
33     end present;
34
35     present BottomPositionReached then
36       emit PressReadyForUnloading
37     end present;
38
39     await tick
40
41   end loop
42
43 end module
```

Contrôleur du convoyeur de dépôt

...The task of the deposit belt is to transport the work pieces unloaded by the second robot arm to the traveling crane. A photoelectric cell is installed at the end of the belt ; it reports when a work piece reaches the end section of the belt. The control program then has to stop the belt. The belt can restart as soon as the traveling crane has picked up the work piece. ... photoelectric cells switch on when a plate intercepts the light ray. Just after the plate has completely passed through it, the light barrier switches off. At this precise moment, the plate is in the correct position to be picked up by the traveling crane ... [204]»

La tâche du contrôleur DBController (cf. figure 8.7) est d'assurer l'acheminement de chaque pièce depuis la zone dans laquelle a été déposée par le robot jusqu'à la zone où elle sera récupérée par la grue. En effet, le contrôleur DBController répète le cycle : attendre le chargement du convoyeur DB, déplacer la pièce vers la grue et attendre le déchargement de la pièce.

Au début le convoyeur DB est prêt pour le chargement et donc une fois le signal DepositBeltLoaded est envoyé par ProductCellController, le contrôleur DBController met en marche le convoyeur DB par l'envoi du signal MovePieceToDepositBeltEnd, ce qui aura comme conséquence le déplacement de la pièce à la zone de déchargement. L'arrivée de la pièce à ladite zone de déchargement impliquera automatiquement la présence du signal PieceAtDepositBeltEnd, ce qui permet au

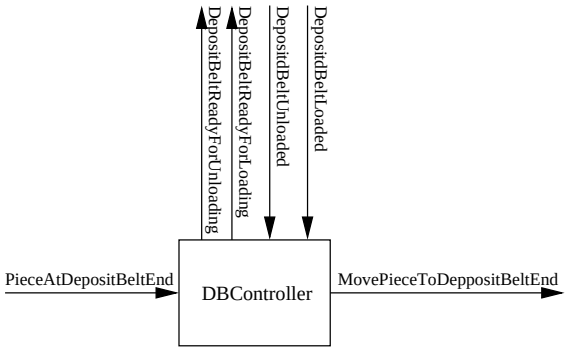


FIG. 8.7: Contrôleur du convoyeur de dépôt.

contrôleur DBController d'informer le contrôleur ProductCellController que le convoyeur DB est prêt pour le déchargement, en lui envoyant le signal DepositBeltReadyForUnloading. La présence du signal DepositBeltUnloaded permet au contrôleur DBController d'envoyer le signal DepositBeltReadyForLoading au contrôleur ProductCellController pour l'informer que le convoyeur DB est prêt pour le chargement. Finalement, DBController reprend de nouveau sa tâche en attendant la présence du signal DepositBeltLoaded.

Nous résumons la précédente formalisation par l'algorithme suivant :

```
1  module DBController:
2    % The Controller of the deposit belt (DB)
3
4    % input signals
5    input PieceAtDepositBeltEnd;      % from DB
6    input DepositBeltLoaded;          % from ProductCellController
7    input DepositBeltUnloaded;        % from ProductCellController
8
9    % output signals
10   output MovePieceToDepositBeltEnd;      % to DB
11   output DepositBeltReadyForUnloading;   % to ProductCellController
12   output DepositBeltReadyForLoading;     % to ProductCellController
13
14   loop
15     present DepositBeltLoaded then
16       emit MovePieceToDepositBeltEnd
17     end present;
18
19     present PieceAtDepositBeltEnd then
20       emit DepositBeltReadyForUnloading
21     end present;
22
23     present DepositBeltUnloaded then
24       emit DepositBeltReadyForLoading
25     end present;
26
27     await tick
28   end loop
29
30 end module
```

Contrôleur de la grue

«...The task of the traveling crane consists in picking up metal plates from the deposit belt, moving them to the feed belt and unloading them there. ...The typical operation of the crane is as follows : 1. After the signal from the photoelectric cell indicates that a work-piece has moved into the unloading area on the deposit belt, the gripper positions itself through horizontal and vertical translations over the deposit belt and picks up the metal plate. 2. The gripper transports the metal plate to the feed belt and unloads it there. [204]»

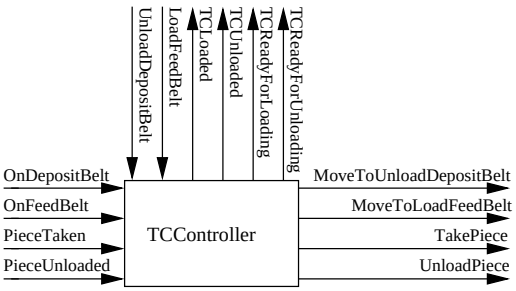


FIG. 8.8: Contrôleur de la grue.

La tâche de la grue est de répéter le cycle décharger le convoyeur DB, déplacer la pièce du convoyeur DB au convoyeur FB et finalement déposer la pièce (charger) sur le convoyeur FB.

Au départ, la grue est à la position de déchargement du convoyeur DB (au dessus du DB) et est prête pour récupérer une pièce. Donc, le signal DepositBeltReadyForUnloading envoyé par ProductCellController permet d'informer le contrôleur TCController (cf. figure 8.8) sur la présence d'une pièce sur le convoyeur DB pour la récupérer. Ainsi, le contrôleur TCController envoie le signal TakePiece à la grue pour qu'elle récupère la pièce. La récupération de la pièce implique automatiquement l'envoi du signal PieceTaken au contrôleur TCController, ce qui permet à ce dernier d'envoyer le signal TCLoaded au contrôleur ProductCellController et d'ordonner la grue à se déplacer à la position de déchargement (au dessus du convoyeur FB) en lui envoyant le signal MoveToLoadFeedBelt. L'arrivée de la grue au dessus du convoyeur FB impliquera automatiquement l'envoi du signal OnFeedBelt au contrôleur TCController, qui ainsi informe le contrôleur ProductCellController, par l'envoi du signal TCReadyForUnloading, que la grue est prête pour déposer la pièce sur le convoyeur FB. La présence du signal LoadFeedBelt permet au contrôleur TCController d'ordonner la grue de déposer la pièce en lui envoyant le signal UnloadPiece. Ensuite, le signal PieceUnloaded est envoyé par la grue pour signaler au contrôleur TCController que la pièce a été déposée. A ce niveau, le contrôleur TCController envoie le signal TCUnloaded au contrôleur ProductCellController pour l'informer que la grue a été déchargée et le signal MoveToDepositBelt à la grue pour quelle se déplace à la position de déchargement (au dessus du convoyeur DB). L'arrivée de la grue à cette position impliquera automatiquement l'envoi du signal OnDepositBelt, ce qui permet au contrôleur TCController d'informer le contrôleur ProductCellController que la grue est prête pour la récupération d'une nouvelle pièce, en lui envoyant le signal TCReadyForLoading. Finalement, le contrôleur TCController répète sa tâche, et cela, après la réception de nouveau le signal UnloadDepositBelt.

Nous avons spécifié le comportement du contrôleur TCController en Esterel comme suit :

```
1  module PressController:
2    % The Controller of Travelling Crane (TC)
3
4    % input signals
5    input LoadFeedBelt;    % from ProductCellController
6    input UnloadDepositBelt; % from ProductCellController
7    input OnFeedBelt;    % from TCMot
8    input OnDepositBelt;     % from TCMot
9    input PieceTaken;    % from TCMot
10   input PieceUnloaded;     % from TCMot
11
12   % output signals
13   output TCLoaded;     % to ProductCellController
14   output TCUnloaded;     % to ProductCellController
15   output TCReadyForLoading; % to ProductCellController
16   output TCReadyForUnloading; % to ProductCellController
17   output MoveToUnloadDepositBelt; % to TCMot
18   output MoveToLoadFeedBelt;  % to TCMot
19   output TakePiece;    % to TCMot
20   output UnloadPiece;    % to TCMot
```

```
21
22
23    loop
24      present UnloadDepositBelt then
25        emit TakePiece;
26      end present;
27
28      present PieceTaken then
29        [
30          emit MoveToLoadFeedBelt
31        ||
32          emit TCLoaded
33        ]
34      end present;
35
36      present OnFeedBelt then
37        emit TCReadyForUnloading;
38      end present;
39
40      present LoadFeedBelt then
41        emit UnloadPiece;
42      end present;
43
44      present PieceUnloaded then
45        [
46          emit MoveToUnloadDepositBelt
47        ||
48          emit TCUnloaded
49        ];
50      end present;
51
52      present OnDepositBelt then
53        emit TCReadyForLoading;
54      end present;
55
56      await tick
57    end loop
58  end module
```

Contrôleur principal de l'atelier

Comme nous l'avons signalé avant, le contrôleur ProductCellController est le responsable sur le fonctionnement global de l'atelier. C'est lui qui définit toutes les étapes du cycle d'usinage ainsi que leur ordonnancement.

Pour chaque zone critique, dite aussi zone de synchronisation, le contrôleur ProductCellController n'autorise un dispositif voulant transférer une pièce à un autre que lorsqu'il reçoit des signaux de la part des deux dispositifs en lui indiquant qu'ils sont prêts pour un tel transfert.

En fonction de la spécification des six contrôleurs de l'atelier, nous avons spécifié le comportement du contrôleur ProductCellController en Esterel comme suit :

```
1   module ProductCellController:
2     % The Controller of the production cellul
3
4     % input signals
5     input RobotReadyForUnloading;        % from RobotController
6     input RobotReadyForLoading;          % from RobotController
7     input RobotLoaded;                   % from RobotController
8     input RobotUnloaded;                 % from RobotController
9     input TCLoaded;                      % from TCController
10    input TCUnloaded;                    % from TCController
11    input TCReadyForLoading;             % from TCController
12    input TCReadyForUnloading;           % from TCController
13    input PressReadyForUnloading;        % from PressController
14    input PressReadyForLoading;          % from PressController
15    input FeedBeltReadyForLoading;       % from FBController
16    input FeedBeltReadyForUnloading;     % from FBController
17    input FeedBeltUnloaded;              % from FBController
18    input TableReadyForUnloading;        % from ERTController
```

```
19    input TableReadyForLoading;          % from ERTController
20    input DepositBeltReadyForUnloading; % from DBController
21    input DepositBeltReadyForLoading;    % from DBController
22
23    % output signals
24    output LoadRobot;             % to RobotController
25    output UnloadRobot;           % to RobotController
26    output LoadFeedBelt;          % to TCController
27    output UnloadDepositBelt;     % to TCController
28    output PressLoaded;           % to PressController
29    output PressUnloaded;         % to PressController
30    output UnloadFeedBelt;        % to FBController
31    output FeedBeltLoaded;        % to FBController
32    output TableUnloaded;         % to ERTController
33    output TableLoaded;           % to ERTController
34    output DepositBeltLoaded;     % to DBController
35    output DepositBeltUnloaded; % to DBController
36
37    var
38      pRobotReadyForUnloading := false : boolean,
39      pRobotReadyForLoading := false : boolean,
40      pTCReadyForLoading := false : boolean,
41      pTCReadyForUnloading := false : boolean,
42      pPressReadyForUnloading := false : boolean,
43      pPressReadyForLoading := false : boolean,
44      pFBReadyForLoading := false : boolean,
45      pFBReadyForUnloading := false : boolean,
46      pTableReadyForUnloading := false : boolean,
47      pTableReadyForLoading := false : boolean,
48      pDBReadyForUnloading := false : boolean,
49      pDBReadyForLoading := false : boolean
50    in
51    loop
52      present RobotReadyForUnloading then
53        pRobotReadyForUnloading := true
54      end present;
55
56      present RobotReadyForLoading then   % initialized to present
57        pRobotReadyForLoading := true
58      end present;
59
60      % Table and Press unloaded, and Robot loaded
61      present RobotLoaded then
62        [
63          emit PressUnloaded
64        ||
65          emit TableUnloaded
66        ]
67      end present;
68
69      % Press and DB loaded, and Robot unloaded
70      present RobotUnloaded then
71        [
72          emit PressLoaded
73        ||
74          emit DepositBeltLoaded
75        ]
76      end present;
77
78      % TC loaded and DB unloaded
79      present TCLoaded then
80        emit DepositBeltUnloaded;
81      end present;
82
83      % Feed Belt loaded and TC unloaded
84      present TCUnloaded then
85        emit FeedBeltLoaded;
86      end present;
87
88      present TCReadyForLoading then    % initialized to present
89        pTCReadyForLoading := true
90      end present;
91
92      present TCReadyForUnloading then
93        pTCReadyForUnloading := true
```

```
94      end present;
95
96      present PressReadyForUnloading then
97        pPressReadyForUnloading := true
98      end present;
99
100     present PressReadyForLoading then
101       pPressReadyForLoading := true
102     end present;
103
104     present FeedBeltReadyForLoading then
105       pFBReadyForLoading := true
106     end present;
107
108     present FeedBeltReadyForUnloading then
109       pFBReadyForUnloading := true
110     end present;
111
112     % FB unloaded and Table loaded
113     present FeedBeltUnloaded then
114       emit TableLoaded
115     end present;
116
117     present TableReadyForUnloading then
118       pTableReadyForUnloading := true
119     end present;
120
121     present TableReadyForLoading then
122       pTableReadyForLoading := true
123     end present;
124
125     present DepositBeltReadyForUnloading then
126       pDBReadyForUnloading := true
127     end present;
128
129     present DepositBeltReadyForLoading then
130       pDBReadyForLoading := true
131     end present;
132
133
134     % load Table and unload Feed Belt
135     if pTableReadyForLoading and pFBReadyForUnloading then
136       emit UnloadFeedBelt;
137       pTableReadyForLoading := false;
138       pFBReadyForUnloading := false;
139     end if;
140
141     % unload Table and Press, and load Robot
142     if pTableReadyForUnloading and pPressReadyForUnloading
143         and pRobotReadyForLoading then
144       emit LoadRobot;
145       pTableReadyForUnloading := false;
146       pPressReadyForUnloading := false;
147       pRobotReadyForLoading := false;
148     end if;
149
150     % load Press and DB, and unload Robot
151     if pPressReadyForLoading and pRobotReadyForUnloading
152         and pDBReadyForLoading then
153       emit UnloadRobot;
154       pPressReadyForLoading := false;
155       pRobotReadyForUnloading := false;
156       pDBReadyForLoading := false;
157     end if;
158
159     % unload DB and load TC
160     if pDBReadyForUnloading and pTCReadyForLoading then
161       emit UnloadDepositBelt;
162       pDBReadyForUnloading := false;
163       pTCReadyForLoading := false;
164     end if;
165
166     % unload TC and load FB
167     if pTCReadyForUnloading and pFBReadyForLoading then
168       emit LoadFeedBelt;
```

```
169 ||          pTCReadyForUnloading := false;
170 ||          pFBReadyForLoading := false;
171 ||        end if;
172 ||
173 ||          await tick
174 ||
175 ||      end loop
176 ||      end var
177 || end module
```

Vérification des propriétés de sûreté par rapport à la spécification

Les propriétés de sûreté doivent être respectées afin de garantir un fonctionnement strict et sûr de l'atelier.

Dans [203], on trouve plusieurs propriétés de sûreté. La plupart d'elles se limitent à la mobilité des dispositifs, l'évite des collisions entre les dispositifs, etc. A titre d'exemple, les propriétés de sûreté pour la table ERT sont :

- La table ne doit pas se déplacer vers le bas une fois la position de chargement est atteinte et elle ne doit pas se déplacer vers le haut une fois la position de déchargement est atteinte ;
- Si elle est dans la position qui permet au robot de récupérer la pièce, la table ne doit pas tourner dans le sens des aiguilles de la montre. De même, si elle est dans la position permettant de récupérer la pièce depuis le convoyeur FB, la table ne doit pas tourner dans le sens inverse des aiguilles de la montre.

Pour vérifier les propriétés de sûreté de nos contrôleurs précédemment spécifiés, nous avons utilisé l'outil XEVE [107], qui est un modèle de vérification permettant une exécution symbolique pour dire si un signal est potentiellement émis ou non.

Après avoir spécifié les contrôleurs de l'atelier, nous avons, à un niveau abstrait, établi le diagramme de blocs plat montrant l'interconnexion de l'ensemble de contrôleurs, voir la figure 8.9.

8.4 Spécification des capteurs de l'atelier suivant la sémantique synchrone

Les contrôleurs de l'atelier agissent en fonction des déplacements des pièces sur les dispositifs. En effet, lorsque les pièces arrivent à des positions bien précises, les contrôleurs doivent être avertis par l'activation de leurs signaux capteurs pour qu'ils puissent se synchroniser et agir sur les dispositifs. C'est les capteurs qui sont responsables sur l'activation des signaux capteurs. Par exemple, le capteur placé au début de la zone critique FB-ERT active le signal capteur PieceAtFeedBeltEnd dés que la pièce arrive à ce niveau pour informer le contrôleur FBController. Ce qui permet à ce dernier de se synchroniser, via le contrôleur principal, avec le contrôleur de la table afin de transférer la pièce à la table.

Nous avons représenté chaque capteur par un même contrôleur paramétrable dont la tâche est de signaler l'arrivée d'une pièce à une position bien précise. Ce contrôleur est appelé PositionController. En effet, à chaque instant t, il lit la position de la pièce à partir de son signal d'entrée, currentPosition, et puis la compare à la valeur de la position désirée. Si les deux positions sont identiques, il envoie le signal positionReached pour activer le signal capteur du contrôleur du dispositif concerné.

Pour le calcul de la position, nous avons supposé que toutes les pièces se déplacent sur tous les dispositifs avec une vitesse constante et nous avons négligé toutes les forces physiques impliquées dans leur déplacement. Cela revient au fait que la modélisation de la partie dynamique de l'atelier est une tâche difficile et sort du cadre de notre travail. De plus, notre but est juste d'utiliser dans la

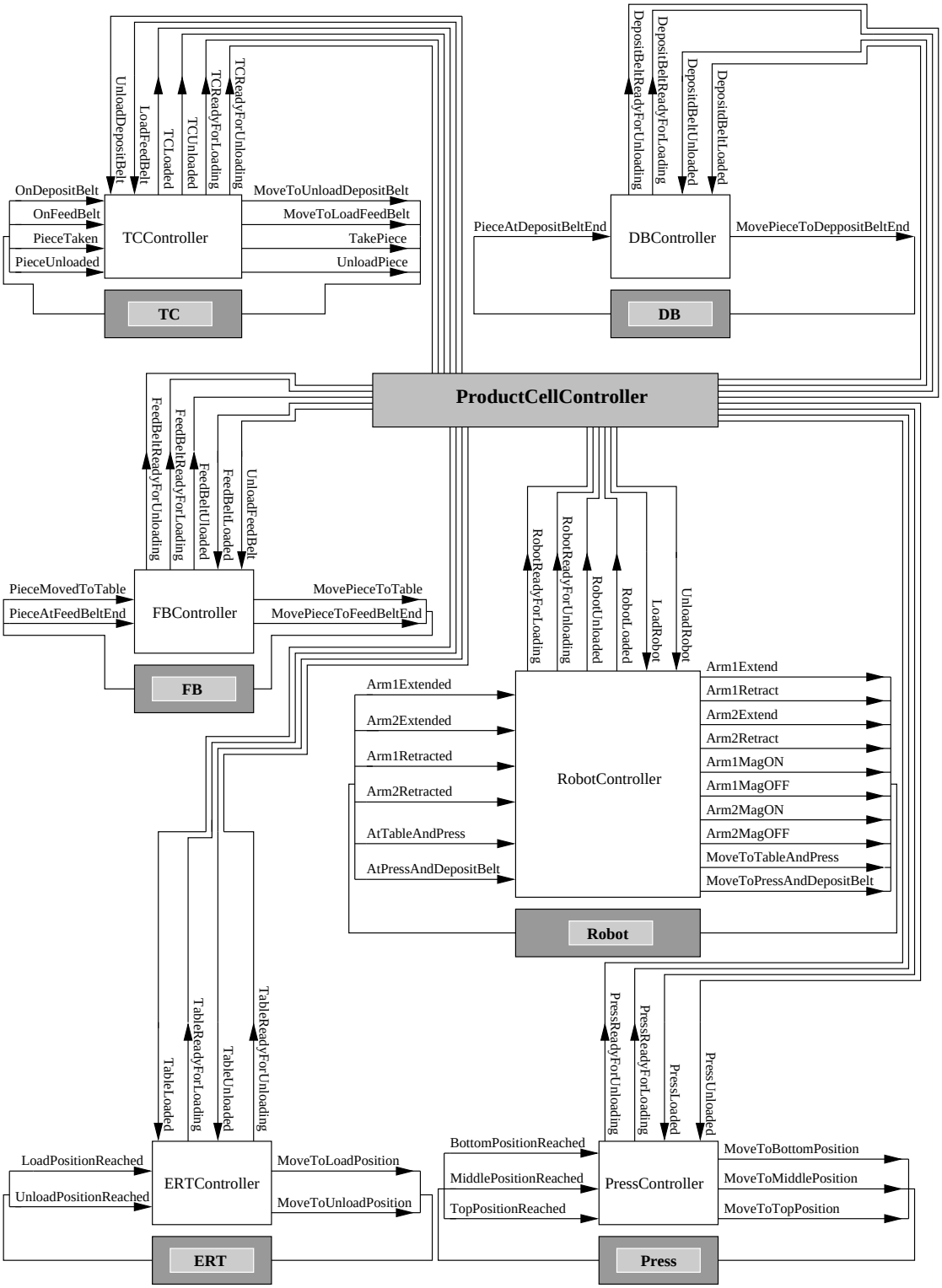


FIG. 8.9: Diagramme de blocs montrant l'iterconnexion des contrôleurs de l'atelier.

section 8.5 des composants domaine-polymorphe ayant des composants Noyaux synchrones sous le modèle de calcul CT (Continuous Time).

La formule permettant de calculer la position d'une pièce à chaque instant t est la suivante :

$$p_p(t) = \int V_p(t) \tag{8.1}$$

Avec

$p_p(t)$ est la position de la pièce p.

$V_p(t)$ est la vitesse de la pièce p.

Nous avons spécifié le contrôleur PositionController pour qu'il soit configurable et ainsi réutilisable afin de représenter tous les capteurs de l'atelier. Pour cela, il a deux constantes, desiredPosition et positiveDirection, qui sont ses paramètres. La première indique à quelle position (position désirée) le contrôleur doit signaler l'arrivée d'une pièce tandis que la deuxième indique le sens de déplacement de la pièce sur le dispositif (positif ou négatif).

Nous avons spécifié le contrôleur PositionController en Esterel comme suit :

Listing 8.1: Module Esterel PositionController

```
module PositionController:
  % Controller of the position.
  % The task of this module is to send a signal
  % when the desired position is reached.

  % input signals
  input currentPosition : double;

  % output signals
  output positionReached;

  % Parameters of controller
  constant desiredPosition : double;
  constant positiveDirection : boolean;


  every immediate currentPosition do
    if positiveDirection then
      if (?currentPosition >= desiredPosition) then
        emit positionReached
      end if
    else
      if (?currentPosition <= desiredPosition) then
        emit positionReached
      end if
    end if
  end every
end module
```

8.5 Conception et simulation basées CDP de l'atelier sous Ptolemy II

8.5.1 Conception du modèle Ptolemy II de l'atelier

La conception du modèle Ptolemy II de l'atelier consiste d'abord à créer les composants représentant les contrôleurs et les capteurs de l'atelier et les composants permettant le calcul des positions des pièces à tout instant, et à choisir les modèles de calcul appropriés, et ensuite, de structurer les composants et de les interconnecter pour avoir le modèle de l'atelier.

Tous les contrôleurs et les capteurs de l'atelier sont représentés par des composants domaine-polymorphes dont les composants Noyaux sont générés à partir de leurs (contrôleurs et capteurs) spécifications en Esterel par notre traducteur présenté dans l'annexe A.

Nous avons conçu le modèle Ptolemy II de l'atelier suivant l'approche hiérarchique. Pour cela, nous avons placé chaque composant domaine-polymorphe représentant un contrôleur et le dispositif qui le contrôle dans un acteur composite (qui est un sous modèle). Ensuite, l'ensemble d'acteurs composites et le composant domaine-polymorphe DPCProductCellController représentant le contrôleur principal de l'atelier sont placés au même niveau hiérarchique, qui est le top-level, voir figure 8.10.

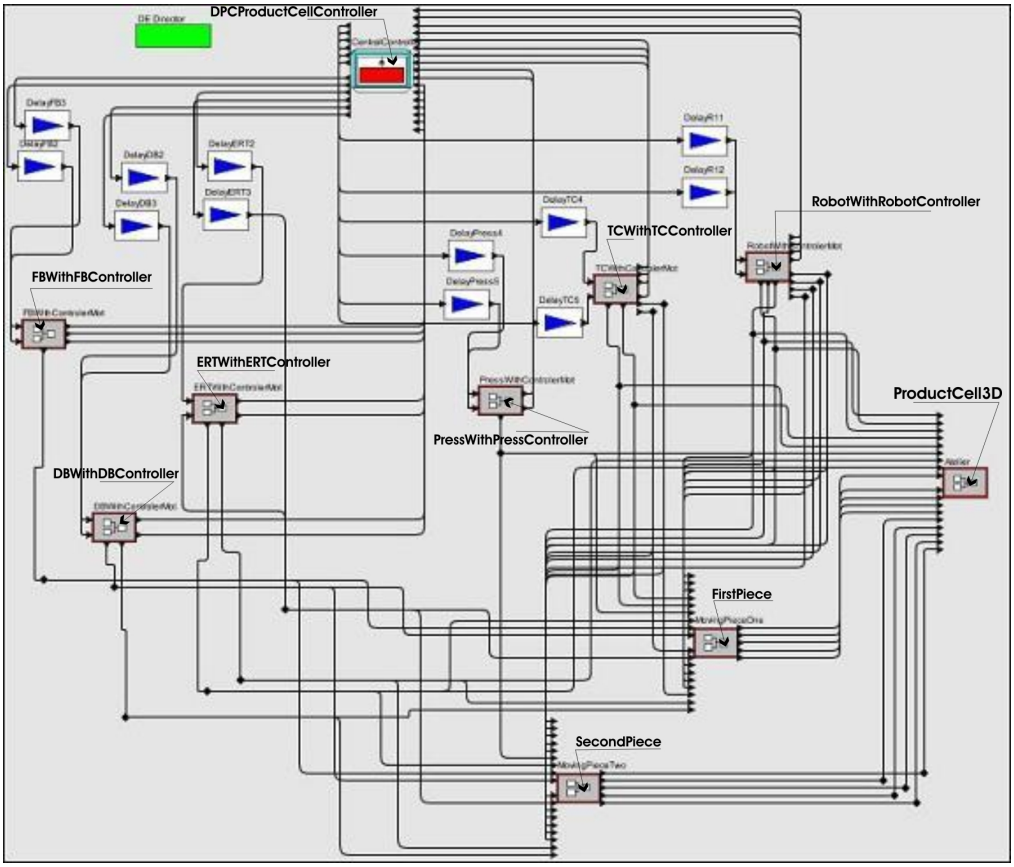


FIG. 8.10: Modèle Ptolemy II de l'atelier basé composants domaine-polymorphes.

La figure 8.10 montre le modèle top-level (niveau haut de la hiérarchie) de l'atelier, qui est un acteur composite opaque ayant le directeur DE (Discrete Event) comme directeur local.

Nous avons utilisé les acteurs atomiques de type Delay prédéfinis par Ptolemy II pour briser les boucles par l'introduction d'un retard. Et comme les composants domaine-polymorphes ont des composants Noyaux synchrones, nous avons pris un retard égal à zéro afin de respecter l'hypothèse synchrone. Nous avons aussi utilisé ce type d'acteur dans les sous modèles (acteurs composites).

Les acteurs composites du niveau top-level sont FBWithFBController, ERTWithERTController, DBWithDBController, PressWithPressController, TCWithTCController, RobotWithRobotController et ProductCell3D.

Chacun des six premiers acteurs composites comporte un composant domaine-polymorphe, qui représente le contrôleur, et un acteur composite, qui représente le dispositif à contrôler.

L'acteur composite représentant le dispositif comporte les acteurs permettant de calculer la position des pièces tout au long de leur déplacement et les composants domaine-polymorphes qui représentent les capteurs.

Comme nous avons conçu les six acteurs de la même façon, en utilisant les mêmes modèles de calcul et la même structuration des niveaux hiérarchiques, nous présentons ici que l'acteur FBWithFBController.

FBWithFBController

La figure 8.11 montre le sous modèle (acteur composite) FBWithFBController du convoyeur FB. Il utilise le modèle de calcul DE et est composé du composant domaine-polymorphe DPCFBController, qui représente le contrôleur du convoyeur FB, des acteurs atomiques de type Delay et de l'acteur composite HSFB.

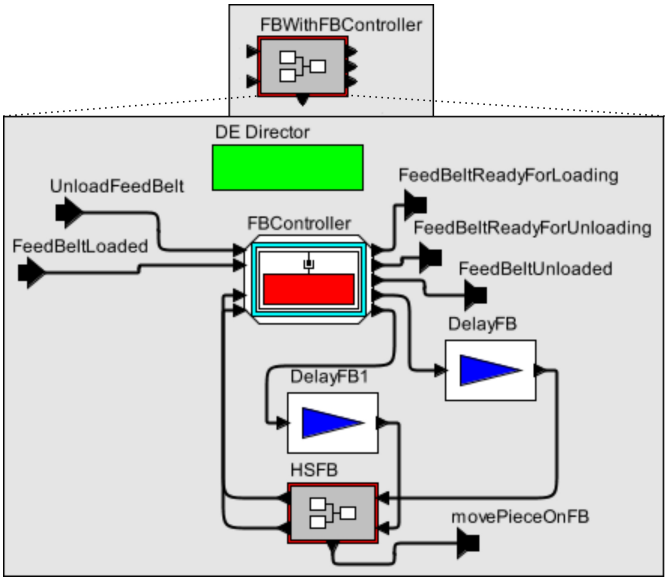


FIG. 8.11: Le sous modèle FBWithFBController.

De même l'acteur composite HSFB est, lui aussi, un sous modèle, qui représente la partie dynamique du convoyeur FB, voir la figure 8.12.

Chaque dispositif de l'atelier passe par plusieurs états dont le nombre dépend de sa tâche. Dans le cas du convoyeur FB, nous comptons trois états, MovingOnFB, Wait et MovingToTable. Le premier état indique que le convoyeur FB est en cours de déplacer la pièce de la position à laquelle a été déposée par la grue TC jusqu'à la zone critique FB-ERT. Le deuxième état indique l'attente du convoyeur FB des signaux actuators du DPCFBController pour passer à l'état suivant. Finalement, le dernier état indique que le convoyeur FB est en cours de faire passer la pièce à la table ERT.

En effet, nous avons représenté la partie dynamique du convoyeur FB par un système hybride [34] composé de modèles de calcul FSM (Finite State Machines) et CT (Continuous Time), voir figure 8.12. Donc, l'acteur composite HSFB a un acteur FSM, appelé StatesOfFB, et un directeur local CT. L'acteur FSM StatesOfFB est composé de trois états du convoyeur FB. Chaque état est raffiné par un sous modèle comportant les acteurs permettant, à chaque instant t, le calcul de la position de la pièce sur le convoyeur FB et le composant domaine-polymorphe, DPCPositionController, représentant le capteur. Ainsi, lorsque la position désirée de la pièce est atteinte, c'est le composant domaine-polymorphe DPCPositionController qui rend les gardes des transitions entre les états vraies, ce qui implique l'exécution des actions des transitions pour activer les signaux capteurs du composant domaine-polymorphe DPCFBController.

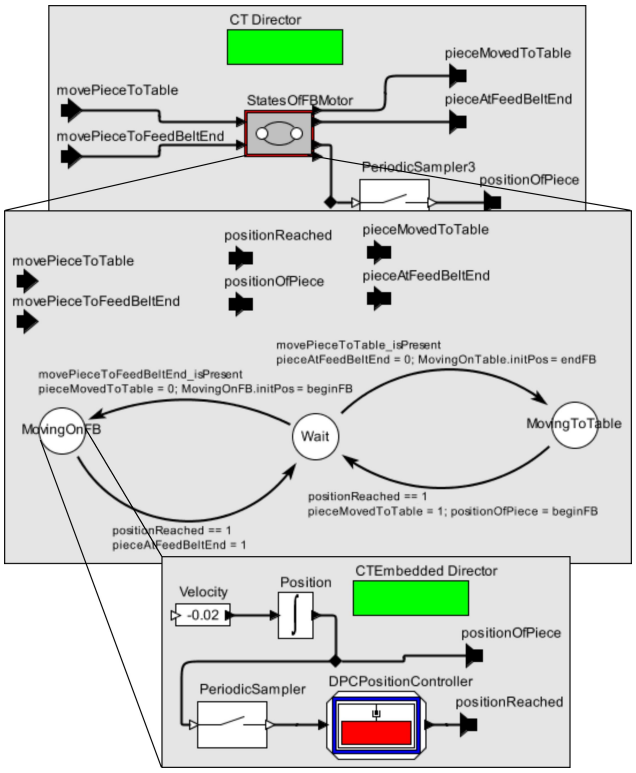


FIG. 8.12: Le sous modèle HSFB représentant la partie dynamique du convoyeur FB.

8.5.2 Simulation du fonctionnement de l'atelier

Pour une simulation graphique en 3 dimensions, nous avons conçu la scène graphique de tous les dispositifs de l'atelier, voir la figure 8.13.

L'acteur composite ProductCell3D a que des ports d'entrée à travers lesquels les acteurs composites calculant les posisitions des pièces font animer la scène graphique.

La simulation du fonctionnement de l'atelier nécessite la détermination du nombre de pièces à utiliser et l'initialisation de l'ensemble de contrôleurs.

Pour le nombre de pièces, nous n'avons pris que deux pièces. Ce nombre est suffisant pour faire fonctionner l'ensemble de dispositifs de l'atelier. De plus, au-delà de ce nombre, le modèle Ptolemy II de l'atelier, montré par la figure 8.10, devient plus compliqué.

L'initialisation des contrôleurs permet d'informer ces derniers sur l'état initial des dispositifs. Et comme les composants domaine-polymorphes représentants les contrôleurs sont sous le contrôle du modèle de calcul DE, nous avons réalisé l'initialisation par envoie des événements dans les méthodes configuration() des composants Noyaux. Les initialisations des dispositifs que nous avons prises sont :

- FB is loaded.
- ERT at load position.
- ERT is ready for loading.
- Arms of Robot are retracted.
- Robot at loading position.
- Robot is ready for loading (i.e for unloading the both Press and ERT).
- Press at bottom position.
- Press is ready for unloading.
- DB is ready for loading.

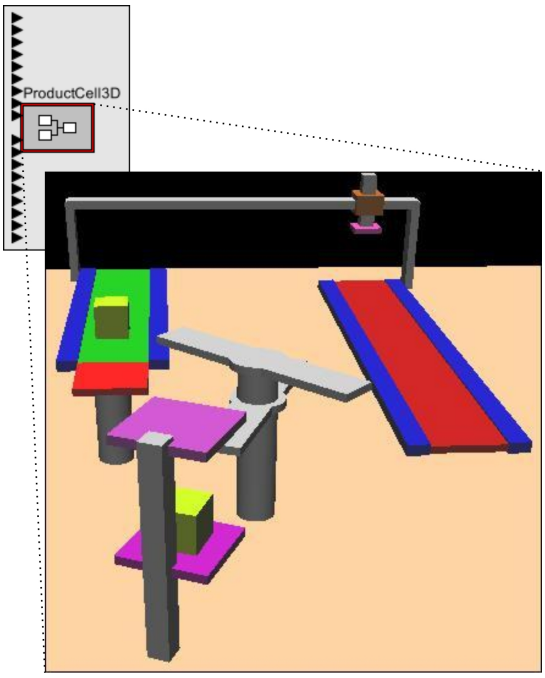


FIG. 8.13: L'atelier en 3D.

- TC at over the deposit belt.
- TC is ready for loading (i.e for unloading the DB).

Après avoir exécuté le modèle Ptolemy II de l'atelier, nous avons constaté que le fonctionnement des dispositifs et leur coopération ont été bien déroulés, ce qui implique que nos composants domaine-polymorphes ont exécuté correctement leurs composants Noyaux synchrones au sein des domaines DE et CT. De plus, dans le domaine CT, nous avons utilisé le composant domaine-polymorphe DPC-PositionController une fois comme un acteur discret (données non persistantes au niveau des ports) et une fois comme un acteur continu (données persistantes au niveau des ports). Dans les deux cas, le composant domaine-polymorphe DPCPositionController a exécuté correctement son composant Noyau.

8.6 Conclusion

Dans le but de valider par simulation de nos concepts, nous avons présenté dans ce chapitre la conception et la simulation basées composants domaine-polymorphes de l'atelier de production automatisé, qui est un système hétérogène mélangeant le contrôle et la dynamique continue.

Le modèle représentant l'atelier a été conçu sous Ptolemy II. Nous avons représenté les contrôleurs et les capteurs de l'atelier par des composants domaine-polymorphes dont les composants Noyaux ont été générés à partir des modules Esterel par notre traducteur (voir annexe A). Par contre, nous avons représenté les déplacements des pièces sur chaque dispositif par un système hybride [34].

Nous avons utilisé les composants domaine-polymorphes, dont les composants Noyaux sont synchrones, dans deux domaines différents, DE et CT. Et la simulation du fonctionnement de l'atelier a bien montré que les composants domaine-polymorphes ont bien fonctionné.

Chapitre 9

Conclusion et Perspectives

Nous avons vu que les systèmes embarqués sont des systèmes hétérogènes. Cette hétérogénéité rend lesdits systèmes embarqués complexes et n'est que le résultat de la combinaison de divers domaines techniques, tel que la mécanique, l'électronique analogique, l'électronique numérique, l'optique, la thermodynamique, etc., imposée à la fois par l'évolution technologique et par les domaines d'application.

Ainsi, pour faire face à la complexité des systèmes embarqués, les outils de modélisation, de conception et de simulation doivent eux aussi suivre cette évolution technologique et proposent de plus en plus des approches et des mécanismes permettant, d'une part, d'exprimer le plus fidèlement possible la structure, les comportements et les propriétés des systèmes afin de garantir une bonne modélisation, une bonne conception et une bonne simulation de ces derniers, et d'autre part, la modularité et la réutilisabilité afin d'augmenter la productivité et de faciliter aux concepteurs des systèmes les processus de modélisation, de conception et de validation ainsi que la maintenabilité.

Dans le deuxième chapitre, nous avons montré qu'actuellement, les outils de modélisation de conception et de simulation des systèmes hétérogènes ne permettent pas l'utilisation directe d'un composant atomique sous un modèle de calcul différent de celui utilisé pour décrire son comportement (sémantique de spécification). En revanche, ces outils permettent une utilisation indirecte d'un tel composant atomique en s'appuyant soit sur des composants domaine-spécifiques, soit sur une des approches hiérarchique et non hiérarchique. Cependant, chacune de ces solutions présente des désavantages.

Le but de cette thèse était donc de rendre possible l'utilisation de composants atomiques dont le comportement est décrit selon une sémantique de spécification, sous des modèles de calcul ayant une sémantique différente, et cela sans passer par les solutions suscitées. Pour cela, nous avons proposé un modèle de composant, appelé composant domaine-polymorphe. Ce dernier est un composant atomique, dynamique, réutilisable, qui a la capacité de s'adapter aux différents modèles de calcul tout en garantissant un comportement interne suivant la sémantique de spécification.

La conception du composant domaine-polymorphe comporte plusieurs étapes. Dans la première étape nous avons montré comment utiliser les composants domaine-spécifiques pour exécuter un comportement suivant sa sémantique de spécification sous différents modèles de calcul, en spécifiant les différentes opérations de l'interface gérant l'hétérogénéité entre la sémantique de spécification et celle du modèle de calcul hôte. Pour cela, nous avons pris la sémantique réactive synchrone comme sémantique de spécification et avons présenté la conception de composants domaine-spécifiques synchrones (CDS Synchrones) pour plusieurs modèles de calcul. Ceci qui nous a permis de donner les règles générales permettant la conception des composants domaine-spécifiques pour d'autres sémantiques de spécification.

Ensuite, partant de la structure abstraite commune aux composants domaine-spécifiques dont le comportement est décrit selon une sémantique de spécification, nous avons présenté dans les

étapes suivantes des modèles de composants intermédiaires jusqu'à parvenir au modèle de composant domaine-polymorphe. Les modèles de composants intermédiaires sont le composant domaine-spécifique modulaire et composable, et le composant domaine-spécifique flexible.

9.1 Apports de la thèse

9.1.1 Modèle de composant domaine-polymorphe

Le composant domaine-polymorphe est composé de :

- *Composant Conteneur* : Il permet au composant domaine-polymorphe d'avoir la structure exigée par le domaine dans lequel il est utilisé. Son rôle est de cacher la complexité des domaines et sert à encapsuler les composants Noyaux, et ainsi à déclencher leurs réactions pour leur permettre d'interagir avec l'environnement (domaine) d'une manière transparente. Le composant Conteneur est donc vu comme une structure d'accueil qui représente un espace d'exécution des composants Noyaux ;
- *Composant Noyau* : Il représente un comportement décrit suivant la sémantique de spécification. Sa spécification est faite une fois pour toute, indépendamment des modèles de calcul sous lesquels on veut l'utiliser ;
- *Deux composants CASs (Conservateur Adaptateur de la Sémantique)* : selon le rôle de la sémantique dont il est spécifique, le composant CAS est utilisé en interne ou en externe. Le CASi est le composant CAS interne, qui est spécifique de la sémantique de spécification du noyau. Le CASe est le composant CAS externe, qui est spécifique de la sémantique du modèle de calcul gouvernant le composant domaine-polymorphe.
  Le composant CASi implémente le modèle de calcul approprié au composant Noyau. Il lui offre les opérations de communication et les opérations de rappel (call-back), ce qui permet au composant Noyau de fonctionner correctement.
  Le composant CASe a comme tâche la manipulation des ports de communication du composant Conteneur et l'exécution des éventuels traitements permettant au composant domaine-polymorphe de fonctionner correctement dans son domaine ;
- *Composant CF (Composant Frontière)* : Il permet aux concepteurs des systèmes d'expliciter le passage des données entre la sémantique du domaine hôte (sémantique externe) et celle de spécification (sémantique interne) à laquelle le composant Noyau doit obéir. Cette explicitation du passage fait partie de la conception du système ;
- *Composant CS (Connecteur des Sémantiques)* : Il permet l'interconnexion des composants CASi et CASe, et utilise le composant CF comme paramètre ;
- *Politique de Transformation* : Elle permet au composant domaine-polymorphe de changer sa structure et son comportement suivant les exigences du domaine hôte. Son implémentation dépend des technologies existantes en génie logiciel, du langage de programmation utilisé et des structures exigées par la plateforme dans laquelle nous utilisons le composant domaine-polymorphe. C'est pourquoi nous avons dégagé cet aspect de changement de structure sous forme d'une politique.
  Cependant, il est difficile d'implémenter cette politique pour que le changement de la structure et du comportement du composant domaine-polymorphe soit complètement automatisé à l'intérieur de la plate-forme de modélisation. Il est en général nécessaire de faire intervenir un utilisateur, ou au mieux un programme externe à la plateforme.

9.1.2 Modèle de composant domaine-spécifique flexible

Le composant domaine-spécifique flexible est un composant intermédiaire généré par notre approche de conception du composant domaine-polymorphe. A part la politique de transformation, le

composant domaine-spécifique flexible est composé des mêmes éléments que le composant domaine-polymorphe. Il possède donc les mêmes avantages, mais nécessite la création d'un composant Conteneur par domaine, alors que le composant domaine-polymorphe utilise un conteneur universel.

9.1.3 Hétérogénéité atomique

Les composants Conteneur et CASe représentent la couche adaptateur du composant domaine-polymorphe tandis que le composant CASi représente sa couche conservateur. Cela permet au composant domaine-polymorphe de fonctionner sous le modèle de calcul du domaine hôte tout en assurant un fonctionnement interne sous le modèle de calcul de spécification. Cette caractéristique fait de lui un composant atomique hétérogène. C'est l'hétérogénéité atomique.

L'hétérogénéité atomique n'est pas une approche concurrente aux autres approches hétérogènes, à savoir l'approche hiérarchique et l'approche non hiérarchique, mais est plutôt complémentaire. Dans le chapitre 6, nous avons, à travers des cas de mélange de modèles de calcul à base de composants domaine-polymorphes, montré que l'hétérogénéité atomique apporte d'autres avantages pour la conception des systèmes embarqués hétérogènes. Nous avons par exemple introduit un mode SDF (Synchronous Dataflow) relaxé, qui utilise des données de bourrage afin de satisfaire les conditions de réaction du composant. Ces données de bourrage doivent correspondre à quelque chose qui a un sens pour l'application concernée. Mais si cette application est conçue en mélangeant du SDF à un autre MoC, c'est que le concepteur a une idée de ce que cela signifie. Nous lui offrons les outils pour exprimer cette idée.

Aussi, contrairement aux autres approches hétérogènes où les changeurs de domaines sont placés entre les sous-modèles du système pour adapter leurs sémantiques (passer d'un modèle de calcul à un autre), l'hétérogénéité atomique permet un changement de domaine au niveau composant atomique, et cela, d'une façon automatique. De plus, l'hétérogénéité atomique *permet l'utilisation de modèles de calcul non supportés par les plateformes dans lesquelles le composant domaine-polymorphe est implémenté*.

9.1.4 Réalisations

Tous les concepts développés dans cette thèse ont été intégrés dans la plateforme Ptolemy II. Cependant, Ptolemy II est entièrement programmé en langage Java, qui a des types statiques, ce qui rend l'implémentation de la politique de transformation impossible. C'est pourquoi nous avons utilisé le composant domaine-spécifique flexible comme composant domaine-polymorphe à intégrer dans Ptolemy II.

Il est désormais possible de modéliser et de concevoir des systèmes hétérogènes à base de composants domaine-polymorphes dans Ptolemy II. De plus, nous avons aussi intégré nos concepts dans Vergil, qui est l'interface graphique de Ptolemy II, afin de rendre l'utilisation des composants domaine-polymorphes simple et rapide, et leur manipulation n'est basée que sur des actions simples : cliquer, glisser, déposer et éditer des paramètres.

Pour permettre l'utilisation de comportements spécifiés suivant la sémantique synchrone dans la conception de systèmes embarqués hétérogènes sous Ptolemy II, nous avons développé un traducteur permettant de générer des composants Ptolemy II, soit sous la forme de composants domaine-spécifiques synchrones, soit sous la forme de composants Noyaux synchrones, à partir de modules Esterel.

Pour la validation de nos concepts par simulation, nous avons modélisé l'atelier de production automatisé, qui est proposé par FZI (Forschungszentrum Informatik de Karlsruhe), comme un système hétérogène mélangeant le contrôle et la dynamique continue, à base de composants domaine-polymorphes.

La partie contrôle a été spécifiée suivant la sémantique synchrone, en utilisant le langage Esterel, tandis que la partie dynamique a été modélisée suivant le modèle de calcul CT (Continuous Time). Ensuite, les deux parties ont été assemblées (connectées) et la simulation du fonctionnement de l'atelier a été faite avec une visualisation en trois dimensions.

9.2 Perspectives

Pour ce qui concerne la suite du travail présenté dans cette thèse, nous envisageons les perspectives suivantes :

- étendre la spécification du composant CAS pour d'autres modèles de calcul. Cependant, nous rappelons que cette spécification n'est pas toujours possible pour certains modèles de calcul. Par exemple, le modèle de calcul FSM (Finite State Machines) fait partie des modèles de calcul pour lesquels il est impossible de spécifier le composant CAS.
- enrichir les composants CAS déjà spécifiés par l'ajout de paramètres et l'utilisation d'exceptions. Celles-ci auront le rôle de signaler le cas où l'utilisateur effectue des manipulations qui violent certaines règles ou la non compatibilité de deux sémantiques ;
- Dans le premier chapitre de cette thèse, nous avons vu que l'approche hétérogène non hiérarchique utilise des composants à interface hétérogène (HICs) et un modèle d'exécution hétérogène pour faire communiquer au même niveau des composants obéissants à des modèles de calcul différents. Cependant, à l'exécution, tout composant HIC a des composants projections qui le représentent sous les modèles de calcul qui régissent ses ports de communication hétérogènes. Pour fonctionner correctement sous son modèle de calcul, le composant projection doit être un composant domaine-spécifique. Comme le composant HIC et ses composants projections ont les mêmes comportements, nous pourrions faire du HIC un composant domaine-polymorphe afin d'éviter la redondance des comportements et augmenter ainsi la réutilisabilité. En effet, pour chaque composant HIC, nous devons utiliser autant de composants Conteneurs que de composants projections et un seul composant Noyau. Ainsi, les composants Conteneurs représentent les projections et le composant Noyau, qui représente le comportement hétérogène, sera utilisé par tous les composants Conteneurs, en le déplaçant avant la réaction d'un conteneur à un autre. Cependant, nous devons spécifier le composant CASi (CAS interne) hétérogène et spécifier toutes les opérations permettant au modèle de calcul hétérogène de déplacer le composant Noyau d'un Conteneur à un autre, et d'activer le composant domaine-polymorphe en tant que composant HIC.
  Pour ce qui concerne la spécification du composant CASi hétérogène, nous pensons que cette tâche est facile à réaliser car il ne s'agit que de transfert des données venant du composant CS (Connecteur des Sémantiques) au composant Noyau et vice versa.

Troisième partie

Annexes

Annexe A

Intégration des Modules ESTEREL dans PTOLEMY II

Dans cet annexe, nous présentons notre traducteur permettant l'intégration des modules Esterel dans Ptolemy II. Ce traducteur a été réalisé lors de mon stage de DEA [206] et amélioré au cours du travail de ma thèse.

A.1 Chaine de compilation d'Esterel

Avant, les versions v1, v2 et v3 du compilateur Esterel traduisent directement les modules Esterel en automates d'états finis [202]. Ces derniers permettent une représentation explicite dont les avantages sont :

- Les configurations d'états accessibles sont directement connues ;
- La traduction des automates en langages de programmation génériques (C, Java, etc.) est très facile ;
- Pour une certaine taille, les automates sont simples à lire par les humains.

A cause de leur temps de génération et de leur taille exponentielle, les automates ont été abandonnés à partir de la version 4 du compilateur Esterel au profit des circuits logiques. Cependant, à partir de ces derniers il est toujours possible de générer des automates.

Les circuits logiques sont générés par le compilateur Esterel depuis des modules Esterel en passant par plusieurs étapes de compilation.

La figure A.1 montre la chaîne de compilation du langage Esterel v5_21. En effet, les fichiers sources ".strl" codant des modules Esterel sont transformés séquentiellement aux formats suivants :

Format ic : Code intermédiaire obtenu à partir du code source ".strl" par le processeur *strlic* dont le rôle est de passer une analyse lexicale et syntaxique ;

Format lc : Code lié obtenu à partir du code ".ic" par le processeur *iclc*. Ce dernier a le rôle d'établir les liens ;

Format sc : Code représentant un circuit logique non trié obtenu à partir du code ".lc" par le processeur *lcsc* ;

Format ssc : Code représentant un circuit logique trié obtenu à partir du code ".sc" par l'un des deux processeurs, *scssc* et *sccausal*. Le rôle du processeur *scssc* est de trier les circuits acyclique ".sc" en circuit ".ssc" tandis que le rôle du processeur *sccausal* est de décycliser d'abord les circuits ".sc" puis de les trier en ".ssc" ;

Format blif : Le format BLIF (Berkeley Logical Interchange Format) est obtenu à partir du code ".ssc" par le processeur *sscblif*. Ce format peut être reconvertis en format ".ssc" par le processeur *blifssc*. Les circuits BLIF peuvent être vérifiés formellement par Xeve [107] ;

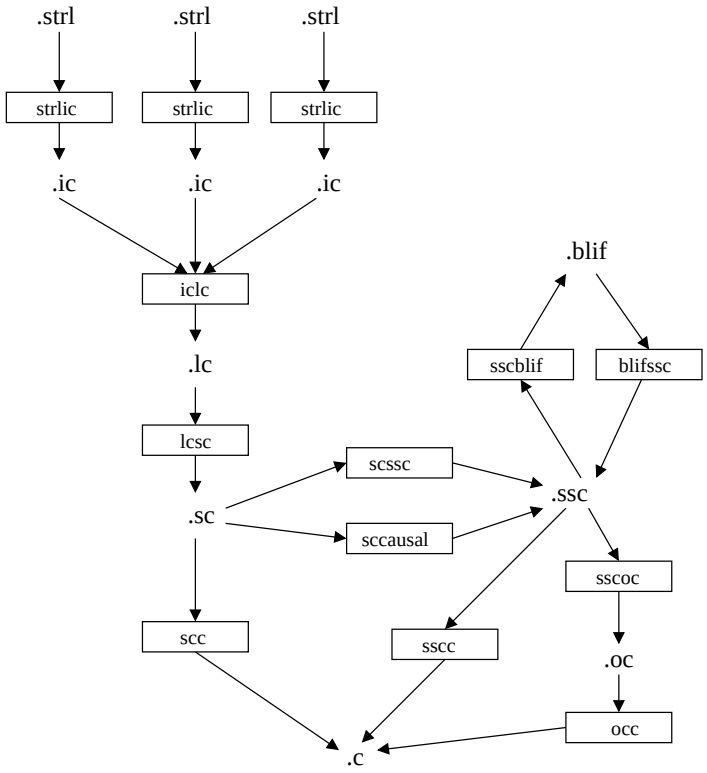


FIG. A.1: Chaîne de compilation d'Esterel.

Format oc : Le format ".oc" contient le code automate commun aux compilateurs Esterel et Lustre. Le processeur *sscoc* transforme le format ".ssc" en format ".oc" ;

Format c : C'est la traduction des modules Esterel en code C exécutable. Le code C est obtenu à partir : des circuits non triés ".sc" par le processeur *scc*, des circuits triés ".ssc" par le processeur *sscc* ou des automates ".oc" par le processeur *occ*.

Nous avons choisi le format ".ssc" à partir du quel nous générons les composants domaine-spécifiques synchrones (CDS Synchrones) et les composants Noyaux synchrones. Ce format est décrit en détail dans la section suivante.

A.2 Description du format ssc des modules Esterel

Tout circuit logique trié est décrit par un ensemble de table stocké dans un fichier textuel portant l'extension ".ssc". Chaque table a une entête, comporte un ensemble de lignes identifiées par des indexes et se termine par le mot-clé end:. Une en-tête est composée du nom de la table, du caractère «:» et du nombre de lignes de la table.

Les tables peuvent utiliser des objets prédéfinis, qui sont, eux même, décrits par des tables. Les objets prédéfinis sont les constates booléennes vrai et faux, les types de base boolean, integer, string, float et double, les fonctions logiques et arithmétiques appliquées sur les booléens et les types de base tel que le and logique, le or logique, l'addition, la soustraction, la division, la comparaison, etc. et les procédures d'affectation.

L'ensemble de tables décrivant tout circuit logique trié sont :

instances	types	constants
functions	procedures	signals
implications	exclusions	variables

tasks	execs	actions
halts	equations	

instances Cette table indique le module Esterel principal (dit aussi racine) et ses éventuels sous modules. L'entrée de cette table portant l'index 0 est toujours dédiée au module racine tandis et les éventuelles autres lignes sont associées aux sous modules. Une ligne (une entrée) dans cette table, qui est dédiée a un sous module, indique le nom du sous module et l'index du module racine qui l'utilise. Par contre, le module principal est indiqué par la ligne root: index_module_racine.

types Dans cette table on trouve tous les types utilisateurs (types prédéfinis non inclus) définis par le programmeur du module Esterel.

Chaque entrée (ligne) de cette table est associée à un type utilisateur et indique son nom et d'éventuels index de procédure et de fonctions appliquées à ce type.

constants Chaque entrée de cette table est associée à une constante (non booléenne) déclarée dans le programme Esterel en indiquant juste son nom ainsi que l'index de son type.

functions Chaque ligne de cette table est associée à une fonction manipulant un type (type prédéfini ou type utilisateur). Une fonction, elle aussi, peut être une fonction prédéfinie ou une fonction utilisateur et dans tous les cas son comportement n'est pas défini au niveau du module Esterel.

Dans une ligne, on trouve donc le nom de la fonction, la liste des index des types des arguments de la fonction et l'index du type de la valeur retournée par la fonction.

procedures Chaque ligne de cette table est associée à une procédure manipulant un type donné (prédéfini ou non). Cette dernière peut être une procédure utilisateur (non prédéfinie), dont le comportement est défini à l'extérieur du module Esterel par l'utilisateur, ou une procédure prédéfinie.

Dans une ligne, on trouve donc le nom de la procédure, la liste des index des types des arguments passés par référence et la liste des index des types des arguments passés par valeur.

actions Cette table regroupe plusieurs types d'actions. Chaque ligne est associée à une action. Parmi les types d'actions nous citons :

- present index_signal : Action tester la présence du signal ;
- dsz index_variable : Action décrémenter la valeur de la variable ;
- output index_signal : Action émettre le signal ;
- call index_procedure list_index_variables list_index_expressions : Action appeler la procédure ;
- reset index_variable : Action remettre la variable à l'état non initialisé ;
- Actions de contrôle d'exécution d'une tâche :
  start: index_task : Lancer l'exécution de la tâche index_task ;
  kill: index_task : Tuer la tâche index_task ;
  return: index_task : Pour récupérer des doonées après la fin normal de la tâche index_task ;
  suspend: index_task : Suspendre l'exécution de la tâche index_task ;
  activate: index_task : Reprendre l'exécution de la tâche index_task ;
- etc.

signals Cette table contient la description des signaux du module Esterel. Chaque entrée de la table est associée à un signal et est de la forme suivante nature nom_signal index_action canal [boolean] dont la signification est :

- nature : Ce champs indique la nature du signal. En effet, il peut prendre l'une des valeurs suivantes : input pour un signal d'entrée, output pour un signal de sortie, inputoutput pour un signal d'entrée/sortie, sensor pour un capteur, local pour un signal local, exception pour un signal d'exception ou return pour un signal de fin d'exécution d'une tâche ;
- nom_signal : Indique le nom du signal ;
- index_action : Il indique l'index de l'action dans la table d'actions associée au signal ;
- canal : Ce champs indique la valeur transportée par un signal. Il peut prendre l'une des valeurs suivantes : pure pour un signal pur, single suivie de l'index de la variable contenant la valeur pour un signal valué émit une fois par instant ou multiple suivie des index de la variable contenant la valeur du signal et de la fonction de combinaison des valeurs pour un signal valué pouvant être émit plusieurs fois par instant ;
- [boolean] : Ce champs est composé du mot clé bool suivi de l'index d'une variable booléenne. Son rôle dépend de la nature du signal. Par exemple, il permet de tester si un signal d'entrée ou d'entrée/sortie est présent.

implications Cette table indique les relations d'implication déclarées dans le module Esterel. Chaque entrée de la table est donc associée à une relation d'implication en indiquant l'index du signal maître et celui du signal esclave.

exclusions Cette table indique les relations d'exclusion entre les signaux déclarées dans le module Esterel. Chaque entrée de la table est donc associée à une relation d'exclusion et indique la liste d'index des signaux impliqués.

variables Cette table indique les variables utilisées dans le module Esterel et les variables générées pour les compteurs d'occurrences et pour les valeurs et les statuts des signaux. Chaque ligne de cette table indique l'index du type de la variable dans la table types et son éventuelle valeur initiale. Dans le cas où elle existe, la valeur initiale est précédée par le mot clé value.

tasks Chaque entrée de cette table indique le nom de tâche ainsi que ses deux listes d'index des types d'arguments. La première liste contient les index des types des arguments passés par référence tandis que la deuxième liste contient les index des types des arguments passés par valeur.

execs Cette table indique les tâches lancées par l'instruction exec. Chaque entrée contient l'index de la tâche à exécuter, l'index du signal return indiquant la fin d'exécution de la tâche et les deux listes d'arguments. La première liste indique les index des variables dans la table variables passées par référence et la deuxième liste indique les index des variables dans la table variables passées par valeur.

halts Cette table ne comporte que des pragmas, qui sont des commentaires spéciaux. Donc, cette table ne sera pas prise en compte par notre traducteur.

equations C'est cette table qui décrit le module Esterel sous forme d'un système d'équations et le tracé de ces dernières donne un circuit logique.

Chaque ligne de la table est associée à une équation. Cette dernière peut être :

a La définition d'une action conditionnée par un fil. Une telle équation est caractérisée par le mot clé act: suivi de l'index de l'action dans la table d'actions, du mot clé cond: et de l'index du fil conditionnant l'action. En effet, l'action n'est exécutée que si le fil de condition est à 1 ;

b La définition d'un fil : Il existe deux types de fil :
   – Les fils standards caractérisés par le mot clé wire: suivi d'une expression ;
   – Les fils de registres caractérisés par le mot clé reg0: ou reg1:. La valeur d'initialisation du registre reg0: (resp. reg1:) est 0 (resp. 1) et elle est suivie d'une expression ;

   Une expression est soit un and multiple qui commence par le mot clé and:, soit un or qui commence par le mot clé or:, soit un synonyme. Ce dernier est composé d'un seul index de fil tandis que les mots clé and: et or: sont suivis d'une liste d'index de fils entre parenthèses séparés par des virgules.

Les équations peuvent utiliser les deux fils prédéfinis qui ont des valeurs fixes. Le premier fil est toujours à 1 et noté $1 tandis que le deuxième fil est toujours à 0 et noté $0.

Exemple Le Listing A.1 montre le format ssc du module Esterel contrôleur de position d'un mobile donné dans le chapitre 8.

Listing A.1: Code ssc du module Esterel PositionController

```
1  ssc5:
2  module: PositionController
3
4  instances: 1
5  root: 0
6  0: PositionController 0 "" "PositionController.strl"
7  end:
8
9  constants: 2
10 0: desiredPosition $4
11 1: positiveDirection $0
12 end:
13
14 signals: 2
15 0: input: currentPosition 1 single: 0 bool: 1
16 1: output: positionReached 3 pure:
17 end:
18
19 variables: 2
20 0: $4 %sigval: 0% %lc: 10 7 0%
21 1: $0 %sigbool: 0% %lc: 10 7 0%
22 end:
23
24 actions: 7
25 0: call: $0 (1) (@$0)
26 1: present: 0 %lc: 10 7 0%
27 2: reset: 0 %lc: 10 7 0%
28 3: output: 1 %lc: 11 8 0%
29 4: if: @1 %lc: 17 2 0%
30 5: if: $43 (0,@0) %lc: 18 3 0%
31 6: if: $41 (0,@0) %lc: 22 3 0%
32 end:
33
34 halts: 2
35 0: %lc: 27 1 0%
36 1: %lc: 16 1 0%
37 end:
38
39 -- nets 35
40 -- optimised nets 13
41 equations: 22
```

```
42 registers: 2
43 return: 12 0
44 halting: 19
45 0: wire: and:(21,! 20)
46 1: wire: and:(0,@1)
47 2: wire: and:(20,! @1)
48 3: act:2 cond:2
49 4: wire: and:(20,@1)
50 5: wire: or:(1,4)
51 6: wire: and:(5, @4)
52 7: wire: and:(6, @5)
53 8: wire: and:(5,! 6)
54 9: wire: and:(8, @6)
55 10: wire: or:(7,9)
56 11: act:3 cond:10
57 12: wire: !$1
58 13: wire: and:(0,! @1)
59 14: wire: and:(21,13)
60 15: wire: and:(6,! 7)
61 16: wire: and:(8,! 9)
62 17: wire: and:(20,! @1)
63 18: wire: or:(7,14,15,9,16,17)
64 19: wire: 18
65 20: reg1: !$1
66 21: reg0: 18 halt:1
67 end:
68 endmodule:
```

A.3 Traduction du format ssc en composants Ptolemy II

Maintenant nous passons à la traduction du code ssc en classe Java codant un composant domaine-spécifique synchrone (CDS Synchrone) ou un composant Noyau synchrone.

Nous avons réalisé un traducteur qui prend en entrée le code ssc du module Esterel et des informations complémentaires en format XML et produit en sortie une classe Java.

Comme c'est montré par la figure A.2, notre traducteur sscptii est composé de deux processeurs :

– L'analyseur lexical/syntaxique : Son rôle n'est pas de vérifier si le code ssc ne comporte pas d'erreur lexicale et/ou syntaxique mais plutôt d'identifier les tables décrivant le circuit logique ainsi que la lecture de toutes les informations qu'elles contiennent. Cet analyseur utilise un parseur que nous avons généré, par l'outil JavaCC, à partir des entités lexicales et la grammaire du code ssc5 ;
– Le générateur de code Java : Son rôle est de générer la classe Java à partir des informations lues par l'analyseur lexical/syntaxique et celles fournies en format XML.

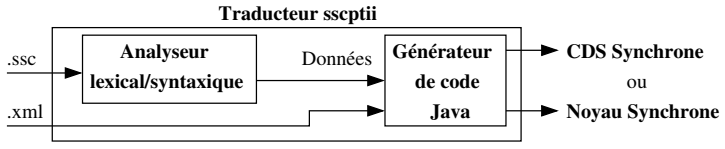


FIG. A.2: Traducteur du code ssc en CDS Synchrone/Noyau Synchrone.

Dans le reste de cette section nous décrivons d'abord les balises XML engendrant les informations complémentaires fournies à notre traducteur et ensuite nous donnons les traductions des différents objets du code ssc en code Java. Cependant, nous signalons que les objets des tables instances et halts représentent des informations complémentaires et en effet les deux tables ne seront pas prises en compte par notre traducteur.

Balises XML engendrant les informations complémentaires

Comme nous l'avons signalé avant, notre traducteur prend en entrée un fichier XML contenant des informations complémentaires et qui sont nécessaires pour une bonne traduction du code ssc en code Java. En effet, le listing A.2 montre les balises clé que nous avions définies afin de bien identifier desdites informations complémentaires fournies par l'utilisateur.

Listing A.2: Les balises XML des informations complémentaires sur le code ssc

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<info>
  <const name="nameOfConstant" parameter="yes/no" InitValue="initValue"/>
  <type name="nameOfUserType" class="pathOfUserType.classOfUserType"/>
  <function name="nameOfFunc" operator="op"/>
  <procedure name="nameOfProc" operator="proc"/>
  <task name="UserTask" fullTaskName="pathOfUserTask.ClassOfUserTask"/>
</info>
```

Les balises sont :

const : Celle balise regroupe des informations concernant une constante déclarée dans un module Esterel. Elle comporte trois attributs. L'attribut name indique le nom de la constante dans le module Esterel, l'attribut parameter indique si cette constante doit être traduite en un paramètre (yes) ou en une constante ordinaire (no) suivant la syntaxe du langage Java et finalement l'attribut InitValue indique la valeur d'initialisation de la constante ;

type : Cette balise comporte deux attributs. L'attribut name indique le nom du type utilisateur déclaré dans le module Esterel et l'attribut class indique le nom complet de la classe représentant ce type. Le nom complet d'une classe est composé du nom de son path (les packages) et son nom ;

function : Elle comporte l'attribut name, qui indique le nom de la fonction déclarée dans le module Esterel, et l'attribut operator, qui indique l'opération effectuée par la fonction ;

procedure : Elle comporte l'attribut name, qui indique le nom de la procédure déclarée dans le module Esterel, et l'attribut operator, qui indique l'opération effectuée par cette procédure ;

task : Cette balise comporte l'attribut name, qui indique le nom de la tâche déclarée dans le module Esterel, et l'attribut class , qui indique le nom complet de la classe représentant la tâche.

Traduction des types

Dans le langage Java, on trouve le type de base, qui peut être int, double, etc. et le type classe. En effet, les types de base du langage Esterel sont traduits implicitement en types de base du langage Java. Par contre, le type utilisateur est forcément un type de classe et il est traduit en une instruction de type import.

Donc, suivant les informations complémentaires, nous résumons comme suit :

Code Esterel & informations	⟶	Code Java
type nomType; <type name="nomType" class="cheminType.classType"/>	⟶	import cheminType.classType;

Traduction des constantes

Suivant les informations complémentaires, nous avons traduit une constante comme suit :

Code Esterel & informations	⟶	Code Java

const nomConst : typeDeConst;		

<const name="nomConst" parameter="yes" InitValue="valInit"/>	⟶	//*Constante traduite en un Parameter*
		public Parameter nomConst;
		// *Création du Parameter*
		nomConst = new Parameter(this, "nomConst");
		// *Attribuer au paramètre le type de la constante*
		nomConst.setTypeEquals(BaseType.TYPECONSTANTE);
		// *Initialisation*
		nomConst.setToken(new TypeConstanteToken(valInit));

<const name="nomConst" parameter="yes" InitValue="valInit"/>	⟶	//*Constante traduite en une constante normale* final TypeConstante nomConst = valInit;

Nous rappelons que le type de la constante est lue par l'analyseur lexical/syntaxique à partir du code ssc.

Traduction des functions

Les fonctions prédéfinies par le compilateur Esterel sont traduites en opérations basées sur les opérateurs qui les correspondent. Par contre, pour les fonctions utilisateurs, nous avons pris en compte que les opérateurs Java non prédéfinis par le compilateur Esterel. En effet, pour une fonction non prédéfinie, l'utilisateur doit indiquer dans l'attribut operator de la balise XML function le symbole opérateur correspond à cette fonction.

Nous signalons qu'il est possible de prendre en compte les fonctions utilisateurs désignant des méthodes de classe retournant des valeurs y compris les constructeurs des classes.

Voici une liste non exhaustive de fonctions prise en compte par notre traducteur :

Symbole d'opérateur	Opérateur dans Java	Description
eq	==	Test d'égalité
ne	!=	Test de non égalité
not	!	Négation
or	||	ou logique
and	&&	and logique
lt	<	Test inférieur
le	<=	Test inférieur ou égal
gt	>	Test supérieur
ge	>=	Test supérieur ou égale
plus	+	Addition
minus	-	Soustraction
times	*	Multiplication
cond	if ... else ...	Renvoi valeur suivant la condition
mod	%	Modulo
div	/	Division
uminus	-	Moins unaire
bitor	|	Or appliqué aux bits
bitand	&	And appliqué aux bits
bitnot	~	Not appliqué aux bits
ls	«	Décalage à gauche
rs	»	Décalage à droite

Traduction des procedures

Il n'existe qu'une seule procédure prédéfinie par le compilateur Esterel, c'est la procédure d'affectation (assignment) que nous avions traduite en opération d'affectation suivant la syntaxe du langage Java. Par contre, pour les procédures utilisateurs, nous avons pris en compte que les procédures qui combinent l'opérateur d'affectation avec d'autres opérateurs. Ces procédures sont traduites en opérations basées sur les opérateurs correspondent. En effet, l'utilisateur doit indiquer dans l'at-

tribut operator de la balise XML procedure le symbole d'opérateur correspondant au nom de la procédure.

Egalement, ici aussi, nous signalons qu'il est possible de prendre en compte une procédure utilisateur désignant des méthodes de classe de type void.

Voici une liste non exhaustive de procédures prise en compte par notre traducteur :

Symbole d'opérateur	Opérateur dans Java	Description
inc	++	Incrémentation
dec	–	Décrémentation
addass	+=	Addition puis affectation
subass	-=	Soustraction puis affectation
mulass	*=	Multiplication puis affectation
modass	%=	Modulo puis affectation
divass	/=	Division puis affectation

Traduction des actions

L'exécution d'une action est déclenchée par la table d'equations. En effet, il existe des actions qui rendent des valeurs booléennes, qui sont symbolisées par les valeurs 1 et 0 et respectivement correspondent aux valeur true et false, par exemple l'action present rend la valeur 1 ou 0, et des actions qui ne rendent pas de valeurs après leur exécution.

Ici, nous donnons la traduction de quelques actions.

Action present Dans le code Java, chaque signal d'entrée et/ou de sortie est lui attribué une variable indiquant son état (present ou absent) et une éventuelle variable indiquant la valeur portée dans le cas ou il s'agit d'un signal valué.

Donc, cette action est traduite en une opération de lecture depuis l'entrée dédiée au signal index_signal, des opérations de mise à jour des variables d'état et de valeur portée, une opération de test de la variable d'état pour savoir si le signal est présent ou absent.

Dans le cas de génération du CDS Synchrone, les données lues seront interprétées avant la mise à jour des variables d'état et de valeur portée par le signal.

Action output Cette action est traduite en opération d'écriture sur la sortie dédiée au signal index_signal. Si le signal est un signal pur, l'opération d'écriture consiste à envoyer une valeur booléenne true ou l'entier 1 pour indiquer au récepteur la présence du signal. Par contre, pour le signal valué, l'opération d'écriture consiste à envoyer sa valeur portée.

De même pour les actions output, dans le cas de génération du CDS Synchrone, les données à envoyer seront interprétées avant leur écriture sur les ports de sortie.

Action dsz Nous avons traduit cette action en une méthode qui prend en entrée la valeur de la variable index_variable, décrémente la valeur de cette variable et rend la valeur 1 (resp. 0) si la valeur dudite variable est nulle (resp. non nulle).

Action reset Cette action est traduite en une no-op car la remise d'une variable à l'état initiale, qui est exprimé par une valeur inconnue, consiste à lui affecter une valeur quelconque.

Actions de contrôle d'exécution des Tâches D'une part, avec le langage Esterel, une tâches est lancée par le module Esterel ainsi les deux s'exécutent en parallèle. Et, le contrôle d'exécution de la tâche est assuré à travers les cinq actions citées précédemment. D'autre part, un programme Java, lui aussi, peut lancer des processus légers, appelés threads, s'exécutant parallèlement à lui et qui peut les contrôler par les opérations *start()*, *notify()*, *destory()* et *wait()* prédéfinies par le langage Java. En effet, une tâche dans le module Esterel est traduite en un thread Java et les actions de contrôle des tâches sont traduites en opérations de contrôle des threads. Pour cela, tous les threads définis par l'utilisateur doivent hériter de la classe abstraite TaskAsynch que nous avions définie. Cette classe hérite de la classe Java Thread dans laquelle nous avons défini les attributs pour les arguments passés par référence, les arguments passés par valeur et le signal return, voir listing A.3.

Listing A.3: Classe mère des tâches Esterel dans Java

```
import java.lang.*;
import java.util.*;

public abstract class TacheAsyn extends Thread{
  // Pour indiquer l'etat du signal return
  public boolean _return = false;

  // argument passes par valeur a la tahce
  public FileParam paramIn = null;

  // argument passes par reference a la tahce
  public FileParam paramOut = null;

  // Pour savoir si la tâche est
  // en activité ou en attente
  public boolean drapeau = false;

  //Pour signaler la fin d'execution
  public void finishedTask() {
    _return = true;
  }
}
```

Nous signalons que l'action present du signal return est traduite en une opération de test de la valeur de l'attribut du thread associé au dit signal return. Si la valeur est true alors l'exécution du thread est arrivée à son terme sinon le thread est en cours d'exécution. Cette valeur est initialisée à false par le CDS Synchrone ou le Noyau Synchrone tandis que sa mise à jour à true est réalisée par le thread à la fin de son exécution par la méthode finishedThread().

Traduction des signals

Les signaux concernés par la traduction sont les signaux d'entrée et/ou de sortie. En effet, un signal est traduit en un port au niveau du CDS Synchrone tandis qu'au niveau du composant Noyau Synchrone il est traduit en un élément de la liste passée au composant Conteneur pour la création des ports lors de la phase d'assemblage. Cet élément contient toutes les données sur le signal à savoir sa nature, son nom, son type, etc.

Traduction des implications et des exclusions

Les contraintes de dépendance imposées sur les signaux d'entrée, l'exclusion et l'implication, doivent être respectées et le seul problème qui peut les violer est bien l'asynchronisme de l'ordre de leur arrivée. Comme solution, nous avons implémenté un séquenceur limité à contrôler la présence des signaux liés par des contraintes et qui consiste tout simplement à lever une exception dans le cas où les contraintes de dépendances sont violées. Cette solution est sévère et pour la relaxer, nous avons ajouté des opérations de mémorisation au séquenceur pour retarder les signaux. Cependant, il faut associer à ce genre de signaux un ordre de priorités pour pouvoir choisir les signaux à retarder au lieu de faire des choix aléatoires et d'avoir des conséquences d'exécutions différentes.

Traduction des variables

Nous avons traduit chaque variable du code ssc en une variable déclarée et utilisée suivant la syntaxe du langage Java. Toute variable traduite porte au niveau du code Java un nom composé de la chaîne de caractères «_var» et de son index dans la table de variables.

Traduction des equations

Nous savons qu'il existe deux types d'équation, équation de type fil et équation de type action. L'équation de type fil est une équation contenant une expression d'évaluation de la valeur du fil tandis que l'équation de type action indique une action conditionnée par un fil.

Avant de donner la traduction des différentes équations, nous donnons d'abord les méthodes utilisées dans l'évaluation des expressions basées sur and, or et !. Ces méthodes utilisent les valeurs 1, 0 et -1 (pour prendre en compte le mode non strict), voir le listing A.4.

Listing A.4: Les méthodes d'évaluation des expréssions

```
////////////////////////////////////////
///        logics functions            ///
private int andLogic(int a, int b) {
  if((a == 1) && (b == 1))
    return 1;
  if((a == 0) || (b == 0))
    return 0;
  return -1;
}

private int orLogic(int a, int b) {
  if((a == 1) || (b == 1))
    return 1;
  if((a == 0) && (b == 0))
    return 0;
  return -1;
}

private int notLogic(int a) {
  if(a == 0)
    return 1;
  if(a == 1)
    return 0;
  return -1;
}
```

Les fils sont déclarés dans le code Java comme variables. En effet, dans le code Java, un fil standard portera un nom composé de la chaîne de caractères «_wire» et son index tandis qu'un fil de type registre portera un nom composé de la chaîne de caractères «_reg» et son index.

Une équation est donc traduite en :

- *_wireIndexFil = évaluation expression* pour un fil standard ;
- *_regIndexFil = évaluation expression* pour un fil de type registre ;
- *if(_wireIndexFil) exécuter action indexAction* pour une équation de type action.

Répartition du code traduit sur les méthodes d'exécution

Dans les méthodes d'initialisation initialize() et configuration() des composants respectivement CDS Synchrone et Noyau Synchrone, nous avons placé les instructions d'initialisation des états des signaux (een entrée et/ou en sortie), des registres, des fils et d'autres éléments dépendant du module Esterel traduit. Les états des signaux et les fils sont initialisés à 0 pour le mode strict et à -1 pour le mode non strict tandis qu'un registre est initialisé à 0 s'il est de type reg0 et à 1 s'il est de type reg1.

Dans la méthode prefire() (resp. pretreatment()) du CDS Synchrone (resp. Noyau Synchrone), nous avons placé le code traduisant les actions de lecture des entrées correspondant aux signaux d'entrée et dans la méthode fire() (resp. treatment()), nous avons placé le code traduisant les équations de type action et de type fil standard. Par contre, dans la méthode postfire() (resp. postreatment()), nous avons placé le code traduisant les équations de type registre, qui représente la mise à jour de l'état du module synchrone, avec les instructions de réinitialisation des états des signaux et les fils.

Exemple de traduction

Le listing A.5 donne le code Java du composant Noyau Corepositioncontroller généré par notre traducteur à partir du code ssc du module Esterel PositionController donné par le listing A.1.

Listing A.5: Code Java du composant Noyau Synchrone Corepositioncontroller

```
/*
 - File name :  Corepositioncontroller.java
 - Date and hourse of creation :  Friday , 1-7-2004 at 18:33
 - From :      PositionController.ssc file
 - By Esterel&Ptolemy Translator 2.0
 - Author : M. Mohamed FEREDJ
 - For any information, please write me at the following adress
     Mohamed.Feredj@supelec.fr
*/

package ptolemy.polymorphcom.demo.modalmodelwithsr;
import ptolemy.polymorphcom.kernel.util.CorePortsList;
import ptolemy.polymorphcom.kernel.util.CorePort;
import ptolemy.polymorphcom.kernel.Core;
import ptolemy.data.*;
import ptolemy.data.type.*;
import ptolemy.data.expr.Parameter;
import ptolemy.kernel.Port;
import ptolemy.kernel.Entity;
import ptolemy.kernel.util.IllegalActionException;
import ptolemy.kernel.util.NameDuplicationException;
import ptolemy.kernel.util.NamedObj;
import java.util.List;

public class Corepositioncontroller extends Core {
  //////////////////////////
  //       Constructor      //
  public Corepositioncontroller() {
    _cPL = new CorePortsList();
    _cPL.append(new CorePort(currentPosition, true, false, "double"));
    _cPL.append(new CorePort(positionReached, false, true, "int"));
  }

  ////////////////////////////////////////////
  //   supports of communication & Parameters   //
  CorePortsList _cPL;
  public Parameter desiredPosition;
  public Parameter positiveDirection;
  String currentPosition = "currentPosition";
  String positionReached = "positionReached";

  ///// configuration method
  public void configuration() throws IllegalActionException {
    if(((Entity)_containerCore).getAttribute("__nonStrictMarker") != null)
```

```
      _nonStrict = -1;
    else
      _nonStrict = 0;

    initializeWires();

    _reg20 = 1;
    _reg21 = 0;

    authorizeActions();


    initializeSignals();

    _sigIn_currentPosition_Value = 0;
  }

  //// pretreatment method
  public boolean pretreatment() throws IllegalActionException {
    // Action "present"
    if(_com.isKnown(currentPosition) && !_notBefore0) {
      if(_com.hasInputData(currentPosition)) {
        _token = _com.take(currentPosition);
        _sigIn_currentPosition_Statut = 1;
        if(_sigIn_currentPosition_Statut > 0)
          _sigIn_currentPosition_Value = ((DoubleToken)_token).doubleValue();
      }else
        _sigIn_currentPosition_Statut = 0;
      _notBefore0 = true;
    }

    return super.pretreatment();
  }

  //// treatment method
  public void treatment() throws IllegalActionException {
    _wire0 = andLogic(_reg21, notLogic(_reg20));
    _wire1 = andLogic(_wire0, _sigIn_currentPosition_Statut);
    _wire2 = andLogic(_reg20, notLogic(_sigIn_currentPosition_Statut));

    if(_action2 == 0) {
      if(_wire2 > 0) {
        _action2 = 1;
        // reset type action to affect the variable state at not initialized
        // does not do anything
      }
    }
    _wire4 = andLogic(_reg20, _sigIn_currentPosition_Statut);
    _wire5 = orLogic(_wire1, _wire4);
    _wire6 = andLogic(_wire5, evaluationActionIf4());
    _wire7 = andLogic(_wire6, evaluationActionIf5());
    _wire8 = andLogic(_wire5, notLogic(_wire6));
    _wire9 = andLogic(_wire8, evaluationActionIf6());
    _wire10 = orLogic(_wire7, _wire9);

    if(_action3 == 0) {
      if(_wire10 > 0) {
        _action3 = 1;
        _com.emit(positionReached, new IntToken(1));
      }
    }
    _wire12 = notLogic(1);
    _wire13 = andLogic(_wire0, notLogic(_sigIn_currentPosition_Statut));
    _wire14 = andLogic(_reg21, _wire13);
    _wire15 = andLogic(_wire6, notLogic(_wire7));
    _wire16 = andLogic(_wire8, notLogic(_wire9));
    _wire17 = andLogic(_reg20, notLogic(_sigIn_currentPosition_Statut));
    _wire18 = orLogic(orLogic(orLogic(orLogic(orLogic(_wire7, _wire14),
      _wire15), _wire9), _wire16), _wire17);
    _wire19 = _wire18;
  }

  //// posttreatment method
  public boolean posttreatment() throws IllegalActionException {
    _reg20 = notLogic(1);
```

```
    _reg21 = _wire18;
    initializeWires();
    initializeSignals();
    authorizeActions();
    return super.posttreatment();
  }

  //// putFinalData method
  public void putFinalData() throws IllegalActionException {
  }

  /**
  * Creates the parameters of the core.
  */
  public void createParameters(NamedObj containerPar)
    throws NameDuplicationException, IllegalActionException {

    desiredPosition = new Parameter(containerPar, "desiredPosition");
    desiredPosition.setTypeEquals(BaseType.DOUBLE);
    desiredPosition.setToken(new DoubleToken(0.0));
    positiveDirection = new Parameter(containerPar, "positiveDirection");
    positiveDirection.setTypeEquals(BaseType.BOOLEAN);
    positiveDirection.setToken(new BooleanToken(true));
  }

  ////////////////////////////////////////////////////////////////
  /// Implementation of declared methods in ConnectionCore interface ///

  /**
  * Return the list of ports to create for this Core.
  */
  public CorePortsList getCoreInputOutputPorts() {
    return _cPL;
  }

  ////////////////////////////////////////
  ///       logics functions           ///

  private int andLogic(int a, int b) {
    if((a == 1) && (b == 1))
      return 1;
    if((a == 0) || (b == 0))
      return 0;
    return -1;
  }

  private int orLogic(int a, int b) {
    if((a == 1) || (b == 1))
      return 1;
    if((a == 0) && (b == 0))
      return 0;
    return -1;
  }

  private int notLogic(int a) {
    if(a == 0)
      return 1;
    if(a == 1)
      return 0;
    return -1;
  }

  ////////////////////////////////////////
  ///   Function to initialize the wires   ///
  private void initializeWires() {
    _wire0 = _nonStrict;
    _wire1 = _nonStrict;
    _wire2 = _nonStrict;
    _wire4 = _nonStrict;
    _wire5 = _nonStrict;
    _wire6 = _nonStrict;
    _wire7 = _nonStrict;
    _wire8 = _nonStrict;
    _wire9 = _nonStrict;
    _wire10 = _nonStrict;
```

```
    _wire12 = _nonStrict;
    _wire13 = _nonStrict;
    _wire14 = _nonStrict;
    _wire15 = _nonStrict;
    _wire16 = _nonStrict;
    _wire17 = _nonStrict;
    _wire18 = _nonStrict;
    _wire19 = _nonStrict;
  }

  ////////////////////////////////////////////////////
  ///   Function to authorize the execution of actions   ///
  private void authorizeActions() {
    _action2 = 0;
    _action3 = 0;
  }

  private void initializeSignals() {
    _sigIn_currentPosition_Statut = _nonStrict;
    _notBefore0 = false;
    _sigOut_positionReached_Statut = _nonStrict;
  }

  ///
  /// Boolean evaluation of the if action
  ///
  private int evaluationActionIf4() throws IllegalActionException {
    if(((BooleanToken)(positiveDirection.getToken())).booleanValue())
      return 1;
    return 0;
  }

  ///
  /// Boolean evaluation of the if action
  ///
  private int evaluationActionIf5() throws IllegalActionException {
    if((_sigIn_currentPosition_Value >=
     ((DoubleToken)(desiredPosition.getToken())).doubleValue()))
      return 1;
    return 0;
  }

  ///
  /// Boolean evaluation of the if action
  ///
  private int evaluationActionIf6() throws IllegalActionException {
    if((_sigIn_currentPosition_Value <=
      ((DoubleToken)(desiredPosition.getToken())).doubleValue()))
      return 1;
    return 0;
  }

  ////////////////////////////////////////////////////
  //             declaration of variables                //
  private int _nonStrict;
  private int _wire0;
  private int _wire1;
  private int _wire2;
  private int _wire4;
  private int _wire5;
  private int _wire6;
  private int _wire7;
  private int _wire8;
  private int _wire9;
  private int _wire10;
  private int _wire12;
  private int _wire13;
  private int _wire14;
  private int _wire15;
  private int _wire16;
  private int _wire17;
  private int _wire18;
  private int _wire19;
  private int _reg20;
  private int _reg21;
```

```
    private int _action2;
    private int _action3;

    private Token _token;

    private int _sigIn_currentPosition_Statut;
    private boolean _notBefore0;
    private double _sigIn_currentPosition_Value;
    private int _sigOut_positionReached_Statut;
}
```

Bibliographie

[1] *http ://www.cert.fr/francais/deri/boniol/Papiers_pdf/RTS2001.pdf*

[2] 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.

[3] RNTL. *système embarqué et temps réel, codéveloppement.* Rapport du groupe de travail. 2001. Disponible sur http ://www.telecom.gouv.fr/rtnl

[4] 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.

[5] J. Martin. *Design of real-time systems.* Prentice Hall, Englewood Cliffs, NJ, 1965.

[6] S. Young. *Real Time Languages : Design and development.* Ellis Horwood Publishers, 1982.

[7] 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.

[8] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, H. Zheng (eds.). *Heterogeneous Concurrent Modeling and Design in Java (Volume 1 : Introduction to Ptolemy II).* Technical Memorandum UCB/ERL M05/21, University of California, Berkeley, CA USA 94720, July 15, 2005.

[9] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, H. Zheng (eds.). *Heterogeneous Concurrent Modeling and Design in Java (Volume 2 : Ptolemy II Software Architecture).* Technical Memorandum UCB/ERL M05/22, University of California, Berkeley, CA USA 94720, July 15, 2005.

[10] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, H. Zheng (eds.). *Heterogeneous Concurrent Modeling and Design in Java (Volume 3 : Ptolemy II Domains).* Technical Memorandum UCB/ERL M05/23, University of California, Berkeley, CA USA 94720, July 15, 2005.

[11] R. Hamouche. *Modélisation des Systèmes Embarqués à base de Composants et d'Aspects.* Thèse de doctorat de l'université d'Every Val d'Essonne. Juin 2004.

[12] M. Mbobi. *Modélisation Hétérogène Non-Hiérarchique.* Thèse de doctorat de l'université Paris XI Orsay. Décembre Juin 2004.

[13] E. M. Clarke, O. Grumberg, D. A Peled. *Model Checking.* The MIT Press, janvier 2000. ISBN 0-262-03270-8.

[14] 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.

[15] 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.

[16] Goel. *Process Network in Ptolemy II.* Technical Memorandum UCB/ERL M98/69, EECS Department, University of California at Berkeley, December 16, 1998.

[17] E. A. Lee and T. M. Parks. *Dataflow Process Networks.* Proceedings of the IEEE, vol. 83, no. 5, pp. 773-801, May, 1995.

[18] E. A. Lee and D. G. Messerschmitt. *Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing.* IEEE Trans. on Computers, January, 1987.

[19] L. Muliadi. *Discrete Event Modeling in Ptolemy II.* Technical Report, Dept. of EECS, University of California, Berkeley, CA 94720, May 1999.

[20] John Davis II. *Order and Containment in Concurrent System Design.* Ph.D. Thesis, Memorandum UCB/ERL M00/47, Electronics Research Laboratory, University of California, Berkeley, September 8, 2000.

[21] 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.

[22] A. Benveniste and G. Berry. *The synchronous approach to reactive and real-time systems.* Proceedings of the IEEE, 79(9), pages 1270-1280, 1991.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] A. Girault, B. Lee, and E. A. Lee. *Hierarchical Finite State Machines with Multiple Concurrency Models.* IEEE Transaction On Computer-Aided Design of Integrated Circuits and Systems, 18(6), Juin 99.

[28] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. *Ptolemy : A Framework for Simulating and Prototyping Heterogeneous Systems.* International Journal of Computer Simulation, special issue on Simulation Software Development, vol. 4, pp. 155-182, April, 1994.

[29] *http ://www.moto-net.com/.*

[30] M. Mbobi, F. Boulanger, M. Feredj, *Non-hierarchical heterogeneity*. International Conference on Computer, Communication and Control Technologies, CCCT'03 31 juillet - 2 août 2003, Orlando, Floride, USA. International Institute of Information and Systemics Volume III, ISBN 980-6560-05-1, p. 430-435.

[31] F. Boulanger, M. Mbobi and M. Feredj. *Flat Heterogeneous Modeling.* In Proceedings of the conference of Internet Processing Systems Interdisciplinaries, Venice, Italy, November 10 to 15, 2004.

[32] R. Alur, T. Dang, J. Esposito, Y. Hur, F. Ivancic, V. Kumar, I. Lee, P. Mishra, G. J. Pappas, and O. Sokolsky. *Hierarchical modeling and analysis of embedded systems.* In Proceedings of the IEEE, 91(1) :11-28, January 2003.

[33] W-T. Chang, S. Ha, and E.A. Lee. *Heterogenous simulation - mixing discreteevent models with dataflow.* Journal of VLSI Signal Processing 15, 127-144, Kluwer Academic Publishers, 1997.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] M. Mbobi, F. Boulanger and M. Feredj. *Execution Model for Non-Hierarchical Heterogeneous Modeling.* 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, ISBN 2004113902, pp 139-144.

[40] S. E. Mattsson and M. Anderson. *The ideas behind omola.* In CACSD 92 : IEEE Symp. Comput. Aided Control Syst. Design, 1992, pp. 23-29.

[41] J. Buck and R., *Vaidyanathan. Heterogenous modeling and simulation of embedded systems in el greco.*. Technical report, Synopsys Inc.

[42] *http ://www.specc.org/.*

[43] *http ://www.systemc.org/.*

[44] E. Christen, K. Bakalar, E. Moser. *Analog and Mixed-Signal Modeling using the VHDL-AMS Language.* 36th Design Automation Conference, New Orleans, June 1999.

[45] *Rosetta 2004.* Valable sur le site http ://www.sldl.org/

[46] *Polis Homepage.* Berkeley University, Hardware/Software design group, 2004, Availlable on line at http ://www-cad.eecs.berkeley.edu/ polis/.

[47] H. Elmqvist, F. E. Cellier, and M. Otter. *Object-oriented modeling of hybrid systems.* In Proceedings of Eur. Simulation Symp., 1993, pp 31-41.

[48] H. Elmqvist, S. E. Mattsson, and M. Otter. *Modelica-The new object-oriented modeling language.* In Proceedings of 12th Eur. Simulation Multiconf., 1998, pp 127-131.

[49] G. Yang, Y. Watanabe, F. Balarin, A. Sangiovanni-Vincentelli. *Separation of Concerns : Overhead in Modeling and Efficient Simulation Techniques.* Fourth ACM International Conference on Embedded Software (EMSOFT'04), september 2004.

[50] *http ://www.mathworks.com*

[51] *http ://www.eecs.berkeley.edu/ polis*

[52] F. Balarin et al. *Hardware-Software Codesign of Embedded Systems, The POLIS Approach.* Kluwer Academic Publishers, 1997.

[53] *SystemC Transaction Level ModelingWorking Group Charter.* June 2003, http ://www.systemc.org/.

[54] *Coware N2C Homepage.* Coware Inc. 2004. http ://www.coware.com

[55] P. Runstadler, R. Crevier. *Virtual Prototyping for System Design and Verification.* Synopsys documentation, 1995.

[56] 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.

[57] *http ://www.ni.com/labview/*

[58] *http ://www.cadence.com/*

[59] *www.boeing.com/assocproducts/easy5/*

[60] Edward A. Lee. *Embedded Software.* Advances in Computers (M. Zelkowitz, editor), Vol. 56, Academic Press, London, 2002.

[61] 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.

[62] 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.

[63] Edward A. Lee and Stephen Neuendorffer. *Actor-oriented models for codesign* In Sandeep Shukla and Jean-Pierre Talpin, editors, Formal Methods and Models for System Design. Kluwer, 2004. to appear.

[64] Edward A. Lee, Stephen Neuendorffer, and Michael J. Wirthlin. *Actor-oriented design of embedded hardware and software systems.* Journal of Circuits, Systems, and Computers, 12(3) :231-260, June 2003.

[65] C. Hewitt. *Viewing control structures as patterns of passing messages.* Journal of Artificial Intelligence, 8(3) :323-363, June 1977.

[66] 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.

[67] 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.

[68] Stephen Neuendorffer. *Actor-Oriented Metaprogramming* PhD Thesis, University of California, Berkeley, December 21, 2004.

[69] 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.

[70] A.A Jerraya et al. *Multilanguage specification for system design and codesign.* Disponible sur http ://tima.imag.fr/publications/files/rr/mss_54.pdf

[71] M. Mbobi, F. Boulanger, M. Feredj. *Integration of a Flat Heterogeneous Domain in Ptolemy II.* The Sixth Biennial Ptolemy Conference, University of California Berkeley, California, USA, May 12, 2005.

[72] M. Mbobi, F. Boulanger, M. Feredj *An Approach of Flat Heterogeneous Modeling based on Heterogeneous Interface Components*. Soumis à Special Issue of the IEEE Transaction on Software Engineering on Interaction and State-based Modeling. Avril 2005.

[73] M. Mbobi, F. Boulanger, M. Feredj *Issues of Hierarchical Heterogeneous Modeling in Component Reusability* The 2005 IEEE International Conference on Information Reuse and Integration (IEEE IRI 2005), Las Vegas, USA, August 15-17, 2005.

[74] M. Mbobi, F. Boulanger, M. Feredj. *Le paradigme acteur dans la modélisation des systèmes hétérogènes.* Soumis à la Revue des Sciences et Technologies de l'Information, Hermès, Numéro spécial.

[75] Chamberlain Fong, *Discrete-Time Dataflow Models for Visual Simulation in Ptolemy II*. Electrical Engineering and Computer Sciences, University of California, Berkeley. UCB/ERL M01/9, December 2000.

[76] E. A. Lee and D. G. Messerschmitt, *Synchronous Data Flow*. Proc. of the IEEE, vol. 75, no. 9, pp. 1235-1245, September, 1987.

[77] S. A. Edwards, *The Specification and Execution of Heterogeneous Synchronous Reactive Systems*. Ph.D. thesis, University of California, Berkeley, May 1997. Available as UCB/ERL M97/31.

[78] Jie Liu and Edward A. Lee, *Component-based Hierarchical Modeling of Systems with Continuous and Discrete Dynamics*. IEEE Symposium on Computer-Aided Control System Design (CACSD'00), Anchorage, Alaska, USA, Sep. 2000.

[79] Banks, J. S. Carson, B. L. Nelson, and D. M. Nicol, *Discrete Event System Simulation*. Prentice Hall, August 15, 2000.

[80] N. Smyth. *Communicating Sequential Processes in Ptolemy II*. Memorandum No. UCB/ERL M98/70 15 Decembre 1998. Electronics Research Laboratory. College of Engineering, University of California, Berkeley 94720.

[81] Jie Liu. *Responsible Frameworks for Heterogeneous Modeling and Design of Embedded Systems.* Ph.D. thesis, Technical Memorandum UCB/ERL M01/41, University of California, Berkeley, CA 94720, December 20th, 2001.

[82] Johan Ecker. *Heterogeneous Modeling and Design in Ptolemy II* Powerpoint Presentation. ECE Seminar Series, Carnegie Mellon, November 29, 2001. Available at http ://ptolemy.eecs.berkeley.edu/presentations/01/

[83] *Modeling Reactive Systems with StateCharts : The StateMate Approch.* iLogix Inc. 1996.

[84] H. Boufaïed. *Machines d'exécution pour les langages synchrones.* Thèse de doctorat, Université de Nice-Sophia Antipolis, novembre 1998.

[85] D. Harel. *Analyse Statecharts : A Visual Formalism for Complex Systems.* Science of Computer Programming, 8 :213-274, 1987.

[86] D. Harel, A. Naamad. *The Statemate Semantics of Statecharts.* ACM Transaction Software Engineering. Method, 5 (4) :477-498, 1996.

[87] F. Maraninchi. *The Argos Language : Graphical Representation of Automata and Description of Reactive Systems.* IEEE Workshop on Visual Languages, October 1991.

[88] F. Maraninchi. *Operational and Compositional Semantics of Synchronous Automaton Compositions.* Third International Conference on Concurrency Theory, August 1992.

[89] G. Berry. *The Esterel v5 language primer. Version 5_91.* Manuel de référence, Centre de Mathématiques Appliqués, Ecole des Mines et INRIA, 2000.

[90] G. Berry, P. Couronné, G. Gonthier. *Programmation synchrone des systèmes réactifs : le langage Esterel.* In Technique et Science Informatique (TSI). Vol. 6(4), pp. 305-316. Hermès, 1987.

[91] G. Berry. *The Foundations of Esterel.* MIT Press, 1998. Edited by C. Stirling, G. Plotkin and M. Tofte.

[92] G. Berry. *The Constructive Semantics of Pure Esterel.* Draft / Not yet published, July 1999.

[93] P. Bournai, B. Chéron, T. Gautier, B. Houssais, P. Le Guernic. *SIGNAL manual.* Manuel de référence, IRISA (Rennes). Juillet 1993.

[94] C. André. *Modélisation de systèmes réactifs par une approche graphique synchrone : SyncCharts.* Présenté à l'école d'été Temps Réel, 9-12 September 2003, Toulouse(France).

[95] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud. *The synchronous data-flow programming language LUSTRE.* In Special issue on Another Look at Real-Time Systems. Proc. of IEEE, Vol. 79, pp. 1305-1320, 1991.

[96] G. Berry. *The effectiveness of synchronous languages for the development of safety-critical systems.* White paper, Esterel Technologies, 2003.

[97] F. Boulanger. *Intégration de modules synchrones dans la programmation par objets.* Thèse de doctorat, Université de Paris 11, Orsay, décembre 1993.

[98] F. Boulanger, C. André, M. A. Péraldi, J. P. Rigault, G. Vidal-Naquet. *Objets et programmation synchrone.* In Modélisation des systèmes réactifs, pp. 55-62, 1996. Brest(France).

[99] F. Boulanger, H. Delebecque, G. Vidal-Naquet. *Intégration de modules synchrones dans un langage à objets.* In Real-Time Systems Conference, pp. 245-260, 1994. Paris (France).

[100] G. Berry. *A hardware implementation of pure Esterel.* Miami, January 1991. ACM Workshop on Formal Methods in VLSI Design.

[101] C ANDRÉ, H BOUFAIED. *Execution Machine for Synchronous Languages.* IDPT-2000 (Integrated Design and Process Technology), Dallas (TX), June 4-8, 2000.

[102] F. Horn, J.B. Stefani. *On programming and supporting multimedia object synchronization.* The Computer Journal. Vol. 36(1)(1993)4-18.

[103] M-A Peraldi. *Conception et Réalisation de Systèmes Temps Réel par une Approche Synchrone.* Thèse de doctorat, Université de Nice-Sophia Antipolis, Juillet 1993.

[104] C. André, H. Boufaïed, D. Gaffé, J.P. Marmorat. *Environnement pour la programmation synchrone des systèmes réactifs.* In Real-Time and Embedded Systems (RTS&ES'96, pp. 27-41, Paris (France), Janvier 1996. Teknea.

[105] J-L. Scharbarg, J. Battut, L. Marcé. *Un automate programmable fondé sur l'approche synchrone du GRAFCET.* Dans Modélisation des Systèmes Réactifs, Brest, Mars 1996. Afcet.

[106] O. Potonniée. *Etude et prototypage en Esterel de la gestion de processus d'un micro-noyau de système d'exploitation réparti avec garantie de service.* Thèse de doctorat, Université Pieere et Marie Curie, Paris 6, Avril 1996.

[107] A. Bouali. *XEVE : An Esterel Verification Environment.* Technical Report 1997, CMA-ENSMP, Sophia Antipolis, France.

[108] M. Auguin. *Co-conception de systèmes spécialisés sur composant.* Ecole Thématique, Seix (Ariège), du 20 au 23 novembre 2000.

[109] *http ://www.esterel-technologies.com*

[110] *http ://java.sun.com/j2se/1.5.0/docs/guide/jni/*

[111] *The ssc format sorted net table (release ssc5).* The ESTEREL Team. CMA, Ecole des Mines and INRIA. March 26, 1998.

[112] K. M. Chandy and J. Misra. *Asynchronous Distributed Simulation Via a Sequence of Parallel Computations.* Communications of the ACM, vol. 24, no. 11, November 1981, pp. 198-205.

[113] R. M. Fujimoto. *Parallel Discrete Event Simulation.* Communications of the ACM, vol. 33, no.10, October 1990, pp 30-53.

[114] *VHDL-A Design Objective Document, version 2.3.* IEEE DASC 1076.1 Working Group, http ://www.vhdl.org/analog/ftp_files/requirements/DOD_v2.3.txt.

[115] J. Misra. *Distributed Discrete-Event Simulation.* Computing Surveys, vol. 18, no. 1, March 1986, pp. 39-65.

[116] G. Kahn. *The Semantics of a Simple Language for Parallel Programming.* Proc. of the IFIP Congress 74, North-Holland Publishing Co., 1974.

[117] G. Kahn and D. B. MacQueen. *Coroutines and Networks of Parallel Processes.* Information Processing 77, B. Gilchrist, editor, North-Holland Publishing Co., 1977.

[118] C. A. R. Hoare. *Communicating Sequential Processes.* Communications of the ACM, Vol. 21, No. 8, August 1978.

[119] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[120] G. R. Andrews. *Concurrent Programming–Principles and Practice.* Addison-Wesley, 1991.

[121] M. G. Hinchey and S. A. Jarvis. *Concurrent Systems : Formal Developments in CSP.* McGraw-Hill, 1995.

[122] J. Liu. *Continuous Time and Mixed-Signal Simulation in Ptolemy II* Master's Report, UCB/ERL Memorandum M98/74, Dept. of EECS, University of California, Berkeley, CA 94720, December 1998.

[123] S. Edwards. *Synchronous Reactive Systems.* Presentation at the University of Texas, Austin, February, 1997

[124] H. Lieberman. *Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems.* In Proceedings of First ACM Conference on Object-Oriented Programming Systems, Languages and Applications, Portland, Oregon, USA, September 1986.

[125] A. Goldberg, D. Robson., *Smalltalk-80 : The language and its implementation*. Addison-Wesley, 1983, pp(11, 12, 14, 15, 27, 57, 86, 173, 206, 207, 209).

[126] B. Stroustrup. *The C++ Programming Language.* Addision-Wesley, 1996.

[127] F. Terrier. *Borland C++*. 1994.

[128] S. B. Lippman. *C++ Primer*. Addision-Wesley, 1991.

[129] S. B. Lippman. *L'essentiel du C++*. Addision-Wesley, 1992.

[130] B. Meyer. *Eiffel The Language.* Prentice-Hall, 1992.

[131] K. Arnold, J. Gosling. *The Java Programming Language.* Addision-Wesley, 1996.

[132] O. J. Dahl et K. Nygaard. *SIMULA-An Algol-Based Simulation Language.* Communication of the ACM, Vol. 9(9), pages 671-678, 1966.

[133] C. V. Lopes, W. L. Hursch. *Separation of Concerns.* College of Computer Science, Northeastern University, Boston, February 1995.

[134] G. Kiczales. *Towards a New Model of Abstraction in the Engineering of Software.* In Proceedings of IMSA'92. Workshop on Refection and Meta-Level Architecture, 1992.

[135] H. Okamura, Y. Ishikawa. *Object Location Control Using Meta-level Programming.* In Proceedings of the ECOOP'94, pp 299-319, Lecture Notes in Computer Science Vol. 821, Springer-Verlag, Bologne, Italie, Juillet 1994.

[136] M. Aksit, J. Bosch, W. van der Sterren, L. Bergmans. *Real-Time Specification Inheritance Anomalies and Real-Time Filters.* In Proceedings of the ECOOP'94, Lecture Notes in Computer Science, Vol. 821, pp 386-407, Springer-Verlag, Bologne, Italie, Juillet 1994.

[137] Sun Microsystems. *Enterprise JavaBeans.* http ://java.sun.com/products/ejb/.

[138] Object Managment Group. *Corba Component Model.* http ://www.omg.org.

[139] Kiczales, G. *Aspect-Oriented Programming.* In Proc. of the ECOOP'97. Vol. LNCS 1241. Springer-Verlag. June 1997.

[140] Tzilla Elrad, Robert E. Filman, Atef Bader. *Aspect-oriented programming. Comm.* ACM, 44(10) :29-32, October 2001.

[141] C. Szyperski. *Component Software : Beyond Object-Oriented Programming.* Addison-Wesley 1999. ACM press édition.

[142] F. Duclos, J. Estublier, P. Morat. *Describing and Using Non-Functional Aspects in Component Based Applications.* In Proc. of the 1st Aspect-Oriented Software Development (AOSD'02). ACM Press, Enshede, Pays-Bas, 2002.

[143] M. N. Bouraqadi Saâdani. *Un MOP Smalltalk pour l'étude de la composition et de la compatibilité des métaclasses. Application à la programmation par aspects.* Thèse de doctorat en sciences pour l'ingénieur de l'université de nantes, 1999.

[144] S. Matsuoka, A. Yonezawa. *Research Directions in Concurrent Object-Oriented Programming.* Chapter Analysis of inheritance anomaly in object-oriented concurrent programming languages. MIT Press, 1993.

[145] B. Meyer. *Object-Oriented Software Construction (second edition).* Prentice Hall, 1997.

[146] B. Meyer. *Conception et Programmation par Objet (pour du logiciel de qualité).* InterEditions, 1990.

[147] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, New York, 1988.

[148] N. M. N. Bouraqadi-Saâdani and Thomas Ledoux. *Le point sur la programmation par aspects.* Technique et science informatique, 20(4) : 271-311, 2001.

[149] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, N. J, 1976.

[150] C. V. Lopes, G. Kiczales. *Recent Developments in AspectJ.* ECOOP'98 Workshop Reader, chapter numéro 1543vLNCS, Springer-Verlag, 1998.

[151] K. Lieberherr, D. Lorenz, M. Mezini. *Programming with aspectual components.* Rapport numéro NU-CCS-99-01, College of Computer Science, Northastern University, Mars 1999.

[152] *http ://eclipse.org/aspectj/*

[153] S. Chiba. *A MetaObjet Protocol for C++.* In Proceeding of OOPSLA'95, pp 285-299, 1995.

[154] S. Chiba, M. Tatsubori. *Yet Another java.lang.Class.* ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems, Brussels, Belgium, Juillet 1998.

[155] G. Kiczales, E. Hilsdate, J. Humgunin, M. Kersten, J. Palm, William G. Griswold. *An overview of AspectJ.* In J. Lindskov Knudsen, editor, ECOOP'01, vol. 2027 of LNCS, pp 327-353. Springer-Verlag, 2001.

[156] R. Pawlak, L. Seinturier, L. Duchien, G. Florin. *JAC : Aflexible and efficient solution for aspect-oriented programming in java.* In Reflection 2001, LNCS 2192, pp 1-24 September 2001.

[157] S. Chiba. *Load-time structural reflection in Java.* In Proceeding of ECOOP'00, LNCS. Springer Verlag, 2000.

[158] B. C. Smith. *Procedural reflection in programming languages.* PhD Thesis, Massachusetts Institute of Technology, 1982.

[159] B. C. Smith. *What do you mean, meta.* Workshop on Reflection and Metalevel Architectures in OO Programming, ECOOP/OOPSLA'90, Octobre 1990.

[160] T. Watanabe, A. Yonezawa. *Reflection in object-oriented concurrent language.* In Proceeding of OOPSLA'88, ACM, 1988.

[161] J. McAffer. *Meta-level Programming with CodA.* ECOOP'95, volume LNC 952, pages 190-214. Springer-Verlag, 1995.

[162] G. Kiczales, A. Paepcke. *Open implementation and metaobject protocols.* Technical report, Expanded tuturial notes, 1994.

[163] R. J. Stroud, Z. Wu. *Advances in Object-Oriented Metalevel Architecture and Reflection.* Chapter Using Metaobject Protocols to satisfy Non-Functional Requirements, pp 31-52. CRC Press, 1996.

[164] J. Dempsey, V. Cahil. *Aspects of system support for distributed computing.* C. Lopes, G. Kiczales, B. Tekinerdogan, K. Mens, Eds. In Proceeding of the Aspect-Oriented Programming Workshop at ECOOP'97, Juin 1997.

[165] C. P. Lunau. *Is composition of metaobjects=aspect oriented programming.* In Proceeding of the Aspect-Oriented Programming Workshop at ECOOP'98, Juillet 1998.

[166] J. Pryor, N. Bastan. *A reflective architecture for the support of aspect oriented programming in smalltalk.* In Proceeding of the Aspect-Oriented Programming Workshop at ECOOP'99, Juillet 1999.

[167] M. Aksit, L. Bergmans. *An object-oriented language-database integration model : The composition-filter approach.* In Proceeding of ECOOP'92, 1992.

[168] M. Assit, B. Teknerdogan. *Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters.* In Proceeding of ECOOP'98, Juillet 1998.

[169] François-Nicola Demers, Jacques Malenfant. *Reflection in Logic, Functional and Object-Oriented Programming : a Short Comparative Study.* In Workshop of IJCAI'95 on Reflection and Meta-Level Architecture and their Application in A.I., pp 29-38, August 1995.

[170] Pattie Maes. *Computational Reflection.* PhD thesis (thèse de doctorat), Artificial Intelligence Laboratory, Vrije University, Brussel, Belgium, 1987.

[171] Pattie Maes. *Concepts and Experiments in Computational Reflection.* In Proceeding of OOPSLA'87, pp 147-155, Orlando, Florida, 1987. ACM.

[172] Brian Foote, Ralph E. Johnson. *Reflective Facilities in Smalltalk-80.* In Proceeding of OOPSLA'89. ACM, 1989.

[173] D.G. Bobrow, R.G Gabriel, J.L. White *Object Oriented Programming - The CLOS perspective.* Chapter CLOS in Context - The Shape of the Design Space. In ObjectOriented Programming. MIT Press, 1993.

[174] Pierre Cointe. *A Tutorial Introduction to Metaclass Architecture as Provided by Class Oriented Languages.* In Proceeding of International Conference on Fifth Generation Computer Systems, ICOT, Tokyo, Japan, November 1988.

[175] Jacques Malenfant. *Abstraction, encapsulation et réflexion dans les langages à prototypes.* Thèse d'habilitation à diriger des recherches, Université de Nantes - Faculté des Sciences et des Techniques, Avril 1997.

[176] Gregor Kiczales, Jim. des Rivières, Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[177] Ira Forman, Scott H. Danforth. *Putting Metaclass to Work.* Addision Wesley, September 1998.

[178] B. Gowing, V. Cahill. *Meta-Object Protocols for C++ : The Iguana Approach.* In Gregor Kiczales, editor, Priceeding of Reflection'96, pp 137-152, San Francisco, California, April 21-23 1996.

[179] N. Bouraqadi. *Java et la réflexion.* Technical Report 2000-4-INFO, Ecole des Mines de Nantes, January 2000. 1st technical report for the RAM project (Reflection for Adaptable Mobility), partially funded by France Telecom R & D.

[180] Sonya E. Keen. *Object-Oriented Programming in Common Lisp : A Programmer's Guide to CLOS.* Addison-Wesley, Reading, Massachussetts, USA, 1989.

[181] J.P. Briot, P. Cointe. *Programming with Explicit Metaclasses in Smalltalk.* In Proceeding of OOPSLA'89, pp 419-431, New Orleans, Louisiana, USA, 1989, ACM.

[182] Ira R. Forman, M.H. Conner, S. Danforth, L.K. Raper. *Release-to-Release Binary Compatiblity in SOM.* In Proceeding of OOPSLA'95, October 1995.

[183] F. Rivard. *Evolution du Comportement des Objets dans les Langages à Classes Réflexives.* Thèse de doctorat, Université de Nantes, Juin 1997.

[184] DUCASSE S. *Intégration Réflexive de Dépendances dans un Modèle à Classes*. PhD thesis, Université de Nice-Sophia Antipolis, 1997.

[185] C. Zimmermann. *Advances in Object-Oriented Metalevel Architectures and Reflection.* Chapter Metalevels, MOPs and What the Fuzz is All About, pp 3-29. CRC Press, 1996.

[186] P. Mulet. *Réflexion et Langages à Prototypes.* Thèse de doctorat, Université de Nantes, Faculté des Sciences et Techniques, Juillet 1995.

[187] I. Welch, R. Stroud. *From Dalang to Kava — the evolution of a reflective Java extension.* In Pierre Cointe, editor, Proceedings of the second international conference Reflection'99, number 1616 in LNCS, pp 2-21. Springer, July 1999.

[188] Z. Wu, S. Schwiderski. *Reflective Java : Making java even more flexible.* Technical report, ANSA, Feb 1997.

[189] D. Caromel, W. Klauser, J. Vayssière. *Towards Seamless Computing and Metacomputing in Java.* Concurrency Practice and Experience, pp 1043-1061, September-November 1998.

[190] N. M. N. Bouraqadi-Saâdani, T. Ledoux, M. Südhotl. *A reflective infrastructure for coarse-grained strong mobility and its tool-based implementation.* Invited presentation at the international Workshop on Experriences with reflective systems. September 2001.

[191] E. Tanter, N. M. N. Bouraqadi-Saâdani, J. Noyé. *Reflex-towards an open reflective extension of java.* In Proccedings of the 3rd international conference on Reflection and Crosscutting Concerns, volume 2192 of LNCS. Springer Verlag, September 2001.

[192] J. Kleinoeder, M. Golm. *Metajava : An efficient run-time meta architecture for java.* Technical report TR-I4-96-03, University of Erlangen-Nuernberg, IMMD IV, 1996.

[193] A. Oliva, L. E. Buzato. *The design and implementation of Guaraná.* In Proceeding of the Fifth USENIX Conference on Object-Oriented Technologies and Systems, pp 203-216. The USENIX Association, 1999.

[194] E. Truyen, B. Vanhaute, Wouter Joosen, P. Verbaeten, B. Norregaard Jorgensen. *Dynamic and selective combination of extensions in component-based applications.* In Proceeding of the 23rd International Conference on Software Engineering (ICSE'01), May 2001.

[195] B. Redmond, V. Cahill. *Iguana/J : Towards a dynamic and efficient reflective architecture for Java.* In ECOOP 2000 Workshop on Reflection and Metalevel Architectures, June 2000.

[196] M. Feredj, F. Boulanger, M. Mbobi. *An Approach for Domain-Polymorph Component Design.* In Proc. of the 2004th IEEE International Conference on Information Reuse and Integration, november 2004.

[197] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns : Elementss of Reusable Object-Oriented Software.* Addision-Wesley, 1995.

[198] *Le langage Python.* http ://www.python.org.

[199] 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.

[200] *http ://www.w3.org/XML/*

[201] *http ://www.w3.org/Graphics/SVG/*

[202] Gérard Berry and Georges Gonthier. *The Esterel Synchronous Programming Language : Design, Semantics, Implementation.* Science of Computer Programming, 19(2) : 87-152. February 1992.

[203] Lewerentz C., Lindner T. (eds.), *Formal Development of Reactive Systems. Case Study Production Cell*. Springer LNCS 891 (1995).

[204] Lindner. T., *Task Description*. Springer LNCS 891 (1995), 9-21.

[205] E. Börger, L. Mearelli, *Integrating ASMs into the Software Development Life Cycle*. Journal of Universal Computer Science, vol. 3, no. 5 (1997), 603-665. Springer Pub. Co.

[206] M. Feredj. *Intégration de modules Esterel dans Ptolemy II*. Rapport de DEA MISI (Méthodes Informatiques des Systèmes Industriels). Université de Versailles en collaboration avec Ecole Supérieure d'Electricité (Supélec) Gif/Yvette. Septembre 2002.